Q127: Boats to Save People

1. Problem Understanding

- You're given:
  - an array people[] of weights
  - unlimited boats
  - each boat can carry at most 2 people
  - total weight inside a boat must be â‰¤ limit
- Goal: Find the minimum number of boats needed.
- This is a classic greedy pairing problem.

---

2. Constraints

- 1 â‰¤ N â‰¤ 5 * 10^4
- 1 â‰¤ people[i] â‰¤ limit â‰¤ 3 * 10^4
- Every person must be placed in a boat
- At most 2 people per boat
- Order does not matter

---

3. Edge Cases

- N = 1 â†’ answer = 1
- Everyone too heavy to pair â†’ each gets its own boat
- Perfect pairs exist â†’ minimum boats
- All weights identical
- Very light + very heavy pairing scenario

---

4. Examples

```
Example 1
N = 2, limit = 3
people = [1, 2]

Output:
1
Pair (1, 2)

Example 2
N = 4, limit = 5
people = [3, 5, 3, 4]

Possible pairs:
(5)
```

```
(4)
(3)
(3)

Output:
4
```

---

5. Approaches

Approach 1: Try All Pairings (Brute Force: TLE)

Idea:

- Try every combination of two people to place into boats.
- âŒ Why it fails
    - O(NÂ²) or worse
    - N up to 50k â†’ impossible

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(NÂ²) or worse
- ðŸ’¾ Space Complexity
    - O(1)

Approach 2: Greedy (Two-Pointer Technique) Optimal

Idea:

- We want to place heavy people first, and try to pair them with the lightest person available.
- Why it works?
    - If the heaviest cannot pair with the lightest, they cannot pair with anyone.
    - If the heaviest can pair with the lightest, it reduces boat count.
- This greedy rule ensures minimal boats.

Steps:

- Sort the array
- Use two pointers:
    - i at start (lightest)
    - j at end (heaviest)
- While i <= j:
    - If people[i] + people[j] <= limit
        - pair them â†’ i++, j--
    - Else
        - send heavy alone â†’ j--
    - Increment boats count

Java Code:

```java
public int numRescueBoats(int[] people, int limit) {
    Arrays.sort(people);
    int i = 0, j = people.length - 1;
    int boats = 0;

    while (i <= j) {
        if (people[i] + people[j] <= limit) {
            i++;   // lightest used
            j--;   // heaviest used
        } else {
            j--;   // heaviest goes alone
        }
        boats++;
    }

    return boats;
}
```

ðŸ' Intuition Behind the Approach:

- The key observation:
  - The heaviest person must be placed in a boat now.
  - If they can be paired with the lightest, it's optimal to do so.
  - If not, pairing them with anyone heavier is impossible â†' they go alone.
- By always using the least possible space per boat and maximizing pairing,
- we guarantee minimum boats.
- This matches the greedy strategy of "minimize waste per step."

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - Sorting â†' O(N log N)
  - Two-pointer scan â†' O(N)
  - Total:
    - O(N log N)
- ðŸ'¾ Space Complexity
  - O(1) extra (sorting in-place, no extra arrays)

---

6. Justification / Proof of Optimality

- The greedy strategy is optimal because pairing the heaviest person with anyone heavier cannot work.
- If the heaviest can pair with the lightest, itâ€™s always optimal to pair them, as:
  - It â€œsavesâ€ one person
  - It avoids wasting a boat
- Sorting + two pointers ensures we try the best possible pairing at each step.
- This is the solution used in major interview problems.

---

7. Variants / Follow-Ups

- Boats with capacity for 3 people
- Boats with variable weight limits
- Each boat can take unlimited people but max total weight K (bin packing light version)
- Grouping students into taxis (same logic)
- Pairing problems in greedy optimization

---

8. Tips & Observations

- Always pair heavy with lightest first
- If they cannot pair, heavy MUST go alone
- Sort only once
- Use two pointers, not nested loops
- When i == j, that one person requires a boat
- Never skip pairing opportunities; greedy is optimal

Q128: Subarray Product Less Than K

---

1. Problem Understanding

- You are given:
  - an array nums[]
  - an integer k
- You must return how many contiguous subarrays have a product < k.
- A subarray must be continuous.

