PreTOI16

Editorial

น้ำหยดลงหินทุกวัน หินมันยังกร่อน by Plurm2545

Observation 1

การถามว่ามีกี่ช่องเป็น 1 ในการทำนายหลังผ่านไป T สหัสวรรษ แปลว่า มีเหลี่ยมจัตุรัสความยาวด้าน $T+1$ กี่รูปในรูปใหญ่ที่เป็น 1 ทั้งสี่เหลี่ยม

Convention

ในเฉลยต่อไปนี้ จะพิจารณาช่อง (r, c) ตั้งแต่ $(1, 1)$ ถึง (N, M) นับจากบนลงล่าง ซ้ายไปขวา แถวก่อนคอลัมน์ และ (r_1, c_1, r_2, c_2) แทนช่องทั้งหมดที่อยู่ในแถวระหว่าง r_1 ไปจนถึง r_2 และ คอลัมน์ c_1 ไปจนถึง c_2 กล่าวคือช่อง (r, c) เป็นสมาชิกของ (r_1, c_1, r_2, c_2) ก็ต่อเมื่อ $r_1 \le r \le r_2$ และ $c_1 \le c \le c_2$

และจะใช้ S แทนความยาวด้านของสี่เหลี่ยม ($S = T + 1$)

Subtask 1 - $N, M \le 20$

Brute force โดยการไล่ r_1, c_1, r_2, c_2 เพื่อหาสี่เหลี่ยม (r_1, c_1, r_2, c_2) เป็นสี่เหลี่ยม เทียบเงื่อนไข $r_2 - r_1 + 1 = c_2 - c_1 + 1$ (ความยาวด้านต้องเท่ากัน) แล้วไล่ r_i, c_i จาก r_1 ถึง r_2 และ c_1 ถึง c_2 ตามลำดับ แล้วนับว่าเป็น 1 ทั้งตารางหรือไม่

Time complexity : $O(N^3M^3)$

(อาจ optimize เวลาเหลือ $N^2M^2\min(N, M)$ ได้หากกรองสี่เหลี่ยมจัตุรัสเท่านั้นก่อนทำการตรวจเช็ค)

Subtask 2 - $N, M \le 150$

มีหลายวิธีเช่น

- Brute force โดยการไล่ r_1, c_1 แทนขอบบนซ้าย แล้วตรวจสอบว่าจะยืดความยาวไปยังช่อง r_2, c_2 ได้หรือไม่
- เก็บ 2D Quick Sum แล้วเช็คทุก (r_1, c_1, r_2, c_2) ว่า เป็นสี่เหลี่ยมจัตุรัส และ มีเลข 1 ทั้งหมดหรือไม่

Time complexity : $O(N^2M^2)$

Subtask 3 - $N, M \leq 400$

- Brute force โดยการไล่ r_1, c_1 แทนขอบบนซ้าย แล้วพิจารณาสี่เหลี่ยมจัตุรัสที่มีขอบบนซ้ายอยู่ที่ช่องนี้ แล้วตรวจสอบว่าเป็น 1 ทั้งสี่เหลี่ยมจัตุรัสหรือไม่
- เก็บ 2D Quick Sum แล้วไล่ (r_1, c_1, S) เพื่อเช็คว่าตารางส่วน $(r_1, c_1, r_1 + S - 1, c_1 + S - 1)$ มี 1 ทั้งตารางหรือไม่

Time complexity : $O(NM \min(N, M))$

Observation 2

หากสี่เหลี่ยมจัตุรัส $(r_1, c_1, r_1 + S, c_2 + S)$ มี 1 ทั้งสี่เหลี่ยม แสดงว่า $(r_1, c_1, r_1 + S - 1, c_2 + S - 1)$ ก็มี 1 ทั้งสี่เหลี่ยมด้วย สำหรับจำนวนเต็มบวก S ใด ๆ

Subtask 4 - $N, M \leq 2\,000$

จาก Observation 2 จะสามารถทำการ binary search ได้ โดยปรับจากทั้งสองวิธีของ subtask 3 ดังนี้ ไล่ (r_1, c_1) แทนจุดบนซ้าย แล้วหาว่า S ที่มากที่สุด ที่สี่เหลี่ยม $(r_1, c_1, r_1 + S - 1, c_1 + S - 1)$ มี 1 ทั้งตารางนั้นเป็นเท่าใด สังเกตว่า หากเราทราบ S มากสุด สำหรับ (r_1, c_1) ใด ๆ แล้ว จะมีคำตอบเพิ่มขึ้นในช่อง $1, 2, 3, \ldots, S$ ซึ่งในขั้นตอนนี้สามารถสร้างอาเรย์เสริมแล้วค่อยทำ Quick Sum ตอนจบได้

Time complexity : $O(NM \log(\min(N, M)))$

เพื่อเป็นการนำทางไปต่อใน Subtask ต่อไป เราสามารถดัดแปลงวิธีการจากการกำหนด (r_1, c_1) แล้วหา S มาเป็นการกำหนด (r_2, c_2) แล้วหาว่า S ที่มากที่สุดที่สี่เหลี่ยม $(r_2 - S + 1, c_2 - S + 1, r_2, c_2)$ มี 1 ทั้งสี่เหลี่ยมเป็นเท่าใด ก็จะได้คำตอบเท่ากันกับแบบเดิม

Observation 3

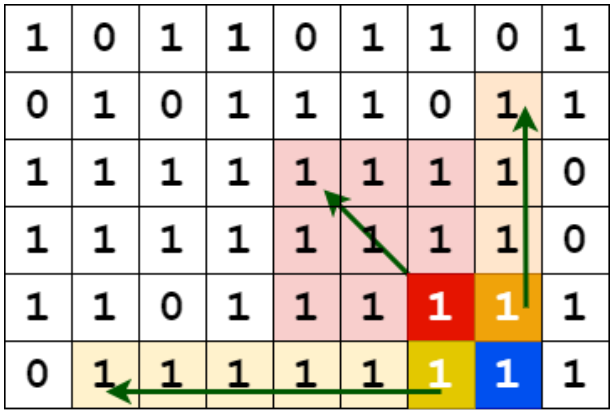
หากเราทราบว่าสี่เหลี่ยม $(r_2 - S + 1, c_2 - S + 1, r_2, c_2)$ เป็นสี่เหลี่ยมจัตุรัสที่มีค่า S สูงสุดเท่าที่เป็นไปได้แล้ว (หาก S มากกว่านี้จะตกขอบ หรือปรากฏ 0 ในสี่เหลี่ยมนั้น) จะได้ว่า S สำหรับการกำหนดขอบล่างขวาไว้ที่ช่อง (r_2+1, c_2+1) นั้นมีค่าไม่เกิน $S + 1$

Subtask 5 - $N, M \leq 4\,500$

จาก Observation 3 เราสังเกตว่าค่า S มากสุดของการกำหนดขอบขวาล่างไว้ที่ช่อง (r_2+1, c_2+1) นั้น ขึ้นอยู่กับ S มากสุดของการกำหนดขอบขวาล่างไว้ที่ (r_2, c_2) และยังขึ้นอยู่กับแถวที่ r_2+1 และคอลัมน์ที่ c_2+1 อีกด้วย

