▶ Stacks / Arrays / Monotonic Stack

✅ Stacks / Arrays / Monotonic Stack – Largest Rectangle in a Histogram

📌 Problem Statement:

You are given an array of bar heights where each bar has a width of `1`.
Find the **area of the largest rectangle** that can be formed inside the histogram.

📥 Example:

```
Input:
heights = [2, 1, 5, 6, 2, 3]

Output:
10

Explanation:
- Rectangle of height 5 and 6 → width 2 → area = 5×2 = 10
```

🔁 Brute Force:

- For each bar, expand left and right while height is ≥ current height.
- Compute area for each such range.
- **Time Complexity**: $O(n^2)$

🧠 Monotonic Stack Intuition:

- Maintain a **stack of indices** of increasing height bars.
- When a bar is **lower** than the top of the stack:
  - Pop from the stack and compute area.
  - Width = current index − index of bar in stack before the popped one − 1
- Do this until stack is empty or top is ≤ current bar.
- Add a dummy bar `0` at the end to flush out remaining bars.

🚀 Optimal Stack Solution

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

```
import java.util.*;
```

```java
public class LargestRectangleHistogram {
    public static int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0;
        int n = heights.length;

        for (int i = 0; i <= n; i++) {
            int h = (i == n) ? 0 : heights[i];

            while (!stack.isEmpty() && h < heights[stack.peek()]) {
                int height = heights[stack.pop()];
                int width = stack.isEmpty() ? i : (i - stack.peek() - 1);
                maxArea = Math.max(maxArea, height * width);
            }

            stack.push(i);
        }

        return maxArea;
    }
}
```

✅ Output

```
Input: [2, 1, 5, 6, 2, 3]
Output: 10

Input: [2, 4]
Output: 4
```

✅ Array Manipulation / Sliding Window / Rotations – Left Rotation

---

📌 Problem Statement:

Given an array of integers and a number `d`, rotate the array **to the left** by `d` positions.

> Elements shifted beyond the first position move to the end.

📥 Example:

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

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

🔁 Brute Force:

- Run a loop `d` times.
- In each iteration:
  - Remove first element.
  - Append it to the end.
- **Time Complexity**: O(d × n)

🧠 Better Intuition:

- Use an auxiliary array:
  - Copy elements from index `d` to `n-1`.
  - Then copy elements from index `0` to `d-1`.

🚀 Optimal Solution (In-place using Reversal Algorithm)

- Reverse:
  - First `d` elements.
  - Then `n-d` elements.
  - Then the whole array.
- **Time Complexity**: O(n)
- **Space Complexity**: O(1)

```java
import java.util.*;

public class LeftRotation {
    public static void rotateLeft(int[] arr, int d) {
        int n = arr.length;
        d %= n;

        reverse(arr, 0, d - 1);
        reverse(arr, d, n - 1);
        reverse(arr, 0, n - 1);
    }

    private static void reverse(int[] arr, int start, int end) {
        while (start < end) {
            int temp = arr[start];
            arr[start++] = arr[end];
```

```
            arr[end--] = temp;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int d = 2;
        rotateLeft(arr, d);
        System.out.println(Arrays.toString(arr));
    }
}
```

✅ Output

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

Input: [10, 20, 30, 40], d = 1
Output: [20, 30, 40, 10]
```

✅ Array Manipulation – Grid Challenge

📌 Problem Statement:

You are given a square grid of characters in the form of an array of strings.
First, sort each row alphabetically.
Then, check if the **columns are also sorted** from top to bottom.

Return "YES" if columns are sorted, otherwise return "NO".

📥 Example:

```
Input:
grid = {
    "ebacd",
    "fghij",
    "olmkn",
    "trpqs",
    "xywuv"
}

Output:
"YES"

Explanation:
After row-wise sorting:
```

```
{
    "abcde",
    "fghij",
    "klmno",
    "pqrst",
    "uvwxy"
}

All columns are sorted.
```

🔁 Brute Force:

- Sort each row.
- For each column, check if it's sorted.
- **Time Complexity**: $O(n^2 \log n)$ for sorting rows + $O(n^2)$ for column check.

🚀 Optimal Solution (Straightforward)

- Sort each row.
- For each column `c`, check if `grid[i][c] <= grid[i+1][c]`
- **Time Complexity**: $O(n^2 \log n)$

✅ Java Code:

