

TCS CodeVita PYQ Problem Solving

Complete Student Revision Notes & C++ Solutions (2018–2023)

This handbook contains curated problem statements, sample test cases, algorithmic notes, and optimal C++ solutions for the top recurring topics in TCS CodeVita.

Topics covered: Arrays & Number Theory, Strings & Hashing, 2D Grid Simulation, Greedy Algorithms & Priority Queues, Dynamic Programming, and Graph Algorithms (Tarjan's Bridges).

#	Topic	Problem Name	Year	Category
1	Arrays & Math	Consecutive Prime Sum	2020	Core Problem
2	Arrays & Math	Square Free Number	2023	Assignment
3	Strings & Hashing	Lexi String	2019	Core Problem
4	Strings & Hashing	String Pair	2022	Assignment
5	Grid Simulation	Constellation	2020	Core Problem
6	Grid Simulation	Chakravyuha	2018	Assignment
7	Greedy & Heaps	Minimize The Sum	2021	Core Problem
8	Greedy & Heaps	Railway Station	2021	Assignment
9	Dynamic Prog.	Uncertain Steps	2019	Core Problem
10	Dynamic Prog.	Minimum Gifts	2020	Assignment
11	Graph Algorithms	Critical Planets	2022	Core Problem
12	Graph Algorithms	Holes and Balls	2019	Assignment

1. Consecutive Prime Sum

Topic: Arrays, Math & Number Theory | Year: 2020 | Category: Core Practice Problem

Problem Statement

Some prime numbers can be expressed as a sum of other consecutive prime numbers.

For example:

$$- 5 = 2 + 3$$

$$- 17 = 2 + 3 + 5 + 7$$

$$- 41 = 2 + 3 + 5 + 7 + 11 + 13$$

Your task is to find the count of all prime numbers less than or equal to N that can be expressed as the sum of **two or more** consecutive prime numbers starting from the first prime number (2).

Input Format

- A single integer N ($2 < N \leq 10^7$).

Output Format

- Print a single integer representing the count of such prime numbers less than or equal to N .

Sample Test Cases

Sample 1

Input:

20

Output:

2

Explanation:

The primes up to 20 are: 2, 3, 5, 7, 11, 13, 17, 19.

Sum of consecutive primes starting from 2:

$$- 2 + 3 = 5 \text{ (Prime, } \leq 20 \text{)} \rightarrow \text{Count} = 1$$

$$- 2 + 3 + 5 = 10 \text{ (Not prime)}$$

$$- 2 + 3 + 5 + 7 = 17 \text{ (Prime, } \leq 20 \text{)} \rightarrow \text{Count} = 2$$

$$- 2 + 3 + 5 + 7 + 11 = 28 (> 20) \rightarrow \text{Stop.}$$

Total valid primes = 2 (namely 5 and 17).

Sample 2

Input:

45

Output:

3

Explanation:

Following from Sample 1:

$$- 2 + 3 + 5 + 7 + 11 + 13 = 41 \text{ (Prime, } \leq 45 \text{)} \rightarrow \text{Count} = 3.$$

$$- 41 + 17 = 58 > 45 \rightarrow \text{Stop.}$$

Total valid primes = 3 (5, 17, 41).

Approach & Notes for Students

1. **Intuition & Mathematical Proof:**

- Why start from 2? The problem strictly mandates consecutive primes starting from the first prime (2). This eliminates $O(P^2)$ sliding window search and reduces the task to evaluating prefix sums of the prime sequence!

- Growth rate of prime prefix sums: $\sum_{i=1}^k p_i \approx \frac{k^2 \ln k}{2}$. Even for large N, the number of primes summed before exceeding N is small, ensuring fast evaluation.

2. **Algorithm Steps:**

- **Step 1:** Generate prime numbers up to N using the **Sieve of Eratosthenes** ($O(N \log \log N)$). Use `std::vector<bool>` for optimal memory (N bits, $\approx 1.25 \text{ MB}$ for 10^7).

- **Step 2:** Collect primes in a sequential array `primes`.

- **Step 3:** Accumulate prefix sums starting with $S = \text{primes}[0] + \text{primes}[1] = 2 + 3 = 5$.

- **Step 4:** For each subsequent prime `primes[i]` ($i \geq 2$), add it to S. If $S > N$, terminate the loop early.

- **Step 5:** Check if S is prime in $O(1)$ time using the boolean sieve lookup. If yes, increment the answer.

3. **Complexity Analysis:**

- **Time Complexity:** $O(N \log \log N)$ for Sieve preprocessing + $O(\sqrt{N / \ln N})$ for prefix evaluation $\approx O(N \log \log N)$.

- **Space Complexity:** $O(N)$ for the bit-vector sieve.

Optimal C++ Solution

```
/**
 * Problem: Consecutive Prime Sum
 * Contest: TCS CodeVita 2020 (Season 9)
 * Topic: Arrays, Math & Number Theory
 * Approach: Sieve of Eratosthenes + Prefix Sum of Primes
 * Time Complexity:  $O(N \log \log N)$ 
 * Space Complexity:  $O(N)$ 
 */

#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    long long n;
    if (!(cin >> n) || n < 5) {
        cout << 0 << "\n";
        return 0;
    }

    // Step 1: Precompute primes up to n using Sieve of Eratosthenes
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;

    for (long long p = 2; p * p <= n; ++p) {
        if (is_prime[p]) {
            for (long long multiple = p * p; multiple <= n; multiple += p) {
                is_prime[multiple] = false;
            }
        }
    }

    // Step 2: Extract primes in increasing order
    vector<long long> primes;
    for (long long i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
    }

    // Step 3: Compute running cumulative sum starting from the first prime (2)
    // At least 2 primes must be summed: 2 + 3 = 5
    long long count = 0;
    long long sum = 0;
    for (int i = 0; i < primes.size(); ++i) {
        sum += primes[i];
        if (sum > n) break;
        if (is_prime[sum]) count++;
    }

    cout << count << "\n";
    return 0;
}
```

```

long long running_sum = 0;

for (size_t i = 0; i < primes.size(); ++i) {
    running_sum += primes[i];

    // If the sum exceeds N, any further additions will also exceed N
    if (running_sum > n) {
        break;
    }

    // Must be sum of at least two consecutive primes (i >= 1)
    if (i >= 1 && is_prime[running_sum]) {
        count++;
    }
}

cout << count << "\n";

return 0;
}

```

2. Square Free Number

Topic: Arrays, Math & Number Theory | Year: 2023 | Category: Self-Practice Assignment

Problem Statement

In mathematics, a **square-free integer** is an integer which is not divisible by any perfect square other than 1. That is, its prime factorization contains no prime factor with an exponent strictly greater than 1.

Given an integer N , find the number of **square-free divisors** of N (excluding 1).

Input Format

- A single line containing an integer N ($1 \leq N \leq 10^9$).

Output Format

- Print a single integer representing the count of square-free divisors of N strictly greater than 1.

Sample Test Cases

Sample 1

Input:

20

Output:

3

Explanation:

The divisors of 20 are: 1, 2, 4, 5, 10, 20.

- Exclude 1.

