Lösningar till KATT 2014

Av problemkommittén.

Strumpmatchning 2

När man tävlar inom problemlösning så är det viktigt att kunna se likheter och skillnader mellan snarlika problem. Här kunde en lösningsteknik från “Strumpmatchning” (från träningspasset) på finalen i Linköping på sätt och vis återanvändas, men en extra finess behövde läggas till för full poäng.

Naiv lösning (40p)

Från lösningsidén till “Strumpmatchning” är det bekant att vi kan hitta det maximala antalet par som kan paras ihop (givet en differens D) i körtid proportionell mot N * log N, där N är antalet strumpor. För att hitta det minimala D som ger tillräckligt många strumppar så kan man loopa över D tills tillräckligt många strumpor blir matchade. Detta ger en körtid proportionell mot (max_D * N + N * log N).

Otillräcklig lösning (70p)

En observation är att om vi har de sorterade strumpstorlekarna [1, 5, 15, 17] så kan vi begränsa oss till att kolla D-värdena [4, 10, 2] - alltså de intilliggande differenserna. Eftersom det kommer finnas N-1 D-värden att testa och varje tar N tid att kolla så blir körtiden kvadratisk (storleksordningen N^2).

Binärsökning (full poäng)

När D kan bli jättestort så blir den naiva lösningen för långsam. Om man istället binärsöker över D så får man en körtid proportionell mot (N * log N + log (max_D) * N). Notera vilken enorm skillnad det är mellan lg_2 (10^15) och 10^15.

Ytterligare en optimering kan göras genom att kombinera idén från binärsökning och 70-poängslösningen. Körtiden blir då proportionell med (N * log N). Det är däremot helt onödigt att krångla till det här! Det är nästan ingen skillnad på log (max_D) och log (max_N), du kan knappa in värdena på miniräknarn och se för dig själv.

Brädspelet

Uppgiften gick ut på att avgöra om en position i spelet var vinnande eller förlorande för personen som börjar. Det var möjligt lösa detta problem på två helt olika sätt, vilket kommer förklaras nedan, men först kan det vara bra att göra några allmänna observationer.

Observation 1:
Storleken på brädet kommer minska för varje omgång. Spelet kan alltså inte hålla på i oändlig tid.
Observation 2:
Båda spelarna följer exakt samma regler. Man behöver alltså inte ta hänsyn till vem av *A* och *B* som är på tur, utan det räcker med att bestämma om en given storlek på brädet är vinnande eller förlorande.

För att bestämma om en position är vinnande eller förlorande så gäller följande vid optimalt spel:

- En position är vinnande om spelaren kan försätta motståndaren i en förlorande position
- En position är förlorande om spelaren bara kan ge motståndaren en vinnande position

Spelet gjordes lite krångligare av att det är motståndaren som väljer vilken av de båda bräd-delarna som ska spelas vidare med. Detta gör att man är i en vinnande position om och endast om man kan dela brädet på ett sätt så att motståndaren får välja mellan två förlorande positioner.

Dynamisk programmering

Indatagränserna på det här problemet var så pass små att samtliga positioner i spelet kan lagras i minnet. Totalt finns det N*M positioner, antalet möjliga storlekar på brädet, vilket är ett ganska litet antal eftersom N och M kunde vara högst 100.

Det är alltså möjligt gå igenom alla positioner och därifrån testa alla drag som spelaren kan göra för att se om det är möjligt att försätta motståndaren i en förlorande position. Ett naturligt sätt att göra detta på är att definiera en rekursiv funktion som givet en brädstorlek avgör om det är en vinnande eller förlorande position. För att undvika att räkna ut samma värden flera gånger är det dock viktigt att spara resultaten när de räknas ut, och direkt returnera det sparade värdet om funktionen anropas med samma värde igen.

Matematisk lösning

Den andra lösningen är lättare att implementera och väldigt mycket snabbare, men mycket mindre uppenbar. Det visar sig att om den största tvåpotensen som delar N är lika med den största tvåpotensen som delar M så är det en förlorande position, annars en vinnande.

