Om formler, bevis och andra komplexa saker*

Jakob Nordström†

Maj 2008

Den 4 juni 1996 exploderade rymdraketen Ariane 5 efter en knapp minut av sin jungfrufärd. Skälet visade sig vara en bit programkod i styrsystemet som ärvts från föregångaren Ariane 4. Ingen hade tänkt på att det var stora skillnader i raketernas banor precis efter start. På Ariane 4 var säkerhetsmarginalerna så stora att man inte ens hade brytt sig om att lägga in felhantering i programkoden, men på Ariane 5 kraschade programmet och slog ut hela styrsystemet. 10 års arbete och 7 miljarder dollar försvann i ett av de dyraste fyrverkerier som mänskligheten skådat.

Misslyckandet med Ariane 5 är bara ett exempel, om än ovanligt effektfullt, på något som är ett vanligt problem inom mjukvaruindustrin. Stora datorprogramsystem har en tendens att bli oerhört komplexa och svåra att analysera. För mindre dramatiska exempel på detta behöver man inte gå längre än persondatorn hemma. Som datoranvändare har vi fått vänja oss vid att våra program för ordbehandling, kalkylblad eller Internet bara kraschar ibland, och att man snällt får starta om datorn. (Ännu mer intressant är kanske att vi bara snällt accepterar att det är så. . .)

Inom hårdvaruutveckling finns liknande problem. Kanske är det mest kända exemplet fallet med Intels Pentiumprocessor 1994. Då upptäckte en forskare att det i vissa sällsynta fall kunde bli lite fel när datorn skulle dividera två tal. Intel försökte först tona ned det hela genom att säga att detta var något som den genomsnittlige användaren aldrig skulle drabbas av. Det var förmodligen helt sant, men som PR-drag betraktat var det en katastrof. Företaget fick senare backa och erbjuda ersättningsprocessorer till en kostnad av 475 miljoner dollar.

Som dessa exempel visar kan felaktig design av mjuk- eller hårdvara få förödande konsekvenser. Att upptäcka och undvika fel är därför ett högprioriterat problem. I stora projekt idag förekommer det att över hälften av tiden läggs på kontroll och verifiering, och kostnaden för simulering och testning står ibland för nära hälften av hela projektbudgeten. Men simulering och testning kan aldrig täcka in alla möjliga fall i allt mer komplexa system. Det kan hjälpa till att hitta fel, men inte att säkerställa att det inte finns några andra fel kvar som inte hittats ännu.

Så vad kan man göra? I den här artikeln skissar jag ett sätt att angripa problemet, och försöker förklara hur det i sin tur är kopplat till den forskning som finns beskriven i min doktorsavhandling.

Att skriva populärvetenskapliga artiklar är knepigt. Det innebär att man måste förenkla och generalisera, inte sällan så mycket att det man påstår i slutändan inte riktigt är sant. Läsaren varnas därför för att det som står här är nästan, men inte helt, olikt den forskning som jag faktiskt har bedrivit. Den som vill veta alla de formellt korrekta detaljerna kan läsa mer i själva avhandlingen, men det här är skrivet för att kunna läsas även av den som inte har några förkunskaper inom datavetenskap.

1 Formalisera mera!

Hur kan man undvika att stora tekniska system blir helt oöverskådliga och börjar leva sina egna liv? Ett sätt att angripa problemet är att använda så kallade *formella metoder*. Om man med hjälp av matematiska verktyg skriver ned specifikationer för systemen och föreslagna designlösningar hur de ska byggas, så

*Detta är en förkortad, svensk version av inledningskapitlet i min doktorsavhandling *Short Proofs May Be Spacious: Understanding Space in Resolution*, som finns på webbsidan `www.csc.kth.se/~jakobn/research/`.

†Aktuell adress: Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. E-post: `jakobn@mit.edu`.

kan man sedan använda matematik för att *bevisa* att ett designat system uppfyller sin specifikation. Formella metoder har tidigare mest varit föremål för akademiska studier, men under de senaste 20 åren har den tillämpade forskningen inom området ökat kraftigt, liksom användningen av dessa metoder ute i industrin.

Här kommer ett väldigt förenklat exempel: Anta att vi vill säkerställa att en processor alltid dividerar två tal korrekt. Det enda processorn gör är att arbeta med ettor och nollor, som svarar mot sant och falskt, och därför kan vi beskriva med matematisk logik hur processorn är uppbyggd. Vi kan också beskriva logiskt att när processorn får in två tal x och y så ska den beräkna x/y och ingenting annat. Nu kan vi skriva allt detta som en stor matematisk formel med en massa variabler som uttrycker ungefär att "processorn ser ut såhär, och division av två tal fungerar såhär, och processorn gör rätt när den dividerar." Om den formeln är sann, så har vi designat processorn korrekt. Om formeln inte är sann vill vi hitta ett konkret motexempel för vilka värden på variablerna som formeln blir falsk. Ett sådant motexempel kan förhoppningsvis ge ledtrådar till var i designen det har blivit fel.

