

2026 USACO.Guide Informatics Tournament Editorial

There were 11 total problems this year, with 7 in Standard and 8 in advanced.

1 Bessie and Trap (Standard A)

Problem Idea: Tristan Sun

Problem Preparation: Tristan Sun

Analysis By: Tristan Sun

Let k be the initial number of keys. After entering room i (0-indexed), Bessie will have $k + b_0 + b_1 + \dots + b_i$ keys. If $i < n - 1$, this quantity must be at least a_{i+1} ; otherwise, Bessie can't enter the next room. It suffices to find the minimum k such that

$$\begin{aligned} k + b_0 + b_1 + \dots + b_i &\geq a_{i+1} & 0 \leq i < n \\ \iff k &\geq \max_{0 \leq i < n} (a_{i+1} - (b_0 + \dots + b_i)). \end{aligned}$$

The last expression can easily be found by processing the array from left to right while maintaining a cumulative sum.

2 Bessie and Rounding (Standard B)

Problem Idea: Rohan Garg

Problem Preparation: Tristan Sun

Analysis By: Tristan Sun

There are two cases, $X < Y$ and $X > Y$.

Case 1. In one operation, X can increase by at most X . Hence, the answer is at least $\lceil \log_2(Y/X) \rceil$. But this lower bound on the answer is also achievable. While $2X \leq Y$, choose $M = 2X$. Once $2X > Y$, choose $M = Y$. It's easy to verify that this algorithm uses exactly $\lceil \log_2(Y/X) \rceil$ operations.

Case 2. Note that we must never allow X to become 0, because then it is impossible to reach Y . How much can X decrease by in a single operation while still remaining positive? First, if $X \leq 3$, then X cannot decrease at all. Otherwise, we must choose a value of M that is less than X in order to decrease X while keeping it positive. It turns out that we can limit ourselves to only choosing values of M such that $X < 2M$; this is roughly because if $2M \leq X$ and the operation decreases the value of X , then choosing $M' = 2M$ instead would decrease the value of X by at least as much.

In order for X to decrease, we must have

$$\begin{aligned} X - M &< 2M - X \\ \iff M &> \frac{2X}{3} \\ \iff M &> \left\lfloor \frac{2X}{3} \right\rfloor \\ \iff M &\geq \left\lfloor \frac{2X}{3} \right\rfloor + 1 \stackrel{\text{def}}{=} f(X) \end{aligned}$$

Clearly, X will decrease the most if $M = f(X)$. Thus, we run the following algorithm to find the answer. While $f(X) \geq Y$, choose $M = f(X)$; otherwise, choose $M = Y$.

3 Bessie and Array (Standard C)

Problem Idea: Sachet Abeysinghe

Problem Preparation: Sachet Abeysinghe

Analysis By: Sachet Abeysinghe

The simple strategy is just to swap the first and last elements, the second and second to last elements, and so on. Thus, if there are no duplicates, the minimum number of swaps is $\lfloor N/2 \rfloor$. However, if there are duplicates, it is unnecessary to swap an element and its mirrored element if they are the same, because that does nothing. So, we subtract one from the answer whenever we encounter that.

Now that we know the minimum swaps X needed, how do we know whether exactly K swaps work? In any case, we will definitely require that $K \geq X$. If that condition is true, then there are two cases:

Case 1. There are no duplicates. In this case, to do more than swaps than X , we need to do additional swaps while preserving the transformed array. This can be accomplished by swapping any two elements, and then swapping the same two elements to undo that, and then repeating. Since this takes sets of two operations, $K - X$ must be even.

Case 2. There are duplicates. In this case, to do more swaps than X , we can simply swap any two equal elements repeatedly, as this does nothing. Thus, any $K \geq X$ works, and no additional conditions need to be imposed.

4 Bessie and Infinite Dungeon (Standard D, Advanced A)

Problem Idea: Kyle Hsiung

Problem Preparation: Tristan Sun

Analysis By: Tristan Sun

Define $dp_{i,c}$ to be an ordered pair. When i heros are dead and there are c coins remaining in your wallet, the first value is the maximum damage dealt to all bosses, the second value is the health of the current boss, and the third value is the index of the current boss. We have the following transitions:

1. Skip the current boss. This corresponds to $dp_{i,c} \rightarrow dp_{i+1,c}$.
2. Fight the current boss. There are two cases. If the hero doesn't do enough damage to kill the current boss, then the second and third values don't change. Otherwise, the second value resets to the health of the next boss while the third value increases by 1. In both cases, the first value decreases by the cost of the current hero.

