1. **Perfect Number -** [**https://codeforces.com/problemset/problem/919/B**](https://codeforces.com/problemset/problem/919/B)

#include<bits/stdc++.h>

using namespace std;

int main() {

    int n;

    cin >> n;

    int count = 0;

    for (int i = 19; ; i++) {

        int temp = i, sum = 0;

        while (temp) {

            sum += temp % 10;

            temp /= 10;

        }

        if (sum == 10) count++;

        if (count == n) {

            cout << i << endl;

            break;

        }

    }

}

1. Aggressive cows - <https://vjudge.net/problem/SPOJ-AGGRCOW>

#include <bits/stdc++.h>

using namespace std;

bool canPlaceCows(vector<int> &stalls, int cows, int minDist) {

    int count = 1;  // First cow placed at first stall

    int lastPos = stalls[0];

    for (int i = 1; i < stalls.size(); i++) {

        if (stalls[i] - lastPos >= minDist) {

            count++;

            lastPos = stalls[i];

        }

        if (count >= cows) return true;

    }

    return false;

}

int aggressiveCows(vector<int> &stalls, int cows) {

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

    int low = 1;  // minimum possible distance is at least 1

    int high = stalls.back() - stalls[0];

    int ans = 0;

    while (low <= high) {

        int mid = low + (high - low) / 2;

        if (canPlaceCows(stalls, cows, mid)) {

            ans = mid;

            low = mid + 1;

        } else {

            high = mid - 1;

        }

    }

    return ans;

}

int main() {

    int t;

    cin >> t;

    while (t--) {

        int n, c;

        cin >> n >> c;

        vector<int> stalls(n);

        for (int i = 0; i < n; i++) {

            cin >> stalls[i];

        }

        int result = aggressiveCows(stalls, c);

        cout << result << endl;

    }

    return 0;

}

1. **Chat Ban -** [**https://vjudge.net/problem/CodeForces-1612C**](https://vjudge.net/problem/CodeForces-1612C)

Newton’s School =>

#include<bits/stdc++.h>

using namespace std;

#define ll long long

int main()

{

    ios\_base::sync\_with\_stdio(false);

    cin.tie(NULL);

    cout.tie(NULL);

    ll t, k, x, ans, l, c;

    long double x1, ans1;

    cin>>t;

    for(;t--;)

    {

        cin>>k>>x;

        c=k\*k;

        if(c<=x){

            cout<<2\*k-1<<"\n";

            continue;

        }

        c=(k\*(k+1))/2;

        if(c<=x){

            x=x-c;

            x=(k\*(k-1))/2-x;

            x1=x;

            ans1=floor((sqrt(1+8\*x1)-1)/2);

            ans=2\*k-1-ans1;

            cout<<ans<<"\n";

        }else{

            x1=x;

            ans1=ceil((sqrt(1+8\*x1)-1)/2);

            ans=ans1;

            cout<<ans<<"\n";

        }

    }

}

Another Approach 🡺

#include <iostream>

using namespace std;

typedef long long ll;

ll emotesSent(ll k, ll m) {

    if (m <= k) {

        return m \* (m + 1) / 2;

    } else {

        ll full = k \* (k + 1) / 2;

        ll rem = m - k;

        ll tail = k - 1;

        ll last = tail - rem + 1;

        return full + (tail + last) \* rem / 2;

    }

}

ll solve(ll k, ll x) {

    ll low = 1, high = 2 \* k - 1, ans = 2 \* k - 1;

    while (low <= high) {

        ll mid = (low + high) / 2;

        if (emotesSent(k, mid) >= x) {

            ans = mid;

            high = mid - 1;

        } else {

            low = mid + 1;

        }

    }

    return ans;

}

int main() {

    int t;

    cin >> t;

    while (t--) {

        ll k, x;

        cin >> k >> x;

        cout << solve(k, x) << "\n";

    }

    return 0;

}

**⏱️ Time Complexity**

**Per Test Case:**

* All operations are constant-time except for sqrt() and some arithmetic.
* So: **O(1)** per test case

1. **K-th Not Divisible by n -** [**https://vjudge.net/problem/CodeForces-1352C**](https://vjudge.net/problem/CodeForces-1352C)

Approach :

1 2 ‘3’ 4 5 ‘6’ 7 8 ‘9’ 10 11 12

n=3

k=7

🡪 10 = k + x

So, we have to find, x = ?