- Divisors:

- 2: Prime (Square-free) $\rightarrow$ Valid

- 4: 2^2 (Divisible by 4) $\rightarrow$ Invalid

- 5: Prime (Square-free) $\rightarrow$ Valid

- 10: 2×5 (Square-free) $\rightarrow$ Valid

- 20: Divisible by 4 $\rightarrow$ Invalid

The square-free divisors are 2, 5, 10. Total count = 3.

Sample 2

Input:

72

Output:

3

Explanation:

Prime factorization: $72 = 2^3 \times 3^2$.

Distinct prime factors are {2, 3}.

Square-free divisors > 1: 2, 3, 6. Total count = 3.

Approach & Notes for Students

1. **Brute Force vs Combinatorial Insight:**

- **Brute Force:** Find all divisors up to $\sqrt{N}$, check each for square-freeness in $O(\text{divisors} \times \sqrt{d})$.

- **Optimal Combinatorics:** Every square-free number is a product of distinct primes with power 1.

- For a number N , any square-free divisor is formed by selecting a non-empty subset of the **distinct prime factors** of N .
- If N has k distinct prime factors $\{p_1, p_2, \dots, p_k\}$, every combination $p_{i_1} \cdot p_{i_2} \cdot \dots \cdot p_{i_m}$ divides N because each p_i divides N with exponent ≥ 1 .
- Therefore, the number of non-empty square-free divisors is simply:

$$2^k - 1$$

where k is the number of distinct prime factors of N !

2. **Algorithm Steps:**

- Factorize N by checking divisibility up to $\sqrt{N}$.
- Count distinct prime factors k .
- Compute answer = $(2^k - 1)$.
- **Time Complexity:** $O(\sqrt{N})$.
- **Space Complexity:** $O(1)$.

Optimal C++ Solution

```
/**
 * Problem: Square Free Number
 * Contest: TCS CodeVita 2023 (Season 11)
 * Topic: Arrays, Math & Number Theory
 * Approach: Prime Factorization & Combinatorics ( $2^k - 1$ )
 * Time Complexity:  $O(\sqrt{N})$ 
 * Space Complexity:  $O(1)$ 
 */

#include <iostream>

using namespace std;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    long long n;
    if (!(cin >> n) || n <= 1) {
        cout << 0 << "\n";
        return 0;
    }

    // Count distinct prime factors of N
    long long distinct_prime_factors = 0;
    long long temp = n;

    // Check factor 2
    if (temp % 2 == 0) {
        distinct_prime_factors++;
        while (temp % 2 == 0) {
            temp /= 2;
        }
    }

    // Check odd factors up to sqrt(temp)
    for (long long d = 3; d * d <= temp; d += 2) {
        if (temp % d == 0) {
            distinct_prime_factors++;
            while (temp % d == 0) {
                temp /= d;
            }
        }
    }

    // If remaining temp > 1, it's a prime factor
    if (temp > 1) {
        distinct_prime_factors++;
    }

    // Every non-empty subset of the distinct prime factors forms a unique square-free divisor.
    // Total valid divisors =  $2^k - 1$  (excluding 1)
```

```
    long long ans = (1LL << distinct_prime_factors) - 1LL;
    cout << ans << "\n";
    return 0;
}
```

3. Lexi String

Topic: Strings & Hashing | Year: 2019 | Category: Core Practice Problem

Problem Statement

Little Bobby likes to play with strings. One day, he invents a new game: instead of the standard English alphabetical ordering (`a, b, c, ... z`), he defines a custom order P consisting of distinct lowercase English letters.

Given the custom order string P and another string S, rearrange the characters of S such that they follow the relative ordering given in P. If a character appears multiple times in S, all occurrences must be preserved and grouped according to the order of P.

Input Format

- The first line contains an integer T, denoting the number of test cases.
- For each test case:
 - The first line contains string P (the custom alphabetical order).
 - The second line contains string S (the string to be sorted).

Output Format

- For each test case, print the rearranged string S on a new line.

Sample Test Cases

Sample 1

****Input:****

```
2
polikujmnhytgbvfredcxswqaz
abcd
qwryupcsfhogjxvzkbndteiama
svholfd
```

****Output:****

```
bdca
sfhovdl
```

****Explanation:****

- In the first test case:
 - Custom alphabet P: `p o l i k u j m n h y t g b v f r e d c x s w q a z`
 - Position of `b` is index 13.
 - Position of `d` is index 18.
 - Position of `c` is index 19.
 - Position of `a` is index 24.
 - Ordering: `b` < `d` < `c` < `a` $\rightarrow$ Result is `bdca`.

Approach & Notes for Students

1. ****Intuition & Approach Comparison:****

- ****Approach 1 (Comparison-based Sort):**** Define a custom comparator using a rank hash table `rank[char]`. Sort S using `std::sort`.
- Time Complexity: $O(|S| \log |S|)$.
- ****Approach 2 (Counting / Bucket Sort - Optimal):****
- Since the alphabet consists of only lowercase letters (≤ 26), count the frequency of each character in S using a frequency array in $O(|S|)$.
- Traverse string P in order from left to right. For each character c in P, append c to the result string `frequency[c]` times.
- Time Complexity: $O(|P| + |S|) = O(|S|)$.

- Space Complexity: $O(1)$ auxiliary space.

2. **Common Pitfalls & Edge Cases:**

- Multiple test cases: Always clear frequency maps between test cases.

- Blank lines / trailing whitespaces: Ensure input reading safely handles newlines.

Optimal C++ Solution

```
/**
 * Problem: Lexi String
 * Contest: TCS CodeVita 2019 (Round 1)
 * Topic: Strings, Custom Sorting & Hashing
 * Approach: Frequency Counting / Bucket Sorting in  $O(|S|)$ 
 * Time Complexity:  $O(|P| + |S|)$ 
 * Space Complexity:  $O(1)$  auxiliary (fixed 256 array)
 */

#include <iostream>
#include <string>
#include <vector>

using namespace std;

void solve() {
    string p, s;
    if (!(cin >> p >> s)) return;

    // Step 1: Count frequency of characters in string S
    vector<int> freq(256, 0);
    for (char c : s) {
        freq[(unsigned char)c]++;
    }

    // Step 2: Build the resulting string based on order in P
    string result = "";
    result.reserve(s.length());

    for (char c : p) {
        if (freq[(unsigned char)c] > 0) {
            result.append(freq[(unsigned char)c], c);
            freq[(unsigned char)c] = 0; // Mark as processed
        }
    }

    // Step 3: Append any remaining characters (if any not present in P)
    for (int i = 0; i < 256; ++i) {
        if (freq[i] > 0) {
            result.append(freq[i], (char)i);
        }
    }

    cout << result << "\n";
}

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int t;
    if (cin >> t) {
        while (t--) {
            solve();
        }
    }

    return 0;
}
```

4. String Pair

Topic: Strings & Hashing | Year: 2022 | Category: Self-Practice Assignment

Problem Statement

One person hands you an array of N integers (each integer between 0 and 100 inclusive). You are required to find the number of unordered pairs $(arr[i], arr[j])$ where $i < j$ such that their sum equals a target value D .

However, the value of D is not given directly! You must determine D using the following steps:

1. Convert each integer $arr[i]$ in the input into its lowercase English text representation without spaces or hyphens.

- Note: For `100`, use `"hundred"` (not `"one hundred"`).

- For compound numbers (e.g., `42`), concatenate directly: `"fortytwo"`.

