PreTOI 2017

Editorial

1A - Gui Pachinko by WhippedCream

สังเกตว่าลำดับการเพิ่มเข้าและดูดออกของลูกบอลในเครื่องนั้นจะเป็นลำดับที่ถูกกำหนดมาแล้ว และการดูดลูกบอลออกจากเครื่อง K ลูก คือการ undo operation การปล่อยลูกบอล K ครั้งล่าสุด กล่าวอีกนัยหนึ่งคือ จำนวนลูกบอลที่เหลือในเครื่องนี้จะเพียงพอสำหรับการตอบว่าลูกบอลสุดท้ายที่ถูกปล่อยลงไปจะไปอยู่ที่ช่องไหน และเพียงพอสำหรับการตอบผลรวมของเลขช่องที่มีลูกบอลอยู่

ตอนนี้สิ่งที่เราต้องหาคือช่องสุดท้ายของการตกของลูกบอลแต่ละลูกในเครื่อง โดยลำดับนี้สามารถหาด้วยวิธีการตรงๆ คือไล่ใส่ลูกบอลไปทีละตัวแล้ว update ว่าช่องนั้นเต็มเรื่อยๆ แล้วเลือกท่อที่มีสมบัติตามโจทย์กำหนด วิธีนี้เราจะใช้เวลา $O(N^2)$ สำหรับแต่ละลูก ซึ่งก็จะเป็น $O(N^3)$ โดยรวม

ในการปรับปรุง runtime เราสังเกตว่า ลำดับที่ลูกบอลตกไปนั้น จะเป็นลำดับเดียวกับ post-order traversal ใน tree โดย visit ลูกของ node หนึ่งตามค่าที่มากที่สุดใน subtree นั้นจากน้อยไปมาก ในการทำเช่นนั้นถ้าสำหรับทุกๆ node เราหาค่าที่มากที่สุดใน subtree ใดๆใหม่ทุกครั้ง เราจะใช้เวลาเป็น $O(N^2)$ จากการ visit N node โดยที่แต่ละ node หาตัวที่มากที่สุดอย่างมากประมาณ N ตัว

เพื่อที่จะทำให้ runtime ดีขึ้น ให้เราสังเกตว่า การหาค่าที่มากที่สุดสำหรับ subtree ของแต่ละ child ของแต่ละ node จะเป็นปัญหาย่อยที่ซ้อนทับกัน ดังนั้นเราไม่จำเป็นที่จะต้องคำนวณค่าแต่ละค่าใหม่ทุกครั้ง สิ่งที่เราจะทำ นั่นคือ dynamic programming บนแต่ละ node เพื่อหา node ที่มีค่ามากที่สุดใน subtree ของมัน จากสมการ $M(u) = \max\{\max\{M(v)\}, u\}$ สำหรับทุกๆ v ที่เป็น child ของ node u ซึ่งจะสามารถคำนวณค่า dynamic programming นี้ได้ใน $O(N)$

เมื่อเราคำนวณตารางนี้เสร็จแล้ว ให้เรา sort ลำดับของลูกใน post order traversal ตามค่าของ M(v) จากน้อยไปมาก แล้วทำ post order traversal ตามลำดับนั้นๆ เราจะเก็บลำดับการ visit ของแต่ละ node ไว้ใน global array (หรือ vector, stack ก็ได้) การทำเช่นนี้จะทำได้ใน $O(N \log N)$

เมื่อได้ post order traversal (เรียกว่า A[i]) ปัญหาก็จบแล้ว แค่เก็บจำนวนลูกบอลที่อยู่ในเครื่องในปัจจุบัน (สมมติว่าเป็น k) จะได้ว่า บอลจะอยู่ในช่อง A[1], A[2], A[3], ..., A[k] (สมมติให้ลูกแรกอยู่ที่ index 1) และสำหรับ operation 1 เราก็

ตอบค่า A[k] ได้เลย สำหรับ operation 3 เราต้องตอบค่า A[1]+A[2]+A[3]+...+A[k] ซึ่งสามารถตอบได้ใน O(1) โดยการ quicksum หรือสร้างอีก array B[i] ที่มีสมบัติว่า A[1] = B[1] และ B[i] = B[i-1] + A[i] ซึ่งจะทำให้ B[k] = A[1] + A[2] + A[3] + ... + A[k] ดังนั้น algorithm ของเราจะทำงานใน O(N log N + Q) เมื่อ Q คือจำนวน operation