---

2. Constraints

- 1 â‰¤ n â‰¤ 3 * 10^4
- 1 â‰¤ nums[i] â‰¤ 1000
- 0 â‰¤ k â‰¤ 10^6
- Time must be better than O(NÂ²) (since N is 30k)

---

3. Edge Cases

- k <= 1 â†’ answer = 0
  - Because product of positive integers â‰¥ 1 cannot be < 1.
- All elements > k â†’ only single elements < k count
- All elements = 1 â†’ every subarray counts

---

4. Examples

```
Example 1
Input

4
10 5 2 5
100
Output

8
Explanation

The 8 subarrays that have product less than 100 are: [10], [5], [2], [5], [10, 5],
[5, 2], [2, 5], [5, 2, 5]

Note that [10, 5, 2] is not included as the product of 100 is not strictly less
than k.

Example 2
Input

3
1 2 3
0
Output

0
Explanation

No subarray is possible with product less than K.
```

---

5. Approaches

Approach 1: Brute Force (Nested Loops)

Idea:

- Check every possible subarray, compute product, count if < k.
- â□Œ Why it fails
  - Worst case:
  - 30,000 elements
  - 450 million subarrays
  - Too slow

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(NÂ²)
- ðŸ'¾ Space Complexity
  - O(1)

Approach 2: Sliding Window (Optimal)

Idea:

- Since all nums[i] > 0:
  - Expanding the window increases product
  - Shrinking the window decreases product
- So we use a sliding window:
  - Use left and right pointers
  - Maintain product of current window
  - If product < k, then:
    - all subarrays ending at right and starting from [left..right] are valid
    - count += (right - left + 1)
  - Otherwise shrink from left
- This works because for each right, every subarray inside the window is valid.

Steps:

- Initialize:
  - product = 1
  - left = 0
  - count = 0
- For each right from 0â†'n-1:
  - multiply product with nums[right]
  - while product â‰¥ k:
  - divide product by nums[left] and move left
  - now product < k, so:
  - add (right - left + 1) to count
  - (these are valid subarrays ending at right)

Java Code:

```
public int numSubarrayProductLessThanK(int[] nums, int k) {
    if (k <= 1) return 0;

    int left = 0;
    int count = 0;
    long product = 1;

    for (int right = 0; right < nums.length; right++) {
        product *= nums[right];

        while (product >= k) {
            product /= nums[left];
            left++;
        }

        count += (right - left + 1);
    }
```

```
        return count;
    }
```

ðŸ’ Intuition Behind the Approach:

- Every time we expand our window by including nums[right],
- the only reason product could become â‰¥ k is because the window is too large.
- Shrinking from the left gradually reduces product until the window is valid again.
- Once valid:
  - Every subarray ending at right is valid:
    - [left..right]
    - [left+1..right]
    - [left+2..right]
    - ...
    - [right..right]
  - Count = (right - left + 1)
- Since each element enters and leaves the window once, total operations are linear.
- It is a classic sliding-window two-pointer trick.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - Sliding window â†’ O(N)
  - Because each element is multiplied once and divided once.
- ðŸ’¾ Space Complexity
  - O(1)
  - Only variables used.

---

6. Justification / Proof of Optimality

- Sliding window works because:
  - All numbers are positive
  - Increasing window increases product
  - Decreasing window decreases product
- Therefore, window is monotonic and expanding/shrinking is valid.
- This gives the optimal solution.

---

7. Variants / Follow-Ups

- Subarray sum < k (same logic)
- Subarray averages < k
- Subarray with product â‰¤ k
- Count of subarrays with at most K distinct elements
- Longest subarray with product < k
- All follow the sliding-window pattern.

---

8. Tips & Observations

- If k <= 1, return 0 immediately
- Use long for product (prevent overflow)
- Never reset productâ€"always shrink window naturally
- Positive numbers guarantee monotonic window behavior
- (right - left + 1) is the key formula to count subarrays

Q129: Pair in array with difference k

---

1. Problem Understanding

- Youâ€™re given an array nums and an integer k.
- A k-diff pair is defined as a pair (nums[i], nums[j]) such that:
  - i != j
  - nums[i] - nums[j] == k
  - Pairs must be unique, meaning (1,3) counted once even if elements repeat multiple times.

---

2. Constraints

- 1 <= n <= 10000
- 0 <= k <= 10^7
- 1 <= nums[i] <= 10^7

---

3. Edge Cases

- When k < 0: No pair possible because difference cannot be negative.
- When k == 0: We are looking for numbers that appear at least twice.
- Duplicate elements must not create duplicate pairs.
- Large numbers but manageable constraints.

---

4. Examples

```
Example 1

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

Output: 2

Pairs: (1,3), (3,5)

Example 2

Input: nums = [1, 3, 1, 5, 4], k = 0
```

```
Output: 1

Pair: (1,1) since 1 appears twice.
```

---

5. Approaches

Approach 1: Using HashSet + HashMap (Most Efficient)

Idea:

- For k > 0:
  - If a number x exists, check if x + k also exists.
- For k == 0:
  - Count numbers that occur at least twice.
- Use set/hashmap to ensure uniqueness.

Steps:

- If k < 0 â†' return 0.
- Build a frequency map of all numbers.
- If k == 0 â†' count numbers with frequency >= 2.
- If k > 0 â†' for each number x, check if x + k exists.
- Count such unique pairs.

