Infosys SP/DSE Complete Problem Repository

Table of Contents

Volume B: Problems 4-13 + All Variations (Continued)

Continuation from Volume A

PROBLEM 4: String Cutting Patterns (GCD Frequency)

Problem Statement

Given an interesting string S of length N that can be **rearranged in any order**.

Cut this string into **contiguous pieces** such that after cutting, **all pieces are equal** to one another.

Cannot:

- Rearrange characters within cut pieces
- Join pieces together

Task: Find maximum number of pieces.

Note: Minimum answer is 1 (no cutting).

Examples

Example 1:

```
Input: "zzzzz"
Output: 5
Explanation: Cut into ["z", "z", "z", "z", "z"]
```

Example 2:

```
Input: "ababcc"
Output: 2
Explanation: Rearrange to "abcabc", cut into ["abc", "abc"]
```

Example 3:

```
Input: "abccdcabacda"
Output: 2
Explanation: Rearrange to form 2 equal pieces
```

Theory & Approach

Key Insight

If we can cut into k equal pieces, each piece has length `n/k`.

For this to work, **every character's frequency must be divisible by k**.

Mathematical Formulation:

Let `freq[c]` = frequency of character c.

Maximum k = **GCD of all character frequencies**.

Proof:

- If k divides all frequencies, we can distribute characters evenly
- If k doesn't divide some frequency, impossible to split evenly
- Largest such k is GCD

C++ Solution

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

int maxPieces(string s) {
    // Count frequency of each character
    unordered_map&lt;char, int&gt; freq;
    for (char c : s) {
        freq[c]++;
    }

    // Find GCD of all frequencies
    int gcdVal = 0;
    for (auto&amp; [ch, count] : freq) {
        gcdVal = __gcd(gcdVal, count);
    }

    return gcdVal;
}

int main() {
    string s;
    cin &gt;&gt; s;
    cout &lt;&lt; maxPieces(s) &lt;&lt; endl;
    return 0;
}
```

Time Complexity: O(n + C log V) where C = unique characters, V = max frequency
Space Complexity: O(C) — frequency map

Python Solution

```
from math import gcd
from functools import reduce
from collections import Counter

def max_pieces(s):
    # Count frequencies
    freq = Counter(s)

    # Find GCD of all frequencies
    frequencies = list(freq.values())
    return reduce(gcd, frequencies)

# Input<a></a>
s = input()
print(max_pieces(s))
```

Variations

Variation 1: Return Actual Pieces

```
vector&lt;string&gt; getPieces(string s) {
    int k = maxPieces(s);
    int pieceLen = s.length() / k;

    // Count frequencies
    unordered_map&lt;char, int&gt; freq;
    for (char c : s) freq[c]++;

    // Build pieces
    vector&lt;string&gt; pieces(k, "");
    for (auto&amp; [ch, count] : freq) {
        int perPiece = count / k;
        for (int i = 0; i &lt; k; i++) {
            pieces[i] += string(perPiece, ch);
        }
    }

    return pieces;
}
```

Variation 2: Minimum Cuts (Not Maximum Pieces)

```
int minCuts(string s) {
    return maxPieces(s) - 1;
}
```

Edge Cases

```
// Test Case 1: All same character
Input: "aaaaa"
Output: 5 (GCD of [5] = 5)

// Test Case 2: All different characters
Input: "abcde"
Output: 1 (GCD of [1,1,1,1,1] = 1)

// Test Case 3: Two characters, equal frequency
Input: "aabb"
Output: 2 (GCD of [2,2] = 2)
```

```
// Test Case 4: Coprime frequencies
Input: "aabbbcccc"  // freq: a=2, b=3, c=4
Output: 1 (GCD of [2,3,4] = 1)

// Test Case 5: Single character
Input: "a"
Output: 1
```

PROBLEM 5: Exercise Energy Optimization

Problem Statement

You have energy E units. There are N exercises, each draining A[i] energy.

You feel **tired** if energy reaches ≤ 0.

Constraints:

- Each exercise can be performed **at most 2 times**
- Want to become tired with **minimum number of exercises**

Return -1 if performing all exercises (each twice) doesn't make you tired.

Examples

Example 1:

```
Input:
E = 6
N = 2
A = [1, 2]

Output: 4
Explanation: Do exercise 2 twice (2+2=4), then exercise 1 twice (1+1=2). Total = 6.
```

Example 2:

```
Input:
E = 10
N = 2
A = [1, 2]

Output: -1
Explanation: Max drain = 2*(1+2) = 6 &amp;lt; 10. Cannot tire.
```

Example 3:

```
Input:
E = 2
N = 3
A = [1, 5, 2]

Output: 1
Explanation: Do exercise with drain 5 once. Energy = 2-5 = -3 ≤ 0.
```

Theory & Approach

Greedy Strategy

Goal: Drain energy as fast as possible.

Approach:

1. **Sort exercises by energy drain (descending)**
2. **Greedily pick largest drain exercises**
3. **Each exercise can be done twice**

Why Greedy Works: Larger drains reach 0 faster.

C++ Solution

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

int minExercises(int E, vector&lt;int&gt;&amp; A) {
    // Sort descending
    sort(A.rbegin(), A.rend());

    int count = 0;

    for (int drain : A) {
        // Each exercise can be done twice
        for (int times = 0; times &lt; 2 &amp;&amp; E &gt; 0; times++) {
            E -= drain;
            count++;

            if (E &lt;= 0) {
                return count;
            }
        }
    }

    // Couldn't tire even after all exercises twice
    return -1;
}

int main() {
    int E, N;
    cin &gt;&gt; E &gt;&gt; N;

    vector&lt;int&gt; A(N);
    for (int i = 0; i &lt; N; i++) {
        cin &gt;&gt; A[i];
    }

    int result = minExercises(E, A);
    cout &lt;&lt; result &lt;&lt; endl;
    return 0;
}
```

Time Complexity: O(N log N) — sorting
Space Complexity: O(1) — in-place sorting

Python Solution

```
def min_exercises(E, A):
    # Sort descending
    A_sorted = sorted(A, reverse=True)

    count = 0

    for drain in A_sorted:
        # Each exercise can be done twice
        for _ in range(2):
            if E &lt;= 0:
                return count
            E -= drain
            count += 1

    # Couldn't tire
    return -1 if E &gt; 0 else count

# Input<a></a>
E = int(input())
N = int(input())
A = [int(input()) for _ in range(N)]

# Output<a></a>
print(min_exercises(E, A))
```

Variations

Variation 1: Unlimited Exercise Repetitions

```
int minExercisesUnlimited(int E, vector&lt;int&gt;&amp; A) {
    int maxDrain = *max_element(A.begin(), A.end());

    if (maxDrain &gt;= E) {
        return 1; // Single largest exercise
    }

    // Total possible drain
    int totalDrain = accumulate(A.begin(), A.end(), 0);

    if (totalDrain == 0) return -1;

    // Number of full rounds + remainder
    int fullRounds = E / totalDrain;
    int remainder = E % totalDrain;

    if (remainder == 0) {
        return fullRounds * A.size();
    }

    // Greedily drain remainder
    sort(A.rbegin(), A.rend());
    int count = fullRounds * A.size();

    for (int drain : A) {
        if (remainder &lt;= 0) break;
        remainder -= drain;
        count++;
    }

    return count;
}
```

Edge Cases

```
// Test Case 1: Single exercise, exactly matches energy
Input: E=5, N=1, A=[5]
Output: 1

// Test Case 2: All exercises = 0 drain
Input: E=10, N=3, A=[0,0,0]
Output: -1

// Test Case 3: Energy = 0
Input: E=0, N=2, A=[1,2]
Output: 0 (already tired)

// Test Case 4: Negative energy possible
Input: E=3, N=2, A=[2,1]
Output: 2 (do exercise with 2 twice? Or once each?)
// Greedy: 2, 2 → total 4 &gt; 3 → answer 2
```