1B - 1 by zoomswk

สังเกตว่าตัวเลขบนลูกบอลสีดำจะไม่สามารถเปลี่ยนแปลงได้เลย อย่าเพิ่งเลือกระบายสีตั้งแต่เริ่มเกม ลองพิจารณาตั้งแต่ลูกบอลลูกซ้ายสุดไปขวาสุดทีละอัน

เมื่อพิจารณาลูกบอล i เราจะตรวจสอบว่า มีเซตของลูกบอลสีดำก่อนหน้า ที่มีผลคูณของตัวเลขเท่ากับ Ai หรือไม่

ถ้าไม่มี เราก็ควรระบายลูกบอลลูกนี้เป็นสีดำ เพราะแน่นอนว่าให้มันเป็นสีขาวไปก็ไม่เกิดประโยชน์ (ยกเว้นกรณีที่ลูกบอลลูกนี้เป็นเลข 1) แต่ในทางกลับกัน ถ้าเราให้มันเป็นสีดำ อาจจะช่วยทำให้ลูกบอลทางขวามือสามารถกลายเป็น 1 ได้ก็ได้

ถ้ามี เราก็ควรให้ลูกบอลลูกนี้เป็นสีขาว เพราะถ้ามีเซตของลูกบอลสีดำด้านซ้ายที่คูณกันได้เท่ากับจำนวนบนลูกบอลนี้ เวลาลูกบอลทางขวามือต้องการค่าบนลูกบอลลูกนี้ เราก็สามารถใช้ลูกบอลทั้งกลุ่มที่ว่าแทนลูกบอลที่เราพิจารณาอยู่ได้

แล้วจะตรวจสอบยังไงว่ามีเซตของลูกบอลสีดำก่อนหน้าที่มีผลคูณของตัวเลขเท่ากับ Ai หรือเปล่า

สำหรับ Subtask ที่ 1 และ 4 เนื่องจาก N ≤ 16 เราสามารถ brute-force การเลือกลูกบอลทุกรูปแบบได้
(เลือกหรือไม่เลือกลูกบอลก่อนหน้าแต่ละลูก) จะใช้เวลาทำงาน O(N*2^N)

สำหรับ Subtask อื่นๆ จากข้อกำหนดที่ว่าไม่มีจำนวนเฉพาะนอกจาก 2, 3, และ 5 หารมันลงตัว
แสดงว่าเราสามารถเขียนจำนวน Ai ในรูปของ $2^x 3^y 5^z$ ได้
เราสามารถแก้ปัญหานี้ได้ด้วย Dynamic Programming 3 มิติ (หรือเขียนใน 1 มิติก็ได้) คล้าย ๆ กับปัญหา subset sum หรือ knapsack

หากลองรันโปรแกรมเพื่อตรวจสอบจำนวนในช่วง [1, 100 000] จะพบว่ามีทั้งสิ้น X = 313 จำนวน
หรือคำนวณแบบคร่าวๆ มากๆ ก็ได้ จะได้ว่ามีไม่เกิน X = $(\log 100000)^3$ / (log 2 log 3 log 5) = 1245 จำนวน (มาจาก $\log_2 100000 \log_3 100000 \log_5 100000$)

หากรัน DP บนตัวเลขเหล่านี้ ก็เหมือนจะได้ว่ามันทำงานใน O(XN) ซึ่งก็ดูจะทันเวลาอยู่แล้ว

แต่ที่จริงหากพิจารณาให้ดีกว่านี้ ก็จะพบว่า อัลกอริทึมจะทำงานในเพียงแค่ $O(N + X^2)$ ซึ่งผ่านได้สบายๆ

ที่ได้ $O(N + X^2)$ ก็เป็นเพราะมีตัวเลขในข้อมูลนำเข้าได้แค่ X แบบ เมื่อเราเจอตัวเลขซ้ำ (แน่นอนว่าเจอเยอะมาก สำหรับ N สูง ๆ) ก็ไม่ต้องรัน DP ใหม่ ดังนั้นจะรันทั้งสิ้นอย่างมาก X ครั้ง (ระหว่างการไล่) ครั้งละ O(X) กลายเป็น $O(X^2)$ และการไล่ลูกบอลทุกลูกใช้เวลา O(N) รวมกันเป็น $O(N + X^2)$