This solution runs in $O(MC + N)$.

Less efficient solutions which ran in $O(MC \log N + N)$ time were also accepted.

5 Bessie and Groups (Standard E, Advanced B)

Problem Idea: Arnav Gokhale

Problem Preparation: Arnav Gokhale

Analysis By: Tristan Sun

We will describe how to solve for a fixed k in $O(n/k)$. By running the algorithm for all $k \mid n$, the problem can be solved in time equal to the sum of the divisors of n . For $n = 98280$, the sum of divisors is maximal and equal to 403200.

First, we check that it's actually possible to rearrange the groups into a sorted array. One requirement is that the values within each group are sorted. Another requirement is that the range of values inside each group doesn't strictly overlap with that of other groups; this can be checked in several ways, i.e. with a sparse table. After checking these two conditions, we no longer care about the detailed structure inside each group; all that matters is their relative order.

The number of swaps of adjacent elements required to sort an array equals the number of inversions in the array; this is because each swap removes at most one inversion. For more information, see [this Codeforces blog](#). Now the problem is reduced to finding which cyclic shift of the array minimizes $(\# \text{ left shifts to reach this cyclic shift}) + (\# \text{ inversions})$. There are various ways to do this. We will describe how to keep track the number of inversions while elements are taken from beginning of the array and moved to the back. When $x = a_0$ is removed from the beginning of the array, inversions with values $a_i < a_0$ are deleted. When x is added to the end of the array, inversions with values $a_i > a_0$ are created. Both of these can be determined by remembering, for each group, how many groups are smaller and how many are larger.

6 Bessie at the Bank (Standard F, Advanced C)

Problem Idea: Sizhe Zheng

Problem Preparation: Sizhe Zheng and Tristan Sun

Analysis By: Sizhe Zheng

This problem can be reframed as a divisibility-invariant problem over a dynamic array (which can be implemented using Persistent Arrays). After each correction, Bessie wants to compute the sum of all her favorite numbers d_i that divide every number in the current security code.

The key observation is that a number d_i divides all elements of the array if and only if it divides their greatest common divisor. Rather than explicitly maintaining the GCD after each update (which is difficult to do efficiently in a persistent setting) we take advantage of the fact that the number of favorite values is small (< 30) and encode divisibility information using bitmasks.

For each number in the array, we construct a bitmask with one bit per favorite value. Bit i is set if and only if d_i divides that number. Under this representation, a favorite number divides all elements of the array exactly when its bit is set in the bitwise AND of the masks for every position in the array.

To support updates while preserving previous versions of the array, we use a persistent segment tree. Each leaf stores the bitmask corresponding to one array element, and each internal node stores the

bitwise AND of its two children. When an update occurs, only $O(\log n)$ nodes need to be copied, while the rest of the structure is shared with earlier versions.

After each correction, the root of the corresponding segment tree version contains the AND of all element masks, which directly encodes the set of favorite numbers dividing every element of the current security code. To answer the query, we simply sum the values d_i whose bits are set in this root mask. The total running time is $O((N + Q) \log N)$, with each query requiring only $O(D)$ to sum the selected favorite numbers, where $D \leq 30$. This is efficient enough to handle the problem constraints comfortably.

Here is Sizhe Zheng's Java implementation:

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

public class BessieAtTheBank {
    static long[] divisors;

    // Convert a number to a d-bit mask (d <= 30)
    static int toMask(long x) {
        int mask = 0;
        for (int i = 0; i < divisors.length; i++) {
            // Bit i is set iff divisors[i] divides x
            if (x % divisors[i] == 0) mask |= (1 << i);
        }
        return mask;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer st = new StringTokenizer(br.readLine());

        int n = Integer.parseInt(st.nextToken());
        int q = Integer.parseInt(st.nextToken());
        int d = Integer.parseInt(st.nextToken());

        divisors = new long[d];
        st = new StringTokenizer(br.readLine());
        for (int i = 0; i < d; i++) divisors[i] = Long.parseLong(st.nextToken());

        int[] arr = new int[n];
        st = new StringTokenizer(br.readLine());
        for (int i = 0; i < n; i++) arr[i] = toMask(Long.parseLong(st.nextToken()));

        PersistentArray tree = new PersistentArray(n);
        PersistentNode[] versions = new PersistentNode[q + 1];
    }
}
```