Java Code:

```java
public static int findPairs(int[] nums, int k) {
    if(k < 0) return 0;

    HashMap<Integer, Integer> map = new HashMap<>();
    for(int x : nums) map.put(x, map.getOrDefault(x, 0) + 1);

    int count = 0;

    if(k == 0){
        for(int key : map.keySet()){
            if(map.get(key) >= 2) count++;
        }
    } else {
        for(int key : map.keySet()){
            if(map.containsKey(key + k)) count++;
        }
    }

    return count;
}
```

ðŸ’ Intuition Behind the Approach:

- A difference k basically means:
- Find pairs where second number is exactly k more than the first.
- HashMap allows O(1) lookup to check whether the "partner element" exists.
- Using keys avoids double counting duplicates.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(n)
    - Because:
        - Building hashmap â†’ O(n)
        - Iterating keys â†’ O(n)
        - Each lookup â†’ O(1)
- ðŸ’¾ Space Complexity
    - O(n)
    - Because hashmap stores frequencies of up to n elements.

Approach 2: Sorting + Two Pointers

Idea:

- Sort the array.
- Use two pointers i and j and ensure:
    - nums[j] - nums[i] == k
    - i != j
- Skip duplicates to maintain unique pairs.

Steps:

- Sort array.
- Keep two pointers:
    - If difference < k â†’ move j.
    - If difference > k â†’ move i.
    - If equal â†’ record pair, move both and skip duplicates.

Java Code:

```
public static int findPairs(int[] nums, int k) {
    Arrays.sort(nums);
    int i = 0, j = 1, count = 0;
    int n = nums.length;

    while(i < n && j < n){
        if(i == j){
            j++;
            continue;
        }
```

```
        int diff = nums[j] - nums[i];

        if(diff < k){
            j++;
        } else if(diff > k){
            i++;
        } else {
            count++;
            int a = nums[i];
            int b = nums[j];
            while(i < n && nums[i] == a) i++;
            while(j < n && nums[j] == b) j++;
        }
    }

    return count;
}
```

ðŸ' Intuition Behind the Approach:

- Sorting groups equal elements together â†' easy to skip duplicates.
- The difference increases automatically as j moves right.
- Two pointers ensure linear scanning instead of nested loops.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n log n)
  - Because of sorting.
    - Two-pointer scan is O(n).
- ðŸ'¾ Space Complexity
  - O(1) or O(n) depending on sorting implementation.

Approach 3: Brute Force (For Understanding Only)

Idea:

- Compare each pair (i,j) using nested loops.
- Use a set to ensure unique pairs.

Java Code:

```
public static int findPairs(int[] nums, int k) {
    HashSet<String> set = new HashSet<>();
    int n = nums.length;

    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(nums[i] - nums[j] == k){
                int a = nums[j];
                int b = nums[i];
```

```
                set.add(a + "#" + b);
            }
            if(nums[j] - nums[i] == k){
                int a = nums[i];
                int b = nums[j];
                set.add(a + "#" + b);
            }
        }
    }

    return set.size();
}
```

ðŸ' Intuition Behind the Approach:

- Straightforward checking of all possible pairs.
- Using a string key avoids counting a pair twice.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n^2)
  - Because all pairs are checked.
- ðŸ'¾ Space Complexity
  - O(n) due to set storing unique pairs.

---

6. Justification / Proof of Optimality

- Approach 1 is optimal because hash lookups allow instant pairing.
- Sorting approach is also good but slower.
- Brute force is only for understanding.

---

7. Variants / Follow-Ups

- Count unordered k-diff pairs â†' modify condition to abs(nums[i] - nums[j]) == k.
- Return the pairs themselves, not just count.
- Handle extremely large inputs (stream-based frequency counting).

---

8. Tips & Observations

- When k == 0, the problem becomes count duplicates.
- Avoid overcounting: sets/maps are crucial.
- Difference pairs are directional; (a,b) is same as (b,a) if difference fixed.

Q130: Maximum width jump

---

1. Problem Understanding

- You are given an array nums.
- A ramp is a pair (i, j) such that:
  - i < j
  - nums[i] <= nums[j]
  - The width is:
    - width = j - i
- Your task is to find the maximum width among all valid ramps.
- If no ramp exists, return 0.

---

2. Constraints

- 2 <= n <= 50000
- Values up to 5 * 10^4
- Must use O(n) or O(n log n) approach.

