Lösningsförslag KATT 2019

Longest increasing pub-sequence

Ett generellt problem i såna här uppgifter är att flyttalsjämförelser kan leda till svårupptäckta buggar. Till exempel om man skriver **if(a == b)** kan man missa if-satsen även om flyttalen a och b borde vara samma, eftersom de har avrundats lite olika. Istället kan man kolla om $|a - b| < \epsilon$ för något jättelitet tal ϵ (typ 10^{-10}). Men det är lätt att glömma bort någonstans, eller att sätta epsilon för litet eller för stort, så om det är möjligt är det bäst att bara undvika flyttal. Antagligen det bästa sättet att undvika flyttal i det här problemet är att bara kolla på kvadraten av avstånd, för då blir allt heltal. Men heltalen kan bli väldigt stora, $2 \cdot 10^{18}$, så använd 64-bitars heltal (**long long** i C++) för att undvika overflow.

Delpoäng

För 10 poäng finns det bara tre fall: $n = 2$, likbent triangel, och icke-likbent triangel. Man skulle kunna tro att liksidiga trianglar var ett fall, men eftersom punkterna måste ha heltalskoordinater kan det inte inträffa. För att få 10 poäng till borde vilken brute-force lösning som helst klara sig. För att få 50 poäng kan man använda DP. Först, förberäkna alla parvis avstånd $\text{dist}(i, j)$ mellan punkter. Låt $dp(i, j)$ vara maximala antalet besök som slutar med att vi kommer till punkt i från punkt j. Funktionen kan beräknas med följande rekursion:

$$dp(i, j) = 1 + \max_{\text{dist(j,k)<dist(i,j)}} dp(j, k)$$

Nu blir svaret max av alla $dp(i, j)$. Om man hittar maximum genom att loopa igenom alla k får man en DP med $O(n^2)$ tillstånd, men som tar $O(n^3)$ tid att räkna ut, vilket räcker för 50 poäng.

Full poäng

För att få full poäng gäller det att på något sätt få bort en faktor n från den kubiska DP:n, genom att hitta maximum på något snabbare sätt. Det finns fler

än ett sätt att göra det på. Tänk om när vi behövde räkna ut $dp(i,j)$ redan hade räknat ut alla $dp(j,k)$ där $\text{dist}(i,j) < \text{dist}(j,k)$. Tänk om vi dessutom höll koll på $dp_{max}(j)$, maximala värdet på $dp(j,k)$ för alla k som vi behandlat hittills. Då skulle vi kunna få ut $dp(i,j)$ i konstant tid och DP:n skulle bara ta $O(n^2)$ tid att räkna ut. Det här går att åstadkomma genom att lösa DP:n botten upp:

1. Börja med att sortera alla ordnade par (i,j) av punkter med avseende på $\text{dist}(i,j)$.
2. Gå igenom paren i den ordningen, håll koll på $dp_{max}(i)$ för alla i, och låt $dp(i,j) = 1 + dp_{max}(j)$.
3. Det här funkar nästan, enda problemet är att det kan finnas par som har exakt samma avstånd, vilket får till följd att vi uppdaterar $dp_{max}(j)$ för tidigt och får felaktiga svar. För att lösa det får man vänta med att uppdatera $dp_{max}(j)$ till det dyker upp något avstånd som är olikt det förra avståndet.

Nu tar själva DP:n bara $O(n^2)$ tid, det dyra steget blir sorteringen av punkterna som tar $O(n^2 \log(n))$, och det räcker för 100 poäng.

Xorcisten

Delpoäng

För 13 poäng kan man testa alla x för varje query i $O(nqa_i)$. För 50 poäng gäller det att hitta x i $O(n \log(a_i))$ per query, vilket kan göras genom att hitta den högsta biten av x, sedan den näst högsta osv. Det blir några fall som dyker upp, men låt oss skippa detaljerna. Nästa nivå för 88 poäng var lite grann till för fullpoängslösningar som var för långsamma av någon anledning. Till exempel finns det en segmentträdslösning som tar $O(q \log(n) \log(a_i))$.

Full poäng

För att få full poäng måste vi ändra perspektiv. Låt oss kalla mängden av alla x sådana att

$$a_1 \oplus x \leq \cdots \leq a_n \oplus x$$

för S_x. Så det vi vill få fram är $\min(S_x)$, vilket kanske inte vore omöjligt om vi kunde hålla koll på S_x. Men S_x kan ju bli väldigt stor, så det går inte att ha den i form av en lista med element eller liknande. Tänk om S_x hade en speciell struktur som lät oss hålla koll på den på ett kompakt sätt. Det visar sig stämma. För ett index i som uppfyller $1 \leq i \leq n-1$, låt S_i betäckna mängden av alla x så att