Ett trick som man kan använda för att hitta sådana “enkla” lösningar är att kolla på många fall där N och M är små och försöka se vad som är gemensamt för alla vinnande respektive förlorande positioner. Så i det här fallet kan vi till exempel göra en tabell över resultatet för alla startbräden med höjd och bredd högst 8, där ‘V’ betyder att en position är vinnande och ‘F’ att den är förlorande:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	F	V	F	V	F	V	F	V
2	V	F	V	V	V	F	V	V
3	F	V	F	V	F	V	F	V
4	V	V	V	F	V	V	V	V
5	F	V	F	V	F	V	F	V
6	V	F	V	V	V	F	V	V
7	F	V	F	V	F	V	F	V
8	V	V	V	V	V	V	V	F

Studerar man tabellen noga så kan man se att när höjden är udda så ger varannan bredd vinst och varannan förlust. När höjden är 2 verkar det som att var fjärde bredd ger vinst, och detsamma gäller när höjden är 6. När höjden är 4 är det statistiska underlaget inte särskilt stort i ovanstående tabell, men det verkar i alla fall som en rimlig hypotes att endast var åttonde bredd ger förlust.

Det verkar alltså som att om höjden M=u*2^k, där u är udda och k ett heltal, så är var 2^(k+1):a bredd en förlorande position, och resten vinnande. Mer specifikt så ser vi att personen som börjar förlorar om och endast om N=t*2^k, där t är udda (och k är samma som för M).

Om detta antagande stämmer så är det enkelt att se vem som vinner, vi måste bara räkna ut den största tvåpotens 2^m som delar M och den högsta tvåpotens 2^n som delar N. Då vinner Berit om och endast om m=n.

Men hur bevisar vi att vår hypotes stämmer? Rent tekniskt sett så kollar inte Kattis om vi kan bevisa det, man får inte mindre poäng om man inte har ett bevis. Men det är ofta en bra idé att ha åtminstone en vag idé om varför hypotesen skulle gälla.

Förklaring 1

För att resonera sig fram till den matematiska lösningen så kan man göra följande 2 observationer:

Observation 1:
Antag att vi vill dela en sida med udda storlek i två delar. Oavsett hur vi gör så kommer motståndaren att få välja mellan en sida med udda storlek och en med jämn storlek.
Observation 2:
Om vi vill dela en sida med jämn storlek så är det alltid möjligt att tvinga motståndaren till att ta en med udda storlek (vi kan till exempel dela upp sida *N* i delarna *1* och *N-1*).

För att hitta lösningen så gäller det att kolla på vad *N* och *M* blir modulo *2*. Vi kan dela upp positionerna i följande 3 typer:
Typ 1: 2 udda sidor
Typ 2: 1 udda sida, 1 jämn sida
Typ 3: 2 jämna sidor

Eftersom vi vet att position 1x1 är förlorande så kan vi gissa på att alla positioner av typ 1 är förlorande. Om vi är i en sådan position så har vi antingen förlorat direkt (1x1) eller så kan motståndaren alltid ta sig till en position av typ 2 (observation 1). Från en position av typ 2 är det dock alltid möjligt att tvinga motståndaren till en position där båda sidorna är udda (observation 2). Detta visar att positioner av typ 1 är förlorande och positioner av typ 2 är vinnande. Frågan är vad som gäller för positioner av typ 3?

Faktum är att det går att dra samma resonemang men modulo 4 istället. Eftersom vi vet att båda sidorna är jämna så kan de vara lika med 0 eller 2 mod 4. Om en sida är lika med 2 mod 4 så måste den antingen delas upp i två udda delar (i vilket fall motståndaren hamnar i en vinnande position) eller så kan motståndaren alltid välja mellan en sida av längd 0 eller 2 mod 4. Om en sida har längd 0 mod 4 så är det dock alltid möjligt att tvinga motståndaren att välja en sida av längd 2 mod 4. På samma sätt som ovan följer att en position med två sidor som är lika med 2 mod 4 är förlorande, och att en position där ena sidan är 0 mod 4 och den andra är 2 mod 4 är vinnande.

Om båda sidorna är 0 mod 4 så kan man istället göra samma resonemang mod 8, eller mer generellt till 2^k. Lösningen blir att loopa över tal 2^k för k = 0,1,2… och hitta det minsta k för vilket N != 0 mod 2^k eller M != 0 mod 2^k. Om N == M mod 2^k för detta k så är det en förlorande position, annars en vinnande. Alternativt formulerat så är en position vinnande om den högsta tvåpotens som delar N inte är samma som den högsta tvåpotens som delar M.