หากเราตั้ง ความแข็งแกร่ง ของรองเท้าที่จะใช้ไว้ ก็จะสามารถรู้ได้ว่าระยะทางที่สั้นที่สุดระหว่างเมือง 1 กับเมือง N เป็นเท่าไหร่ โดยใช้ Dijkstra’s ที่ทำงานใน O(N + M log N) โดยจะไม่เดินผ่านถนนที่อันตรายกว่าความแข็งแกร่งที่เราตั้งไว้เด็ดขาด หลังจากนั้นเราก็ตรวจสอบว่า ระยะทางที่ได้ น้อยกว่าหรือเท่ากับ T หรือเปล่า
ก็จะรู้ได้ว่า ความแข็งแกร่งเท่านี้ พอสำหรับการเดินทางภายใน T วินาทีไหม

ถ้าทดลองกับรองเท้าทุกรุ่น แล้วหาราคาที่ถูกที่สุด ก็จะได้อัลกอริทึมที่ทำงานใน O(K (N + M log N))
ซึ่งพอสำหรับแก้ปัญหาย่อยที่ 1, 2, และ 3

แต่สังเกตได้ว่า ยิ่งรองเท้าแข็งแกร่งมาก ระยะเวลาที่ใช้ในการเดินทางจาก 1 ไป N ก็จะน้อยลงตาม (หรืออาจจะเท่าเดิม แต่จะไม่มากขึ้น) ดังนั้นเราสามารถ binary search ความแข็งแกร่งที่จำเป็น แล้วรัน Dijkstra’s ตามที่อธิบายไว้ แล้วหารองเท้าที่ถูกที่สุดที่มีความแข็งแกร่งอย่างน้อยเท่ากับความแข็งแกร่งที่ได้ใน O(K) ก็จะได้อัลกอริทึมที่ทำงานใน O(log(100,000) (N + M log N) + K)

อีกทางหนึ่งก็คือ sort รองเท้าจากราคาน้อยไปราคามาก ใน O(K log K) แน่นอนว่าถ้าราคาแพงแล้วความแข็งแกร่งน้อยกว่าก็ไม่ต้องพิจารณาได้เลย ทีนี้เราก็จะได้ลำดับของรองเท้าเรียงจากราคาน้อยไปมาก และแข็งแกร่งน้อยไปมากเช่นกัน เราสามารถ binary search บนลำดับนี้ แล้วรัน Dijkstra’s เช่นเคย จะได้อัลกอริทึมที่ทำงานใน O(K log K + (N + M log N) log K)

2A - Seven Gems by JETHO

ปัญหาข้อนี้ เกี่ยวกับการทำ breadth-first search (BFS) บน STATE ที่มีลักษณะพิเศษ ดังนี้

1. จำนวนของอัญมณีที่ถืออยู่ มีผลต่อตัวเลือกในการเลือกเดินต่อ
2. รูปแบบ Set ของอัญมณีทั้ง 7 ในแต่ละตำแหน่งบนแผนที่ ที่ถืออยู่ มีผลต่อตัวเลือกในการเลือกเดินต่อ
3. เลขบนนาฬิกา มีผลต่อตัวเลือกในการเลือกเดินต่อ

จาก ลักษณะพิเศษข้างต้น เราสามารถจำกัดรูปแบบ Set ของ STATE ทั้งหมดได้เป็น STATE = POS × TIME × GEMS เมื่อ POS = Set ของตำแหน่งในตาราง, TIME = Set ของเวลาบนเข็มนาฬิกา และ GEMS = Set ของการถืออัญมณี ซึ่งจะมีทั้งหมด 2^7 แบบ

Subtask 1 : $1 \le N , M \le 20$ และคำตอบมีค่าไม่เกิน 100

สามารถผ่านได้ด้วย update STATE ทุก STATE ทั้งหมด 100 รอบ เผื่อหาคำตอบ โดย STATE = POS × TIME × GEMS

Subtask 2 : $1 \le N , M \le 100$ และ คำตอบมีค่าไม่เกิน 1000

สามารถผ่านได้ด้วย BFS โดยเช็ค STATE = POS × TIME × GEMS และไม่ทำการไปยัง STATE ซ้ำและ มี Limit ที่เดินที่ 1,000 ก้าวเดิน

Subtask 3 :

สามารถผ่านได้ด้วย การทำ BFS + Implementation ที่ดี

2B - Joddad Valley by RayaBurong25.1

ข้อนี้ เห็นได้ชัดว่าเป็น Single Source Shortest Path แต่ความยากของโจทย์นี้คือความพลิกแพลงภายในตัวโจทย์ โดย subtask แต่ละอันพยายามออกแบบให้คิดได้เป็นขั้นเป็นตอน