We can see the pattern/logic , like this => ( such as, ( 7 + 3 )/3 = 3.3333… ~ 3 )

x = (k + x) / n

=> nx = k + x;

=> x = k / (n - 1)

For,

1st test case : n=3, k=7

( 7+3 ) % 3 != 0

ans = 7 + 7 / (3-1)

2nd test case : n=4, k=12

( 12+4 ) % 4 == 0

ans = 12 + 12 / (4-1) – 1

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

int main(){

    ll t;cin>>t;

    while (t--)

    {

        ll n, k;cin>>n>>k;

        ll x = k/(n-1);

        if((k+x)%n == 0){

            cout<<k+x-1<<endl;

        }else{

            cout<<k+x<<endl;

        }

    }

    return 0;

}

**Time Complexity**

* For each test case:
  + Division, modulo, addition, and conditional check: all are **O(1)** operations.

Let:

* t = number of test cases

Then:

* **Total Time Complexity:** O(t)

**Space Complexity**

* No extra data structures used.
* Just a few variables per test case.

So:

* **Total Space Complexity:** O(1)

**E- 10474 Where is the Marble? –** [**https://vjudge.net/problem/UVA-10474**](https://vjudge.net/problem/UVA-10474)

[**https://onlinejudge.org/external/104/10474.pdf**](https://onlinejudge.org/external/104/10474.pdf)

#include <bits/stdc++.h>

using namespace std;

int main() {

    int N, Q;

    int caseNum = 1;

    while (cin >> N >> Q, N || Q) {

        vector<int> marbles(N);

        for (int i = 0; i < N; ++i) {

            cin >> marbles[i];

        }

        // Sort the marbles

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

        cout << "CASE# " << caseNum++ << ":\n";

        while (Q--) {

            int query;

            cin >> query;

            // lower\_bound returns an iterator to the first element >= query

            auto it = lower\_bound(marbles.begin(), marbles.end(), query);

            if (it != marbles.end() && \*it == query) {

                // +1 for 1-based index

                cout << query << " found at " << (it - marbles.begin() + 1) << '\n';

            } else {

                cout << query << " not found\n";

            }

        }

    }

    return 0;

}

**⏱ Time Complexity**

* For each test case:
  + Sorting marbles: O(N log N)
  + Query processing: Q × O(log N) due to binary search

Overall:

* Efficient even for maximum constraints (N, Q ≤ 10,000).

**F. Points in Segments -** [**https://vjudge.net/problem/LightOJ-1088**](https://vjudge.net/problem/LightOJ-1088)

**✅ Approaches / Explanation:**

* **Test case 1**:
  + Points: [1, 4, 6, 8, 10] (Already **sorted**)
  + Segments (queries):
    1. [0, 5]
    2. [6, 10]
    3. [7, 100000]

**🔍 Segment 1: [0, 5]**

We want all points p such that 0 ≤ p ≤ 5.

From the points [1, 4, 6, 8, 10], the points 1 and 4 fall in this range → **Answer: 2**

**🔹 Binary Search Method:**

* Use lower\_bound(0) → first point ≥ 0 → index 0
* Use upper\_bound(5) → first point > 5 → index 2
* Count = upper\_bound - lower\_bound = 2 - 0 = 2

**🔍 Segment 2: [6, 10]**

We want points p such that 6 ≤ p ≤ 10.