---

3. Edge Cases

- Strictly decreasing array â†’ no ramp â†’ return 0
- Multiple valid ramps; pick max width
- Duplicate values help (since <= condition)

---

4. Examples

```
Example 1
nums = [6, 0, 8, 2, 1, 5]
max ramp = (1, 5) because nums[1]=0 <= nums[5]=5
width = 5 - 1 = 4
Output 4
Example 2
nums = [9, 8, 1, 0, 1, 9, 4, 0, 4, 1]
max ramp = (2, 9) => 1 <= 1
width = 7
Output 7
```

---

5. Approaches

Approach 1: Monotonic Decreasing Stack + Backward Scan (Optimal O(n))

Idea:

- Build a stack of indices where values are strictly decreasing.
- These are the best possible left-side candidates.
- Traverse from right (j = n-1 to 0):

    - While the top of the stack forms a valid ramp:
        - compute width
        - pop (we already found the best j for that index)

Java Code:

```
public static int maxWidthRamp(int[] nums) {
    Stack<Integer> st = new Stack<>();

    for (int i = 0; i < nums.length; i++) {
        if (st.isEmpty() || nums[i] < nums[st.peek()]) {
            st.push(i);
        }
    }

    int ans = 0;

    for (int j = nums.length - 1; j >= 0; j--) {
        while (!st.isEmpty() && nums[st.peek()] <= nums[j]) {
            ans = Math.max(ans, j - st.peek());
            st.pop();
        }
    }

    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- A left index with a bigger value than previous ones is useless â†' skip
- Stack keeps only good smallest-left candidates
- Scanning from right ensures maximum width

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
    - O(n) because:
        - each index is pushed once
        - each popped at most once
- ðŸ'¾ Space Complexity
    - O(n)

Approach 2: Sorting + Two Pointers

Idea:

- Create pairs (value, index)
- Sort by value (if tie â†' index)
- Now all value[i] <= value[j] is guaranteed in sorted order.
- Use two pointers to maintain:

    - minLeftIndex
    - currentRightIndex

Steps:

- Create arr = [(nums[i], i)] pairs
- Sort arr by value increasing
- Maintain:
    - minIndex = +âˆž
    - For each pair (value, index):
        - answer = max(answer, index - minIndex)
        - update minIndex
- This is true two-pointers, because:
    - sorted array â†' left â‰¤ right automatically
    - only track minimal index so far as left pointer
    - current index acts as right pointer

Java Code:

```
public static int maxWidthRamp(int[] nums) {

    int n = nums.length;

    // arr[i] = {nums[i], i} â†' store (value, originalIndex)
    int[][] arr = new int[n][2];

    // Fill the pair array
    for (int i = 0; i < n; i++) {
        arr[i][0] = nums[i];  // value
        arr[i][1] = i;        // index
    }

    // Sort by value (ascending).
    // After sorting, for any two pairs a and b:
    // a[0] <= b[0] ensures nums[left] <= nums[right]
    Arrays.sort(arr, (a, b) -> a[0] - b[0]);

    int minIndex = Integer.MAX_VALUE;  // track smallest index seen so far
    int ans = 0;                       // maximum width ramp

    // Iterate over sorted pairs (value, index)
    for (int[] p : arr) {

        int idx = p[1];  // original index of current element

        // If current index is smaller â†' becomes new left boundary
        if (idx < minIndex) {
            minIndex = idx;
        }

        // idx > minIndex â†' we found a valid ramp: minIndex < idx AND
```

```
nums[minIndex] <= nums[idx]
        else {
            // width = rightIndex - leftIndex
            ans = Math.max(ans, idx - minIndex);
        }
    }

    return ans;
}
```

ðŸ’ Intuition Behind the Approach:

- Sorting by value ensures nums[left] <= nums[right]
- Only check whether the index order also matches (leftIndex < rightIndex)
- Maintain the smallest left index so far (this becomes the left pointer)
- Current element acts as right pointer
- This becomes a clean, stable 2-pointer system on sorted value-index pairs.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n log n) due to sorting
- ðŸ’¾ Space Complexity
  - O(n)

Approach 3: Brute Force (O(nÂ²), not recommended)

Idea:

- Try every (i, j) pair from right to left.

Java Code:

```
public static int maxWidthRamp(int[] nums) {
    int ans = 0;
    int n = nums.length;

    for (int i = 0; i < n; i++) {
        for (int j = n - 1; j > i; j--) {
            if (nums[i] <= nums[j]) {
                ans = Math.max(ans, j - i);
                break;
            }
        }
    }

    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- Simple but too slow.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n^2)
- ðŸ'¾ Space Complexity
  - O(1)

---

6. Justification / Proof of Optimality

- Monotonic stack gives best performance: O(n)
- Sorting + two pointers is clean and intuitive: O(n log n)
- Brute force is useful only for understanding.