Subtask 1: เป็นการถามเพียงหนึ่งครั้งและ $c = 0$ สามารถใช้ Dijkstra หรือ algorithm SSSP อื่น ๆ ได้ตรง ๆ โดยเก็บ state ความเร็ว และปรับ state เวลา ด้วยระยะทาง/ความเร็ว (ฟังก์ชัน compare ใน priority queue อาจเทียบ state เวลา)

Subtask 2: เมื่อ $c \neq 0$ ก็เพียงแค่ ปรับความเร็ว ด้วย $c \times$ระยะทาง อาจเก็บเป็นอีก state ก็ได้ แต่ไม่จำเป็น เพราะแต่ละคำถาม เป็นปัญหาที่ independent จากกัน

Subtask 3: ใน subtask นี้ เพิ่มอีกสอง Operation คือ add และ delete ซึ่งถ้าหาก represent กราฟโดยใช้ Adjacency List ก็สามารถทำได้ง่าย คือการเติม vector ที่ว่างเปล่า (add) และการลบทั้ง vector (delete)

Subtask 4: พอจำนวน edge เพิ่มมากขึ้น เป็นการ hint ว่า จำนวน edge ที่เยอะมาก ต้องไม่ส่งผลต่อความเร็วของอัลกอริทึม เมื่อจำนวน edge มากเกินกว่า $n(n-1)/2$ แล้ว แสดงว่าต้องมี multiple edge หรือ edge loop ในกราฟ เมื่อพิจารณาดูแล้ว จะได้ว่า

1. ในกรณีของ multiple edge ใช้เฉพาะ edge ที่มีความยาวน้อยที่สุด edge ที่เหลือสามารถลบทิ้งได้
2. ในกรณีของ edge loop ไม่จำเป็นต้องใส่ในกราฟเพราะไม่มีประโยชน์

ดังนั้น วิธีแก้ที่ง่ายที่สุดคือการ represent graph ด้วย Adjacency Matrix ทำให้การ add คือการหา min length ของแต่ละช่อง (ตอนแรก สำหรับทุก j AdjMat[a][j] และ AdjMat[j][a] จะเป็น INF) และการ delete คือการ reset ค่ากลับเป็น infinity

อันที่จริงสามารถ represent กราฟโดยใช้ Adjacency Matrix และใช้การ optimize เพื่อให้เร็วขึ้นได้ (คนแต่งโจทย์ใช้ทั้ง map และ vector เก็บ edge ในกราฟ โดยการ add/delete ให้ทำบน map และการ travel ให้ copy ทุก edge จาก map ไป vector แล้วใช้ vector ระหว่างทำ Dijkstra ซึ่งอาจจะเป็นการทำให้ยุ่งยากโดยใช่เหตุก็ได้ 5555)

ทุกอย่างในโจทย์ข้อนี้ควรใช้ double ในการคำนวณ และอย่าลืมว่าตอนตอบให้ตอบทศนิยม 6 ตำแหน่งเท่านั้น

2C - Ultimate Werewolf by PeaTT~

ในข้อนี้ เราสามารถมองปัญหาเป็น undirected graph ที่มี P โหนดและ P เส้นได้ โดยกราฟนี้จะมีเส้นเชื่อมจากระหว่างโหนด u กับ v ถ้าคนที่ u โหวต v (เป็นไปได้ที่จะมีเส้นซ้ำสองเส้นระหว่างสองโหนดนี้ ถ้าต่างคนต่างโหวตหากัน แต่จะไม่มีลูปหาตัวเอง) และเพื่อความเข้าใจง่าย แทนที่เราจะหาจำนวน Villager ที่น้อยที่สุด เราจะหาจำนวน Werewolf ที่มากที่สุดแทน ก็คือ เราจะหา subset ของโหนดที่ใหญ่ที่สุดที่จะเป็น Werewolf ที่ไม่มีสองโหนดที่เลือกมามีเส้นเชื่อมหากันโดยตรง เราจะได้ว่าปัญหานี้มันก็คือ **Maximum Independent Set (MIS)** บนกราฟพิเศษนี้นั่นเอง ในที่นี้เราจะสนใจแค่ขนาดของ MIS ของกราฟนั้น

ข้อสังเกตแรกคือ แต่ละ component ของกราฟนั้น เราสามารถคิดแยกกันได้เลย แล้วก็ค่อยนำคำตอบมาบวกรวมกัน ดังนั้นตั้งแต่นี้ไป เราจะสมมติว่ากราฟนั้นเชื่อมต่อกันหมด