From the list: 6, 8, 10 fall in range → **Answer: 3**

**🔹 Binary Search:**

* lower\_bound(6) = index 2
* upper\_bound(10) = index 5
* Count = 5 - 2 = 3

**🔍 Segment 3: [7, 100000]**

We want points p such that 7 ≤ p ≤ 100000.

From the list: 8, 10 fall in this range → **Answer: 2**

**🔹 Binary Search:**

* lower\_bound(7) = index 3
* upper\_bound(100000) = index 5
* Count = 5 - 3 = 2

Code :

#include <bits/stdc++.h>

using namespace std;

int main() {

    ios::sync\_with\_stdio(false); // Fast I/O

    cin.tie(nullptr);            // Disable C-style sync

    int T;

    cin >> T;

    for (int cs = 1; cs <= T; ++cs) {

        int n, q;

        cin >> n >> q;

        vector<int> points(n);

        for (int i = 0; i < n; ++i) {

            cin >> points[i];

        }

        cout << "Case " << cs << ":\n";

        while (q--) {

            int A, B;

            cin >> A >> B;

            // Use binary search (lower\_bound and upper\_bound)

            int left = lower\_bound(points.begin(), points.end(), A) - points.begin();

            int right = upper\_bound(points.begin(), points.end(), B) - points.begin();

            cout << (right - left) << '\n';

        }

    }

    return 0;

}

**📈 Time Complexity:**

* Each query runs in **O(log n)** due to binary search
* Total: O((n + q) \* log n) → efficient for constraints (n ≤ 1e5, q ≤ 5e4)

**G. Minion Chef and Bananas -** [**https://vjudge.net/problem/CodeChef-MINEAT**](https://vjudge.net/problem/CodeChef-MINEAT)

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

typedef long long ll;

bool canEatAll(const vector<ll>& A, ll H, ll K) {

    ll totalHours = 0;

    for (ll bananas : A) {

        totalHours += (bananas + K - 1) / K;  // ceil(bananas / K)

    }

    return totalHours <= H;

}

int main() {

    int T;

    cin >> T;

    while (T--) {

        ll N, H;

        cin >> N >> H;

        vector<ll> A(N);

        ll maxBananas = 0;

        for (ll i = 0; i < N; i++) {

            cin >> A[i];

            maxBananas = max(maxBananas, A[i]);

        }

        // Binary search for the minimum possible K

        ll low = 1, high = maxBananas, result = maxBananas;

        while (low <= high) {

            ll mid = low + (high - low) / 2;

            if (canEatAll(A, H, mid)) {

                result = mid;  // try to find smaller K

                high = mid - 1;

            } else {

                low = mid + 1;

            }

        }

        cout << result << endl;

    }

    return 0;

}

**⏱ Time Complexity:**

* Binary search: O(log(max(A)))
* For each check: O(N)
* Total: O(T \* N \* log(max(A))) → Efficient even for large inputs.

**H. Shake Shake Shaky -** [**https://vjudge.net/problem/SPOJ-MAIN8\_C**](https://vjudge.net/problem/SPOJ-MAIN8_C)

**Approach to Solve**

This is a **search problem** — more specifically, a **Binary Search on the answer**.

**Why Binary Search?**

* We are looking for the **maximum number of candies per student** (x) that can be given.
* If we can give x candies to each student, then we can definitely give **less than x** candies too.
* If we **cannot** give x candies, then we cannot give **more than x** candies either.
* This monotonic property allows binary search.

**Steps:**

1. **Find search space**:
   * Minimum = 1 (at least 1 candy per student)
   * Maximum = max(candies in any box) (cannot give more than biggest box’s candies).
2. **Binary Search**:
   * For a mid value mid, check if we can serve **K students** such that each gets mid candies.
   * To check:
     + For each box with candies[i], students served = candies[i] / mid.
     + Sum up over all boxes.
     + If total served ≥ K → Possible, move to higher side (low = mid + 1).
     + Else, move to lower side (high = mid - 1).
3. **Store the last successful mid** as the answer.