Formlerna som man får på detta sätt är vanligen enormt stora, så att försöka räkna på dem med penna och papper är ingen bra idé. Det man vill ha är ett dataprogram som analyserar en formel och antingen bekräftar att formeln är sann eller hittar ett motexempel. Ett sådant program kallas för en *automatisk teorembevisare*.

Formell verifiering är ett område där sådana program är användbara, men listan över tillämpningar är mycket längre än bara design och verifiering av hård- och mjukvara. Automatiska teorembevisare används också bland annat för att lösa schemaläggningsproblem, inom forskning om artificiell intelligens, och till och med för att bevisa saker i teoretisk matematik.

2 Att bevisa saker är komplext

Såhär långt verkar det bara vara goda nyheter. Vi börjar med ett svårt problem, formulerar om det med hjälp av matematisk modellering, och sedan får datorn ta hand om resten. Så nu är problemet löst, eller?

Nej, tyvärr inte. För den dåliga nyheten är att det verkar vara en mycket svår uppgift för en dator att bevisa logiska formler. Man kan tycka att det borde vara enkelt: varför inte bara låta datorn kontrollera alla möjliga fall? Är inte det just den typ av monotona rutinberäkningar som är datorerna paradgren? Problemet är att det det finns alldeles för många fall att kontrollera. Låt oss säga att formeln har N stycken variabler. Varje variabel kan vara antingen sann eller falsk, så sammanlagt får vi 2^N olika fall. Och om vår formel innehåller, säg, en miljon variabler (vilket är realistiskt för verkliga problem) så betyder det att vi får $2^{1000000}$ fall att kontrollera. Detta är ett tal med mer än 300 000 siffror. För att förstå litegrand hur ohyggligt stort det är kan vi göra följande tankeexperiment: även om varje atom i hela det kända universum vore en modern superdator som hade kört i full fart ända sedan tidens början för 13,7 miljarder år sedan, så skulle alla dessa superdatorer tillsammans bara ha hunnit beta av en fullkomligt försumbar del av alla fallen vid det här laget. Så vi skulle inte riktigt ha tid att vänta på att beräkningarna skulle bli klara. . .

Här kommer nästa fascinerande faktum: trots att det finns många avancerade automatiska teorembevisare som fungerar väldigt väl mestadelen av tiden, så har var och en av dem problemet att det finns formler som är så knepiga för dem att de väsentligen inte kan göra något bättre än att testa alla fall. Det är till och med så att det verkar som att för varje dataprogram, eller *algoritm*, som tar en formel som indata och avgör om den är sann eller inte, så måste det finnas formler som algoritmen bara inte rår på (som jag förklarade ovan så är alternativet att testa alla möjliga fall inte genomförbart i praktiken). Jag skriver att det *verkar* vara så, för det är ingen som vet säkert om det måste vara så, eller om vi bara inte har lyckats konstruera tillräckligt bra algoritmer ännu. Denna fråga är faktiskt ett av det stora, öppna problemen inom den moderna matematiken och datavetenskapen. Ett av de sju berömda *millennieproblemen* som Clay Mathematics Institute satt upp som de stora utmaningarna inför det nya årtusendet (`www.claymath.org/millennium/`), och som kan göra den som löser det en miljon dollar rikare, handlar just om detta. (I teknisk jargong är det känt som problemet om huruvida *P* är lika med *NP* eller inte, men vad det betyder får vi ta någon annan gång.)

Att bevisa logiska formler verkar alltså vara ett svårt problem. Inom området *beviskomplexitet* vill vi förstå bättre precis *hur svårt* problemet är (fast inte nödvändigtvis med hoppet om att tjäna en miljon dollar). För att göra det så tar vi till ett originellt grepp – vi struntar i dataprogrammen som faktiskt bevisar formlerna!

Varje automatisk teorembevisare använder någon uppsättning regler för att bevisa formler, och vi kan tänka oss att vi ber programmet att skriva ner vilka regler den använder för att komma fram till sina slutsatser. Resultatet av detta är att när en algoritm har bevisat en formel så kommer det också att finnas ett nedskrivet *bevis* som förklarar varför formeln är sann. Olika algoritmer fungerar på olika sätt, och det sätt en algoritm fungerar på avgör vilka regler som den kan använda i sina bevis. På så sätt svarar varje automatisk teorembevisare mot ett så kallat *bevissystem*. Det är sådana bevissystem som vi är intresserade av att studera inom beviskomplexitet.

Vad kan det ge oss att studera bevissystem? Om en formel som vi vill bevisa är väldigt liten så kanske det finns små bevis för den. Om formeln istället är stor så verkar det naturligt att förvänta sig att också beviset måste vara stort. Vad vi är intresserade av är att mäta hur snabbt bevisstorleken växer *som funktion av formelstorleken.* Om det finns någorlunda små bevis för alla formler mätt i formelstorleken så kan vi hoppas att det också ska gå att hitta dessa bevis snabbt. Men om den minimala bevisstorleken växer väldigt snabbt i ett bevissystem när formlerna blir större, då finns inget hopp om att vi ska kunna bevisa formlerna snabbt. Hur kan vi veta det? Jo, även om algoritmen skulle ha blåtur och hitta det bästa beviset som finns, så vet vi att alla bevis – inklusive det bästa – måste innehålla väldigt många bevissteg. Därför måste algoritmen ta lång tid på sig innan den har hunnit gå igenom alla steg i beviset.