```

// Build initial version (version 0)
versions[0] = tree.build(arr);
StringBuilder out = new StringBuilder();
out.append(tree.getSum(versions[0])).append("\n");

for (int i = 1; i <= q; i++) {
    st = new StringTokenizer(br.readLine());
    int prevVersion = Integer.parseInt(st.nextToken());
    int idx = Integer.parseInt(st.nextToken()) - 1;
    int newMask = toMask(Long.parseLong(st.nextToken()));

    versions[i] = tree.update(versions[prevVersion], idx, newMask);

    long sum = tree.getSum(versions[i]);
    out.append(sum).append("\n");
}
System.out.println(out.toString().trim());
}

static class PersistentNode {
    PersistentNode left, right;
    int mask; // AND of all masks in the segment

    // Leaf node
    PersistentNode(int mask) {
        this.mask = mask;
    }

    // Merge two nodes
    PersistentNode(PersistentNode left, PersistentNode right) {
        this.left = left;
        this.right = right;
        this.mask = left.mask & right.mask;
    }
}

static class PersistentArray {
    private final int n;

    public PersistentArray(int n) {
        this.n = n;
    }

    private PersistentNode build(int[] masks, int l, int r) {
        if (l == r) return new PersistentNode(masks[l]);
        int mid = (l + r) >>> 1;

```

```

        PersistentNode left = build(masks, l, mid);
        PersistentNode right = build(masks, mid + 1, r);
        return new PersistentNode(left, right);
    }

    private PersistentNode update
        (PersistentNode prev, int l, int r, int idx, int newMask) {
        if (l == r) return new PersistentNode(newMask);
        int mid = (l + r) >>> 1;
        if (idx <= mid) {
            PersistentNode newLeft = update(prev.left, l, mid, idx, newMask);
            return new PersistentNode(newLeft, prev.right);
        } else {
            PersistentNode newRight =
                update(prev.right, mid + 1, r, idx, newMask);
            return new PersistentNode(prev.left, newRight);
        }
    }

    public PersistentNode build(int[] masks) {
        return build(masks, 0, n - 1);
    }

    public PersistentNode update(PersistentNode prev, int idx, int newMask) {
        return update(prev, 0, n - 1, idx, newMask);
    }