2. Count the total number of vowels (`a`, `e`, `i`, `o`, `u`) across all textual representations of the numbers in the array. This sum is your target value D .

3. Count how many pairs of numbers $(arr[i], arr[j])$ with $i < j$ satisfy:

$$arr[i] + arr[j] = D$$

4. Output the final pair count converted into its English word representation (e.g., $0 \rightarrow$ `"zero"`, $3 \rightarrow$ `"three"`).

- **Exception:** If the pair count is strictly greater than 100, output `"greater 100"`.

Input Format

- First line: An integer N ($1 \leq N \leq 10^5$).

- Second line: N space-separated integers $arr[i]$ ($0 \leq arr[i] \leq 100$).

Output Format

- A single string representing the English word of the pair count, or `"greater 100"`.

Sample Test Cases

Sample 1

Input:

```
5
1 2 3 4 5
```

Output:

```
one
```

Explanation:

- Word representations:

- `1` $\rightarrow$ "one" (2 vowels: o, e)

- `2` $\rightarrow$ "two" (1 vowel: o)

- `3` $\rightarrow$ "three" (2 vowels: e, e)

- `4` $\rightarrow$ "four" (2 vowels: o, u)

- `5` $\rightarrow$ "five" (2 vowels: i, e)

- Total vowels $D = 2 + 1 + 2 + 2 + 2 = 9$.

- Pairs in `[1, 2, 3, 4, 5]` summing to 9:

- $4 + 5 = 9$ (1 pair).

Output: `"one"`.

Approach & Notes for Students

1. **Number to Word Mappings:**

- Precompute words for 0 through 100.

- Distinct vowels: `a`, `e`, `i`, `o`, `u`.

2. **Two-Sum Pair Counting in O(N)**

- Because N can be up to 10^5 , an $O(N^2)$ nested loop will cause **Time Limit Exceeded (TLE)**.

- Use a frequency array / hash map `freq` to count pairs in O(N) time:

- For each element x, add `freq[D - x]` to pair count, then increment `freq[x]++`.

3. **Number to English Output**

- Map final count (0 ≤ count ≤ 100) back to English words.

- If count > 100, print "greater 100".

Optimal C++ Solution

```
/**
 * Problem: String Pair
 * Contest: TCS CodeVita 2022 / Season 9
 * Topic: Strings, Custom Formatting & Hashing
 * Approach: Number-to-words table, vowel count, O(N) Two-Sum frequency map
 * Time Complexity: O(N)
 * Space Complexity: O(1) auxiliary (values ≤ 100)
 */

#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>

using namespace std;

// Convert number (0 to 100) into English string without spaces
string numberToWord(int n) {
    if (n == 0) return "zero";
    if (n == 100) return "hundred"; // CodeVita specification: "hundred", not "one hundred"

    static const vector<string> ones = {
        "", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine",
        "ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen", "sixteen",
        "seventeen", "eighteen", "nineteen"
    };

    static const vector<string> tens = {
        "", "", "twenty", "thirty", "forty", "fifty", "sixty", "seventy", "eighty", "ninety"
    };

    if (n < 20) {
        return ones[n];
    } else {
        return tens[n / 10] + ones[n % 10];
    }
}

// Count vowels in a string
int countVowels(const string& str) {
    int cnt = 0;
    for (char c : str) {
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
            cnt++;
        }
    }
    return cnt;
}

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    if (!(cin >> n)) return 0;

    vector<int> arr(n);
```

```

int total_vowels = 0;

for (int i = 0; i < n; ++i) {
    cin >> arr[i];
    string word = numberToWord(arr[i]);
    total_vowels += countVowels(word);
}

int d = total_vowels;

// Find unordered pairs (arr[i], arr[j]) with i < j such that arr[i] + arr[j] == D
// Use hash map / frequency array for O(N) counting
unordered_map<int, long long> freq;
long long pair_count = 0;

for (int i = 0; i < n; ++i) {
    int complement = d - arr[i];
    if (freq.find(complement) != freq.end()) {
        pair_count += freq[complement];
    }
    freq[arr[i]]++;
}

// Output formatting
if (pair_count > 100) {
    cout << "greater 100\n";
} else {
    cout << numberToWord((int)pair_count) << "\n";
}

return 0;
}

```

5. Constellation

Topic: 2D Grid Simulation | Year: 2020 | Category: Core Practice Problem

Problem Statement

Three scientists are observing the sky using an advanced telescope. The telescope outputs a $3 \times N$ matrix composed of three characters:

- `.` represents empty dark space.
- `#` represents a boundary separating galaxies.
- `\*` represents stars.

Within each galaxy, groups of stars form constellation patterns that look like English vowels (`A`, `E`, `I`, `O`, `U`). Each vowel is formed inside a non-overlapping 3×3 block.

The 5 vowel templates are defined as follows:

| Vowel | Row 0 | Row 1 | Row 2 |

| :---: | :---: | :---: | :---: |

| **A** | `.\*.` | `\*\*\*` | `\*.\*` |

| **E** | `\*\*\*` | `\*\*\*` | `\*\*\*` |

| **I** | `\*\*\*` | `.\*.` | `\*\*\*` |

| **O** | `\*\*\*` | `\*.\*` | `\*\*\*` |

| **U** | `\*.\*` | `\*.\*` | `\*\*\*` |

Your task is to scan the telescope matrix from left to right, decode all stars into their corresponding vowels, preserve the `#` galaxy separators, and print the decoded constellation sequence.

Input Format

- The first line contains an integer N ($1 \leq N \leq 10^5$), representing the number of columns in the grid.
- The next 3 lines contain strings of length N representing the telescope scan rows.

Output Format

- Print a single line containing the extracted vowels and galaxy delimiters (`#`) in order.

Sample Test Cases

Sample 1

Input:

```
18
*.*#####*.*
*.*#*.*#*.*
#####*.*
```

Output:

U#O#I#E#A

Explanation:

- Columns 0–2: Matches vowel `U`
- Column 3: `#` (Galaxy delimiter)
- Columns 4–6: Matches vowel `O`
- Column 7: `#` (Galaxy delimiter)
- Columns 8–10: Matches vowel `I`
- Column 11: `#` (Galaxy delimiter)
- Columns 12–14: Matches vowel `E`

- Columns 15–17: Matches vowel `A`

Result: `U#O#I#EA`

Approach & Notes for Students

1. **Intuition & Parsing Strategy:**

- Instead of complex multi-pass sliding windows or connected components, use a **single column-pointer traversal** $c \in [0, N-1]$.

- At column c :

- Check if column c is a separator `#`: Print `#`, increment $c \mathrel{+}= 1$.

- Check if column c is empty (`.` across all 3 rows): Skip column, increment $c \mathrel{+}= 1$.

- Otherwise, column c is the start of a 3×3 vowel!

2. **O(1) Vowel Identification Logic:**

- Look at the 3×3 block across columns $c, c+1, c+2$:

- If Row 0 is `.\*.` $\rightarrow$ **A**

- If Row 0 is `\*..\*` $\rightarrow$ **U**

- If Row 0 is `\*\*\*`:

- If Row 1 is `\*\*\*` $\rightarrow$ **E**

- If Row 1 is `.\*.` $\rightarrow$ **I**

- If Row 1 is `\*..\*` $\rightarrow$ **O**

- After identifying the vowel, append it and jump $c \mathrel{+}= 3$ to avoid processing internal columns of the star.