---

7. Variants / Follow-Ups

- Count number of valid ramps
- Return the pair itself
- Reverse condition: find (i, j) such that nums[i] >= nums[j]
- 2D variant in matrices

---

8. Tips & Observations

- Monotonic structures drastically reduce search space
- Sorting transforms ramp condition into simple index comparison
- Always check decreasing patterns â€" they usually suggest stacks

---

Q131: Maximum Consecutive Ones 2

1. Problem Understanding

- You are given a binary array nums and an integer k.
- You may flip at most k zeroes to ones. Find the maximum length of a contiguous subarray that contains only 1s after at most k flips.

---

2. Constraints

- 1 <= nums.length <= 10^5
- 0 <= nums[i] <= 1
- 0 <= k <= nums.length
- Need an O(n) or O(n log n) solution for large n.

3. Edge Cases

- k == 0: standard maximum consecutive ones (no flips).
- k >= count(0s): you can flip all zeros â†' answer = n.
- All zeros or all ones arrays.
- n = 1.
- Very large k relative to n.

4. Examples

```
Example 1
Input: n=11, k=2
nums = [1,1,1,0,0,0,1,1,1,1,0]
Output: 6
Explanation: flip two zeros to get [1,1,1,0,0,1,1,1,1,1,1] â†' longest 6.

Example 2
Input: n=4, k=4
nums = [0,0,0,1]
Output: 4
Explanation: flip all zeros â†' [1,1,1,1].
```

5. Approaches

Approach 1: Sliding Window / Two Pointers (Optimal, O(n))

Idea:

- Maintain a window [l, r] that contains at most k zeros. Expand r and when zeros exceed k, move l until zeros â‰¤ k. Track the maximum window length.

Steps:

- Initialize l = 0, zeros = 0, ans = 0.
- For r from 0 to n-1:
  - If nums[r] == 0 â†' zeros++.
  - While zeros > k: if nums[l] == 0 â†' zeros--; l++.
  - Update ans = max(ans, r - l + 1).
- Return ans.

Java Code:

```java
public static int longestOnes(int[] nums, int k) {
    int l = 0;
    int zeros = 0;
    int ans = 0;
```

```
    for (int r = 0; r < nums.length; r++) {
        if (nums[r] == 0) zeros++;
        while (zeros > k) {
            if (nums[l] == 0) zeros--;
            l++;
        }
        ans = Math.max(ans, r - l + 1);
    }
    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- We want the largest contiguous block with â‰¤ k zeros â€" that is exactly a variable-length window constrained by a count.
- Expanding r increases candidate length; contracting l maintains feasibility.
- Each index enters and leaves the window at most once â†' linear time.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n) â€" each r moves once; l moves at most n times.
  - Why: the while loop increments l only when zero count exceeds k; total increments across entire scan â‰¤ n.
- ðŸ'¾ Space Complexity
  - O(1) â€" only counters and pointers.

Approach 2: Prefix Sum of Zero Indices + Binary Search (O(n log n))

Idea:

- Store indices of all zeros.
- To build a window that flips at most k zeros, use the zero-list to quickly compute the farthest right boundary.
- For each zero-index, find the (k+1)-th zero to the right â†' defines max window

Steps:

- Build zeroIdx list of positions where nums[i] == 0.
- If zeroIdx.size() <= k â†' you can flip all zeros â†' answer = n.
- Otherwise, for each i:
  - Let leftZero = zeroIdx[i]
  - Let rightZero = zeroIdx[i + k]
  - The max window is between previous zero and next zero boundaries.

Java Code:

```
public static int longestOnesPrefix(int[] nums, int k) {
    ArrayList<Integer> zeros = new ArrayList<>();
```

```
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == 0) zeros.add(i);
    }

    if (zeros.size() <= k) return nums.length;

    int ans = 0;

    for (int i = 0; i + k < zeros.size(); i++) {
        int leftZero = zeros.get(i);
        int rightZero = zeros.get(i + k);

        int leftBoundary = (i == 0) ? 0 : zeros.get(i - 1) + 1;
        int rightBoundary = (i + k == zeros.size() - 1) ? nums.length - 1 :
zeros.get(i + k + 1) - 1;

        ans = Math.max(ans, rightBoundary - leftBoundary + 1);
    }

    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- The array becomes divided by zeros.
- Flipping k consecutive zeros means we take a window between the (i-1)-th zero and (i+k+1)-th zero.
- Zero positions give perfect left/right window boundaries.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - Collecting zeros: O(n)
  - Looping over zeros: O(z) where z = number of zeros
  - Total: O(n)
- ðŸ'¾ Space Complexity
  - O(z) to store zero indices.

Approach 3: Brute Force (O(n^2), For understanding only)

Idea:

- For each start i, expand j and count zeros, stop when zeros exceed k, update maximum length.