ข้อสังเกตถัดไปคือ ถ้ากราฟมันเป็น**ต้นไม้** นั่นคือ มี P โหนดและ P-1 เส้น เราสามารถขนาดของหา MIS ได้โดยใช้ dynamic programming (DP) บนต้นไม้นี้ โดยเราจะเลือกโหนดนึง r มาเป็นรากแล้วก็ทำ depth-first search (DFS) จากมันไป เราก็จะได้ rooted tree มาต้นหนึ่ง ระหว่างนั้นเราจะพิจารณาทีละ subtree กับแต่ละโหนด u (เราไม่สนใจ parent ของ u ในปัญหาย่อยนี้) โดยเรามีทางเลือกสองทาง คือ

1. จะให้คนที่ u เป็น Werewolf (เลือกโหนด u ใส่ subset) จะได้ว่าลูกแต่ละตัวของ u นั้นต้องเป็น Villager โดยความสัมพันธ์ที่ได้คือ **dp[u][werewolf] := 1 + sum v in children(u) {dp[v][villager]}** (ถ้า u ไม่มีลูก จะได้ว่าส่วนของ sum นั้นจะเป็น 0)
2. จะให้คนที่ u เป็น Villager (ไม่เลือกโหนด u ใส่ subset) จะได้ว่าลูกแต่ละตัวของ u นั้นจะเป็นอะไรก็ได้ จะได้ความสัมพันธ์ **dp[u][villager] := sum v in children(u) {max(dp[v][villager], dp[v][werewolf])}**

ขนาดของ MIS ในต้นไม้นี้ก็คือ **max(dp[r][villager], dp[r][werewolf])** แต่ในโจทย์ข้อนี้ กราฟมันไม่ใช่ต้นไม้ซะทีเดียว แต่มันมีเส้นเชื่อมเพิ่มมาอีกเส้นหนึ่ง สมมติให้เป็นเส้นระหว่าง p กับ q (ย้ำว่ามันอาจจะซ้ำกับเส้นบนต้นไม้เดิมได้) โดยวิธีการหาเส้นนี้ในกราฟนั้นทำได้สองวิธี คือ

1. จะทำ DFS แล้วหา back edge ใน DFS tree
2. ใช้ disjoint-set union โดยเส้น (p, q) นั้นจะไม่ทำให้จำนวน component ลดลง

ส่วนวิธีหา MIS บนกราฟ **“ต้นไม้+1”** นั้น เราสามารถดัดแปลงวิธีเดิมข้างต้นสำหรับต้นไม้ได้ โดยเราจะลองเลือกหรือไม่เลือก p กับ q ทุก ๆ แบบทั้งหมด 3 รูปแบบ คือ

1. ไม่เลือก p และ q ทั้งคู่ (p, q: villager)
2. เลือก p แต่ไม่เลือก q (p: werewolf, q: villager)
3. เลือก q แต่ไม่เลือก p (p: villager, q: werewolf)

สำหรับในแต่ละแบบนี้ เราจะกำหนดรูปแบบตามที่ตั้งไว้ แล้วก็ทำ DFS / DP ตามปกติ ยกเว้นเวลาคำนวณ u = p หรือ u = q โดยเราอาจจะห้ามไม่ให้ใช้รูปแบบตรงข้ามนั้นด้วยการตั้งค่าคำตอบที่ผิดให้เป็นลบมาก ๆ (เช่น -(P+1) หรือ -10^9 ก็ได้) สุดท้ายเราก็เก็บคำตอบจากแต่ละรูปแบบ ได้เป็นคำตอบของกราฟ (component) นี้ไป

สรุปของอีกวิธีคร่าว ๆ : สังเกตว่ากราฟพิเศษนี้ (สมมติว่ามีแค่ 1 component) จะมีวง (cycle) อยู่วงนึงเสมอ และส่วนที่เหลือจะเป็นกิ่งไม้ที่ยื่นมาจากแต่ละโหนดในวงนั้น วิธีแก้ปัญหาคือ เราจะคำนวณหาวงบนกราฟนี้ คำนวณคำตอบ DP จากแต่ละกิ่งต้นไม้ที่ยื่นออกมา (เอาเส้นในวงออกให้หมดแล้วให้แต่ละโหนดบนวงเป็นรากต้นไม้ของมัน) แล้วก็ใช้ DP อีกแบบคำนวณคำตอบบนวงนั้น (กำหนดรูปแบบของโหนดหนึ่งแล้วก็คลี่วงออกเป็นเส้น)