```java
public class GridChallenge {
    public static String gridChallenge(String[] grid) {
        int n = grid.length;

        // Sort each row
        for (int i = 0; i < n; i++) {
            char[] chars = grid[i].toCharArray();
            Arrays.sort(chars);
            grid[i] = new String(chars);
        }

        // Check columns
        for (int col = 0; col < grid[0].length(); col++) {
            for (int row = 0; row < n - 1; row++) {
                if (grid[row].charAt(col) > grid[row + 1].charAt(col)) {
                    return "NO";
                }
            }
        }
```

```java
        return "YES";
    }

    public static void main(String[] args) {
        String[] grid = {"ebacd", "fghij", "olmkn", "trpqs", "xywuv"};
        System.out.println(gridChallenge(grid));  // Output: YES
    }
}
```

✅ Output

```
Input:
["ebacd", "fghij", "olmkn", "trpqs", "xywuv"]
Output: "YES"

Input:
["zyx", "wvu", "tsr"]
Output: "NO"
```

✅ Array Manipulation / Greedy – Mark and Toys

📌 Problem Statement:

Mark wants to buy as many toys as possible with a given budget.
You are given a list of toy prices and an integer budget `k`.
Determine the **maximum number of toys** he can purchase.

📥 Example:

```
Input:
prices = [1, 12, 5, 111, 200, 1000, 10]
k = 50

Output:
4

Explanation:
He can buy [1, 5, 10, 12] = 28 total spent → max 4 toys.
```

🔁 Brute Force:

- Generate all subsets and check which ones fit under budget.
- **Time Complexity**: Exponential → Not practical.

🧠 Greedy Intuition:

- Sort the prices.
- Start buying the **cheapest toys first** until the budget runs out.

🚀 Optimal Greedy Solution

- Sort prices.
- Keep a running sum.
- **Time Complexity**: O(n log n)

✅ Java Code:

```
import java.util.*;

public class MarkAndToys {
    public static int maximumToys(int[] prices, int k) {
        Arrays.sort(prices);
        int count = 0;
        int total = 0;

        for (int price : prices) {
            if (total + price <= k) {
                total += price;
                count++;
            } else {
                break;
            }
        }

        return count;
    }

    public static void main(String[] args) {
        int[] prices = {1, 12, 5, 111, 200, 1000, 10};
        int k = 50;
        System.out.println(maximumToys(prices, k));  // Output: 4
    }
}
```

✅ Output

```
Input:
prices = [1, 12, 5, 111, 200, 1000, 10], k = 50
```

```
Output: 4
```

✅ Array Manipulation / Greedy – Permuting Two Arrays

📌 Problem Statement:

You are given two arrays `A` and `B`, both of length `n`, and an integer `k`.
Can you permute both arrays such that **A[i] + B[i] ≥ k for all i**?

Return `"YES"` if it's possible, otherwise return `"NO"`.

📥 Example:

```
Input:
A = [2, 1, 3]
B = [7, 8, 9]
k = 10

Output:
"YES"

Explanation:
Sorted A = [1, 2, 3], B = [9, 8, 7]
A[0]+B[0]=10, A[1]+B[1]=10, A[2]+B[2]=10 ✅
```

🔁 Brute Force:

- Try all permutations of both arrays.
- For each permutation, check the sum condition.
- **Time Complexity**: O(n!²) — not practical.

🚀 Optimal Greedy Solution:

- Sort A in ascending order.
- Sort B in **descending** order.
- Check if `A[i] + B[i] ≥ k` for all i.
- **Time Complexity**: O(n log n)

✅ Java Code:

```
import java.util.*;

public class PermutingTwoArrays {
    public static String twoArrays(int k, int[] A, int[] B) {
```

```
        Arrays.sort(A);
        Arrays.sort(B);
        int n = A.length;

        // Reverse B to make it descending
        for (int i = 0; i < n / 2; i++) {
            int temp = B[i];
            B[i] = B[n - 1 - i];
            B[n - 1 - i] = temp;
        }

        for (int i = 0; i < n; i++) {
            if (A[i] + B[i] < k) return "NO";
        }
        return "YES";
    }

    public static void main(String[] args) {
        int[] A = {2, 1, 3};
        int[] B = {7, 8, 9};
        int k = 10;
        System.out.println(twoArrays(k, A, B));  // Output: YES
    }
}
```

✅ Output