3. **Complexity Analysis:**

- **Time Complexity:** $O(N)$ — every column is inspected at most once.

- **Space Complexity:** $O(N)$ to store the 3 row strings.

Optimal C++ Solution

```
/**
 * Problem: Constellation
 * Contest: TCS CodeVita 2020 (Season 9)
 * Topic: 2D Grid Simulation & Matrix Traversal
 * Approach: Column-wise scanning, O(1) pattern identification, O(N) total time
 * Time Complexity: O(N)
 * Space Complexity: O(N)
 */

#include <iostream>
#include <vector>
#include <string>

using namespace std;

// Decode a 3x3 block starting at column c into the corresponding vowel
char decodeVowel(const vector<string>& grid, int c) {
    string r0 = grid[0].substr(c, 3);
    string r1 = grid[1].substr(c, 3);
    string r2 = grid[2].substr(c, 3);

    if (r0 == ".*.") {
        return 'A';
    } else if (r0 == ".*.*") {
        return 'U';
    } else if (r0 == "****") {
        if (r1 == "****") return 'E';
        if (r1 == ".*.") return 'I';
        if (r1 == ".*.*") return 'O';
    }

    return '?'; // Fallback for invalid/unmatched pattern
}

int main() {
```

```

// Fast I/O
ios_base::sync_with_stdio(false);
cin.tie(NULL);

int n;
if (!(cin >> n)) return 0;

vector<string> grid(3);
for (int i = 0; i < 3; ++i) {
    cin >> grid[i];
}

string result = "";
int c = 0;

while (c < n) {
    // Case 1: Galaxy boundary '#'
    if (grid[0][c] == '#' || grid[1][c] == '#' || grid[2][c] == '#') {
        result += '#';
        c++;
    }
    // Case 2: Empty space column (all '.')
    else if (grid[0][c] == '.' && grid[1][c] == '.' && grid[2][c] == '.') {
        c++;
    }
    // Case 3: Start of a 3x3 star (vowel)
    else {
        if (c + 2 < n) {
            char vowel = decodeVowel(grid, c);
            if (vowel != '?') {
                result += vowel;
            }
            c += 3; // Advance past the 3x3 star
        } else {
            c++;
        }
    }
}

cout << result << "\n";

return 0;
}

```

6. Chakravyuha

Topic: 2D Grid Simulation | Year: 2018 | Category: Self-Practice Assignment

Problem Statement

During the Kurukshetra war, Dronacharya created the deadly multi-tier defensive formation known as the **Chakravyuha**. To break through the formation, Abhimanyu needed to navigate through an $N \times N$ circular spiral grid starting from the outer boundary at position (0, 0) and spiraling inwards to the center.

The army places warriors with power values from 1 to N^2 along the spiral path.

- In addition, **Power Points** are established at special checkpoints:
- Coordinate (0, 0) is always a Power Point (granted at the entry with power 1).
- Any position where the warrior power value is divisible by **11** ($k \bmod{11} == 0$) is also designated as a Power Point.

Your task is to:

1. Construct and print the $N \times N$ spiral matrix (tab-separated ``t``).
2. Print the total count of Power Points.
3. Print the 0-indexed coordinates ``(r, c)`` of all Power Points in the order they are reached during the inward spiral.

Input Format

- A single integer N ($1 \leq N \leq 100$), denoting the size of the matrix.

Output Format

- Print the $N \times N$ matrix where values in each row are separated by a tab (``t``).
- On the next line, print: ``Total Power points : <count>``
- On subsequent lines, print the coordinates of each power point in the format: ``(r, c)``

Sample Test Cases

Sample 1

Input:

5

Output:

```
1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
Total Power points : 3
(0, 0)
(4, 2)
(3, 2)
```

Explanation:

- Entries along the spiral:
- (0, 0) has value 1 $\rightarrow$ Initial Power Point
- (4, 2) has value 11 $\rightarrow$ Power Point
- (3, 2) has value 22 $\rightarrow$ Power Point
- Total Power points = 3.

Approach & Notes for Students

1. **Spiral Traversal Algorithm:**
- Maintain 4 boundaries: ``top = 0``, ``bottom = N - 1``, ``left = 0``, ``right = N - 1``.

- Traverse:

1. Left to right along `top`, then `top++`
2. Top to bottom along `right`, then `right--`
3. Right to left along `bottom`, then `bottom--`
4. Bottom to top along `left`, then `left++`

- Maintain a counter k from 1 to N^2 .

- When $k \equiv 1 \text{ or } k \pmod{11} \equiv 0$, record the coordinate (r, c) in a list.

2. **Formatting Nuances:**

- Elements within each row must be tab-separated (`\t`).

- Space before and after colon in `"Total Power points : 3"`.

- Coordinates must be printed as `(r, c)`.

Optimal C++ Solution

```
/**
 * Problem: Chakravyuha
 * Contest: TCS CodeVita 2018
 * Topic: 2D Grid Simulation & Matrix Traversal
 * Approach: Spiral Matrix filling with Power Points tracking
 * Time Complexity:  $O(N^2)$ 
 * Space Complexity:  $O(N^2)$ 
 */

#include <iostream>
#include <vector>

using namespace std;

struct Point {
    int r, c;
};

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    if (!(cin >> n)) return 0;

    vector<vector<int>> matrix(n, vector<int>(n, 0));
    vector<Point> power_points;

    // (0, 0) is always granted as a power point
    power_points.push_back({0, 0});

    int top = 0, bottom = n - 1;
    int left = 0, right = n - 1;
    int val = 1;

    while (top <= bottom && left <= right) {
        // Move Left to Right
        for (int c = left; c <= right; ++c) {
            matrix[top][c] = val;
            if (val > 1 && val % 11 == 0) {
                power_points.push_back({top, c});
            }
            val++;
        }
        top++;

        // Move Top to Bottom
        for (int r = top; r <= bottom; ++r) {
            matrix[r][right] = val;
            if (val > 1 && val % 11 == 0) {
                power_points.push_back({r, right});
            }
            val++;
        }
        right--;

        // Move Right to Left
        for (int c = right; c >= left; --c) {
            matrix[bottom][c] = val;
            if (val > 1 && val % 11 == 0) {
                power_points.push_back({bottom, c});
            }
            val++;
        }
        bottom--;

        // Move Bottom to Top
        for (int r = bottom; r >= top; --r) {
            matrix[r][left] = val;
            if (val > 1 && val % 11 == 0) {
                power_points.push_back({r, left});
            }
            val++;
        }
        left++;
    }

    // Print the matrix
    for (int r = 0; r < n; ++r) {
        for (int c = 0; c < n; ++c) {
            cout << matrix[r][c] << " ";
        }
        cout << "\n";
    }

    // Print the power points
    cout << "Total Power points : " << power_points.size() << "\n";
}
```

```

    }
    right--;

    // Move Right to Left
    if (top <= bottom) {
        for (int c = right; c >= left; --c) {
            matrix[bottom][c] = val;
            if (val > 1 && val % 11 == 0) {
                power_points.push_back({bottom, c});
            }
            val++;
        }
        bottom--;
    }

    // Move Bottom to Top
    if (left <= right) {
        for (int r = bottom; r >= top; --r) {
            matrix[r][left] = val;
            if (val > 1 && val % 11 == 0) {
                power_points.push_back({r, left});
            }
            val++;
        }
        left++;
    }
}

// Print Spiral Matrix tab-separated
for (int r = 0; r < n; ++r) {
    for (int c = 0; c < n; ++c) {
        cout << matrix[r][c];
        if (c < n - 1) {
            cout << "\t";
        }
    }
    cout << "\n";
}

// Print Total Power points
cout << "Total Power points : " << power_points.size() << "\n";

// Print Power Point Coordinates
for (const auto& pt : power_points) {
    cout << "(" << pt.r << ", " << pt.c << ")\n";
}

return 0;
}

```

7. Minimize The Sum

Topic: Greedy & Priority Queues | Year: 2021 | Category: Core Practice Problem

Problem Statement

You are given an array of N positive integers and a number K . You are allowed to perform at most K operations on the array.