Med andra ord kan storleken hos bevis i ett bevissystem ge oss intressant information om hur effektiv vi kan förvänta oss att motsvarande automatiska teorembevisare är. Att studera bevissystem är också intressant av en rad andra, mer teoretiska, skäl, men att diskutera dem skulle leda utanför vad vi kan klara av att täcka in i denna populärvetenskapliga artikel.

3 Vad är egentligen ett bevis?

Diskussionen om bevis och bevissystem ovan blev på något sätt ganska abstrakt. Vad är ett "bevissystem"? Och vad menas med ett "bevis"? Låt oss försöka bli lite mer konkreta.

Exakt vad är ett bevis? Det är en fråga som är nästan omöjlig att svara på. Till exempel är min doktorsavhandling full med (påstådda) bevis. Några av dem presenteras in i minsta detalj, med sida upp och sida ned av tekniska bevissteg. I andra bevis sägs det bara att vissa påståenden är "uppenbara" eller "enkla att kontrollera" och läsaren får själv försöka fylla i detaljerna. Om man vill starta en animerad diskussion bland teoretiska datavetare så skulle ett förslag kunna vara att fråga exakt vilken nivå av detaljer som är den rätta när man ska ge ett bevis. Det finns nog lika många åsikter om detta som det finns forskare.

Som tur är behöver vi inte ge något hundraprocentigt heltäckande svar på frågan. Vi behöver bara hitta en definition som fungerar för de mycket begränsade formella bevissystem som vi studerar inom beviskomplexitet. Och här visar det sig att rätt definition är ungefär följande: *"Ett bevis är något som hjälper dig att enkelt verifiera ett påstående som annars kanske skulle vara svårt att kontrollera."*

Min avhandling handlar om bevissystem för logik. Notationen i sådana system kan tyvärr kännas lite hotfull och tar en stund att vänja sig vid. Här är vi ute efter exempel som man kan förstå på en gång! Därför kommer vi att koncentrera oss på heltalen $1, 2, 3, 4, \ldots$ istället för på logik.

Vi startar med några fakta som inte har med beviskomplexitet att göra, men som kan vara intressanta i alla fall. Alla heltal större än 1 kan "delas upp" i en produkt av heltal (också större än 1). Till exempel så kan vi skriva $77 = 7 \cdot 11$ och $30 = 2 \cdot 3 \cdot 5$. Ett av de grundläggande resultaten inom matematiken säger att om vi delar upp ett tal i "minimala beståndsdelar", i den meningen att heltalen i produkten själva inte kan skrivas som produkter av mindre heltal, så är denna uppdelning, eller *faktorisering*, unik. Detta är känt som *Aritmetikens fundamentalsats.*

I exemplen ovan är det lätt att kontrollera att $2, 3, 5, 7$ och 11 inte kan skrivas som produkter av mindre heltal, så uppdelningarna är minimala. De "minimala komponenterna" i en faktorisering kallas för *primtal*. Primtalen kan inte delas upp i mindre beståndsdelar utan kan bara skrivas som en ”produkt” av ett enda tal, nämligen sig själva.

Anta nu att något ger oss ett heltal N och frågar oss hur faktoriseringen av N ser ut. För att ta konkreta exempel, låt oss tänka på de två talen $N_1 = 25957$ och $N_2 = 510510$. Hur kan man skriva dem som produkter av mindre heltal? Eller de kanske är primtal? (Fundera gärna på detta en liten stund innan du läser vidare – det ena talet är lätt att faktorisera medan det andra är svårare).

Det här problemet kanske ser enkelt ut, men det ligger i den absoluta forskningsfronten inom datavetenskap. Enorma mängder tid och pengar har investerats för att försöka konstruera bra faktoriseringsalgoritmer. Ännu har ingen lyckats. Eller vi *tror* att ingen har lyckats. Det är en välkänd hemlighet att myndigheter som Försvarets radioanstalt (FRA) i Sverige och National Security Agency (NSA) i USA arbetar intensivt med detta problem, de också, men de publicerar inte sina resultat.

Varför är man så intresserad av faktorisering? Jo, därför att de flesta forskare tror att problemet är så svårt att det omöjligt kan finnas några effektiva faktoriseringsalgoritmer. Och även om vi inte vet säkert att det är så, så är konsensus att det är väldigt sannolikt. Så sannolikt att det finns många IT-säkerhetssystem där hela säkerheten bygger på att faktorisering måste vara svårt (annars är systemen inte säkra). Mycket av modern kryptografi fungerar på det här sättet. Och det här är inte bara teoretisk grundforskning – förmodligen är exempelvis säkerhetssystemet för din internetbank byggt på att faktorisering av stora tal, eller något annat liknande problem, förmodas vara så svårt att det i praktiken är omöjligt att lösa även med mycket kraftfulla datorer (sådana som finns på FRA och NSA, till exempel).

