Haltproblemet

Dagens föreläsning

- Motivation och bakgrund
- Haltproblemet
- Uppräkningsbarhet, diagonalisering och beräkningens begränsningar
- Sammanfattning

Motivation och bakgrund

Det här är nog den coolaste föreläsningen på kursen – vi ska se ett bevis för att det inte går att avgöra om program är korrekt skrivna och avslutas förr eller senare, och dessutom skall vi få se ett resonemang som säger att mängden av program som kan skrivas är uppräkningsbar (vad det betyder kommer vi strax in på)!

Eftersom det bara finns en uppräkningsbar mängd program som kan skrivas, men mängden problem inte är uppräkningsbar (det visar vi också), är det alltså uppenbart att vissa problem helt enkelt inte går att lösa på algoritmisk väg!

Motivation och bakgrund, forts.

Hilbert ställde upp ett antal problem 1928, bland annat det som kallas för "Entscheidungsproblem" – alltså "avgörbarhetsproblemet". Det gick ut på att lösa frågan huruvida det går att skapa en dator, där man som indata anger en beskrivning av ett formellt språk och ett matematiskt påstående. Svaret från datorn skall vara `true` eller `false`, beroende på huruvida påståendet är sant eller ej. Datorn behöver inte rättfärdiga sitt svar, utan måste bara vara korrekt.

Alan Turing bevisade 1936 att det är omöjligt att skapa en sådan dator, genom att reducera "haltproblemet" för Turingmaskiner till Hilberts avgörbarhetsproblem. Vi ska nu ta en närmare titt på haltproblemet.

Universella Turingmaskiner

För att kunna föra vårt resonemang om haltproblemet, måste vi introducera konceptet ”universell Turingmaskin”. Det är en Turingmaskin som kan simulera en annan Turingmaskin.

Det är förmodligen inte svårt att göra en koppling till datorns värld, där vi bland annat har Java Virtual Machine, emulatorer för diverse spelkonsoller och så vidare.

Fundera gärna själva på hur en universell TM kan skapas! Vi behöver dock mest bara acceptera att de går att skapa.

Haltmängden

Vi definierar följande mängd:

$$HALT = \{\langle M \rangle \langle w \rangle \mid \text{Turingmaskinen } M \text{ stannar vid indata } w\}$$

Precis som förra gången innebär notationen $\langle M \rangle$ en strängrepresentation av M – tänk ”källkod”.

Vi vill visa att $HALT$ är **oavgörbar**. Den är dock *semiavgörbar*, eftersom en universell TM som simulerar maskinen M bara behöver stanna då M gör det.

Haltmängdens oavgörbarhet

Hela nästa föreläsning kommer handla om konsekvenser av att haltmängden är oavgörbar. Men redan idag kan vi fundera lite kring vad det egentligen innebär.

Vi kan i generella fall inte säga huruvida ett program $P \in HALT$. Vi kan alltså inte veta om ett program kommer stanna eller ej! Det går i specialfall (`while(true);`), men inte i generella fallet.

Haltmängdens oavgörbarhet, forts.

Stannar följande program, eller fastnar det i en evig slinga?

```
program searchForOddPerfectNumber()
  var int n = 1     // arbitrary-precision integer
  loop {
    var int sumOfFactors = 0
    for factor from 1 to n - 1
    if factor is a factor of n
      sumOfFactors = sumOfFactors + factor
      if sumOfFactors = n then
        exit loop
      n = n + 2
  }
return
```

Haltmängdens oavgörbarhet, forts.

Om någon kommer på svaret på föregående bild, så rekommenderar jag att den personen skickar in beviset till någon matematisk journal – problemet har inte lösts på flera hundra år!

(Ett perfekt tal är ett heltal som kan uttryckas som summan av dess positiva delare/divisorer, som exempelvis $6 = 1 + 2 + 3$. Hittills är bara positiva perfekta tal kända.)

Bevisdags!

Att det var svårt att veta om algoritmen för att hitta ett udda perfekt tal kommer stanna är dock inte ett bevis för att $HALT$ är oavgörbart, möjligtvis har det dock gjort oss mer mottagliga för att så kan vara fallet.

Om vi antar (för motsägelse) att $HALT$ är avgörbart, låt oss låtsas att `halt(M, w)` är ett korrekt program som undersöker vad M hade gjort vid indata w och returnerar `true` om M stannar vid indata w, `false` annars.

Bevis, forts.

Vi skapar ett program vi kallar för `lura(M)` som fungerar enligt följande:

```
program lura(M)
  if halt(M, M)
    loop forever
  else
    return
```

Alltså, vi tar ett program representerat med en sträng som indata och avgör sen med hjälp av `halt` om programmet skulle stanna med sin egna strängrepresentation som indata. Om så är fallet fastnar vi medvetet i en evig slinga, annars (om `halt` avgör att programmet inte stannar, alltså) stannar vi.

Bevis, forts.

Vad händer då vi kör `lura(lura)`?

Vi har uppenbarligen bara två möjliga fall – antingen stannar vi i en evig slinga, eller så stannar programmet. Men vad innebär det att `lura(lura)` har stannat eller fastnat i en evig slinga?

Bevis, forts.

Fall 1: säg att `lura(lura)` hamnade i en evig slinga. Då måste det innebära att `halt(lura, lura)` returnerade `true` – men det är ju omöjligt, för vi fastnade i en evig slinga!

Fall 2: om `lura(lura)` stannade, måste det innebära att `halt(lura, lura)` returnerade `false` – det är också omöjligt, vi vet ju att `lura(lura)` stannar!