In each operation, you can choose any element $arr[i]$ from the array and replace it with its half rounded down:

$$arr[i] = \lfloor arr[i] / 2 \rfloor$$

Your task is to find the **minimum possible sum** of all array elements after performing at most K operations.

Input Format

- The first line contains two integers: N ($1 \leq N \leq 10^5$) and K ($1 \leq K \leq 10^5$).
- The second line contains N space-separated positive integers $arr[i]$ ($1 \leq arr[i] \leq 10^9$).

Output Format

- Print a single integer representing the minimum total sum of the array after at most K operations.

Sample Test Cases

Sample 1

Input:

```
4 3
20 7 5 4
```

Output:

```
17
```

Explanation:

- Initial array: `[20, 7, 5, 4]`, Sum = 36.
- **Operation 1:** Max element is 20. Replace with $\lfloor 20/2 \rfloor = 10$. Array becomes `[10, 7, 5, 4]`.
- **Operation 2:** Max element is 10. Replace with $\lfloor 10/2 \rfloor = 5$. Array becomes `[5, 7, 5, 4]`.
- **Operation 3:** Max element is 7. Replace with $\lfloor 7/2 \rfloor = 3$. Array becomes `[5, 3, 5, 4]`.
- Final Sum: $5 + 3 + 5 + 4 = 17$.

Approach & Notes for Students

1. Greedy Choice Property:

- Why always pick the maximum? The reduction in sum achieved by dividing x is $x - \lfloor x/2 \rfloor = \lceil x/2 \rceil$. This reduction is monotonically non-decreasing with respect to x . Therefore, dividing the currently largest element always yields the maximum possible reduction at each step!

2. Data Structure Selection:

- Finding the maximum repeatedly among N elements across K steps:
- Linear Scan: $O(K \times N) \approx 10^{10}$ operations $\rightarrow$ **TLE!**
- Max-Heap (`std::priority_queue` in C++):
- Extract max: $O(\log N)$
- Insert halved element: $O(\log N)$
- Total time: $O(N + K \log N) \approx 10^5 \log_2(10^5) \approx 1.7 \times 10^6$ operations $\rightarrow$ **Blazing Fast** ($< 0.05s$)!

3. Key Edge Cases & Overflow Protection:

- Sum can exceed 32-bit integer limits: Always use `long long` for sum accumulation.
- Early stopping: If the maximum element in the heap becomes 0, further operations will do nothing; break immediately.

Optimal C++ Solution

```
/**
 * Problem: Minimize The Sum
 * Contest: TCS CodeVita 2021 (Season 10)
 * Topic: Greedy Algorithms & Priority Queues
 * Approach: Max-Heap (std::priority_queue) for repeated maximum reduction
 * Time Complexity:  $O(N + K \log N)$ 
 * Space Complexity:  $O(N)$ 
 */

#include <iostream>
#include <vector>
#include <queue>
#include <numeric>

using namespace std;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, k;
    if (!(cin >> n >> k)) return 0;

    priority_queue<long long> max_heap;

    for (int i = 0; i < n; ++i) {
        long long val;
        cin >> val;
        max_heap.push(val);
    }

    // Perform up to K greedy operations
    while (k-- > 0 && !max_heap.empty()) {
        long long top = max_heap.top();
        max_heap.pop();

        if (top == 0) {
            max_heap.push(0);
            break; // If the largest element is 0, no further reduction is possible
        }

        long long halved = top / 2;
        max_heap.push(halved);
    }

    // Accumulate the final sum using long long to avoid integer overflow
    long long total_sum = 0;
    while (!max_heap.empty()) {
        total_sum += max_heap.top();
        max_heap.pop();
    }

    cout << total_sum << "\n";

    return 0;
}
```

8. Railway Station

Topic: Greedy & Interval Scheduling | Year: 2021 | Category: Self-Practice Assignment

Problem Statement

Given the schedule of N trains arriving at a railway station, determine the **minimum number of platforms** required so that no train is made to wait.

For each train, you are provided with:

- Its **arrival time**
- Its **stoppage duration** (time it remains at the station)

The departure time of a train is calculated as:

$$\text{Departure Time} = \text{Arrival Time} + \text{Stoppage Duration}$$

Critical CodeVita Rule:

> **Boundary Conflict:** If Train A's departure time is T and Train B's arrival time is also T, **they CANNOT** share the same platform! A separate platform is required for Train B.

Input Format

- The first line contains an integer N ($1 \leq N \leq 10^5$), the number of trains.
- The next N lines each contain two space-separated integers: `arrival stoppage`` ($0 \leq \text{arrival} \leq 10^9, 1 \leq \text{stoppage} \leq 10^9$).

Output Format

- Print a single integer representing the minimum number of platforms required.

Sample Test Cases

Sample 1

Input:

```
3
10 2
5 10
12 5
```

Output:

```
3
```

Explanation:

- Train 1: Arrives at 10, departs at $10 + 2 = 12$.
- Train 2: Arrives at 5, departs at $5 + 10 = 15$.
- Train 3: Arrives at 12, departs at $12 + 5 = 17$.

At time $t = 12$:

- Train 1 is departing at 12.
- Train 3 is arriving at 12.
- Due to the CodeVita collision rule, Train 1 and Train 3 cannot share a platform at $t = 12$.
- Train 2 is also occupying a platform (until 15).
- Hence, at time 12, all 3 trains require separate platforms.

Minimum platforms needed = `3`.

Approach & Notes for Students

1. **Two-Pointer Chronological Sweep ($O(N \log N)$):**

- Separate the train data into two arrays: `arr` (arrival times) and `dep` (departure times).
 - Sort both arrays in ascending order.
 - Use two pointers $i = 0$ (for arrivals) and $j = 0$ (for departures).
 - Iterate through the sorted events:
 - If $arr[i] \leq dep[j]$: An arrival happens before or concurrently with departure. Increment current platforms ($curr_platforms++$), update $max_platforms$, and advance $i++$.
 - If $arr[i] > dep[j]$: Departure occurred strictly before arrival. Decrement current platforms ($curr_platforms--$) and advance $j++$.
2. **Why Standard Platform Code Fails on CodeVita:**
- On standard platforms, when arrival equals departure, departure is processed first ($arr[i] < dep[j]$), which reuses the platform.
 - In TCS CodeVita, arrival \(\leq\) departure MUST require a new platform! Using $\leq$ is the critical differentiator.