Men låt oss återvända till beviskomplexitet. Betrakta följande variant av faktoriseringsproblemet: anta att vi får två heltal N och k från en person som hävdar att faktoriseringen av N innehåller precis k primtal. Säg till exempel att någon påstår att $N_1 = 25957$ är en produkt av 2 primtal. Kan vi avgöra om detta är sant? Vad skulle vara ett bevis som övertygade oss?

Det känns som om det skulle kunna bli svårt att avgöra om detta är sant eller inte på egen hand, eftersom vi inte vet hur man faktoriserar heltal effektivt. Men eftersom den person som vi talar med tydligen känner till faktorerna så kan vi ju be honom eller henne att bevisa påståendet genom att tala om vilka de är. Om vi som svar får listan

$$\begin{array}{r} 101 \\ 257 \end{array}$$

så kan vi ganska enkelt kontrollera att beviset är korrekt. För det första innehåller listan två tal. För det andra så blir resultatet 25957 om vi multiplicerar talen. För det tredje kan vi se efter att varken 101 eller 257 kan skrivas som en produkt av mindre heltal, så de är primtal. Notera att alla de här beräkningarna är enkla att göra för hand. Och om talet N är större, så att det blir svårt att räkna med penna och papper, så finns det effektiva algoritmer både för multiplikation och primtalstestning, så i sådana fall kan vi låta en dator kontrollera beviset för oss.

Ett bevissystem har två viktiga egenskaper. För det första så kan vi *aldrig luras att acceptera ett felaktigt påstående.* Tänk om personen vi talade med istället hade sagt att 25957 har 6 faktorer. Vi ber att få beviset, och får kanske listan

$$\begin{array}{r} 2 \\ 2 \\ 3 \\ 3 \\ 7 \\ 103 \end{array}$$

Det är förstås nära – produkten blir 25956 – men vi har ändå inga problem att upptäcka felet och kan förkasta det felaktiga beviset.

För det andra så *finns det alltid bevis för sanna påståenden.* Givetvis kan det hända ibland att det blir fel i bevis (det händer faktiskt ganska ofta även i vetenskapliga matematiska artiklar). Om någon skulle säga till oss att de två faktorerna i 25957 är

$$\begin{array}{r} 100 \\ 258 \end{array}$$

så kan vi inte acceptera detta som ett bevis eftersom det är uppenbart fel. Men poängen är att eftersom det faktiskt är sant att 25957 har precis 2 faktorer så finns det ett annat, korrekt bevis som kan övertyga oss om detta faktum (nämligen det första beviset ovan). Att *hitta* ett sådant bevis kan vara en helt annan sak, men vi vet åtminstone att det måste *finnas*.

När vi väl har ett bevissystem så vill vi förstå dess egenskaper. Två viktiga egenskaper, som jag har koncentrerat mig på i min forskning, är *bevislängd* och *bevisminne*. För att förklara vad detta är så tittar vi på vårt andra tal $N_2 = 510510$, som vi påstår har 7 faktorer. Beviset för detta är listan

$$\begin{array}{r} 2 \\ 3 \\ 5 \\ 7 \\ 11 \\ 13 \\ 17 \end{array}$$

som har 7 rader, så längden av beviset är 7.

Bevisminnet är (grovt talat) det antal saker man behöver hålla reda på medan man kontrollerar beviset. När vi läser listan ovan så kan vi multiplicera talen ett i taget och hålla reda på hur många tal vi tagit hand om hittills. Allt vi behöver minnas är alltså (1) hur många faktorer vi har tittat på hittills, (2) vad produkten av alla dessa faktorer är, och (3) talet som vi tittar på just nu. När vi har gått igenom hela listan så kan vi kontrollera att den innehöll 7 faktorer och att produkten av dem är 510510. Eftersom det är så är allt i ordning och vi kan acceptera beviset. För att kontrollera beviset behövde vi bara minnas 3 saker, så vi säger att bevisminnet är 3.

Observera att även om de bevis som vi får är väldigt långa (om heltalet N har ett stort antal faktorer), så är alla bevis i vårt bevissystemet ”enkla” i den meningen att man bara behöver minne 3 för att verifiera dem oavsett hur långa de är. För mer generella bevissystem så är sambandet inte så enkelt, utan där är det en intressant fråga hur längd och minne beror på varandra.

I min forskning har jag studerat bevissystemet *resolution* för logiska formler. Resolution är ett viktigt bevissystem eftersom det är grunden för de bästa automatiska teorembevisarna som finns idag. Ett öppet problem för resolution har varit följande fråga:

> *Anta att ett bevis är kort. Är då beviset enkelt också i den meningen att det bara kräver lite minne? Eller kan det vara så att det finns korta bevis som ändå är så komplicerade att man måste hålla reda på väsentligen hela beviset medan man kontrollerar det?*

Huvudresultatet i min doktorsavhandling svarar på denna fråga genom att visa att det faktiskt finns formler som har korta bevis men att dessa bevis måste vara så invecklade att man inte kan kontrollera dem ”lokalt” rad för rad (som vi gjorde ovan). Istället måste man minnas nästan hela beviset samtidigt för att kunna verifiera det.