**Code :**

#include <bits/stdc++.h>

using namespace std;

bool canDistribute(vector<long long> &candies, long long K, long long mid) {

    long long count = 0;

    for (long long c : candies) {

        count += c / mid; // students from this box

        if (count >= K) return true; // early exit

    }

    return count >= K;

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    int T;

    cin >> T;

    while (T--) {

        int N;

        long long K;

        cin >> N >> K;

        vector<long long> candies(N);

        long long maxCandy = 0;

        for (int i = 0; i < N; i++) {

            cin >> candies[i];

            maxCandy = max(maxCandy, candies[i]);

        }

        long long low = 1, high = maxCandy, ans = 0;

        while (low <= high) {

            long long mid = (low + high) / 2;

            if (canDistribute(candies, K, mid)) {

                ans = mid;      // possible, try bigger

                low = mid + 1;

            } else {

                high = mid - 1; // not possible, try smaller

            }

        }

        cout << ans << "\n";

    }

    return 0;

}

I. Phone Numbers- <https://vjudge.net/problem/CodeForces-940C>

<https://codeforces.com/problemset/problem/940/C>

**Problem Restatement**

We are given:

* A string s of length n
* An integer k (length of required string t)
* We must find the **smallest lexicographical string t of length k**:
  + The **set of characters** in t must be a **subset** of characters in s.
  + t must be **lexicographically greater** than s.

**Key Points**

1. We only care about the **set of letters** in s (duplicates don't matter).
2. We want **the smallest lexicographical string** that is **just greater** than s.
3. Two cases:
   * If k > n:  
     → We can extend s by adding the **smallest character** from its set until length k.
   * If k <= n:  
     → We find the **rightmost position** where we can replace a character with the next bigger available character from our set, then fill the rest with the smallest available character.

**Step-by-Step Approach**

**Step 1:**

Find the **sorted set** of characters in s.  
Example:  
s = "ayy" → unique letters = {a, y} → sorted = "ay"

**Step 2:**

Two scenarios:

**Case 1: k > n**

We can simply append smallest char to s until length k:

* Example: s = "ba", k = 3  
  smallest char = 'a' → "ba" + 'a' → "baa"

**Case 2: k <= n**

We:

1. Start from position k-1 (rightmost position in t).
2. Try to replace it with the next larger available letter from our sorted set.
3. If possible → replace, and fill all positions after it with the smallest letter.
4. If not possible → move one position left and repeat.

**Example Walkthrough**

Example:

ini

CopyEdit

n = 3, k = 3

s = "abc"

* unique sorted letters = "abc"
* k = n, so we try to modify:
  + From right:  
    position 2 (c) → no bigger letter available. Move left.
  + position 1 (b) → next letter after b is c.  
    Replace → "ac", then fill remaining with smallest (a) → "aca"

✅ Answer: "aca"

**Code :**

#include <bits/stdc++.h>

using namespace std;

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    int n, k;

    cin >> n >> k;

    string s;

    cin >> s;

    // Step 1: get sorted unique letters

    set<char> st(s.begin(), s.end());

    vector<char> letters(st.begin(), st.end());

    if (k > n) {

        // Case 1: just append smallest char until length k

        cout << s;

        for (int i = n; i < k; i++) cout << letters[0];

        cout << "\n";

        return 0;

    }

    // Case 2: modify s

    string t = s.substr(0, k); // start with first k chars of s

    // Go from right to left to find position to change

    for (int i = k - 1; i >= 0; i--) {

        // find next larger letter than t[i] in letters[]

        auto it = upper\_bound(letters.begin(), letters.end(), t[i]);

        if (it != letters.end()) {

            t[i] = \*it; // replace with next bigger char

            // fill rest with smallest char

            for (int j = i + 1; j < k; j++) t[j] = letters[0];

            cout << t << "\n";

            return 0;