Java Code:

```
public static int longestOnesBrute(int[] nums, int k) {
    int ans = 0;

    for (int i = 0; i < nums.length; i++) {
        int zeros = 0;
```

```
        for (int j = i; j < nums.length; j++) {
            if (nums[j] == 0) zeros++;
            if (zeros > k) break;

            ans = Math.max(ans, j - i + 1);
        }
    }

    return ans;
}
```

ðŸ’ Intuition Behind the Approach:

- Expands every possible window â€" explores all choices.
- Too slow for large input but good for conceptual understanding.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n^2)
  - Why: nested loops.
- ðŸ'¾ Space Complexity
  - O(1).

---

6. Justification / Proof of Optimality

- Sliding window is the optimal solution because it naturally models â€œat most k bad itemsâ€.
- Prefix-zero-index solution is useful when analyzing zero boundaries or performing k-group constraints.
- Brute force only for learning correctness pattern.

---

7. Variants / Follow-Ups

- Flip exactly k zeros, not at most.
- Longest substring with at most k replacements (string version).
- Max consecutive ones with cost-per-flip scenarios.
- Generalized to non-binary arrays using sum â‰¤ k constraints.

---

8. Tips & Observations

- Anytime you hear â€œflip at most k itemsâ€ â†' sliding window with a count.
- Zeros act as boundaries; analyzing their positions gives alternate solutions.
- When k is 0, the sliding window collapses to simple consecutive ones counting.

Q132: Max Number of K-Sum Pairs

---

1. Problem Understanding

- We are given an array nums and an integer k.
- You can perform operations where you remove two numbers whose sum is exactly k.
- Return the maximum number of operations you can perform.

---

2. Constraints

- 1 <= nums.length <= 100000
- 1 <= nums[i] <= 10^9
- 0 <= k <= 10^9
- Large input â†' must aim for O(n) or O(n log n) solutions.

---

3. Edge Cases

- All numbers greater than k â†' no pairs.
- k = 0 edge half-case (pairs must be identical numbers).
- Frequent duplicates.
- Large values â†' only integer arithmetic needed.
- Only one valid pair existing.

---

4. Examples

```
Example 1
nums = [1,2,3,4], k = 5 â†' output = 2
(1+4), (2+3)

Example 2
nums = [3,1,3,4,3], k = 6 â†' output = 1
Only possible pair: (3,3)
```

---

5. Approaches

Approach 1: Brute Force (O(nÂ²))

Idea:

- For each element, search for its complement k - nums[i].
- If found and not used, remove both.
- Continue until no more pairs.

Java Code:

```
```

```java
public static int maxOperations(int[] nums, int k) {
    int n = nums.length;
    boolean[] used = new boolean[n];
    int ops = 0;

    for (int i = 0; i < n; i++) {
        if (used[i]) continue;
        for (int j = i + 1; j < n; j++) {
            if (!used[j] && nums[i] + nums[j] == k) {
                used[i] = used[j] = true;
                ops++;
                break;
            }
        }
    }
    return ops;
}
```

ðŸ' Intuition Behind the Approach:

- Directly check all pairs.
- Works but extremely slow for large inputs.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(nÂ²) because every pair of indices is checked.
- ðŸ'¾ Space Complexity
    - O(n) for the used[] array.

Approach 2: Sorting + Two Pointers (O(n log n))

Idea:

- Sort the array.
- Use two pointers (l, r):
- If nums[l] + nums[r] == k â†' count operation, move both.
- If sum < k â†' move left.
- If sum > k â†' move right.

Java Code:

```java
public static int maxOperations(int[] nums, int k) {
    Arrays.sort(nums);
    int l = 0, r = nums.length - 1, ops = 0;

    while (l < r) {
        int sum = nums[l] + nums[r];
        if (sum == k) {
            ops++;
```

```
            l++;
            r--;
        } else if (sum < k) {
            l++;
        } else {
            r--;
        }
    }
    return ops;
}
```

ðŸ' Intuition Behind the Approach:

- Sorting helps control the sum direction.
- Two pointers efficiently find matching pairs.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n log n) due to sorting.
  - Two-pointer scan is O(n).
- ðŸ'¾ Space Complexity
  - O(1) ignoring sorting space.

Approach 3: HashMap Counting (Optimal O(n))

Idea:

- Use a frequency map:
  - For each number x, its required partner is k - x.
  - If partner exists in map:
    - Use one from map â†' count operation.
  - Else:
    - Store current number in map.
- Very efficient since each element is processed once.