Alternativ (mer matematisk) förklaring

Om hypotesen ska bevisas formellt bör vi använda oss av ett induktionsbevis, det vill säga att vi bevisar att hypotesen stämmer för positionen (M, N) givet att den stämmer för alla positioner (M', N') sådana att M'<=M, N'<=N och (M, N) != (M', N').

Då uppkommer två fall, dels fallet då den största tvåpotens som delar M är lika med den största tvåpotens som delar N, och dels fallet då tvåpotenserna är olika. Låt för enkelhetens skull v(x) beteckna den största tvåpotens som delar x.
I det första fallet vill vi visa att oavsett vilket drag spelaren med motorsågen gör så kan den andra spelaren välja en bit så att den resulterande positionen är vinnande. Så anta att M=u*2^k och att N=t*2^k, där u och t är udda. Antag utan inskränkning (detta uttrycket kallas även WLOG [Without Loss Of Generality]) att spelaren med motorsågen delar brädet vinkelrätt mot sidan med längden M så att vi får två bräden med dimensionerna PxN och QxN. Vi vill visa att något av brädena PxN och QxN är vinnande, och enligt induktionsantagandet är det ekvivalent med att η(N) inte är lika med η(P) eller att η(N) inte är lika med η(Q). Så antag att det motsatta gäller (vi använder oss alltså av ett motsägelsebevis), det vill säga att v(N)=v(P) och v(N)=v(Q). Vi kan alltså skriva P=p*2^(v(N)) och Q=q*2^(v(N)), där p och q är udda.
M=P+Q=p*2^(v(N))+q*2^(v(N))=(p+q)*2^(v(N)). Men eftersom p och q är udda så är p+q jämnt, alltså är v(M)>=v(N)+1, vilket är en motsägelse. Vi är därmed färdiga med det första fallet.
Det andra fallet behandlas på ett liknande sätt. Givet att v(M) != v(N) så kan vi anta utan inskränkning (WLOG) att v(M) > v(N). Då kan personen vid draget applicera motorsågen så att det bildas två bräden med dimensionerna 2^(v(N))xN respektive (M-2^(v(N)))xN, och eftersom v(2^(v(N)))=v(N)=v(M-2^(v(N))) så ger induktionsantagandet att båda dessa bräden är förlorande, och vi är därmed färdiga!

Bitmagi

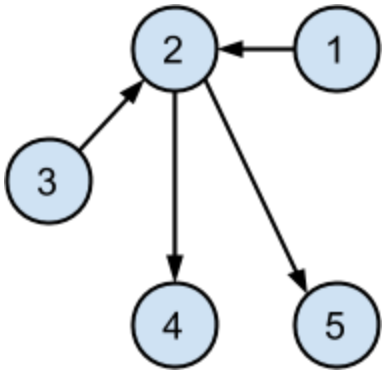
Den största tvåpotensen som delar ett positivt tal x går enkelt att hitta med bithacks (se http://graphics.stanford.edu/~seander/bithacks.html för fler): ((x ^ (x-1)) + 1) >> 1. Detta fungerar eftersom de positioner som skiljer sig mellan de binära talen x och x-1 är precis den lägst satta ettan i x, och positionerna under den. Man kan använda den trivian för att skriva en riktigt kort Python-lösning på brädspelet:

```
a,b=map(int,raw_input().split())
print"BA"[a^a-1^b^b-1>0]
```

Bokrecensioner

Första delpoängen (‘<’)

Vi skapar en graf med N noder. Varje nod representerar en bok. Vi drar en riktad kant mellan nod i och j om vi i indata har givet att a_i < a_j. Så här kommer det att se ut för det första exempeltestfallet:



Finns det en cykel i denna graf så är svaret “-1”. Finns det en väg i denna graf (som följer pilar) längre än M så är också svaret “-1”. I annat fall finns en lösning, och vi hittar den genom att göra en topologisk sortering. Vi börjar helt enkelt på de noder som inte har några pilar “till sig” och tilldelar recensionerna girigt tills alla noder fått rätt värde. Googla “topologisk sortering” för mer info!