        }

    }

}

**Complexity**

* **Time**:
  + O(n log n) for set creation and sorting
  + O(k log m) for binary search per position (m = number of unique letters ≤ 26)
* **Space**: O(n) for storing letters

J. Recursive Queries - <https://vjudge.net/problem/CodeForces-932B>

<https://codeforces.com/problemset/problem/932/B>



**🧩 Problem Understanding**

You're given two functions:

* **f(n)**: product of digits of n
* **g(n)**: recursively applies f until the result is a single-digit number

**✨ Function Definition**

f(n): product of digits of n, ignoring zeros

g(n):

if n < 10 → return n

else → return g(f(n))

**🔍 Task**

You’re given **Q queries**. Each query contains three integers: l, r, and k. You must count how many integers x in the range [l, r] satisfy g(x) = k.

**📌 Example Test Case**

**Input:**

4

22 73 9

45 64 6

47 55 7

2 62 4

**Output:**

1

4

0

8

**Explanation:**

Let’s walk through the first query: 22 73 9

We need to count how many numbers x in [22, 73] satisfy g(x) = 9.

* Try x = 33:
  + Digits: 3 and 3 → product = 9
  + g(33) = g(9) = 9 ✅
* Try x = 39:
  + Digits: 3 and 9 → product = 27 → g(27) = g(2×7) = g(14) = g(1×4) = g(4) = 4 ❌

Only x = 33 satisfies g(x) = 9 in that range → **Answer: 1**

**🧠 Solution Approach**

**🔍 Key Observations:**

* g(x) always reduces to a digit between 1 and 9.
* We can precompute g(x) for all x from 1 to 10⁶.
* For each digit k from 1 to 9, we build a prefix sum array cnt[k][x] that stores how many numbers ≤ x have g(x) = k.

**⚙️ Steps:**

1. **Precompute** g(x) for all x in [1, 10⁶].
2. **Build prefix sums**: For each digit k, store cumulative counts of g(x) = k.
3. **Answer each query in O(1)** using:

cnt[k][r]−cnt[k][l−1]

**Code:**

#include <bits/stdc++.h>

using namespace std;

const int MAX = 1e6 + 5;

int cnt[10][MAX]; // cnt[k][i] = count of g(x) == k for x in [1..i]

// Correct g(x) function: skips zeros in digit product

int g(int x) {

while (x >= 10) {

int prod = 1;

while (x > 0) {

int d = x % 10;

if (d != 0) prod \*= d;

x /= 10;

}

x = prod;

}

return x;

}

// Precompute g(x) and build prefix sums

void preprocess() {

for (int i = 1; i < MAX; ++i) {

int val = g(i);

for (int k = 1; k <= 9; ++k) {

cnt[k][i] = cnt[k][i - 1] + (val == k);

}

}

}

// Answer each query in O(1)

int query(int l, int r, int k) {

return cnt[k][r] - cnt[k][l - 1];

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

preprocess();

int Q;

cin >> Q;

while (Q--) {

int l, r, k;

cin >> l >> r >> k;

cout << query(l, r, k) << '\n';

}

return 0;

}

**🧮 Time and Space Complexity**

| **Operation** | **Complexity** |
| --- | --- |
| Preprocessing | O(N × D) where D is number of digits in x (≈ log₁₀N) |
| Query Processing | O(1) per query |
| Space | O(9 × N) for prefix sums |

K. Print The Pattern - <https://vjudge.net/problem/SPOJ-ABACABA>

**🧩 Problem Understanding**

You're given a number N (1 ≤ N ≤ 15). You need to print a specific pattern based on this number.

**🔁 Pattern Rule:**

The pattern follows a recursive structure:

* P(1) = A
* P(2) = P(1) + B + P(1) = ABA
* P(3) = P(2) + C + P(2) = ABACABA
* P(4) = P(3) + D + P(3) = ABACABADABACABA
* ...
* P(N) = P(N-1) + char(N) + P(N-1)

Where char(N) is the N-th uppercase letter: 'A' + N - 1

**🧠 Solution Approach**

This is a classic recursive pattern generation problem.