Optimal C++ Solution

```
/**
 * Problem: Railway Station
 * Contest: TCS CodeVita 2021 (Season 10)
 * Topic: Greedy Algorithms & Interval Scheduling
 * Approach: Chronological sweep using sorted arrival and departure arrays (Two Pointers)
 * Time Complexity:  $O(N \log N)$ 
 * Space Complexity:  $O(N)$ 
 */

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    if (!(cin >> n) || n <= 0) {
        cout << 0 << "\n";
        return 0;
    }

    vector<long long> arrivals(n);
    vector<long long> departures(n);

    for (int i = 0; i < n; ++i) {
        long long arr, stoppage;
        cin >> arr >> stoppage;
        arrivals[i] = arr;
        departures[i] = arr + stoppage;
    }

    // Sort both arrival and departure times independently
    sort(arrivals.begin(), arrivals.end());
    sort(departures.begin(), departures.end());

    int i = 0, j = 0;
    long long curr_platforms = 0;
    long long max_platforms = 0;

    // Sweep across timeline
    while (i < n && j < n) {
        // Crucial CodeVita rule: if arrival == departure, they cannot share a platform
        // Hence: arrivals[i] <= departures[j] demands an extra platform
        if (arrivals[i] <= departures[j]) {
            curr_platforms++;
            max_platforms = max(max_platforms, curr_platforms);
            i++;
        } else {
            curr_platforms--;
            j++;
        }
    }
}
```

```
}  
cout << max_platforms << "\n";  
return 0;  
}
```

9. Uncertain Steps

Topic: Dynamic Programming & Combinatorics | Year: 2019 | Category: Core Practice Problem

Problem Statement

Codu is standing at the ground level (step 0) and wants to climb a staircase of N steps to reach the top (step N).

Under normal conditions, Codu can take either:

- 1 step, or
- 2 steps in a single move.

However, Codu also possesses a special booster jump that allows taking a 3-step jump, but this superpower can be used at most once during the entire journey!

Your task is to calculate the total number of distinct ways Codu can reach step N . Since the answer can be very large, output the answer modulo $10^9 + 7$.

Input Format

- A single integer N ($1 \leq N \leq 10^6$), representing the target step.

Output Format

- Print a single integer representing the total number of ways modulo $10^9 + 7$.

Sample Test Cases

Sample 1

Input:

3

Output:

4

Explanation:

The 4 possible ways to reach step 3 are:

- 1 + 1 + 1 (0 booster jumps)
- 1 + 2 (0 booster jumps)
- 2 + 1 (0 booster jumps)
- 3 (1 booster jump used)

Total ways = 4.

Sample 2

Input:

4

Output:

7

Explanation:

- Ways without booster jump (Fibonacci-like):
 - 1+1+1+1, 1+1+2, 1+2+1, 2+1+1, 2+2 (5 ways)
- Ways with exactly one 3-step booster jump:
 - 3 + 1
 - 1 + 3 (2 ways)

Total ways = 5 + 2 = 7.

Approach & Notes for Students

1. **State Space Formulation:**

- Standard stair climbing is 1D DP ($dp[i] = dp[i-1] + dp[i-2]$).
- Here, we have a constrained binary state: Has the 3-step jump been used or not?
- Define two DP states:
- $dp_0[i]$: Number of ways to reach step i **without** using the 3-step jump.
- $dp_1[i]$: Number of ways to reach step i **using exactly one** 3-step jump.

2. **Transition Equations:**

- **For $dp_0[i]$:**

$$dp_0[i] = (dp_0[i-1] + dp_0[i-2]) \pmod{10^9+7}$$

- **For $dp_1[i]$:**

$$dp_1[i] = (dp_1[i-1] + dp_1[i-2] + dp_0[i-3]) \pmod{10^9+7}$$

(Explanation: Can take 1 or 2 steps from a state where the jump was already used, OR take the 3-step jump from step $i-3$ where the jump had not yet been used).

3. **Base Cases:**

- $dp_0[0] = 1$, $dp_0[1] = 1$, $dp_0[2] = 2$
- $dp_1[0] = 0$, $dp_1[1] = 0$, $dp_1[2] = 0$
- At step $i = 3$: $dp_1[3] = dp_1[2] + dp_1[1] + dp_0[0] = 0 + 0 + 1 = 1$.

4. **Complexity & Optimization:**

- Time Complexity: $O(N)$
- Space Complexity: $O(1)$ by maintaining only the last 3 variables, or $O(N)$ array for clarity.

Optimal C++ Solution

```
/**
 * Problem: Uncertain Steps
 * Contest: TCS CodeVita 2019 (Round 1)
 * Topic: Dynamic Programming & Combinatorics
 * Approach: Multi-state Dynamic Programming with modular arithmetic
 * Time Complexity:  $O(N)$ 
 * Space Complexity:  $O(N)$  or  $O(1)$ 
 */

#include <iostream>
#include <vector>

using namespace std;

const long long MOD = 1000000007;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    if (!(cin >> n) || n <= 0) {
        cout << 0 << "\n";
        return 0;
    }

    if (n == 1) {
        cout << 1 << "\n";
        return 0;
    }
    if (n == 2) {
        cout << 2 << "\n";
        return 0;
    }
}
```

```

if (n == 3) {
    cout << 4 << "\n";
    return 0;
}

// dp0[i] = ways to reach step i without using the 3-step jump
// dp1[i] = ways to reach step i using exactly one 3-step jump
vector<long long> dp0(n + 1, 0);
vector<long long> dp1(n + 1, 0);

// Base cases
dp0[0] = 1;
dp0[1] = 1;
dp0[2] = 2;

dp1[0] = 0;
dp1[1] = 0;
dp1[2] = 0;

for (int i = 3; i <= n; ++i) {
    // Standard Fibonacci transitions without the jump
    dp0[i] = (dp0[i - 1] + dp0[i - 2]) % MOD;

    // Transitions with jump:
    // 1. Jump already used: take 1 step or 2 steps from dp1
    // 2. Jump used right now: take 3 steps from dp0[i-3]
    dp1[i] = (dp1[i - 1] + dp1[i - 2] + dp0[i - 3]) % MOD;
}

long long total_ways = (dp0[n] + dp1[n]) % MOD;

cout << total_ways << "\n";

return 0;
}

```

10. Minimum Gifts

Topic: Dynamic Programming & Greedy Two-Pass | Year: 2020 | Category: Self-Practice Assignment

Problem Statement

A company organizes an annual festival where gifts are distributed to N employees standing in a straight line. Each employee has an assigned performance rank based on their year-round contributions.

The HR manager needs to distribute gifts according to the following strict conditions:

1. Every employee must receive at least **1** gift.
2. If an employee has a strictly higher rank than their immediate left neighbor, they must receive strictly more gifts than the left neighbor.
3. If an employee has a strictly higher rank than their immediate right neighbor, they must receive strictly more gifts than the right neighbor.

Calculate the **minimum total number of gifts** the company needs to purchase across all employees.

Input Format

- The first line contains an integer T , the number of test cases.
- For each testcase:
 - The first line contains an integer N ($1 \leq N \leq 10^5$), denoting the number of employees.
 - The second line contains N space-separated integers representing the performance ranks of the employees ($1 \leq \text{rank}[i] \leq 10^5$).