Andra delpoängen (‘<’, ‘=’)

Om a_i = a_j innebär det att alla jämförelser med bok i också gäller för bok j. Precis som för första delpoängen ska vi representera böckerna med noder i en graf där en kant från nod i till nod j innebär att a_i < a_j. Antag att vi har en mängd böcker som måste ha samma betyg a_i_1 = a_i_2 = … = a_i_k. Börja med att se till att det inte finns någon motsägelse bland dessa (t.ex a_i_1 = a_i_2, a_i_2 = a_i_3, a_i_1 < a_i_3). Om vi hittar en motsägelse är svaret “-1”. Om det inte finns motsägelser kan vi representera den mängden noder som en enda nod. Välj en representant, t.ex. nod i_1. För varje kant som går från en nod i mängden låter vi den kanten gå från nod i_1. För varje kant som går till en nod i mängden låter vi den kanten gå till nod i_1. Om vi hanterar alla likheter på det sättet har vi en graf där alla kanter representerar ‘<’, utan likheter, precis som för första delpoängen. Problemet löses med en topologisk sortering.

Full poäng

För full poäng behöver vi kunna hanter ‘<=’. Börja med att bygga en graf där varje bok representeras av en nod. Om a_i <= a_j skapar vi en kant från nod i till nod j. Om a_i = a_j skapar vi en kant från nod i till nod j och en kant från nod i till nod j. Vi kan ignorera ‘<’ just nu. I den här grafen vill vi söka efter Strongly Connected Components (SCC) (googla för mer information). I varje SCC i grafen måste de ingående böckerna ha samma betyg. Vi kan ta reda på om det finns motsägelser (genom att titta på jämförelserna med ‘<’) och i så fall är svaret

“-1”. Annars representerar vi varje SCC som en nod på samma sätt som i lösningen för andra delpoängen. Nu kan vi konstruera en graf som bara innehåller ‘<’ och ‘<=’. Om det i denna graf finns cykler är svaret “-1”. Om vi har en graf utan cykler löses problemet med topologisk sortering. I det här fallet blir det lite mer komplicerat än tidigare eftersom vi måste hålla reda på vilka kanter som är ‘<’ och vilka som är ‘<=’. Om en kant med kravet ‘<’ går från nod i till j kan vi sätta a_j = max(a_j, a_i+1). Om en kant med kravet ‘<=’ går från nod i till j kan vi sätta a_j = max(a_j, a_i). Om vi gör så varje gång vi hanterar en kant till nod j tilldelas a_j minsta möjliga värde, vilket är precis vad vi vill.

Att gruppera noder i grafen i SCC:er kan göras i linjär tid med t.ex. Tarjan’s algoritm. Se wikipedia för mer information.

Låttexter

Första delpoängen

För de första 20 poängen räckte tidskomplexiteten O(Q*N) och det enklaste sättet att göra detta på var följande algoritm:
För varje ord sparar vi dess längd. Denna längd är enkel att räkna ut när vi läser in definitionen på ordet, om det är en konkatenering (sammanslagning) av två ord är längden bara summan av de två ordens längder och annars får vi hela strängen given och det är bara att räkna antalet tecken.
Antag nu att vi vill hitta tecken nummer x i strängen S.
Om S var givet i indata direkt (inte som en konkatenering av två andra ord) så hittar vi svaret direkt, annars är S=T+U, där T och U är två andra (kortare!) ord. Om x är mindre än eller lika med längden L(T) av T vet vi att det x:te tecknet i S är lika med det x:te tecknet i T, och vi kan anropa funktionen “hitta x:te tecknet i T”, rekursivt. Om x är större än L(T) är det x:te tecknet i S lika med det (x-L(T)):e tecknet i U, och vi anropar rekursivt funktionen “hitta (x-L(T)):e tecknet i U”. Eftersom längden på det ord vi letar i alltid är kortare än i föregående rekursion vet vi att vi kommer anropa funktionen högst en gång för varje ord, det vill säga högst N gånger. Efter det kan vara Q queries blir den totala komplexiteten O(QN), vilket räcker gott och väl för de första 20 poängen, men inte för några fler.