**Strategy:**

* Use a recursive function build(N) that returns the pattern for level N.
* Base case: N == 1 → return "A"
* Recursive case: build(N) = build(N-1) + char(N) + build(N-1)

**Code:**

#include <iostream>

using namespace std;

string build(int n) {

    if (n == 1) return "A";

    string prev = build(n - 1);

    char mid = 'A' + n - 1;

    return prev + mid + prev;

}

int main() {

    int N;

    cin >> N;

    cout << build(N);

    return 0;

}

**🧮 Time and Space Complexity**

| **Aspect** | **Complexity** |
| --- | --- |
| Time | O(2ⁿ) — exponential growth of pattern |
| Space | O(2ⁿ) — due to string concatenation |

For N = 15, the output length is 2¹⁵ - 1 = 32767 characters, which is acceptable within the constraints.

**L. Tower of Hanoi Movement – Easy -** [**https://vjudge.net/problem/SPOJ-TOHMOVE1**](https://vjudge.net/problem/SPOJ-TOHMOVE1)

**1. Problem Understanding**

We’re given the **Tower of Hanoi** puzzle, but we **don’t** need to print the full sequence of moves.  
Instead:

* We have **N disks**.
* We want to know **what happens in the a-th move** of the **optimal** solution sequence.

Rules:

1. Only one disk moves at a time.
2. A larger disk can’t go on top of a smaller disk.
3. Moves follow the optimal Tower of Hanoi strategy.

**Example from the problem**

**Case:** N = 3, a = 5

Full move sequence for N=3:

javascript

CopyEdit

1 : A => C

2 : A => B

1 : C => B

3 : A => C

1 : B => A ← 5th move

2 : B => C

1 : A => C

Output for this case:

javascript

CopyEdit

1 : B => A

Meaning: On the 5th move, disk 1 moves from rod B to rod A.

**2. Approach**

We use the recursive nature of Tower of Hanoi to **directly jump to the a-th move** instead of simulating all moves.

**Tower of Hanoi recursive structure**

For N disks (from from to to using aux):

1. Move N-1 disks from from → aux (using to) → **(First phase)**
2. Move disk N from from → to → **(Middle move)**
3. Move N-1 disks from aux → to (using from) → **(Last phase)**

Number of moves in first phase = 2^(N-1) - 1 (let’s call this half).

**Algorithm logic**

1. **Base case**: If n == 1 → only move disk 1 from from to to.
2. **Check where a lies**:
   * If a <= half → it's in **first phase** → recursively search in n-1 disks moving from from → aux.
   * If a == half + 1 → it's the **middle move** → print n : from => to.
   * If a > half + 1 → it's in **last phase** → recursively search in n-1 disks moving from aux → to.

**Code :**

#include <bits/stdc++.h>

using namespace std;

void findMove(int n, long long a, char from, char aux, char to) {

    if (n == 1) {

        cout << "1 : " << from << " => " << to << "\n";

        return;

    }

    long long half = (1LL << (n - 1)) - 1; // moves in first part

    if (a <= half) {

        // Move lies in first phase

        findMove(n - 1, a, from, to, aux);

    }

    else if (a == half + 1) {

        // This is the move of largest disk

        cout << n << " : " << from << " => " << to << "\n";

    }

    else {

        // Move lies in second phase

        findMove(n - 1, a - (half + 1), aux, from, to);

    }

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    int T;

    cin >> T;

    while (T--) {

        int N;

        long long a;

        cin >> N >> a;

        findMove(N, a, 'A', 'B', 'C');

    }

    return 0;

}

**3. Complexity Analysis**

* **Time Complexity**:
  + Each recursive step reduces n by 1 → O(N) in worst case.
  + Much faster than O(2^N) simulation.
* **Space Complexity**:
  + Recursion depth = N → O(N) stack space.