Det här resultatet är ett exempel på vad som kallas en *undre gräns*, i detta fall en undre gräns som säger att vissa formler alltid kräver en stor mängd minne. Förutom att den undre gräns som jag har visat besvarar en känd öppen fråga inom beviskomplexitet så är den också intressant för automatiska teorembevisare. Lite slarvigt så kan man tolka resultatet som att det finns formler som är väldigt enkla att bevisa om man angriper dem ”på rätt sätt”, men att hitta rätt sätt är litegrand som att leta efter en nål i en höstack.

4 Varför är det svårt att visa undre gränser?

Att försöka bevisa undre gränser är en ganska speciell verksamhet. Inom det mesta av datavetenskapen, och inom vetenskapen i stort, så försöker man visa hur man kan *göra* saker: konstruera, beräkna, bevisa, uppfinna... En undre gräns är något rakt motsatt – den visar att någonting *inte går att göra*.

Om man till exempel vill visa att ett problem kan lösas effektivt med hjälp av datorer så är spelreglerna ganska tydliga: man behöver komma på rätt idé för en algoritm, använda den för att konstruera ett dataprogram, och sedan bevisa att programmet är korrekt. Men för att visa att ett problem *inte* kan lösas effektivt med dator så måste vi visa att *inget dataprogram, oavsett hur det är uppbyggt*, kan lösa problemet snabbare än någon viss tidsgräns. Observera att vi inte kan göra några som helst antaganden om hur dataprogrammen är konstruerade. De kan göra oerhört märkliga saker. Kanske finns det en helt genial lösning till det här problemet som ingen har tänkt på förut! Kanske finns det något program som verkar helt knäppt vid första anblicken, men som ändå på något sätt lyckas lösa problemet oerhört snabbt. Vi kan inte *anta* att så inte är fallet – vi måste *bevisa* det.

Som avslutning på den här artikeln tänkte jag presentera en fallstudie som visar varför det kan vara väldigt svårt att bevisa undre gränser. Återigen undviker vi att ge oss in på logikens område, eftersom vi vill tala om saker som inte kräver förkunskaper inom datavetenskap eller högre matematik. Istället ska vi diskutera hur man kan sortera tal effektivt. Problemet som vi vill att datorn ska lösa för oss är detta:

> *Givet en lista med N olika heltal i godtycklig ordning, räkna upp dessa tal sorterade i stigande ordning.*

Vi förstår att det förmodligen måste ta längre tid att sortera 100 tal än att sortera bara 10 tal. Det vi är intresserade av är att visa undre gränser för hur mycket tid som behövs uttryckt som en funktion av antalet heltal N som ska sorteras. Som ett löpande exempel kommer vi att använda listan

$$670,\ 45,\ 490,\ 642,\ 475,\ 124,\ 802,\ 266$$

med $N = 8$ tal att sortera.

En naturlig idé, som man kan använda till exempel när man får kort från en kortlek och vill sortera sin hand, är att ha de osorterade korten till höger och sedan stoppa in dem ett och ett på rätt plats i en sorterad mängd kort till vänster. Den här algoritmen kallas *insättningssortering* och i figur 1 finns en illustration av hur den fungerar för vår exempellista.

1.	**670**		45	490	642	475	124	802	266
2.	**45**	670		490	642	475	124	802	266
3.	45	**490**	670		642	475	124	802	266
4.	45	490	**642**	670		475	124	802	266
5.	45	**475**	490	642	670		124	802	266
6.	45	**124**	475	490	642	670		802	266
7.	45	124	475	490	642	670	**802**		266
8.	45	124	**266**	475	490	642	670	802	

Figur 1: Exempel på insättningssortering (sorterade tal till vänster, osorterade tal till höger).

Hur lång tid tar insättningssortering? Det är en snabb metod för små listor, men när talen att sortera blir fler så börjar det gå sämre. När man ska sortera in de sista talen i listan så kan man behöva gå igenom en stor andel av de tal som redan sorterats för att hitta rätt plats att stoppa in det aktuella talet. Se till exempel på omgång 8 i figur 1 när det sista talet 266 ska sorteras in. Observera att datorn inte har något sätt att direkt ”se” var talet ska hamna. För datorn är 266 bara ett mönster av ettor och nollor som kan jämföras med andra mönster av ettor och nollor som svarar mot andra tal. Så vi måste gå igenom listan antingen från vänster eller höger, och om rätt position är någonstans i mitten så kommer det ta nästan $N/2$ steg att hitta dit. Man kan visa att det här kan förväntas hända för en ganska stor andel av talen. Insättningssortering tar därför kvadratisk tid mätt i antalet tal som ska sorteras, det vill säga tid proportionellt mot N^2.