$$a_i \oplus x \leq a_{i+1} \oplus x.$$

För att ett tal x ska vara i S_x krävs att alla dessa olikheter uppfylls, så

$$S_x = \bigcap_{i=1}^{n-1} S_i$$

dvs S_x är snittet av alla S_i (jag använder lite mängdteorinotation här, hoppas det är OK). Låt oss undersöka hur S_i ser ut. Ifall $a_i = a_{i+1}$ så är S_i alla möjliga tal. Annars kommer a_i och a_{i+1} att skilja sig vid någon bit, och låt oss säga att den högsta biten där de skiljer sig är m. Det visar sig nu att S_i består av alla tal vars bitar kan vara vad som helst, utom den m:te biten som måste vara antingen 0 eller 1 beroende på om $a_i < a_{i+1}$ eller inte. S_i kan representeras på ett kompakt sätt: den består av en massa begränsningar på formen "bit nummer j kan vara 0", "bit nummer j får inte vara 1", osv. Såna mängder kan representeras av en bitsträng b av längd $2\log(a_i)$ (60 i vårt fall), där talet b_{2j} avgör om bit nummer j hos talen i S_i kan vara noll, och b_{2j+1} avgör om bit nummer j kan vara ett. I vårt fall kommer alla b_j att vara 1 utom b_{2m} eller b_{2m+1}. Det som är bra med den här representationen är att snittet av två såna här mängder kommer ha en motsvarande sträng som är bitwise AND av de två strängarna! Det här låter oss hitta S_x. Notera att 60 bitars bitsträngar får plats i ett 64-bitars heltal vilket låter oss utföra bitwise AND väldigt snabbt. Ett sätt att hålla koll på AND av alla S_i är att använda ett segmentträd. Men det behövs egentligen inget segmentträd, allt vi behöver hålla reda på är hur många nollor som finns på varje position j över alla S_i. Om svaret är noll så är motsvarande positition i bitwise AND 1, annars så är det 0. När vi väl har hittat bitsträngen som motsvarar S_x kan vi hitta $\min(S_x)$ på ett girigt sätt, och det märks också om svaret är -1 (om någon bit varken får vara noll eller ett). Komplexiteten blir $O(q\log(a_i))$.

Myrkolonin

Delpoäng

För 18 poäng kan man för varje query loopa igenom alla punkter i rektangeln och göra en DFS från de punkter som tillhör grafen för att räkna antalet komponenter. För ytterligare 17 poäng kan man faktiskt göra exakt samma sak, fast loopa igenom alla n graf-punkter istället.

För 68 poäng krävs en hel del mer. Först, den del av grafen som hamnar inuti en rektangel kommer bestå av ett antal komponenter med träd, dvs en skog. En skog med n noder och m kanter har alltid $n-m$ komponenter. Så det vi vill hitta är antalet noder och kanter i rektangeln. För att hitta antalet noder kan vi tänka oss ett $A \times B$-rutnät M där det finns en etta i de rutor där det finns en nod, och en nolla annars. Antalet noder i en viss rektangel blir nu summan av

alla element i motsvarande rektangel i M. Att hitta summan av tal i rektanglar av rutnät kan göras med *2D-prefixsummor*. Här kommer en kort introduktion till 2D-prefixsummor:

Låt $S_{i,j}$ vara summan av alla tal i rektangeln $(0, 0, i, j)$, dvs.

$$S_{i,j} = \sum_{a \leq i, b \leq j} M_{a,b}.$$

Vi kan hitta $S_{i,j}$ för alla (i, j) på ett lite DP-liknande sätt, genom att låta

$$S_{i,j} = M_{i,j} + S_{i-1,j} + S_{i,j-1} - S_{i-1,j-1}.$$

När vi väl har hittat alla $S_{i,j}$ i $O(AB)$ kan vi också hitta summan av alla element i godtyckliga rektanglar (a, b, c, d) i konstant tid! Summan blir nämligen

$$S_{c,d} - S_{c,b-1} - S_{a-1,d} + S_{a-1,b-1}.$$

Så, nu kan vi hitta antalet noder i alla rektanglar. För att hitta antalet kanter kan vi använda 2D-prefixsummor igen, t.ex. genom att lägga en etta i rutan till höger om en horisontell kant och nedanför en vertikal. Då kommer prefixsumman ge oss nästan rätt svar, förutom att vi dessutom får med en massa kanter som korsar rektangeln längs den vänstra och övre sidan. För att subtrahera antalet korsande kanter kan man använda prefixsummor eller lägga alla kanter i en massa listor beroende på x,y koordinat och binärsöka i dem. Man kanske till och med hinner göra det i $O(A + B)$. Det här räcker i alla fall för 68 poäng.

Full poäng

För att få full poäng finns en ful och en fin lösning. Den fula lösningen är att i princip göra samma sak som för 68 poäng. Problemet är att nu kan vi inte använda 2D-prefixsummor för att rutnätet är för stort, men problemet är ekvivalent med att räkna antalet punkter i en massa rektanglar. Ni som var med i coconut-gruppen på lägret kanske minns att Simon förklarade hur man löser det här med persistenta segmentträd. Det är ett sätt att göra det på, det finns också en offlinelösning som går ut på att dela in alla rektangelqueries i fyra små rektanglar med hörn i $(0, 0)$ och sortera dem. I vilket fall som helst blir det ganska jobbigt att implementera, och vi måste fortfarande räkna de korsande kanterna (men det kan göras på samma sätt som i 68-poängslösningen).

Som tur är finns det en finare lösning. Det visar sig att vi kan hitta antalet komponenter i en rektangel enbart genom att kolla på de kanter som korsar rektangeln! Det räcker inte att bara räkna antalet korsande kanter, för varje komponent kan korsa rektangeln på flera ställen. Men vi kanske kan göra vissa kanter "speciella" på ett sätt så att varje komponent får exakt en speciell kant. Det kan göras på följande sätt. Rota trädet i en godtycklig nod, och rikta

kanterna så att de pekar bort från roten (det här kan göras med en DFS). Varje komponent kommer nu ha exakt en kant som korsar rektangeln och pekar inåt mot rektangeln! Det enda undantaget är om roten tillhör en komponent, då kommer den komponenten inte ha några inåtpekande kanter. Så allt vi behöver göra är att räkna antalet inåtpekande kanter, och addera ett om roten är i rektangeln. Att räkna antalet inåtpekande kanter kan göras på samma sätt som när vi räknade korsande kanter innan, lägg dem i en lista beroende på koordinat och åt vilket håll kanten pekar, och binärsök i de här listorna. Komplexiteten blir $O(n + q \log(n))$.