Output Format

- For each test case, print a single integer representing the minimum total number of gifts required.

Sample Test Cases

Sample 1

Input:

```
2
5
1 2 1 5 2
2
1 2
```

Output:

```
7
3
```

Explanation:

- **Test Case 1:**
 - Employee ranks: `[1, 2, 1, 5, 2]`
 - Minimal gift allocation satisfying both left and right neighbor constraints:
 - Employee 0 (rank 1): 1 gift
 - Employee 1 (rank 2): 2 gifts (higher than left neighbor)
 - Employee 2 (rank 1): 1 gift
 - Employee 3 (rank 5): 2 gifts (higher than both neighbors)
 - Employee 4 (rank 2): 1 gift
 - Total gifts = $1 + 2 + 1 + 2 + 1 = 7$.
- **Test Case 2:**
 - Employee ranks: `[1, 2]`

- Gifts: $[1, 2] \rightarrow \text{Total} = 3$.

Approach & Notes for Students

1. **Two-Pass Greedy / DP Strategy ($O(N)$ Time, $O(N)$ Space):**

- Initialize an array `gifts` of size N with all values set to 1.

- **Pass 1 (Left to Right):**

- For $i = 1$ to $N - 1$:

- If $\text{rank}[i] > \text{rank}[i - 1]$, then $\text{gifts}[i] = \text{gifts}[i - 1] + 1$.

- **Pass 2 (Right to Left):**

- For $i = N - 2$ down to 0:

- If $\text{rank}[i] > \text{rank}[i + 1]$, then $\text{gifts}[i] = \max(\text{gifts}[i], \text{gifts}[i + 1] + 1)$.

- Accumulate the sum of all elements in `gifts` using `long long`.

2. **Why Two Passes are Necessary:**

- A single pass cannot satisfy bidirectional neighbor dependencies. The left-to-right pass satisfies all left-neighbor requirements, and the right-to-left pass satisfies right-neighbor requirements without violating the previously satisfied left constraints (guaranteed by taking $\max$).

Optimal C++ Solution

```
/**
 * Problem: Minimum Gifts
 * Contest: TCS CodeVita 2020 (Season 9)
 * Topic: Dynamic Programming & Greedy Two-Pass
 * Approach: Two-pass prefix & suffix comparison (Candy Problem)
 * Time Complexity:  $O(N)$ 
 * Space Complexity:  $O(N)$ 
 */

#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

using namespace std;

void solve() {
    int n;
    if (!(cin >> n) || n <= 0) return;

    vector<long long> ranks(n);
    for (int i = 0; i < n; ++i) {
        cin >> ranks[i];
    }

    // Step 1: Every employee receives at least 1 gift
    vector<long long> gifts(n, 1);

    // Step 2: Left-to-right pass (satisfy left neighbor condition)
    for (int i = 1; i < n; ++i) {
        if (ranks[i] > ranks[i - 1]) {
            gifts[i] = gifts[i - 1] + 1;
        }
    }

    // Step 3: Right-to-left pass (satisfy right neighbor condition)
    for (int i = n - 2; i >= 0; --i) {
        if (ranks[i] > ranks[i + 1]) {
            gifts[i] = max(gifts[i], gifts[i + 1] + 1);
        }
    }

    // Step 4: Sum all gifts using 64-bit integer
    long long total_gifts = 0;
    for (int i = 0; i < n; ++i) {
        total_gifts += gifts[i];
    }
}
```

```
    cout << total_gifts << "\n";  
}  
  
int main() {  
    // Fast I/O  
    ios_base::sync_with_stdio(false);  
    cin.tie(NULL);  
  
    int t;  
    if (cin >> t) {  
        while (t--) {  
            solve();  
        }  
    }  
  
    return 0;  
}
```

11. Critical Planets

Topic: Graph Algorithms & Bridge Finding | Year: 2022 | Category: Core Practice Problem

Problem Statement

In an interstellar galactic empire, there are N planets numbered from 0 to $N - 1$, interconnected by M bidirectional space transit tunnels.

The space council wants to safeguard the stability of the interstellar transit network. A transit tunnel between two planets is deemed **critical** (a bridge) if blocking it disconnects any part of the network, making travel between certain pairs of planets impossible.

Any planet that serves as an endpoint of at least one critical transit tunnel is declared a **Critical Planet**.

Your objective is to find all Critical Planets and print their IDs in ascending order. If no critical planets exist in the network, print `-1``.

Input Format

- The first line contains two integers: M (number of transit paths) and N (number of planets), separated by a space ($0 \leq M \leq 10^4$, $1 \leq N \leq 7000$).
- The next M lines each contain two integers u and v ($0 \leq u, v < N$), representing a bidirectional transit tunnel between planet u and planet v .

Output Format

- Print the IDs of all critical planets in ascending order, each on a new line.
- If no critical planets exist, print `-1``.

Sample Test Cases

Sample 1

Input:

```
3 4
0 1
1 2
2 3
```

Output:

```
0
1
2
3
```

Explanation:

- The network forms a simple linear chain: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.
- Every tunnel $(0, 1)$, $(1, 2)$, $(2, 3)$ is a critical bridge whose removal would disconnect the network.
- The endpoints of these bridges are planets `0, 1, 2, 3``.

Sample 2

Input:

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

Output:

```
2
3
```

Explanation:

- Planets `0, 1, 2` form a triangle (cycle), so no individual tunnel inside that cluster is critical.
- Planets `3, 4, 5` form a triangle (cycle), so no individual tunnel inside that cluster is critical.
- The tunnel between planet `2` and planet `3` connects the two clusters. If tunnel (2, 3) is severed, cluster `{0, 1, 2}` cannot communicate with cluster `{3, 4, 5}`.
- Thus, (2, 3) is the only critical bridge. The critical planets are `2` and `3`.

Approach & Notes for Students

1. **Crucial Distinction: Articulation Points vs Bridge Endpoints:**

- In standard graph theory, an **Articulation Point** (Cut Vertex) is a node whose removal increases disconnected components. On a linear graph 0-1-2-3, articulation points are only 1 and 2 (removing leaf 0 doesn't disconnect the remaining nodes).
- However, in this problem, critical planets are explicitly defined as the **endpoints of critical edges (bridges)**!
- On a bridge u-v, BOTH endpoints u and v are critical. In 0-1-2-3, all 3 edges are bridges, so vertices `{0, 1, 2, 3}` are all critical planets.

2. **Tarjan's Bridge-Finding Algorithm:**

- Maintain:
 - `disc[u]`: Discovery timestamp of node u during DFS.
 - `low[u]`: Lowest discovery time reachable from u via DFS tree and back-edges.
- For an edge u to v:
 - If v is the parent of u, skip.
 - If v is already visited (back-edge), update $\text{low}[u] = \min(\text{low}[u], \text{disc}[v])$.
 - If v is unvisited:
 - Recurse on v.
 - Update $\text{low}[u] = \min(\text{low}[u], \text{low}[v])$.
- **Bridge Condition:** If $\text{low}[v] > \text{disc}[u]$, edge (u, v) is a BRIDGE! Add both u and v to our set of critical planets.