สมมติให้

- $B_{r,c}$ แทน ค่า S มากสุดที่ทำให้ ตาราง (ไม่จำเป็นต้องจัตุรัส) ในช่อง $(r-S+1, c, r, c)$ เป็น 1 ทั้งหมด
- $C_{r,c}$ แทน ค่า S มากสุดที่ทำให้ ตารางในช่อง $(r, c-S+1, r, c)$ เป็น 1 ทั้งหมด
- $D_{r,c}$ แทนคำตอบจาก subtask 4 นั่นคือ S มากสุดที่ทำให้ตารางในช่อง $(r-S+1, c-S+1, r, c)$ เป็น 1 ทั้งหมด



รูปที่ 1: รูปแสดงการเก็บค่า B, C, D โดยค่า B แสดงด้วยสีส้ม ค่า C แสดงด้วยสีเหลือง และค่า D แสดงด้วยสีแดง

จากรูปที่ 1 เราจะเห็นได้ว่า

- $B_{5,8} = 4$
- $C_{6,7} = 6$
- $D_{5,7} = 3$

เราจะสังเกตได้ว่า

- $B_{r,c} = B_{r-1,c} + 1$ หากช่อง (r, c) เป็น 1 และเป็น 0 หากช่อง (r, c) เป็น 0
- $C_{r,c} = C_{r,c-1} + 1$ หากช่อง (r, c) เป็น 1 และเป็น 0 หากช่อง (r, c) เป็น 0
- $D_{r,c} = min(D_{r-1,c-1}, C_{r,c-1}, B_{r-1,c}) + 1$ หากช่อง (r, c) เป็น 1 และเป็น 0 หากช่อง (r, c) เป็น 0

จากตัวอย่างจึงได้ว่า $D_{6,8} = min(D_{5,7}, C_{6,7}, B_{5,6}) + 1 = min(3, 6, 4) + 1 = 3 + 1 = 4$ ตรงตามต้องการ

เมื่อทำ Dynamic Programming ตามวิธีดังกล่าว ก็จะทำให้ทราบว่า S มากสุดของแต่ละช่อง (r_2, c_2) เป็นเท่าใด

Time complexity : $O(NM)$, Space complexity : $13NM$ bytes

Subtask 6 - $N, M \leq 6\,000$

จาก Subtask 5 นั้นมีการเก็บอาเรย์ถึง 3 อาเรย์ซึ่งเป็นการเปลืองต่อทรัพยากรอย่างมาก หากมีการปรับลดด้วยวิธีการใดก็ตามที่ลดได้พอสมควร จะทำให้ผ่านใน subtask นี้ หนึ่งในวิธีคือการเปลี่ยนชนิดข้อมูลอาเรย์ B, C, D จาก int (4 bytes) เป็น short (2 bytes) จะทำให้ memory usage จาก $3 \cdot 4 \cdot NM$ เหลือ $3 \cdot 2 \cdot NM$

Time complexity : $O(NM)$ **, Space complexity :** $7NM$ **bytes**

นอกจากนั้นยังมีอีกวิธีที่จะทำให้ได้ถึง Subtask 7 เลยด้วย คือการสังเกตว่าเราไม่จำเป็นจะต้องเก็บทั้ง B, C, D แต่เก็บเพียงแค่ D ก็เพียงพอแล้ว (สามารถใช้แทนที่ค่า B และ C ได้เลย)

Time complexity : $O(NM)$ **, Space complexity :** $5NM$ **bytes**

Subtask 7 - $N, M \leq 7\,000$

จาก Subtask 5 และ 6 อีกวิธีการหนึ่งที่มีประสิทธิภาพสูงมากในการลดหน่วยความจำคือ สังเกตว่าการหาค่าช่อง $B_{r,c}$, $C_{r,c}$ และ $D_{r,c}$ จะไปดึงค่าในแถวที่ $r-1$ และ r เท่านั้น นั่นแสดงว่าเราไม่จำเป็นต้องเก็บทั้งตาราง แต่สามารถทำ memory shift ได้ (เก็บเฉพาะแถวปัจจุบันกับแถวก่อนหน้า) และทำ memory shift บนตารางหลักด้วย

Time complexity : $O(NM)$ **, Space complexity :** $O(M)$

Alternate Solution

จริง ๆ แล้ว จะมีอีกวิธีที่แตกต่างออกไปจากแนวทางเฉลยเลย ซึ่งทางผู้จัดข้อนี้ก็ยังไม่แน่ใจว่าจะผ่าน Subtask สุดท้ายหรือไม่ คือสังเกตว่า เราสามารถมองว่าปัญหาข้อนี้คือการกัดกร่อนจริง ๆ ได้ กล่าวคือ นึกสภาพว่าช่อง 0 แทนไฟ และช่อง 1 แทนพื้นที่โล่ง จะเห็นได้ว่า หลังเวลาผ่านไป 2 สหัสวรรษ ผลลัพธ์ที่ได้จะเป็นแบบเดียวกับ การกระจายไฟไปยังช่องรอบ ๆ 8 ทิศ และหากทำซ้ำ จะได้ผลลัพธ์เมื่อเวลาผ่านไป $4, 6, 8, \ldots$ สหัสวรรษ ต่อมา เราสามารถนำรูปภาพของการกระจาย 1 สหัสวรรษ จากวิธีการโดยตรง (แทนค่า (r, c) ด้วย $\min(r, c), (r+1, c), (r, c+1), (r+1, c+1)$) แล้วค่อยทำการกระจายต่อด้วยวิธีการกระจายไฟ

วิธีการข้างต้นหากทำตรง ๆ จะได้เพียง Subtask 3 แต่สังเกตว่าเราไม่จำเป็นต้องค่อย ๆ กระจายไฟ แต่เราสามารถนับว่ามีกี่ช่องที่จะหายไป ณ เวลาขณะใดขณะหนึ่ง เราจึงสามารถทำ Breadth-First Search แบบ 8 ทิศได้ เพื่อนับระยะทางต่าง ๆ ซึ่งหากทำแบบปกติ ด้วยขนาดตารางและความช้าของ `std::queue` จะทำให้ผ่านเพียง Subtask 4 แต่หาก optimize วิธีการดำเนินการและการจัดเก็บ (เช่นใช้โครงสร้างอื่นแทน `std::queue`, ใช้ `std::bitset` เก็บตาราง) อาจทำให้ผ่านมากกว่านั้นได้

Time complexity : $O(N^2)$ **, Space complexity :** $O(N^2)$

Remark

บางคนสามารถทำวิธีเดียวกับวิธีที่ดีที่สุดได้ แต่เกิดปัญหาไม่ผ่าน เนื่องจากสาเหตุบางประการ เช่น

- มีการประกาศฟังก์ชันแยกนอก `main`
- ใช้ `scanf("%c")`
- ใช้ `getchar`
- ใช้ `std::cin`
- ใช้ `std::cout << std::endl`
- ทำการคำนวณซ้ำซ้อน (มี loop หรือ conditional มากเกินควร)
- เรียกช่องอาเรย์สองมิติแบบคอลัมน์ก่อนแถว (เช่น นิยาม $D_{i,j}$ เป็น $D_{j,i}$ ของเฉลยข้างต้น)

สาเหตุเหล่านี้ เป็น สาเหตุที่จะทำให้การทำงานของโปรแกรมช้าลง (บางสาเหตุส่งผลเพียงเล็กน้อย) โดยสาเหตุเหล่านี้จะไม่ค่อยได้เจอในการเขียนโปรแกรมแข่งขันทั่วไป แต่บางครั้งจะมีผลมากในบางการแข่งขัน ข้อนี้จึงมีการจำกัดเวลาอย่างเข้มงวด (ก่อนหน้านี้ทางผู้จัดเคยจัดเป็น 1 วินาทีเท่านั้น แต่โค้ดเฉลยส่งไม่ผ่านบางรอบ จึงขยายขอบเขตเวลามากขึ้น)