Men det är kanske ändå det bästa vi kan hoppas på? Det är N tal som ska sorteras, och vi behöver veta hur alla talen förhåller sig till varandra för att kunna sortera dem i rätt ordning. Eftersom vi har N tal så betyder det att det finns $N(N-1) \approx N^2$ olika par av tal. Därför kan det kännas rimligt att förvänta sig att det skulle behövas ungefär N^2 jämförelser innan vi har rett ut relationerna mellan talen i alla dessa par. Om vi bara räknar jämförelser och ignorerar andra saker som algoritmen också behöver göra, så ser vi att det verkar som om den tid som behövs för att sortera N tal måste vara proportionell mot N^2, eller hur?

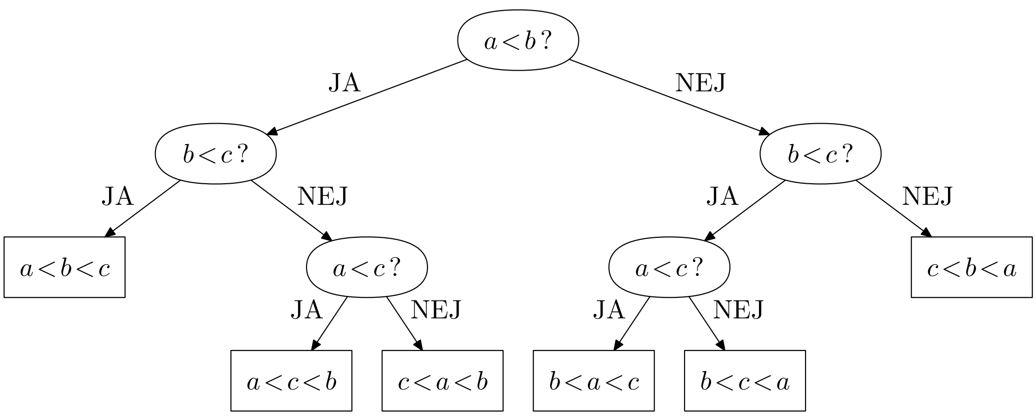
Nej, det visar sig vara fel. Låt oss titta på en helt annan sorteringsidé. Anta att vi delar upp vår lista med tal i två halvor $670, 45, 490, 642$ och $475, 124, 802, 266$. Anta vidare, med lite önsketänkande, att vi kan sortera dessa halvor på något smidigt sätt så att vi får $45, 490, 642, 670$ och $124, 266, 475, 802$. Då kan vi snabbt slå ihop de två sorterade listorna till en större sorterad lista. Börja med att titta på det första talet i varje lista. Talet 45 är minst, så det ska vara först i den stora listan. Nu är 124 mindre än 490, så det talet kommer tvåa, och av samma anledning väljer vi sedan 266 och 475 från den andra listan. Sedan ser vi att 802 är större än 490, så då byter vi till första listan igen och tömmer den helt, eftersom alla tal som är kvar i listan är mindre än 802. Slutligen lägger vi till 802 sist i vår stora lista, som nu är helt sorterad! Observera att till skillnad från för insättningssorteringen så kunde vi nu hela tiden direkt hitta rätt plats för varje tal utan att behöva gå igenom listan av tal som redan sorterats.

Men det verkar finnas en hake här: hur sorterar vi de mindre listorna till att börja med? Tja, varför inte använda samma idé en gång till? Dela upp den första listan $670, 45, 490, 642$ i två dellistor $670, 45$ och $490, 642$, sortera dessa listor, och slå ihop dem. Men det verkar ju bara bli ett cirkelresonemang – hur ska vi sortera de listorna då? Jo, de listorna har bara två tal, så de är enkla att ta hand om: om talen står i fel ordning byter vi plats på dem och annars låter vi dem vara. Och om man hela tiden delar upp större listor i mindre så får man förr eller senare små listor som kan sorteras enkelt på detta sätt, så vi har faktiskt inget cirkelresonemang alls.

Den här algoritmen kallas *fusionssortering* (eller *mergesortering* på svengelska), eftersom den bygger på att slå ihop små listor till större listor. Den kanske verkar tillkrånglad, men det visar sig att den är dramatiskt mycket snabbare än insättningssortering. Även om det är långt ifrån uppenbart (och vi ska inte försöka oss på ett bevis här) så går den i tid proportionell mot $N \ln N$, det vill säga N gånger naturliga logaritmen av N, istället för N^2. För lite större datamängder innebär den skillnaden att sorteringen går på några sekunder istället för att ta timmar eller dagar!

Jaha, så nu har vi sänkt tidsgränsen för sortering från N^2 till $N \ln N$. Går det att sortera ännu mer effektivt? Eller är det omöjligt att sortera snabbare än $N \ln N$? Låt oss försöka bevisa en undre gräns!