```
Input:
A = [2, 1, 3], B = [7, 8, 9], k = 10

Output:
"YES"
```

✅ Heap / Priority Queue – Jesse and Cookies

📌 Problem Statement:

Jesse loves cookies. Each cookie has a **sweetness level**. Jesse wants to **combine** the two least sweet cookies until all cookies have **at least sweetness** `k`.

To combine two cookies with sweetness `a` and `b`:
New cookie = `a + 2*b`

Determine the **minimum number of operations** required.
Return `-1` if it's not possible.

📥 Example:

```
Input:
cookies = [1, 2, 3, 9, 10, 12]
k = 7

Output:
2

Explanation:
- Combine 1 + 2×2 = 5 → [3, 5, 9, 10, 12]
- Combine 3 + 2×5 = 13 → [9, 10, 12, 13]
Now all ≥ 7 ✅
```

🔁 Brute Force:

- Sort array on every iteration.
- Pick smallest 2, combine, insert again.
- **Time Complexity**: $O(n^2 \log n)$

🚀 Optimal (Min Heap):

- Use a **min-heap** (priority queue).
- Pop 2 smallest elements.
- Push combined cookie.
- Repeat until heap top ≥ `k` or heap size < 2.
- **Time Complexity**: $O(n \log n)$

✅ Java Code:

```java
import java.util.*;

public class JesseAndCookies {
    public static int cookies(int k, int[] A) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : A) minHeap.add(num);

        int operations = 0;

        while (minHeap.size() > 1 && minHeap.peek() < k) {
            int least = minHeap.poll();
            int secondLeast = minHeap.poll();
            int combined = least + 2 * secondLeast;
```

```
            minHeap.add(combined);
            operations++;
        }

        return (minHeap.peek() >= k) ? operations : -1;
    }

    public static void main(String[] args) {
        int[] cookies = {1, 2, 3, 9, 10, 12};
        int k = 7;
        System.out.println(cookies(k, cookies));  // Output: 2
    }
}
```

✅ Output

```
Input:
cookies = [1, 2, 3, 9, 10, 12], k = 7

Output: 2
```

✅ Heap / Priority Queue – Find the Running Median

📌 **Problem Statement:**

Given a stream of integers, for each new element added, print the **median** of all elements seen so far.

📥 **Example:**

```
Input stream: [12, 4, 5, 3, 8, 7]

Output:
12
8
5
4.5
5
6
```

🤔 Intuition:

To efficiently compute the **running median**:

- Use **two heaps**:
  - Max-heap for **lower half**
  - Min-heap for **upper half**

- Ensure:
    - Size difference ≤ 1
    - Max-heap top ≤ Min-heap top

Median is:

- If equal size → average of tops
- Else → top of the bigger heap

🔁 Brute Force:

- Sort the entire array on each insertion.
- Find the middle element.
- **Time Complexity**: $O(n^2 \log n)$

🚀 Optimal Heap-Based Solution:

- **MaxHeap**: for smaller half
- **MinHeap**: for larger half
- Keep heaps balanced.
- **Time Complexity**: $O(n \log n)$

✅ Java Code:

```
import java.util.*;

public class RunningMedian {
    public static List<Double> runningMedian(int[] nums) {
        List<Double> result = new ArrayList<>();
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>
(Collections.reverseOrder());  // left
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();  // right

        for (int num : nums) {
            if (maxHeap.isEmpty() || num <= maxHeap.peek()) {
                maxHeap.offer(num);
            } else {
                minHeap.offer(num);
            }

            // Balance heaps
            if (maxHeap.size() > minHeap.size() + 1) {
                minHeap.offer(maxHeap.poll());
            } else if (minHeap.size() > maxHeap.size()) {
```

```java
                maxHeap.offer(minHeap.poll());
            }

            // Get median
            if (maxHeap.size() == minHeap.size()) {
                result.add((maxHeap.peek() + minHeap.peek()) / 2.0);
            } else {
                result.add((double) maxHeap.peek());
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {12, 4, 5, 3, 8, 7};
        List<Double> medians = runningMedian(nums);
        for (Double m : medians) {
            System.out.println(m);
        }
    }
}
```

✅ Output:

```
12.0
8.0
5.0
4.5
5.0
6.0
```