    public long getSum(PersistentNode root) {
        long sum = 0;
        int rootMask = root.mask;
        for (int j = 0; j < divisors.length; j++) {
            if ((rootMask & (1 << j)) != 0) {
                sum += divisors[j];
            }
        }
        return sum;
    }
}

```

7 Bessie and Kaprekar (Standard G, Advanced D)

Problem Idea: khluo

Problem Preparation: khluo and Tristan Sun

Analysis By: khluo

Observe that $K_n(x)$ depends only on the multiset of the digits of x , which we will call s . There are $\binom{n+9}{9}$ distinct s and therefore at most $\binom{n+9}{9}$ distinct values of $K_n(x)$.

For each s , we can compute the number of corresponding values of x and the value of $K_n(x)$ for all such x , each in $O(n)$.

What remains is to determine the total number of x from which each potential y is reachable, which is possible in $O\left(\binom{n+9}{9}\right)$ time. After that, each query can be answered in constant time.

8 Bessie and GCD (Advanced E)

Problem Idea: duckling

Problem Preparation: duckling

Analysis By: Tristan Sun

Instead of finding $f(1, r)$, we will find $f(1, r)$ and $f(1, l - 1)$. But first, let's learn to solve an easier problem. Define $g(r)$ to be the number of coprime pairs of positive integers (a, b) such that $1 \leq a + b \leq r$. Note that the condition $\gcd(a, b) = 1$ is the same as $\gcd(a, r) = 1$. Thus,

$$g(r) = \sum_{i=1}^r \phi(i).$$

where $\phi(x)$ is Euler's Totient Function. To solve the original problem, note that

$$f(1, r) = \sum_{p|r} g(r/p) = \sum_{p|r} \sum_{i=1}^{r/p} \phi(i) = \sum_{p|r} \Phi(r/p).$$

Here's how to compute $f(1, r)$ quickly for all r . First, calculate $\phi(x)$ and its prefix sums $\Phi(x)$. One way to do this is by using [linear sieve](#). Next, for each prime p , iterate over its multiples kp and increase the corresponding value of $f(1, r)$ by $\Phi(k)$. The total time complexity is $O(n \log \log n)$.

9 Bessie and XOR (Advanced F)

Problem Idea: Rohan Garg

Problem Preparation: Tristan Sun

Analysis By: Tristan Sun

First of all, note that it's unnecessary to perform an operation on the same segment twice. Also, instead of thinking about XORing entire segments, we consider the difference array d of a where the effect of XORing the segment $[l, r]$ by x is the same as XORing d_l and d_r by x . Thus, the problem can be rephrased as follows. You are given $n + 1$ nodes with values written on them and m edges (l, r) . Assign a weight to edge such that the XOR of the weights of incident edges on each node equals the value written on the node, or report that it's impossible.

Observe that the values written on the nodes of each connected component must have total XOR 0 because each edge has two endpoints, meaning that it leaves 0 net contribution to the total XOR.

If any connected component has nonzero XOR, report that it's impossible. Otherwise, for each component, find an arbitrary spanning tree. We will assign weights to the edges of this spanning tree recursively. Specifically, root the tree at any vertex and find any leaf. Assign the weight of the edge incident on said leaf to be equal to the leaf's value. Repeat this process with the leaf and the edge removed from the tree. In the end, the root will have the correct XOR because the total XOR of all values in the component is 0.

10 Bessie and Gifts (Advanced G)

Problem Idea: ???

Problem Preparation: Tristan Sun

Analysis By: Tristan Sun

This problem can be solved using Divide and Conquer. Writeup soon.

11 Bessie and Heist (Advanced H)

Problem Idea: Aryan

Problem Preparation: Tristan Sun

Analysis By: Aryan

All shots making the same angle of incidence with the crystal's sides is equivalent to saying their delta, $|t - s| \pmod n$, is constant for all shots, so let's define $d = |t - s| \pmod n$ and only consider shots with delta d . Any shot with delta d can be made exactly with an equivalent shot of delta $n - d$, so we can additionally constrain d to be at most $\lfloor \frac{n}{2} \rfloor$.

Due to the law of reflection, the laser will first hit side s , then side $t = s + d \pmod n$, then side $s + 2d \pmod n$, and so on. The laser will hit sides according to the arithmetic series $s + kd \pmod n$.

If the beam ends due to reaching a shattered side, it must end on the original drilled side s . This is because every side x can only be reached from a single predecessor side $x - d$. The first time the beam revisits a side must be a side which the predecessor was previously unvisited, which is only true for side s . The maximum length of a beam therefore satisfies

$$s + kd = s \pmod n \quad k > 0$$

which happens when

$$k = \frac{n}{\gcd(n, d)}.$$

A laser of delta d can only affect sides of the same residue class mod $\gcd(n, d)$. This splits the array into $\gcd(n, d)$ different disjoint sets based on residue, each of length $\frac{n}{\gcd(n, d)}$. The elements of these sets can be ordered in a cyclic array with adjacent elements d apart in the original array to follow the laser's orbit, which I'll call an orbit. Denote the k -th element of the r -th such orbit O_r to be $a_{r+kd \pmod n}$. All elements with index k in O_r satisfy $k \equiv r \pmod{\gcd(n, d)}$.

However, most shots have a smaller energy than $\frac{n}{\gcd(n,d)}$. A shot with energy e corresponds to a subarray of length

$$\min\left(e, \frac{n}{\gcd(n,d)}\right)$$

of any orbit. Suppose we take s_r shots from orbit O_r . Then, we can interpret this as each orbit costing s_r shots in return for a profit of the maximum sum of s_r disjoint subarrays of O_r , with a budget of m shots and a goal to maximize total profit. If we can calculate the maximum sum of r disjoint subarrays of a cyclic array, this reduces to a classic hashmap problem.

You can compute the maximum sum of k subarrays of noncyclic array v by maintaining 2 dp arrays:

$\text{dp}_1[i][j]$ = maximum sum of j subarrays where the j 'th subarray ends at i

$\text{dp}_2[i][j]$ = maximum sum of j subarrays ending anywhere $\leq i$.

The transitions are:

$$\text{dp}_1[i][j] = \max(\text{dp}_1[i-1][j] + a_i, \text{dp}_2[i-1][j-1] + a_i)$$

$$\text{dp}_2[i][j] = \max(\text{dp}_1[i][j], \text{dp}_2[i-1][j])$$

This can be extended to cyclic subarrays by noticing that only one subarray can actually “wrap around” (continuing from n to 1) at a time. If a subarray wraps around, it must leave a gap between subarrays. If no subarray wraps around, we can use the non-cyclic dp as before, and if a subarray does wrap around, we can minimize and subtract off the k non-cyclic subarrays between. If the sum of elements in our array is S , the quantity we are looking for is:

$$\max \text{ disjoint subarrays}, S - \min \text{ disjoint subarrays}$$

We can loop from $d = 1$ to $d = \lfloor \frac{n}{2} \rfloor$ and compute the maximum sum of shattered sides as described above. The answer is the maximum of such values among all d .

For max subarray sum calculations we do an

$$O\left(m \cdot \frac{n}{\gcd(n,d)}\right)$$

dp loop $\gcd(n,d)$ times. Afterwards we do an $O(\gcd(n,d)m^2)$ knapsack calculation. This is $O(mn + \gcd(n,d)m^2)$ calculations for each value of d . Because

$$\sum_{d=1}^n \gcd(n,d) = O(n \log(n))$$

after computing this for all values of d , the final answer will take

$$O(m^2n + n \log(n)m^2)$$

time which comfortably passes constraints.