Som vi noterade ovan så vet vi inte hur algoritmerna kan tänkas fungera, så vi koncentrerar oss på något som de rimligen måste göra, nämligen jämföra talen. Låt oss räkna jämförelserna som algoritmerna gör och bevisa en undre gräns för hur många som minst behövs. Vi kan representera alla möjliga utfall av jämförelserna genom att rita en figur, som i figur 2 för en exempelalgoritm som sorterar tre tal. Inuti ovalerna skriver vi vilka jämförelser som görs, och pilarna från ovalerna visar de två möjliga utfallen. Det här blir vad datavetare kallar ett *träd* (fast vi brukar insistera på att rita våra träd upp och ned).



Figur 2: Beslutsträd för att sortera tre tal a, b, c.

Varje sorteringsalgoritm svarar mot ett sådant träd på följande sätt. Algoritmen börjar med någon första jämförelse, som vi ritar in i *roten* på trädet (som alltså är högst upp i figuren). Beroende på resultatet av jämförelsen så kan algoritmen sedan göra två olika saker, vilket svarar mot att gå längs med en av de två grenarna från roten. Så snart algoritmen vet hur listan ska sorteras stannar den och matar ut talen i rätt ordning. Vi markerar detta i vårt träd genom att rita ett fyrkantigt *löv* där det står hur talen är ordnade. Eftersom det här trädet hjälper oss att fatta vårt *beslut* om hur talen ska sorteras så brukar det kallas för ett *beslutsträd*.

Vi ska använda oss av beslutsträdet för att bevisa en undre gräns för sortering. För att göra det så noterar vi först tre saker:

1. Antalet jämförelser som algoritmen behöver i värsta fallet är den längsta vägen från roten till något löv längs grenarna i trädet. Listan som vi får att sortera kan ju nämligen se ut hursomhelst, så alla vägar genom trädet är möjliga scenarier för algoritmen.

2. Det måste finnas åtminstone ett löv i trädet för varje möjligt svar. Som vi just noterande kan listan med tal att sortera vara ordnad hursomhelst, så alla möjliga sätt att ordna talen är potentiella kandidater för rätt sorteringsordning. Eftersom alla de svar som algoritmen kan ge finns i löven så måste det finnas ett löv för varje möjligt svar.

3. Antalet möjliga olika svar när man sorterar tre tal är $3 \cdot 2 \cdot 1 = 6$ stycken. För att se att det är så kan man resonera såhär: Vilket som helst av de 3 talen kan vara minst. När vi vet vilket tal det är så finns det 2 kandidater kvar för mittentalet, och sedan finns det bara 1 tal kvar som måste vara det största (se löven i figur 2 för de olika varianterna). Mer allmänt gäller att om vi sorterar N tal så finns det

   $$N(N-1)(N-2)\ldots \text{ (fortsätt multiplicera med mindre tal) } \ldots 3 \cdot 2 \cdot 1$$

   möjliga svar, eftersom det minsta talet kan vara vilket som helst av de N talen, det näst minsta kan vara vilket som helst av de $N-1$ andra talen, det näst näst minsta kan vara vilket som helst av de $N-2$ kvarvarande talen, och så vidare.

Med hjälp av dessa observationer kan vi nu bevisa vår undre gräns i en tvåstegsraket.

I det första steget så kan man visa (och det är inte svårt) att ett beslutsträd med L stycken löv måste ha något löv som är på avstånd åtminstone $\ln L$ (logaritmen av L) från roten. Om alla vägar i trädet skulle vara kortare än så skulle det helt enkelt inte få plats L löv i trädet. Som ett konkret exempel på detta kan vi observera att figur 2 visar att det är omöjligt att sortera 3 tal med bara 2 jämförelser. Varför? Jo, därför att det finns 6 möjliga svar, men om alla vägar i trädet skulle ha längd högst 2 så skulle vi inte få plats med fler än 4 löv.

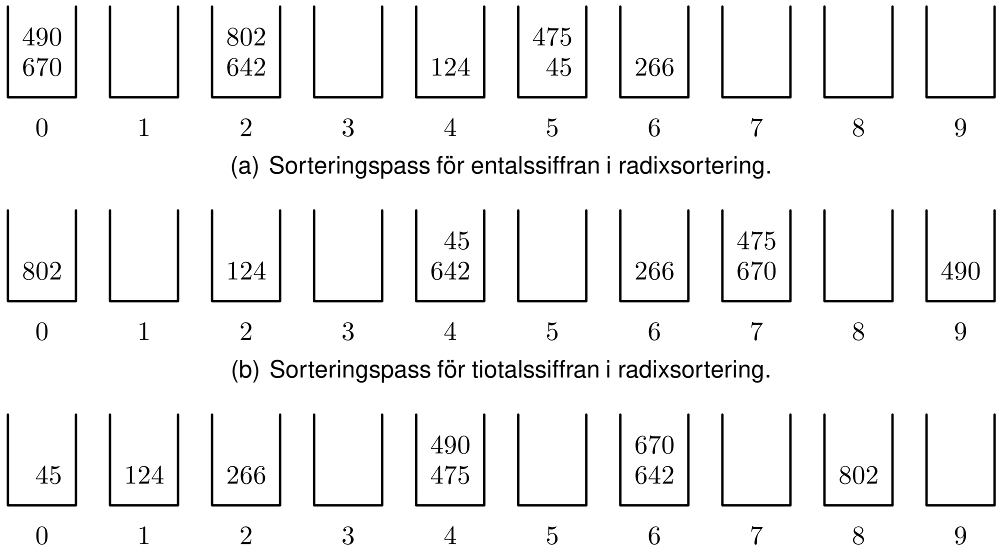
I det andra stegat så kan man räkna ut (men det är svårare) att