PROBLEM 6-9: Medium Level Problems

PROBLEM 6: Array Integer Sequences (DP)

Full solution provided in Volume A - PDF 2

Summary:

- DP state: `dp[length][num]` = count of arrays ending with num
- Recurrence: Sum over divisors
- Complexity: O(k × n × √n)

PROBLEM 7: Ugliness Minimization

Full solution provided in Volume A - PDF 2

Summary:

- Binary string manipulation
- Greedy: swap vs flip based on cost
- Convert to decimal with modulo

PROBLEM 8: Maximum Vacation Days (Greedy Intervals)

Problem Statement

Andy has N days numbered 1 to N. He has M obligations on specific days D[i].

He can **cancel at most K obligations** to maximize consecutive free days.

Find maximum vacation length.

Theory & Approach

Sliding Window on Obligations

Key Insight: Sort obligations, then find longest gap with ≤K obligations inside.

Algorithm:

1. Sort obligation days
2. Use sliding window: keep at most K obligations in window
3. Longest window = answer

C++ Solution

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

int maxVacation(int N, int M, int K, vector&lt;int&gt;&amp; D) {
    if (K &gt;= M) {
        return N; // Cancel all obligations
    }

    sort(D.begin(), D.end());

    int maxDays = 0;

    // Try windows with K obligations to cancel
    for (int start = 0; start + K &lt; M; start++) {
        int end = start + K;

        // Gap between D[start-1] and D[end+1]
        int leftBound = (start == 0) ? 0 : D[start - 1];
        int rightBound = (end + 1 &gt;= M) ? N + 1 : D[end + 1];

        int vacationDays = rightBound - leftBound - 1;
        maxDays = max(maxDays, vacationDays);
    }

    // Also check gaps at start and end
    if (K &lt; M) {
        // From day 1 to D[K]
        maxDays = max(maxDays, D[K] - 1);

        // From D[M-K-1] to day N
        maxDays = max(maxDays, N - D[M - K - 1]);
    }

    return maxDays;
}

int main() {
    int N, M, K;
    cin &gt;&gt; N &gt;&gt; M &gt;&gt; K;

    vector&lt;int&gt; D(M);
    for (int i = 0; i &lt; M; i++) {
        cin &gt;&gt; D[i];
    }

    cout &lt;&lt; maxVacation(N, M, K, D) &lt;&lt; endl;
    return 0;
}
```

Time Complexity: O(M log M) — sorting
Space Complexity: O(1)

Python Solution

```
def max_vacation(N, M, K, D):
    if K &gt;= M:
        return N

    D.sort()
    max_days = 0

    # Sliding window
    for start in range(M - K):
        end = start + K

        left_bound = 0 if start == 0 else D[start - 1]
        right_bound = N + 1 if end + 1 &gt;= M else D[end + 1]

        vacation_days = right_bound - left_bound - 1
        max_days = max(max_days, vacation_days)

    # Edge cases
    if K &lt; M:
        max_days = max(max_days, D[K] - 1)
        max_days = max(max_days, N - D[M - K - 1])