3. **Complexity Analysis:**

- **Time Complexity:** $O(V + E)$ where $V = N$, $E = M$.
- **Space Complexity:** $O(V + E)$ for adjacency list and recursion stack.

Optimal C++ Solution

```
/**
 * Problem: Critical Planets
 * Contest: TCS CodeVita 2022
 * Topic: Graph Algorithms & Bridge Finding (Tarjan's DFS)
 * Approach: Tarjan's Bridge Finding Algorithm - low[v] > disc[u]
 * Time Complexity: O(V + E)
 * Space Complexity: O(V + E)
 */

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;

int timer = 0;
vector<int> disc, low;
vector<bool> visited;
vector<vector<int>> adj;
set<int> critical_planets;

void dfs(int u, int parent) {
    visited[u] = true;
    disc[u] = low[u] = ++timer;
    for (int v : adj[u]) {

```

```

        if (v == parent) {
            continue;
        }

        if (visited[v]) {
            // Back-edge found
            low[u] = min(low[u], disc[v]);
        } else {
            // Forward tree edge
            dfs(v, u);
            low[u] = min(low[u], low[v]);

            // Tarjan's Bridge Condition:
            // If the lowest reachable vertex from subtree rooted at v is strictly
            // discovered after u, then edge (u, v) is a bridge.
            if (low[v] > disc[u]) {
                critical_planets.insert(u);
                critical_planets.insert(v);
            }
        }
    }
}

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int m, n;
    if (!(cin >> m >> n)) return 0;

    adj.assign(n, vector<int>());
    disc.assign(n, -1);
    low.assign(n, -1);
    visited.assign(n, false);
    timer = 0;
    critical_planets.clear();

    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        if (u >= 0 && u < n && v >= 0 && v < n && u != v) {
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
    }

    // Traverse all components (handles disconnected graphs)
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i, -1);
        }
    }

    if (critical_planets.empty()) {
        cout << -1 << "\n";
    } else {
        for (int planet : critical_planets) {
            cout << planet << "\n";
        }
    }

    return 0;
}

```

12. Holes and Balls

Topic: Simulation & Traversal | Year: 2019 | Category: Self-Practice Assignment

Problem Statement

There is an inclined ramp with N holes arranged in order from bottom to top, labeled 1 to N.

- The diameter of hole i is $H[i]$.
- The capacity of hole i is exactly equal to its 1-based index i (i.e., hole 1 holds at most 1 ball, hole 2 holds at most 2 balls, ..., hole N holds at most N balls).

A sequence of M balls with varying diameters are rolled down the incline one by one from the top (starting at hole N and rolling towards hole 1).

Movement Rules for Each Ball:

1. When a ball encounters hole k (starting from $k = N$ down to 1):
 - If the ball's diameter $\leq H[k]$ AND hole k is not yet full (currently holds $< k$ balls), the ball falls into hole k and stays there.
 - Otherwise, the ball rolls past hole k to hole $k - 1$.
2. If a ball rolls past hole 1 without falling into any hole, it rolls off the ramp into position 0.

Your task is to determine the final resting position of each ball in the order they are released.

Input Format

- First line: Integer N ($1 \leq N \leq 50$), denoting the number of holes.
- Second line: N space-separated integers representing the diameters of holes $H[1]$, $H[2]$, ..., $H[N]$.
- Third line: Integer M ($1 \leq M \leq 1000$), denoting the number of balls.
- Fourth line: M space-separated integers representing the diameters of the M balls in the order of release.

Output Format

- Print M space-separated integers representing the resting hole index ($1 \dots N$) or 0 for each ball.

Sample Test Cases

Sample 1

****Input:****

```
3
21 3 6
11
20 15 5 7 10 4 2 1 3 6 8
```

****Output:****

```
1 0 3 0 0 3 3 2 2 0 0
```

****Explanation:****

- Holes:
 - Hole 1: Diameter = 21, Capacity = 1
 - Hole 2: Diameter = 3, Capacity = 2
 - Hole 3: Diameter = 6, Capacity = 3
- Balls roll from Hole 3 down to Hole 1:
- Ball 20: Too large for Hole 3 and 2; fits into Hole 1 (capacity now 1/1, full). Output: `1`.
- Ball 15: Too large for Hole 3 and 2; Hole 1 full. Rolls off $\rightarrow$ Output: `0`.
- Ball 5: Fits in Hole 3 (capacity 1/3) $\rightarrow$ Output: `3`.
- Ball 7: Too large for all $\rightarrow$ Output: `0`.

- Ball 10: Too large for all $\rightarrow$ Output: `0`.
- Ball 4: Fits in Hole 3 (capacity 2/3) $\rightarrow$ Output: `3`.
- Ball 2: Fits in Hole 3 (capacity 3/3, full) $\rightarrow$ Output: `3`.
- Ball 1: Hole 3 full; fits into Hole 2 (capacity 1/2) $\rightarrow$ Output: `2`.
- Ball 3: Fits into Hole 2 (capacity 2/2, full) $\rightarrow$ Output: `2`.
- Ball 6: All fitting holes full $\rightarrow$ Output: `0`.
- Ball 8: Too large for all $\rightarrow$ Output: `0`.

Approach & Notes for Students

1. **Simulation Logic:**

- Maintain a `count[k]` array tracking the number of balls currently in hole k .
- Max capacity for hole k is k .
- For each ball diameter d :
- Loop k backwards from N down to 1 :
- If $d \leq H[k]$ and `count[k] < k`:
- `count[k]++`
- Record k as the ball's position, and break.
- If the loop finishes without placing the ball, record 0 .

2. **Complexity Analysis:**

- Time Complexity: $O(M \times N)$ — with $M \leq 1000$ and $N \leq 50$, total operations $\approx 5 \times 10^4$ (< 0.01 seconds).
- Space Complexity: $O(N)$ for hole data.

Optimal C++ Solution

```
/**
 * Problem: Holes and Balls
 * Contest: TCS CodeVita 2019 (Round 1)
 * Topic: Simulation & Array / Stack Traversal
 * Approach: Reverse simulation of rolling balls into capacity-constrained holes
 * Time Complexity:  $O(M * N)$ 
 * Space Complexity:  $O(N + M)$ 
 */

#include <iostream>
#include <vector>

using namespace std;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    if (!(cin >> n) || n <= 0) return 0;

    // 1-indexed hole diameters and capacities
    // Hole i has diameter H[i] and max capacity i
    vector<long long> H(n + 1);
    for (int i = 1; i <= n; ++i) {
        cin >> H[i];
    }

    int m;
    if (!(cin >> m) || m <= 0) return 0;

    vector<int> current_count(n + 1, 0);
    vector<int> result(m);

    for (int i = 0; i < m; ++i) {
```

```

    long long ball_dia;
    cin >> ball_dia;

    int settled_hole = 0;

    // Ball rolls from top hole N down to bottom hole 1
    for (int k = n; k >= 1; --k) {
        if (ball_dia <= H[k] && current_count[k] < k) {
            settled_hole = k;
            current_count[k]++;
            break;
        }
    }

    result[i] = settled_hole;
}

// Output space-separated answers
for (int i = 0; i < m; ++i) {
    cout << result[i] << (i + 1 < m ? " " : "");
}
cout << "\n";

return 0;
}

```