$$\ln\bigl(N(N-1)(N-2)\cdots 3 \cdot 2 \cdot 1\bigr) \approx N \ln N \ .$$

Men det betyder att eftersom det finns $N(N-1)(N-2)\cdots 3 \cdot 2 \cdot 1$ möjliga svar på vilken sorteringsordning som är den rätta, och alla dessa svar behöver åtminstone ett löv var, så måste varje algoritm som sorterar N tal göra proportionellt mot $N \ln N$ jämförelser i värsta fallet. Detta måste alltså vara en undre gräns för den tid som behövs för sorteringen. (Kom ihåg att vi helt ignorerar alla andra saker som algoritmen också kommer att behöva göra.) Följaktligen är fusionssorteringen som vi beskrev ovan en optimal sorteringsalgoritm!

Nästan. Det är verkligen sant att $N \ln N$ är en undre gräns för så kallade *jämförelsebaserade* sorteringsalgoritmer. Men tänk om det finns sorteringsalgoritmer som inte gör jämförelser? Det låter kanske som en konstig tanke att inte jämföra det som man ska sortera, men kom ihåg att vi varnade för att de algoritmer som vi försöker visa undre gränser för kan verka vara knäppa.

Så här kommer en knäpp algoritm. Den fungerar på följande sätt (beskrivningen är nog enklare att följa om man samtidigt tittar på figur 3 nedan). Vi har tio *hinkar* som vi märker 0, 1, 2, 3, $\ldots$, 8, 9. Eftersom talen i vår lista har tre siffror så kommer vi att göra tre sorteringspass över listan. I det första passet så sorterar vi alla tal på entalssiffran. Det vill säga, 670 placeras i hink 0, 45 placeras i hink 5, 490 placeras i hink 0 ovanpå 670, 642 placeras i hink 2, 475 placeras i hink 5 ovanpå 45, och så vidare som i figur 3(a). Sedan vänder vi upp och ned på hinkarna, häller ut talen, och går igenom dem igen från vänster till höger, och uppifrån och ned för varje "hinkhög". Den här gången sorterar vi talen på tiotalssiffran, så 670 placeras i hink 7, 490 placeras i hink 9, 642 placeras i hink 4, 802 placeras i hink 0, etc. Resultatet av detta andra sorteringspass finns i figur 3(b). I tredje sorteringspasset så vänder vi hinkarna upp och ned och går igenom talet igen från vänster till höger och sorterar den på hundratalssiffran. Eftersom 45 saknar hundratalssiffra så placeras talet i hink 0. Detta leder till att talen hamnar i hinkar som i figur 3(c). Slutligen vänder vi alla hinkar upp och ned och räknar upp alla talen från vänster till höger, och uppifrån och ned för tal i samma hink (efter att ha vänt hinken upp och ned, som sagt). Det är enkelt att kontrollera att om vi gör det med talen i figur 3(c) så får vi våra åtta tal sorterade i rätt ordning.



(a) Sorteringspass för entalssiffran i radixsortering.

(b) Sorteringspass för tiotalssiffran i radixsortering.

(c) Sorteringspass för hundratalssiffran i radixsortering.

Figur 3: Exempel på radixsortering av listan 670, 45, 490, 642, 475, 124, 802, 266.

Den här algoritmen kallas för *siffersortering* eller *radixsortering*. Den kanske verkar lite magisk, men den fungerar. Och dessutom slår den fusionssortering (och vår undre gräns) genom att gå i tid proportionell mot N istället för $N \ln N$.

Så hur är det med N, då? Är åtminstone *det* en undre gräns för den tid som behövs för sortering? Ja, det är det. För att kunna sortera N tal så måste en algoritm åtminstone *titta* på varje tal minst en gång. Eftersom det finns N tal i listan så måste detta ta tid proportionellt mot N. (Och den här gången går det att bevisa att det faktiskt är så.)

Därmed har vi kommit till slutet på denna artikel. Jag hoppas att avsnitten 1 och 2 har kunnat förmedla åtminstone litegrand om vad beviskomplexitet är och vad det kan vara bra för. Slutet av avsnitt 3 gav förhoppningsvis något slags känsla för vilken typ av problem jag har studerat i min forskning, och de mer invecklade resonemangen i avsnitt 4 kanske gav en indikation på varför det ibland kan vara väldigt knepigt att visa undre gränser.

Och ifall du tyckte att detta var intressant så finns det en hel doktorsavhandling som bara väntar på att bli läst…