ดังนั้น การทำข้อนี้ให้ผ่านจึงไม่ได้ขึ้นอยู่กับประสิทธิภาพของแนวคิดเพียงอย่างเดียว แต่จะต้องเขียนโปรแกรมอย่างมีประสิทธิภาพด้วย

Solution Code:

```cpp
#include <bits/stdc++.h>
using namespace std;
int dp[2][7005];
int evp[7005];
int ans[7005];
char line[7005];
int main() {
  int n, m;
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    /**
     * Receive the input each line and store in the 'line'
     * array of char.
     *
     * Current line will be line[1], line[2], line[3], ..., line[m]
     */
    scanf("%s", line + 1);
    int cur = i % 2;
    int old = 1 - cur;
    for (int j = 1; j <= m; j++) {
      if (line[j] == '0')
        continue;
      /**
       * dp formula as described in the editorial.
       * D_(i),(j) will be dp[cur][j].
       * And D_(i-1),(j) will be dp[old][j].
       */
      dp[cur][j] = min({dp[old][j - 1], dp[cur][j - 1],
                        dp[old][j]}) + 1;
      evp[dp[cur][j]]++;
    }
    for (int j = 0; j <= m; j++)
      dp[old][j] = 0;
  }
  for (int i = min(n, m); i > 0; i--)
    ans[i] = ans[i + 1] + evp[i];
  for (int i = 1; i <= min(n, m); i++)
    printf("%d\n", ans[i]);
  return 0;
}
```

Lion Army by win11905

Subtask 1, 5, 9 - $N, M \leq 10$

เนื่องจากค่า N และ M มีค่าน้อยมาก เราสามารถแจกแจงวิธีการจัดกองกำลังทั้งหมด 2^N วิธีได้ด้วย recursion หรือ bitwise operation แล้วตรวจสอบว่าวิธีที่จัดไว้ตรงตาม M เงื่อนไขที่กำหนดหรือไม่ ให้ตอบจำนวนวิธีที่ผ่านการตรวจสอบเงื่อนไขทั้งหมด

Time complexity : $O(2^N M)$ **, Space complexity :** $O(N + M)$

Subtask 2, 6, 10 - $N, M \leq 1\,000$

เราจะพิจารณาโจทย์ข้อนี้เป็นปัญหากราฟ โดยมี N nodes (หมายเลข 1 ถึง N) แทนสิงโตแต่ละตัว ในตอนแรก จะยังไม่มี edge เชื่อมระหว่าง node ใด ๆ หลังการเพิ่มข้อมูลครั้งที่ i เราจะสร้างเส้นเชื่อมระหว่าง node a_i และ b_i พร้อมระบุค่า t_i ประกอบเส้นนั้นไว้ แล้วทำการนับจำนวนวิธีจัดกลุ่ม ดังนี้

สำหรับแต่ละ component (กลุ่ม node ที่ถูกเชื่อมกัน) ให้ใช้วิธี depth-first search เพื่อลงสี node 2 สี แทนการจัดกลุ่มสิงโตแต่ละตัวลงกลุ่ม A หรือ H โดย node แรกที่เราเริ่มต้นการ DFS จะสามารถลง A หรือ H ก็ได้ เมื่อเดินทางไปตาม edge ที่ระบุค่า $t_i = 0$ หรือ 1 ไว้ node อื่น ๆ ใน component จะถูกบังคับสีโดยอัตโนมัติ

ระหว่างการ DFS ถ้าจำเป็นต้องลงสีขัดแย้งกับที่เคยลงไว้แล้ว (เช่น สิงโตสามตัวเชื่อมกับเป็นสามเหลี่ยม ทุกตัวเกลียดกันหมด ทำให้ไม่สามารถจัดเป็นสองกลุ่มได้) แปลว่าเราไม่สามารถจัดกลุ่มสิงโตทั้งหมดได้ จำนวนวิธีสำหรับข้อมูลนั้น (และข้อมูลถัด ๆ ไป) จะเป็น 0 ทันที

หากไม่เกิดการขัดแย้งขึ้น สังเกตว่าแต่ละ component จะมีวิธีการลงสีเพียง 2 แบบเท่านั้น แบบแรกคือวิธีที่เราให้ node เริ่มต้นเป็น A แล้ว node อื่นถูกบังคับ ส่วนอีกแบบคือเริ่มเป็น H แทน ทำให้ node อื่น ๆ ถูกกลับสีทั้งหมด เราสามารถลงสีแต่ละ component ได้อิสระต่อกัน ดังนั้น จำนวนวิธีจะเท่ากับ 2^c เมื่อ c เท่ากับจำนวน component ของกราฟ (การหาค่า 2^c จะต้องใช้วิธีการลูปคูณใน modulo $10^9 + 7$ เท่านั้น ฟังก์ชัน pow ในภาษา C จะให้คำตอบเป็นตัวแปรประเภท double ที่มีตำแหน่งเลขนัยสำคัญไม่ละเอียดพอ)

ทั้งนี้ หากผู้เข้าแข่งขันไม่ทราบวิธีเก็บข้อมูล edge ที่มี weight t_i ประกอบใน adjacency list ผู้เข้าแข่งขันจะต้องเลือกจัดการกรณี $t_i = 0$ หรือ 1 อย่างใดอย่างหนึ่งเท่านั้น ทำให้ได้คะแนนใน subtask 2 หรือ 6 อย่างเดียว (สำหรับ $t_i = 0$ เป็นการตรวจสอบ bipartite graph ด้วยวิธีคล้าย ๆ ที่กล่าวไป ส่วน $t_i = 1$ เป็นการนับจำนวน component ด้วยวิธี flood fill ทั่วไป)

Time complexity : $O(M(N + M))$ **, Space complexity :** $O(N + M)$

Subtask 5 –8 - $t_i = 1$

พิจารณาโจทย์ข้อนี้เป็นปัญหากราฟตามที่กล่าวในหัวข้อก่อนหน้า เนื่องจาก $t_i = 1$ เสมอ จึงสามารถคำนวณคำตอบเป็น 2^c เมื่อ $c =$ จำนวน component ในกราฟได้ทันทีโดยไม่ต้อง depth-first search เพื่อตรวจสอบการขัดแย้ง

การนับจำนวน component ให้ได้อย่างรวดเร็วโดยไม่ต้อง depth-first search สามารถทำได้โดยใช้ Union-find Disjoint Set ที่มี path compression และ/หรือ union by rank

Time complexity : $O(N + M \cdot \alpha(N))$ **, Space complexity :** $O(N + M)$

Subtask 3, 7, 11 - $N, M \leq 80\,000$

Subtask นี้เป็นการผสมผสานระหว่างวิธีใน subtask 2, 6, 10 และ 5–8 โดยใช้ข้อสังเกตดังนี้: คำตอบจะเป็น 2^c เมื่อ c เท่ากับจำนวน component ในกราฟเสมอ จนกระทั่งเกิดการขัดแย้งของข้อมูล ทำให้คำตอบเป็น 0 เสมอไม่ว่าจะเพิ่มข้อมูลอื่นใดอีก