    return max_days

# Input<a></a>
N = int(input())
M = int(input())
K = int(input())
D = [int(input()) for _ in range(M)]

print(max_vacation(N, M, K, D))
```

PROBLEM 9: Heroes vs Villains Battle

Problem Statement

M heroes (each health H) battle N villains (health V[i]).

Battle Rules:

- If H > V[i]: Villain defeated, hero health -= V[i]
- If H < V[i]: Hero defeated
- If H = V[i]: Both defeated

Heroes fight villains **in order** (villain 1, then 2, etc.).

Task: Find minimum villains to remove **from front** so heroes win.

Theory & Approach

Prefix Sum + Binary Search

Key Insight: Total hero health = M × H. Check suffix sums of villains.

Algorithm:

1. If total villain health ≤ total hero health → no removal needed
2. Binary search on number to remove from front
3. Check if remaining villains can be defeated

C++ Solution

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

int minRemove(int N, int M, int H, vector&lt;int&gt;&amp;amp; V) {
    long long totalHeroHealth = (long long)M * H;
    long long totalVillainHealth = accumulate(V.begin(), V.end(), 0LL);

    if (totalVillainHealth &lt;= totalHeroHealth) {
        return 0; // No removal needed
    }

    // Find minimum to remove from front
    long long suffixSum = 0;
    for (int remove = N - 1; remove &gt;= 0; remove--) {
        suffixSum += V[remove];

        if (suffixSum &gt;= totalHeroHealth) {
            return remove; // Need to remove first 'remove' villains
        }
    }

    return N; // Remove all
}

int main() {
    int N, M, H;
    cin &gt;&gt; N &gt;&gt; M &gt;&gt; H;

    vector&lt;int&gt; V(N);
    for (int i = 0; i &lt; N; i++) {
        cin &gt;&gt; V[i];
    }

    cout &lt;&lt; minRemove(N, M, H, V) &lt;&lt; endl;
    return 0;
}
```

Time Complexity: O(N)
Space Complexity: O(1)

Python Solution

```
def min_remove(N, M, H, V):
    total_hero_health = M * H
    total_villain_health = sum(V)

    if total_villain_health &lt;= total_hero_health:
        return 0

    # Check suffix sums
    suffix_sum = 0
    for remove in range(N - 1, -1, -1):
        suffix_sum += V[remove]

        if suffix_sum &gt;= total_hero_health:
            return remove

    return N

# Input<a></a>
N = int(input())
M = int(input())
H = int(input())
V = [int(input()) for _ in range(N)]
```

```
print(min_remove(N, M, H, V))
```

PROBLEM 10-13: Hard Level Problems

PROBLEM 10: Maximum XOR Subset

See Volume A for full analysis

Summary:

- Select at most n/2 elements to maximize XOR
- Use Gaussian elimination on XOR basis
- Complexity: O(n × log(max_value))

PROBLEM 11: Xor-Sum Maximization

See Volume A for full analysis

Summary:

- Find x in [0,K] maximizing sum(x XOR A[i])
- Greedy bit-by-bit construction
- Complexity: O(n × log K)

PROBLEM 12: Road Terrain Transformation

See Volume A for full analysis

Summary:

- Binary search on number of days
- Day d can dig 2^(d-1) meters
- Complexity: O(n × log(max_height))

PROBLEM 13: Longest Bitwise Subsequence

Problem Statement

Find longest increasing subsequence where for adjacent elements:

$$(A[i]\&A[i+1]) \times 2lt;(A[i]|A[i+1]) \quad (1)$$

Theory & Approach

DP with Bitwise Check

State: `dp[i]` = LIS ending at index i satisfying condition

Recurrence:

```
for i in 0 to n-1:
    dp[i] = 1
    for j in 0 to i-1:
        if A[j] &lt; A[i] and check_condition(A[j], A[i]):
            dp[i] = max(dp[i], dp[j] + 1)
```

C++ Solution

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

bool checkCondition(int a, int b) {
    return (a &amp; b) * 2 &lt; (a | b);
}

int longestBitwiseSubseq(vector&lt;int&gt;&amp; A) {
    int n = A.size();
    vector&lt;int&gt; dp(n, 1);

    for (int i = 1; i &lt; n; i++) {
        for (int j = 0; j &lt; i; j++) {
            if (A[j] &lt; A[i] &amp;&amp; checkCondition(A[j], A[i])) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }

    return *max_element(dp.begin(), dp.end());
}

int main() {
    int n;
    cin &gt;&gt; n;

    vector&lt;int&gt; A(n);
    for (int i = 0; i &lt; n; i++) {
        cin &gt;&gt; A[i];
    }

    cout &lt;&lt; longestBitwiseSubseq(A) &lt;&lt; endl;
    return 0;
}
```

Time Complexity: $O(n^2)$
Space Complexity: O(n)

References

[1] PrepInsta. (2025). Infosys SP and DSE Coding Questions (Complete Set)

[2] Codeforces: XOR Basis and Linear Algebra Problems

[3] LeetCode: Longest Increasing Subsequence Variations