```

#include <bits/stdc++.h>
using namespace std;
#define ll long long
const ll MOD = 1e9 + 7;
const ll INF = 1e18;

ll gcd(ll a, ll b) {
    return b ? gcd(b, a % b) : a;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int T;
    cin >> T;
    while (T--) {
        int n, m;
        cin >> n >> m;
        vector<ll> a(n);
        for (int i = 0; i < n; i++) cin >> a[i];

        ll ans = 0;
        for (int d = 1; d <= n/2; d++) {
            int g = gcd(n, d);

            // construct orbits
            vector<vector<ll>> orbits(g, vector<ll>(n/g));
            for (int r = 0; r < g; r++) {
                for (int k = 0; k < n/g; k++) {
                    orbits[r][k] = a[(r + (k) * d) % n];
                }
            }

            // compute subarray dp for each orbit
            vector<ll> overall(m+1, -INF);
            overall[0] = 0;

            for (int r = 0; r < g; r++) {
                vector<ll>& orbit = orbits[r];
                ll total = 0;
                for (int x : orbit)
                    total += x;

                vector<vector<ll>> dpLocalMax(n/g+1, vector<ll>(m+1, -INF));
                vector<vector<ll>> dpGlobalMax(n/g+1, vector<ll>(m+1, -INF));
                vector<vector<ll>> dpLocalMin(n/g+1, vector<ll>(m+1, INF));
            }
        }
    }
}

```

```

vector<vector<ll>> dpGlobalMin(n/g+1, vector<ll>(m+1, INF));
dpGlobalMax[0][0] = 0;
dpGlobalMin[0][0] = 0;

for (int i = 1; i <= n/g; i++) {
    for (int j = 0; j <= m; j++) {
        if (j > 0 && dpGlobalMax[i-1][j-1] != -INF)
            dpLocalMax[i][j] = dpGlobalMax[i-1][j-1] + orbit[i-1];
        if (j > 0 && dpGlobalMin[i-1][j-1] != INF)
            dpLocalMin[i][j] = dpGlobalMin[i-1][j-1] + orbit[i-1];
        if (dpLocalMax[i-1][j] != -INF)
            dpLocalMax[i][j] = max(dpLocalMax[i][j],
                                   dpLocalMax[i-1][j] + orbit[i-1]);
        if (dpLocalMin[i-1][j] != INF)
            dpLocalMin[i][j] = min(dpLocalMin[i][j],
                                   dpLocalMin[i-1][j] + orbit[i-1]);
        dpGlobalMax[i][j] = max(dpGlobalMax[i-1][j],
                                dpLocalMax[i][j]);
        dpGlobalMin[i][j] = min(dpGlobalMin[i-1][j],
                                dpLocalMin[i][j]);
    }
}

vector<ll> orbit_dp(m+1, -INF);
for (int j = 0; j <= m; j++) {
    orbit_dp[j] = dpGlobalMax[n/g][j];
    if (j > 0 && dpGlobalMin[n/g][j] != INF) {
        orbit_dp[j] =
            max(orbit_dp[j], total - dpGlobalMin[n/g][j]);
    }
}

// knapsack to find best use of shots
for (int cap = m; cap >= 0; cap--) {
    if (overall[cap] == -INF) continue;
    for (int take = 1; take + cap <= m; take++) {
        if (orbit_dp[take] == -INF) continue;
        overall[cap + take] = max(overall[cap + take],
                                   overall[cap] + orbit_dp[take]);
    }
}

for (int i = 0; i <= m; i++)
    ans = max(ans, overall[i]);
}

```

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