Full poäng

På denna uppgift finns det många olika lösningar som ger full poäng, alla ganska kluriga.
Den kanske enklaste lösningen har komplexiteten O(N log N + Q log N log L), där L är det högsta tillåtna värdet på R (indexen för de tecken vi vill hitta i den sista strängen). Den går ut på att vi preprocessar givna data genom att för varje ord beräkna så kallade “majoritetsbarn” på avstånd 2^k från ordet. Ett majoritetsbarn på avstånd 1 från ett ord S=T+U definieras som T om L(T)>=U och U annars. Om S inte är en konkatenering är majoritetsbarnet på avstånd 1 odefinierat. För alla k>0 definieras ett majoritetsbarn på avstånd 2^k till ett ord S som majoritetsbarnet på avstånd 2^(k-1) till majoritetsbarnet på avstånd 2^(k-1) till S, eller odefinierat om majoritetsbarnet på avstånd 2^(k-1) till S är odefinierat.
Vi använder sedan majoritetsbarnen i våra queries genom att försöka hitta ett majoritetsbarn som tecknet vi söker efter är en del av. Mer precist så söker vi det största m sådant att majoritetsbarnet C på avstånd m till S sådant att C inkluderar position x i S. Vi hittar detta m genom att först kontrollera vilket det största k är sådant att majoritetsbarnet på avstånd 2^k inkluderar position x i S, därefter kollar vi om majoritetsbarnet på avstånd 2^k+2^(k-1) inkluderar x, om det gör det kollar vi sedan på majoritetsbarnet på avstånd 2^k+2^(k-1)+2^(k-2) annars på det på avstånd 2^k+2^(k-2), och så vidare tills vi fastställt m. (I princip innebär detta att vi gör en binärsökning efter vilket majoritetsbarn vi ska använda, och vilka tvåpotenser vi ska använda motsvarar ettor i den binära representationen av m.) Detta steg tar O(log N).
Vi observerar nu att C’s majoritetsbarn (på avstånd 1) inte innehåller x. Ty om det gjorde det skulle ju majoritetsbarnet på avstånd m+1, vilket är detsamma som C’s majoritetsbarn, innehålla

x, en motsägelse eftersom m var maximalt. Därmed vet vi att C's icke-majoritetsbarn måste innehålla x. Eftersom icke-majoritetsbarnet är mindre eller lika i längd än majoritetsbarnet, betyder det att vi har hittat ett ord av längd högst hälften av längden på S. Vi kan nu upprepa nu samma procedur rekursivt tills vi kommer till ett ord vars värde är givet direkt i indata. Eftersom vi halverar i varje steg, och kan dissa de tecken i ett ord med index över L så innebär det att vi kommer rekursera högst log_2 L gånger. Därför blir komplexiteten för algoritmen O(N log N + Q log N log L).

Det är även möjligt att lösa problemet i O((N+Q)log L) genom att använda sig av självbalanserande träd. Exempelvis kan man i logaritmisk tid "konkatenera" två AVL-träd och skapa en balanserad version av resultatet, så länge man ser till så att inga noder pekar till sina föräldrar (så att de kan återanvändas på flera platser i trädet), och så att man skapar nya noder i stället för att ändra i de gamla (som kanske används på andra platser i trädet). Detta kräver i värsta fall O(log L) minne och O(log L) tid per nytt ord vi processar. Därefter är det enkelt att gå igenom träden på samma sätt som för de första delpoängen, men det går istället i O(log L) per query eftersom träden är balanserade.

Ett tredje lösningsidé är att lösa problemet offline, d.v.s. läsa in alla queries och lösa problemet för dem samtidigt. Det man kan göra är t.ex. att hålla en ordnad mängd av (strängindex, vilken query) för varje ord, som talar om vilka queries som ska lösas för det ordet. Sedan kan vi iterera över orden bakifrån, och för varje sammansatt ord dela upp mängden i två delar - den del som ska lösas för det vänstra delordet, och den som ska lösas för det högra. Nyckelidén som får det här att fungera och inte bli O(NQ) i tidskomplexitet är att när vi delar upp mängden i två delar så kommer den ena delen nästan alltid att vara väldigt mycket större än den andra, och ligga på ena ändan av den ordnade mängden, så vi kan ta bort den lilla delen från mängden, och *flytta hela den stora mängden* åt det håll den ska, i O(1) tid, och sätta en flagga som talar om hur mycket varje strängindex borde justeras med (eftersom vi inte vill ändra detta på varje element i mängden, då detta skulle ta linjär tid). Exakta detaljer, och t.ex. hur man hanterar att flytta en mängd till ett ord där det redan finns en mängd, lämnas som en övning åt läsaren. Det är svårt att ge en exakt bedömning på komplexiteten av den här lösningen, men i praktiken är den den snabbaste av de tre som beskrivits.