Java Code:

```
public static int maxOperations(int[] nums, int k) {
    HashMap<Integer, Integer> map = new HashMap<>();
    int ops = 0;

    for (int x : nums) {
        int need = k - x;

        if (map.getOrDefault(need, 0) > 0) {
            ops++;
            map.put(need, map.get(need) - 1);
        } else {
            map.put(x, map.getOrDefault(x, 0) + 1);
```

```
        }
    }
    return ops;
}
```

ðŸ' Intuition Behind the Approach:

- For each number, try to find its partner immediately.
- If partner previously seen â†' form pair.
- Else store number.
- Map ensures constant-time lookup â†' very fast.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(n)
  - Each element inserted or matched exactly once.
- ðŸ'¾ Space Complexity
  - O(n) in worst case (no pairs).

---

6. Justification / Proof of Optimality

- Brute force is too slow for n = 10^5.
- Sorting+two-pointers is good, clean, commonly used.
- HashMap is optimal, used in almost all top solutions.
- HashMap approach avoids sorting and matches pairs instantly.

---

7. Variants / Follow-Ups

- Count unique pairs instead of removing.
- Print actual pairs instead of count.
- Generalize to k-sum using hashing.
- Use multiset/map instead of hashmaps in languages like C++.

---

8. Tips & Observations

- Any problem where you match â€œtwo numbers summing to kâ€ strongly suggests:
  - HashMap
  - Frequency count
  - Two pointers after sorting
- Removing elements = choosing pairs; HashMap excels at this.

Q133: Count Pair Sum

---

1. Problem Understanding

- You are given two sorted, distinct-element arrays arr1 (size m) and arr2 (size n).
- You must count how many pairs (a from arr1, b from arr2) satisfy:
  - a + b == x
- Return only count, not the pairs.

---

2. Constraints

- 1 <= m, n <= 5 * 10^4
- Arrays are sorted
- Elements are distinct inside each array
- 1 <= x <= 2 * 10^5
- Overall operations must ideally be O(m + n) or O(m log n).

---

3. Edge Cases

- All values smaller than x â†' result may be 0.
- All values greater than x â†' result may be 0.
- All pairs valid only once (distinct arrays guarantee no duplicates in each array).
- Very large arrays â†' brute force will TLE.

---

4. Examples

```
Example 1:
arr1 = [1,2,4,5]
arr2 = [3,5,7,8]
x = 9
Pairs â†’ (1,8), (2,7), (4,5) â†’ count = 3.

Example 2:
arr1 = [1,2,3]
arr2 = [4,5,6,7,8]
x = 8
Pairs â†’ (1,7), (2,6), (3,5) â†’ count = 3.
```

---

5. Approaches

Approach 1: Brute Force (Nested Loops) â€" O(m * n)

Idea:

- Check all m*n pairs and count those with sum x.

Java Code:

```
public static int countPairs(int[] arr1, int[] arr2, int x) {
    int count = 0;
    for (int a : arr1) {
        for (int b : arr2) {
            if (a + b == x) count++;
        }
    }
    return count;
}
```

ðŸ' Intuition Behind the Approach:

- Straightforward brute-force checking of all possibilities.
- Simple to write, but too slow for large inputs.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(m * n)
  - Why: Each element of arr1 is paired with each element of arr2.
- ðŸ'¾ Space Complexity
  - O(1).

Approach 2: Binary Search on Second Array â€" O(m log n)

Idea:

- For each a from arr1:
  - Look for x - a in arr2 using binary search.

Java Code:

```
public static int countPairs(int[] arr1, int[] arr2, int x) {
    int count = 0;
    for (int a : arr1) {
        int target = x - a;
        int lo = 0, hi = arr2.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (arr2[mid] == target) {
                count++;
                break;
            } else if (arr2[mid] < target) {
                lo = mid + 1;
            } else {
                hi = mid - 1;
            }
        }
    }
```

```
    return count;
}
```

ðŸ' Intuition Behind the Approach:

- Since arr2 is sorted, binary search helps locate the matching pair in log n time.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(m log n)
  - Why: For each of m elements, binary search takes log n.
- ðŸ'¾ Space Complexity
  - O(1).

Approach 3: HashSet Approach â€" O(m + n) (Better than binary search if arrays weren't sorted)

Idea:

- Store all elements of arr2 in a HashSet, then for each a check if (x - a) exists.

Java Code:

```
public static int countPairs(int[] arr1, int[] arr2, int x) {
    HashSet<Integer> set = new HashSet<>();
    for (int b : arr2) set.add(b);

    int count = 0;
    for (int a : arr1) {
        if (set.contains(x - a)) count++;
    }
    return count;
}
```

ðŸ' Intuition Behind the Approach:

- Hashing gives O(1) lookup for each pair check.

Complexity (Time & Space):

- â±ï¸ Time Complexity
  - O(m + n)
  - Why: Build hash set â†' O(n), loop arr1 â†' O(m).
- ðŸ'¾ Space Complexity
  - O(n) for the hash set.