ให้รับข้อมูลการจัดกลุ่มทั้งหมดมาก่อน แล้วใช้วิธี Binary Search เพื่อหาว่าเราสามารถหาคำตอบตามปกติได้ถึงข้อมูลที่เท่าไหร่ก่อนที่จะเกิดการขัดแย้ง นั่นคือ ในตอนแรกให้ทดลองเพิ่มข้อมูลจนถึงข้อมูลที่ $m = \lceil \frac{1+M}{2} \rceil$ แล้วใช้ Depth-first search ทดลองลงสีก่อน ถ้าทำไม่ได้ แปลว่าจะต้องทดลองเพิ่มข้อมูลน้อยลง ดังนั้นจึงพิจารณาทดลองค่า m ภายในช่วง $[1, m-1]$ แทน แต่ถ้าทำได้ให้พิจารณาช่วง $[m, M]$ ด้วยวิธีการแบ่งครึ่งช่วงลงไปเรื่อย ๆ เช่นนี้ทำให้เราต้องทดลองไม่เกิน $\log_2 M$ ครั้ง (การเขียนโค้ดในส่วนนี้ควรระวังไม่ให้เกิด infinite loop)

เมื่อได้ข้อมูลดังกล่าวแล้ว จึงสร้าง Union-find Disjoint Set ที่มี path compression และ/หรือ union by rank ขึ้นมาแล้วไล่เพิ่มข้อมูล (เชื่อม node เข้าในเซตเดียวกัน) แล้วตอบทีละคำถามเป็น 2^c เสมอ จนกว่าจะถึงตำแหน่งที่คำนวณไว้ หลังจากนั้นจึงตอบ 0 เสมอ (อนึ่ง การคำนวน 2^c ไม่สามารถทำได้โดยการคิด 2^N ไว้ก่อนแล้วหาร 2 ทุกครั้งที่เกิดการรวม component ขึ้น จะต้อง precompute $2^0, 2^1, 2^2, \ldots, 2^N$ ไว้หรือใช้ binary exponentiation เท่านั้น)

Time complexity : $O((N + M) \log M)$ **, Space complexity :** $O(N + M)$

Subtask 4, 8, 12 - $N, M \leq 300\,000$

เนื่องจากวิธี Binary Search ตามที่อธิบายข้างต้นมี time constant สูงเกินอาจทำให้ติด Time Limit และไม่ได้คะแนนเต็ม ดังนั้นจึงต้องพยายาม optimize ให้ทำงานเร็วขึ้นหรือคิดวิธีที่ Time Complexity ดีขึ้น ดังนี้

พิจารณาโจทย์ข้อนี้เป็นปัญหากราฟที่มี $2N$ nodes โดยสิงโตแต่ละตัวจะมี 2 nodes แสดงถึงความเป็นไปได้ที่สิงโตตัวนั้นจะอยู่ในกลุ่ม A หรือกลุ่ม H เพื่อความสะดวก กำหนดให้สิงโตตัวที่ i ($1 \leq i \leq N$) มี node หมายเลข i และ $N+i$ ตามลำดับ

ทุกครั้งที่ได้รับข้อมูลใหม่เป็น a_i, b_i, t_i เราจะเชื่อมกราฟด้วย Union-find Disjoint Set พร้อมนับจำนวน component ดังนี้

- ถ้า $t_i = 1$ สร้างเส้นเชื่อม (a_i, b_i) และ $(a_i + N, b_i + N)$ แสดงถึงความเป็นไปได้ที่ถูกผูกมัดเกี่ยวข้องกัน คือ ถ้าสิงโตตัวหนึ่งอยู่กลุ่ม A อีกตัวหนึ่งจะต้องอยู่กลุ่ม A ด้วย (หรือกลุ่ม H เหมือนกัน)
- ถ้า $t_i = 0$ สร้างเส้นเชื่อม $(a_i, b_i + N)$ และ $(a_i + N, b_i)$ แสดงถึงการที่สิงโตทั้งสองตัวจะต้องมีสีตรงข้ามกันเสมอ

หากพบว่าสิงโตตัวที่ u ใด ๆ มี node u กับ $u + N$ อยู่ใน component เดียวกันแปลว่าเกิดข้อมูลขัดแย้งขึ้น เราไม่สามารถบังคับให้สิงโตตัวหนึ่งอยู่สองกลุ่มพร้อมกันได้ ดังนั้นคำตอบจึงเป็น 0 (หลังแต่ละข้อมูลเราไม่จำเป็นต้องไล่

ตรวจสอบ $u = 1, 2, \ldots, N$ ทั้งหมด ตรวจสอบแค่ $u = a_i$ และ $u = b_i$ ก็พอแล้ว เพราะ node อื่นไม่ได้มีการเปลี่ยนแปลงอะไรเกิดขึ้น) หากไม่มีการขัดแย้ง คำตอบจะเป็น $2^{c/2}$ เมื่อ c เท่ากับจำนวน component ในกราฟ (ต้องหารด้วย 2 เพราะเราสร้างกราฟเกินมาเท่าตัวหนึ่งพอดี)

Time complexity : $O(N + M \cdot \alpha(N))$ **, Space complexity :** $O(N + M)$

หมายเหตุ: $\alpha(N)$ หมายถึง *Inverse Ackermann Function*

Solution Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 600010;
const int M = 1e9+7;

int n, m, cc, par[N];
long long mpow[N];

int find(int x) { return par[x] = (par[x] == x ? x : find(par[x])); }

inline void unite(int a, int b) {
  a = find(a), b = find(b);
  if (a == b)
    return;
  par[a] = b, --cc;
}