Approach 4: Two Pointers (Optimal) â€" O(m + n)

Idea:

- Since both arrays are sorted:
    - Use pointer i = 0 on arr1 (smallest values)
    - Use pointer j = n-1 on arr2 (largest values)
- Move pointers based on arr1[i] + arr2[j]:
    - If sum == x â†' count++, move both
    - If sum < x â†' need bigger sum â†' i++
    - If sum > x â†' need smaller sum â†' j--

Java Code:

```
public static int countPairs(int[] arr1, int[] arr2, int x) {
    int i = 0;
    int j = arr2.length - 1;
    int count = 0;

    while (i < arr1.length && j >= 0) {
        int sum = arr1[i] + arr2[j];

        if (sum == x) {
            count++;
            i++;
            j--;
        } else if (sum < x) {
            i++;    // need larger sum
        } else {
            j--;    // need smaller sum
        }
    }
    return count;
}
```

ðŸ' Intuition Behind the Approach:

- Sorted arrays allow a linear scan from opposite ends.
- You always eliminate one pointer depending on sum, guaranteeing no missed pairs.

Complexity (Time & Space):

- â±ï¸ Time Complexity
    - O(m + n)
    - Why: Each pointer moves monotonically across its array once.
- ðŸ'¾ Space Complexity
    - O(1).

---

6. Justification / Proof of Optimality

- Fully uses sorted property.
- Inspects each element at most once.
- Eliminates impossible sums quickly.

- Best complexity achievable under given constraints.
- Thus Two-Pointer is the optimal approach.

---

7. Variants / Follow-Ups

- Return the pairs instead of count.
- Arrays unsorted â†' sort + two pointers OR use hashset.
- Duplicate elements present â†' handle frequency counts.
- K arrays sum â†' similar expansion as K-sum problems.

---

8. Tips & Observations

- When both arrays sorted â†' two-pointer is usually optimal for sum problems.
- Hashing is best when arrays not sorted.
- Avoid brute force for n = 50,000.

Q134: Maximum Consecutive Ones

---

1. Problem Understanding

- You are given a binary array arr of size n.
- Your task is to find the maximum number of consecutive 1s in the array.

---

2. Constraints

- 1 <= n <= 10^5
- 0 <= arr[i] <= 1
- Need an O(n) final solution.

---

3. Edge Cases

- All zeros â†' answer is 0
- All ones â†' answer is n
- Single element array
- Alternating 1s and 0s
- Large n â†' brute force may be slow

---

4. Examples

```
Example 1
Input: [1, 0, 1, 1]
```

```
Output: 2

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

---

5. Approaches

Approach 1: Brute Force (O(n^2))

Idea:

- Check every index i, and count how many consecutive 1s start from there.

Java Code:

```java
public static int maxConsecutiveOnes(int[] arr) {
    int n = arr.length;
    int ans = 0;

    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = i; j < n; j++) {
            if (arr[j] == 1) count++;
            else break;
        }
        ans = Math.max(ans, count);
    }

    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- For each position, simulate starting a consecutive-ones streak.
- Very slow because we repeatedly scan the same segments.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n^2)
  - Because at each i, inner loop expands until a zero or end.
- ðŸ'¾ Space Complexity
  - O(1)
  - Only counters.

Approach 2: Linear Scan with Reset (Optimal & Clean) â€" O(n)

Idea:

- Traverse the array once:
  - Maintain a currentCount of continuous ones
  - Reset it when you hit a 0
  - Track the maximum streak in ans

Java Code:

```
public static int maxConsecutiveOnes(int[] arr) {
    int ans = 0;
    int count = 0;

    for (int x : arr) {
        if (x == 1) {
            count++;
            ans = Math.max(ans, count);
        } else {
            count = 0;
        }
    }

    return ans;
}
```

ðŸ' Intuition Behind the Approach:

- Consecutive ones naturally form segments.
- Whenever you see a zero, a segment ends â†' reset count.
- Always update maximum during the scan.

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(n) â€" every element processed once.
- ðŸ'¾ Space Complexity
  - O(1) â€" constant counters.

---

6. Justification / Proof of Optimality

- The brute force simulates all possible consecutive streaks, which is unnecessary.
- The optimal linear scan captures the nature of the problem: ones form continuous blocks that can be counted in one pass.
- Given n â‰¤ 10^5, the linear scan is required and the best solution.

---

7. Variants / Follow-Ups

- Count maximum consecutive 0s
- Count maximum consecutive 1s with at most one flip (follow-up variant)
- Count longest subarray of equal elements

---

8. Tips & Observations

- Most "consecutive ones" problems use either:
  - Simple counter, or
  - Sliding window if flipping or skipping is allowed.
- Resetting counters at boundaries is a common pattern.