int main() {
  iota(par, par + N, 0);
  mpow[0] = 1;
  for (int i = 1; i < N; i++)
    mpow[i] = mpow[i - 1] * 2ll % M;
  scanf("%d %d", &n, &m), cc = 2 * n;
  bool valid = true;
  for (int i = 1, t, a, b; i <= m; i++) {
    scanf("%d %d %d", &t, &a, &b);
    if (valid) {
      if (!t)
        unite(a, b + n), unite(b, a + n);
      else
        unite(a, b), unite(a + n, b + n);
      if (find(a) == find(a + n) || find(b) == find(b + n))
        valid = false;
    }
    printf("%lld\n", (valid ? mpow[cc / 2] : 0));
  }
  return 0;
}
```

Humanity Has Declined by AquaBlitz11

Subtask 1 - $N, K, Q \leq 100$

ในการตอบแต่ละ query เราสามารถพิจารณาลูปค่า $k = 1, 2, \ldots, K$ ทีละตัว เพื่อลูปหาว่าใน $A[l_i \ldots r_i]$ มีค่า k ปรากฏหรือไม่ หากไม่พบค่า k แปลว่าไม่มีขนมหวานสำหรับคนแคระหมายเลข k ดังนั้นจึงตอบ NO แต่หากพบทุกค่าปรากฏอยู่ ให้ตอบ YES

Time complexity : $O(NKQ)$ **, Space complexity :** $O(N + Q)$

Subtask 2 - $N \leq 1\,000$, $K \leq 100$, $Q \leq 200\,000$

ก่อนจะตอบแต่ละ query เราสามารถสร้าง $cnt[1 \ldots K]$ ขึ้นมาไว้บันทึกว่าเลข 1 ถึง K เคยปรากฏหรือไม่ โดยเริ่มต้นให้เซตทุกช่องเป็น 0 ก่อน แล้วเมื่อไล่ดูทีละตัวเลขในช่วง $A[l_i \ldots r_i]$ ก็ให้ทำการเพิ่มค่าใน array cnt ตามค่าที่อ่านได้ เมื่อทำจนเสร็จสิ้นแล้ว ให้ตรวจสอบว่า $cnt[1 \ldots K]$ ทุกช่องมีค่าอย่างน้อย 1 หรือไม่ ถ้าใช่แปลว่าเราได้ขนมหวานสำหรับคนแคระทุกคนแล้วจึงตอบ YES แต่ถ้ามีช่องใดที่เป็น 0 แปลว่าขนมหวานมีไม่เพียงพอ ให้ตอบ NO

Time complexity : $O(Q(N + K))$ **, Space complexity :** $O(N + K + Q)$

Subtask 3 - $N, Q \leq 200\,000$, $K \leq 100$

สำหรับข้อนี้ สังเกตว่าค่า $K \leq 100$ ดังนั้น ในแต่ละคำถาม หากเรามีวิธีตรวจสอบว่าค่า $k = 1, 2, \ldots, K$ ปรากฏใน $A[l_i \ldots r_i]$ หรือไม่ โดยไม่ต้องเสียเวลาลูปดูทีละช่อง ก็จะได้คะแนนใน subtask นี้

ก่อนเริ่มตอบคำถามทั้งหมด ให้คำนวณ Prefix Sum Array (หรือที่หลายคนเรียกว่า Quicksum) ไว้ทั้งหมด K array โดย array ที่ k จะทำให้เราสามารถตอบได้อย่างรวดเร็วว่าในช่วง $A[l_i \ldots r_i]$ ใด ๆ มีค่า k ปรากฏทั้งหมดกี่ตัว

- นิยามให้ $qs[k][j] =$ จำนวนครั้งที่ค่า k ปรากฏใน $A[1 \ldots j]$
- สังเกตว่า $qs[k][j] = qs[k][j-1] + 1$ เมื่อ $A[j] = k$ (เจอค่าที่ต้องการ ดังนั้นจึงนับว่าเจอเพิ่ม 1 ครั้ง) และ $qs[k][j] = qs[k][j-1] + 0$ เมื่อ $A[j] \neq k$
- ดังนั้น จำนวนครั้งที่ k ปรากฏใน $A[l_i \ldots r_i]$ เท่ากับ $qs[k][r_i] - qs[k][l_i - 1]$

เมื่อคำนวณไว้แล้วดังนี้ จะสามารถตอบแต่ละคำถามได้ในเวลา $O(K)$ โดยตรวจสอบว่าจำนวนครั้งที่ $k = 1, 2, \ldots, K$ ปรากฏมีค่าอย่างน้อย 1 สำหรับทุกค่า k หรือไม่

Time complexity : $O(K(N + Q))$ **, Space complexity :** $O(KN + Q)$

Subtask 4 - $N, K, Q \leq 200\,000$

การจะได้คะแนนเต็มในข้อนี้จำเป็นจะต้องใช้การสังเกต (observation) ดังนี้

- ถ้า query (l, r) ให้คำตอบเป็น YES แล้ว (l, r') ที่ $r' > r$ ย่อมให้คำตอบเป็น YES เช่นกัน เพราะว่าขนมหวานครบตั้งแต่ตัวที่ l ถึง r แล้ว ถึงเพิ่มขนมหวานมาก็ไม่ได้ทำให้คำตอบเปลี่ยน
- ถ้า query (l, r) ให้คำตอบเป็น NO แล้ว (l, r') ที่ $l \leq r' < r$ ย่อมให้คำตอบเป็น NO เช่นกัน เพราะว่าเดิมขนมหวานไม่พออยู่แล้ว หากจำกัดขอบเขตให้แคบกว่าเดิม ขนมหวานก็ยังคงไม่พออยู่

นั่นคือ สำหรับแต่ละค่า l เราสามารถหาค่า r ขั้นต่ำที่จำเป็นต่อการตอบ YES ในที่นี้จะเรียกค่าดังกล่าวว่า $rgt[l]$ (Minimum right boundary for l) ในกรณีที่ไม่มีค่า r ที่ทำให้คำตอบเป็น YES ได้ให้ถือว่า $rgt[l] = N + 1$

สังเกตว่า $rgt[l] \leq rgt[l + 1]$ เสมอ เพราะเมื่อขอบซ้ายขยับแคบเข้ามา ขอบขวาขั้นต่ำอาจจะต้องขยายมากขึ้นเพื่อทดแทนขนมหวานที่หายไป เราสามารถเขียนโปรแกรมตามข้อสังเกตนี้ได้เลย

- เราจะมีตัวแปร l และ r โดยในตอนแรก ให้เริ่มที่ $l = r = 1$ ทั้งคู่
- สร้าง array $cnt[1 \ldots K]$ ขึ้นมาเพื่อบันทึกว่าในช่วง $A[l \ldots r]$ ที่เราพิจารณาอยู่มีค่า 1 ถึง K ปรากฏตัวละกี่ครั้ง (ตอนแรกทุกช่องจะเป็น 0 ยกเว้นตำแหน่งช่องที่ $A[1]$)
- สร้างตัวแปร sat (satisfied) ที่นับว่าตอนนี้ใน cnt มีกี่ช่องที่มีค่าอย่างน้อย 1 (ตอนแรก $sat = 1$ เว้นแต่ว่า $A[1] > K$ จะได้ $sat = 0$)
- ทุกครั้งที่ขยับค่า l หรือ r เพิ่มขึ้นให้ปรับค่า cnt ตาม $A[r]$ ที่เจอเพิ่มเติม/ลดหายไป อีกทั้งยังปรับ sat ตามด้วย นั่นคือถ้า $cnt[k]$ เพิ่งเพิ่มเป็น 1 แปลว่า sat เพิ่มขึ้นอีก 1 แต่ถ้ามีค่าลดลงเป็น 0 แปลว่า sat หายไป 1
- สำหรับทุก l เราจะพยายามขยับค่า r ไปเรื่อย ๆ จนกว่า $sat = K$ จึงบันทึกคำตอบว่า $rgt[l] = r$, ขยับขอบซ้ายเพิ่มมาเป็น $l + 1$, ขยับค่า r เท่าที่จำเป็น, ขยับขอบซ้ายเพิ่มมาเป็น $l + 2$, ขยับค่า r เท่าที่จำเป็น ฯลฯ เช่นนี้ไปเรื่อย ๆ จน $l = N$ (หากไม่สามารถทำให้ $sat = K$ ได้ แปลว่าสำหรับค่า l ตั้งแต่ค่านั้นเป็นต้นไป ต้องตอบ NO เสมอ เพราะฉะนั้นบันทึกเป็น $rgt[l] = N + 1$ ได้เลย)

เมื่อได้ค่า $rgt[l]$ ทุกช่องแล้ว การตอบคำถามก็สามารถทำได้อย่างง่ายได้ หาก $r_i < rgt[l_i]$ (ขอบขวาไม่ถึงขั้นต่ำที่เราคำนวณไว้) ให้ตอบ NO แต่หาก $r_i \geq rgt[l_i]$ ให้ตอบ YES

Time complexity : $O(N + K + Q)$ **, Space complexity :** $O(N + K + Q)$

Solution Code:

```
#include <bits/stdc++.h>
using namespace std;

const int N = 200010;
const int K = 200010;

int A[N], cnt[K], rgt[N], sat;

int main() {
  int n, k, q;
  scanf("%d %d %d", &n, &k, &q);
  for (int i = 1; i <= n; ++i)
    scanf("%d", &A[i]);

  int j = 0;
  for (int i = 1; i <= n; ++i) {
    while (sat < k && j < n) {
      ++j;
      if (A[j] <= k && ++cnt[A[j]] == 1)
        ++sat;
    }
    rgt[i] = (sat == k ? j : n + 1);
    if (A[i] <= k && --cnt[A[i]] == 0)
      --sat;
  }

  while (q--) {
    int l, r;
    scanf("%d %d", &l, &r);
    if (r >= rgt[l])
      printf("YES\n");
    else
      printf("NO\n");
  }

  return 0;
}
```

Traveling Pooh by my99n

Subtask 1 - $N, Q \leq 5\,000$

ในแต่ละคำถาม สามารถทดลองสลับประตู แล้วให้ปีศาจเปิดประตูไปเรื่อยๆทีละประตู จนเจอ หรือ ไม่เจอ กับโนบิตะได้ ซึ่งจะใช้เวลาในการเปิดประตูทั้งหมดอย่างมาก N ครั้ง ต่อหนึ่ง คำถาม ในการทดลองเปิดประตูไปเรื่อย ๆ สามารถใช้วิธีการ depth-first search บนประตูทุก ๆ บาน

Time complexity : $O(N \cdot Q)$ **, Space complexity :** $O(N)$

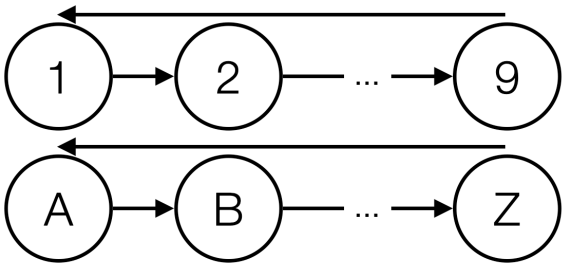
Subtask 2 - $N, Q \leq 200\,000, a_i = b_i$

สังเกตว่าจะไม่มีการสลับประตูเลย ดังนั้นเราสามารถหาไว้ล่วงหน้าได้ ว่า ประตูไหนสามารถเปิดไปถึงกันได้บ้างโดยการทดลองเปิดประตูไปเรื่อย ๆ (depth-first search ประตูทั้งหมด) ไว้ล่วงหน้า โดยในขั้นตอนของการทำ dfs (depth-first search) จะต้องเก็บข้อมูลว่าประตูบานไหนอยู่ในกลุ่มที่เท่าไหร่เอาไว้ด้วย หลังจากนั้นในแต่ละคำถามจะสามารถตอบได้ทันทีว่าประตูสองบานใด ๆ สามารถเปิดไปถึงกันได้หรือไม่ (อยู่ใน component เดียวกัน) ใน $O(1)$ time

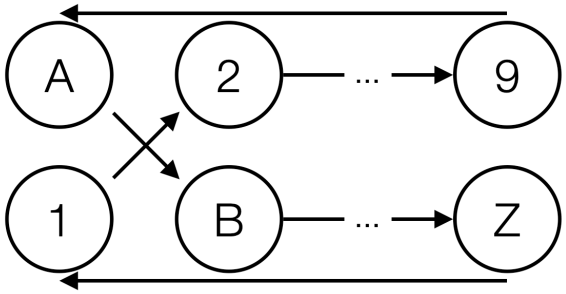
Time complexity : $O(N + Q)$ **, Space complexity :** $O(N)$

Subtask 3 - $N, Q \leq 200\,000$

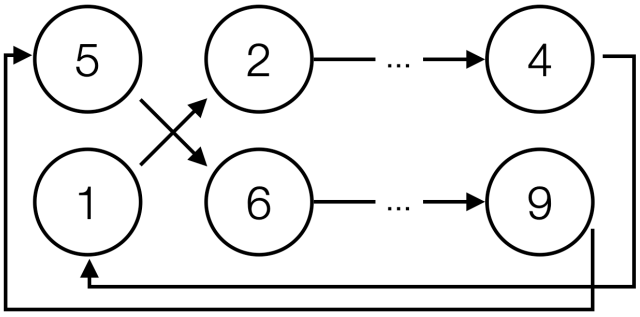
เราสามารถนำวิธีหาคำตอบ Subtask ที่ 2 มาปรับได้ หลังจากที่เรารู้แล้วว่า หากไม่มีการสลับประตู ประตูใดจะเปิดไปถึงกันได้บ้าง และเรารู้ว่า การเปิดประตูจะทำให้เกิดการวนเป็นวงกลมเสมอ เช่นตามรูปที่ 2



รูปที่ 2: ลำดับของการเปิดประตูจนกลับมาที่เดิม (วนเป็นวงกลม)



รูปที่ 3: หลังจากสลับประตูจากคนละวงของลำดับ (1 และ A)



รูปที่ 4: หลังจากสลับประตูจากภายในวงเดียวกัน (1 และ 5)

รูปทั้งสามรูปข้างต้นประกอบด้วยวงกลมซึ่งแทนประตู และลูกศรซึ่งแสดงว่าเมื่อเปิด ประตูแล้วจะไปโผล่ที่ประตูไหน ยกตัวอย่างเช่น เมื่อเปิดประตูที่ 1 จะไปโผล่ที่ประตูที่อยู่ตำแหน่งที่ 1 ซึ่งก็คือประตูที่ 2 เป็นต้น

เมื่อสลับหมายเลขบนประตูสองบานใด ๆ จะทำให้เกิดการเปลี่ยนแปลงได้แค่ 2 แบบเท่านั้นคือสังเกตว่าหากสลับประตูที่ 1 กับ A จากรูปที่ 2 ซึ่งอยู่คนละ component ลำดับจะกลายเป็นตามรูปที่ 3 เนื่องจากตำแหน่งของประตูไม่ได้มีการ เปลี่ยนแปลง ประตูตำแหน่งที่ A ยังคงเป็นประตูเดิมถึงแม้ว่าประตูที่มีหมายเลข A จะถูกย้ายไปที่ตำแหน่งที่ 1 แล้วก็ตาม ในทำนองเดียวกัน ประตูที่ตำแหน่งที่ 1 ยังคงเป็นประตูหมายเลข 2 ถึงแม้ว่าประตูที่มีหมายเลข 1 จะถูกย้ายไปอยู่ที่ตำแหน่ง A

หากสลับประตูที่ 1 กับ 5 ซึ่งอยู่ใน component เดียวกัน ลำดับจะกลายเป็นตามรูปที่ 4 ซึ่งสามารถพิจารณาในทำนองเดียวกันกับกรณีแรก ดังนั้นจึงสามารถหาว่าประตูสองประตูใดๆจะไปถึงกันได้หรือไม่ (อยู่ใน component เดียวกันหรือไม่) หลังจากสลับเพียง 1 ครั้งได้ ใน $O(1)$ โดยการแยกพิจารณาด้วย 2 เงื่อนไขข้างต้น คือ ประตูที่สลับอยู่ใน component เดียวกันหรือไม่ เมื่อมีคำถาม Q คำถามจึงใช้เวลา $O(Q)$

หมายเหตุ: component หมายถึง ส่วนของ graph ที่เชื่อมถึงกัน

Time complexity : $O(N+Q)$ **, Space complexity :** $O(N)$

Solution Code:

```
#include <bits/stdc++.h>
using namespace std;

const int N = 2e5 + 10;
int n, q, s[N], cycle[N], order[N];
bool visited[N];

int cnt;
void dfs(int u, int t) {
  if (visited[u])
    return;
  visited[u] = true, cycle[u] = t, order[u] = ++cnt;
  dfs(s[u], t);
}

bool same(int a, int b) { return (cycle[a] == cycle[b]); }

bool query() {
  int s, e, a, b;
  scanf("%d %d %d %d", &s, &e, &a, &b);
  if (!same(s, e))
    return ((same(a, s) and same(b, e)) or (same(a, e) and
             same(b, s)));
  else {
    if (!same(a, s) or !same(b, s))
      return true;
    if (order[s] > order[e])
      swap(s, e);
    return !((order[s] <= order[a] and order[a] < order[e]) xor
             (order[s] <= order[b] and order[b] < order[e]));
  }
  cout << '\n';
}

int main() {
  int c = 0;
  scanf("%d %d", &n, &q);
  for (int i = 1; i <= n; i++)
    scanf("%d", &s[i]);
  for (int i = 1; i <= n; i++)
    if (!visited[i])
      cnt = 0, dfs(i, ++c);
  while (q--)
    cout << (query() ? "DEAD\n" : "SURVIVE\n");
}
```

Claw Machine by PeppaPigHS

กำหนดให้ K เป็นค่า k จากคำถามที่มากที่สุด

Subtask 1 - $N, Q \leq 10$

เนื่องจาก N และ Q มีค่าไม่เกิน 10 จึงสามารถใช้ Bruteforce ในการลองวิธีการหยิบตุ๊กตาทุกรูปแบบได้ และนับเฉพาะวิธีการหยิบที่ผลรวมความน่ารักของตุ๊กตาเท่ากับ k พอดีภายในช่วง l, r จัดว่าเป็นการไล่สับเซตทั้งหมดที่อาจมีมากถึง 2^N สับเซต

Time complexity : $O(Q \cdot 2^N)$ **หรือ** $O(Q \cdot N \cdot 2^N)$ **, Space complexity :** $O(N + Q)$

Subtask 2 - $N, Q \leq 1\,000$

สังเกตว่าจาก Subtask 1 จะมีการคิดซ้ำซ้อนเป็นจำนวนมาก ดังนั้นเราจึงสามารถใช้ Dynamic Programming เข้ามาช่วยเพื่อลดเวลาการทำงานได้ ดังนั้นสำหรับแต่ละคำถาม กำหนดให้ $dp(i, j)$ คือจำนวนวิธีการหยิบตุ๊กตาที่ทำให้ผลรวมความน่ารักมีค่าเท่ากับ j เมื่อพิจารณาตุ๊กตาตัวที่ l ถึง i เท่านั้น ดังนั้นมี Transition ดังนี้

$$dp(i, j) = dp(i-1, j) + dp(i-1, j-A_i)$$

ซึ่งมาจาก 2 กรณีคือไม่เลือกตุ๊กตาตัวที่ i ทำให้ค่าผลรวมความน่ารักเท่าเดิม กับเลือกตุ๊กตาตัวที่ i ทำให้ค่าผลรวมความน่ารักเพิ่มขึ้น A_i เมื่อทำการคำนวณค่าของ $dp(i, j)$ จนครบทุกค่าแล้ว จะได้ว่าคำตอบของคำถามนี้คือ $dp(r, k)$

Time complexity : $O(Q \cdot N \cdot K)$ **, Space complexity :** $O(N \cdot K)$

Subtask 3 - $A_i = 1$

สำหรับ Subtask นี้ จะสังเกตได้ว่าในคำถามแต่ละคำถาม จะหยิบตุ๊กตา k ตัวเสมอ เพราะตุ๊กตาทุกตัวมีค่าความน่ารักเป็น 1 และเนื่องจากค่าความน่ารักของตุ๊กตาทุกตัวมีค่าเท่ากัน จึงสามารถเลือกตุ๊กตา k ตัวใดๆ ก็ได้ที่อยู่ภายในช่วง l ถึง r กล่าวคือเป็นการเลือกตุ๊กตา k ตัว จากตุ๊กตาทั้งหมด $r - l + 1$ ตัวนั่นเอง

ดังนั้นกำหนดให้ $dp(i, j)$ คือจำนวนวิธีการหยิบตุ๊กตาที่ทำให้ผลรวมความน่ารักมีค่าเท่ากับ j เมื่อพิจารณาตุ๊กตา i ตัว สังเกตว่านิยามมีความคล้ายกับ Subtask 2 แต่ต่างกันตรงที่ลำดับของตุ๊กตาไม่มีความสำคัญ ส่วน Transition จะเป็นดังนี้

$$dp(i, j) = dp(i-1, j) + dp(i-1, j-1)$$

เราจะคำนวณค่า $dp(i, j)$ เพียงครั้งเดียวเท่านั้น และสำหรับคำตอบของแต่ละคำถาม จะเท่ากับ $dp(r - l + 1, k)$

Time complexity : $O(N \cdot K + Q)$ **, Space complexity :** $O(N \cdot K)$

Subtask 4 - $N, Q \leq 100\,000$

จาก Solution ของ Subtask 2 แทนที่เราจะคำนวณค่าของ $dp(i, j)$ ใหม่ทุกครั้งที่มีการถามคำถาม เราสามารถคำนวณเพียงรอบเดียว กำหนดให้ $dp(i, j)$ คือจำนวนวิธีการหยิบตุ๊กตาที่ทำให้ผลรวมความน่ารักมีค่าเท่ากับ j เมื่อพิจารณาตุ๊กตาตัวที่ 1 ถึง i เท่านั้น สำหรับ Transition จะเหมือนกับ Subtask 2

สำหรับการตอบคำถาม กำหนดให้ $f(i)$ คือจำนวนวิธีในการหยิบตุ๊กตาแล้วผลรวมความน่ารักเป็น i เมื่อพิจารณาตุ๊กตาในช่วง $[l, r]$ เท่านั้น เราสามารถหา Transition ของ $f(i)$ ได้ดังนี้

$$f(i) = dp(r, i) - \sum_{j=1}^{i} f(i - j) \cdot dp(l - 1, j)$$

ซึ่งมาจากกรณีที่เราพิจารณาหยิบตุ๊กตาในช่วง $[1, r]$ แล้วลบกรณีที่หยิบตุ๊กตาในช่วง $[1, l)$ ออก หลังจากคำนวณค่า $f(i)$ เรียบร้อยแล้ว คำตอบของคำถามนี้จะเป็น $f(k)$

Time complexity : $O(N \cdot K + Q \cdot K^2)$ **, Space complexity :** $O(N \cdot K)$

Solution Code:

```
#include <bits/stdc++.h>

#define long long long

using namespace std;

const int N = 1e5 + 5;
const int M = 1e9 + 7;

int n, m, A[N];
long dp[N][105], f[105];

int main() {
  scanf("%d %d", &n, &m);
  dp[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    scanf("%d", A + i);
    for (int j = 0; j <= 100; j++) {
      dp[i][j] += dp[i - 1][j];
      if (j >= A[i])
        dp[i][j] += dp[i - 1][j - A[i]];
      dp[i][j] %= M;
    }
  }
  for (int t = 1, a, b, c; t <= m; t++) {
    scanf("%d %d %d", &a, &b, &c);
    for (int i = 0; i <= c; i++) {
      f[i] = dp[b][i];
      for (int j = 1; j <= i; j++) {
        f[i] -= f[i - j] * dp[a - 1][j] % M;
        f[i] = (f[i] + M) % M;
      }
    }
    printf("%lld\n", f[c]);
  }

  return 0;
}
```

Magic Pooh by Autoratch

หมีพูห์จะวิ่งจากเมือง 1 ไปยังเมือง N ผ่านด่านต่างๆ โดยต้องวิ่งให้ใช้เวลาน้อยที่สุด และอาจจะใช้คอปเตอร์เร่งความเร็วได้ สามารถแปลได้ว่า โจทย์ต้องการให้หา Single Source Shortest Path บนกราฟ N โหนด M เส้นเชื่อม และอาจจะเลือกเปลี่ยน น้ำหนัก (ระยะทาง) ของเส้นเชื่อมได้หนึ่งเส้นเป็นค่าที่โจทย์กำหนดให้ (ระยะเวลาที่คอปเตอร์ใช้) หรืออาจจะไม่เปลี่ยนก็ได้

Subtask 1 - $N, M \leq 10, K = 10^9$

เนื่องจาก ค่า $K = 10^9$ จึงทำให้ การใช้คอปเตอร์เร่งความเร็วจะไม่ทำให้ได้เวลาเร็วกว่าการวิ่งผ่านด่านปกติจึงสามารถทำ SSSP ตรง ๆ ได้เลย และ ค่า N และ M มีค่าน้อยกว่าเท่ากับ 10 จึงสามารถทำ Floyd-Warshall ได้

Time complexity : $O(N^3)$ **, Space complexity :** $O(N^2)$

Subtask 2 - $N, M \leq 10$

ค่า K ไม่เท่ากับ 10^9 ทำให้ อาจจะจำเป็นที่จะต้องใช้คอปเตอร์เพื่อที่จะได้วิ่งได้เวลาที่ดีที่สุด จึงไม่สามารถทำ Floyd-Warshall ได้ จึงต้องทำ Bruteforce Recursive แทน

Time complexity : $O(N!)$ **, Space complexity :** $O(N!)$

Subtask 3 - $N, M \leq 1\,000$

เนื่องจากค่า N และ M น้อยกว่าเท่ากับ $1\,000$ จึงสามารถทำ Bellman-Ford ในการหาระยะที่สั้นที่สุดจากโหนด 1 และโหนด N มายังทุกโหนด และใช้วิธีเดียวกับใน Subtask 6 ได้

Time complexity : $O(N \cdot M)$ **, Space complexity :** $O(N + M)$

Subtask 4 - $N, M \leq 100\,000, w_i = 1$

เนื่องจากค่า $w_i = 1$ จึงทำให้ทุกเส้นเชื่อมมีน้ำหนักเป็น 1 จึงสามารถทำ Breadth First Search จากโหนด 1 ไปโหนด N ได้

Time complexity : $O(N + M)$ **, Space complexity :** $O(N + M)$

Subtask 5 - $N, M \leq 100\,000, K = 10^9$

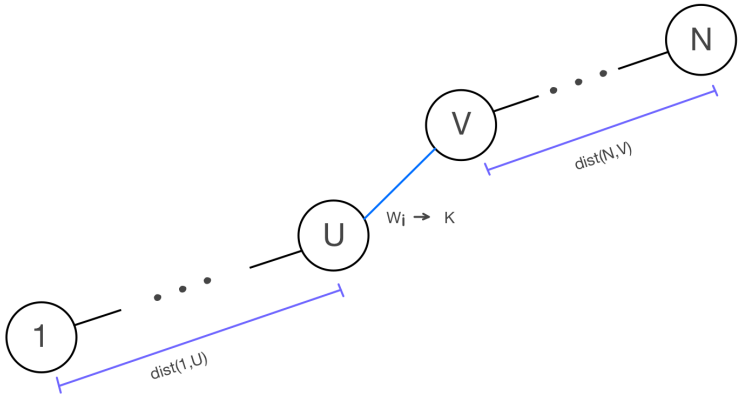
เนื่องจากค่า $K = 10^9$ จึงทำให้ การใช้คอปเตอร์เร่งความเร็วจะไม่ทำให้ได้เวลาเร็วกว่าการวิ่งผ่านด่านปกติจึงสามารถทำ SSSP Dijkstra ตรง ๆ ได้เลย

Time complexity : $O(N + M \log N)$ **, Space complexity :** $O(N + M)$

Subtask 6 - $N, M \leq 100\,000$

เนื่องจากโจทย์ให้เปลี่ยนได้แค่หนึ่งเส้นเชื่อมจึงสามารถไล่ดูทุกเส้นเชื่อมว่า หากเปลี่ยนความยาว(ใช้คอปเตอร์)แล้วจะได้ระยะทางรวมเท่าไหร่ ให้เส้นเชื่อมที่จะเปลี่ยนความยาวเป็นเส้นเชื่อมที่เชื่อมโหนด U และโหนด V

ระยะเวลาที่ใช้ในการวิ่งจากโหนด 1 ไปโหนด N โดยใช้คอปเตอร์บนเส้นเชื่อม (U, V) สามารถคำณวนได้ดังนี้



รูปที่ 5: $min(dist(1, U) + dist(N, V) + K, dist(1, V) + dist(N, U) + K)$

โดยให้ $dist(X, Y)$ แทนระยะเวลาที่สั้นที่สุดที่ใช้ในการวิ่งจากโหนด X ไปโหนด Y

ดังนั้นจะสามารถหาค่า $dist()$ ในสมการด้านบนได้โดยการหาระยะสั้นสุดจากโหนด 1 ไปยังทุกโหนด และจากโหนด N ไปยังทุกโหนด ซึ่งสามารถทำได้โดยการ Dijkstra จากโหนด 1 ไปยังทุกโหนดรอบนึง และจากโหนด N ไปยังทุกโหนดอีกรอบนึง

Time complexity : $O(N + M \log N)$ **, Space complexity :** $O(N + M)$

Solution Code:

```
#include <bits/stdc++.h>
#define pii pair<long long, int>
using namespace std;

const int N = 1e5 + 1;

int n, m, k, t;
vector<pair<int, int>> adj[N];
vector<tuple<int, int, int>> ed;
priority_queue<pii, vector<pii>, greater<pii>> q;
long long dist[2][N], ans = LLONG_MAX;
bool visited[2][N];

void run(int s, int t) {
  fill(dist[t] + 1, dist[t] + n + 1, LLONG_MAX);
  dist[t][s] = 0;
  q.emplace(0, s);
  while (!q.empty()) {
    int u = q.top().second; q.pop();
    if (visited[t][u]) continue;
    visited[t][u] = true;
    for (auto[w, v] : adj[u])
      if (dist[t][u] + w < dist[t][v])
        dist[t][v] = dist[t][u] + w, q.emplace(dist[t][v], v);
  }
}

int main() {
  scanf("%d %d %d %d", &n, &m, &k, &t);
  for (int i = 0; i < m; i++) {
    int a, b, d; scanf("%d %d %d", &a, &b, &d);
    adj[a].emplace_back(d, b);
    adj[b].emplace_back(d, a);
    ed.emplace_back(a, b, d);
  }
  run(1, 0), run(n, 1);
  for (auto[a, b, d] : ed) {
    if (dist[0][a] != LLONG_MAX and dist[1][b] != LLONG_MAX)
      ans = min(ans, dist[0][a] + dist[1][b] + min(d, k));
    if (dist[1][a] != LLONG_MAX and dist[0][b] != LLONG_MAX)
      ans = min(ans, dist[1][a] + dist[0][b] + min(d, k));
  }
  if (ans > t)
    puts("No Honey TT");
  else
    printf("Happy Winnie the Pooh :3\n%lld\n", ans);
}
```