Vårt antagande, att det det finns ett program `halt(M, w)` som avgör om ett program stannar med en viss indata är således falskt. $HALT$ är följdaktligen **oavgörbart**.

”Jaså, du har skrivit program nummer 57!”

Med haltproblemet i bagaget, koncentrerar vi oss nu på påståendet att det bara finns en uppräkningsbar mängd program som kan skrivas.

Om vi vill jämföra storleken av två mängder för att se vilken mängd som är större är det intuitivt att bara räkna elementen i mängderna. Men det fungerar uppenbarligen inte om vi har oändligt stora mängder.

Cantor föreslog en lösning som fungerar för oändliga mängder också – två mängder är lika stora om *elementen i en av mängderna kan paras ihop med elementen i den andra*.

Mängders storlek

Denna mappning $f : A \to B$ mellan mängder måste vara en korrespondens, alltså att $f(i) \neq f(j)$ om $i \neq j$ samt att det för varje $b \in B$ det finns ett $a \in A$ sådant att $f(a) = b$.

Ett exempel på mängder som är lika stora enligt den här definitionen är exempelvis de jämna naturliga talen $\mathcal{E}$ och de naturliga talen $\mathcal{N}$. Det låter helt galet, eftersom det intuitivt bara borde finnas hälften så många jämna tal som jämna och udda tillsammans!

Mappningen från $\mathcal{N}$ till $\mathcal{E}$ är $f(n) = 2n$. Vi ser alltså att så fort vi betraktar ett naturligt tal, så kan det mappas till ett jämt positivt naturligt tal. Uppenbarligen *är* mängderna lika stora!

Mängders storlek, forts.

Vi säger att en mängd A är **uppräkningsbar** om A antingen är ändlig *eller* har samma storlek (enligt Cantors definition) som $\mathcal{N}$.

Det känns kanske som om vi i stort sett kan göra en mappning mellan de naturliga talen och vilken mängd som helst och att allt således skulle vara uppräkningsbart, men så är inte fallet. Vi skall exempelvis se både att de rationella talen är uppräkningsbara, och att de reella talen inte är det.

Beviset bygger på en teknik som kallas **diagonalisering** – varför det heter just så framgår snart.

$\mathcal{Q}$ är uppräkningsbar

De rationella talen $\mathcal{Q}$, alltså tal i bråkform, är till skillnad från de reella talen uppräkningsbara.

Vi kan lätt skapa en oändlig matris där vi skriver upp alla rationella tal – raderna ger täljaren och kolonnerna nämnaren. Det enda problemet vi har är att vi inte får upprepa tal som förekommer flera gånger, exempelvis är ju $\frac{1}{1} = \frac{2}{2} = \frac{3}{3} = \ldots$.

Det är lätt att börja i övre vänstra hörnet med $\frac{1}{1}$ och sen gå på diagonalen och inkludera endast de bråk som inte representerar sådana tal vi redan har inkluderat i mängden. Vi håller reda på "sekvensnummret" och på detta vis skapar vi en mappning mellan de naturliga talen $\mathcal{N}$ och de rationella talen $\mathcal{Q}$.

$\mathcal{R}$ är dock inte uppräkningsbar

För att bevisa att $\mathcal{R}$ är uppräkningsbar måste vi bevisa att det inte går att skapa en mappning mellan $\mathcal{N}$ och $\mathcal{R}$. Om vi för motsägelse antar att så dock är fallet, måste det gälla för alla tal $x \in \mathcal{R}$. Låt oss skapa ett $x \in \mathcal{R}$ för vilket denna mappning ej kan gälla!

Låt oss anta att vår mappning ger oss bland annat följande värden:

n	$f(n)$
1	$1,5672\ldots$
2	$7,1234\ldots$
3	$3,1415\ldots$
$\vdots$	$\vdots$

$\mathcal{R}$ är inte uppräkningsbart, forts.

Låt oss skapa ett $x \in \mathcal{R}$ som inte kan ingå i mappningen.

För att säkerställa att $x \neq f(1)$, låt den första siffran efter decimaltecknet i x vara skiljd från första siffran efter decimaltecknet i $f(1) = 1,5672\ldots$, exempelvis 3. Vi undviker att $x = f(2)$ genom att låta andra decimalsiffran i x vara skiljd från andra decimalsiffran i $f(2)$.

Gör på samma sätt för att säkerställa att $x \neq f(3)$ eller $f(4)$ och så vidare. Kort sagt, skapa x så att den n:te decimalsiffran är skiljd från den n:te decimalsiffran i $f(n)$. Vi går alltså på diagonalen bland decimalsiffrorna i $f(n)$ och ser till att decimalsiffrorna i x skiljer sig från var och en på just den positionen.

Vårt framgångsrika skapande av talet x som inte ingår i mappningen visar att $\mathcal{R}$ ej är uppräkningsbart!

Kopplingen till Turingmaskinerna

Reella och rationella tal är i och för sig intressanta, men för oss är Turingmaskiner intressantare. Mängden av alla Turingmaskiner *är* nämligen uppräkningsbar!

Strängar Σ^* över ett alfabete Σ är uppräkningsbara, eftersom det finns bara en ändlig mängd strängar med längden 0, 1, och så vidare. Det betyder alltså att vi kan skapa en mappning till dem från mängden av naturliga tal.

Eftersom vi vet att vi kan representera Turingmaskiner som strängar (än en gång, tänk "källkod"), och strängarna tydligen är uppräkningsbara är mängden av Turingmaskiner uppräkningsbar också!

Mängden av alla språk (skrivet $\mathcal{L}$) över ett alfabete är dock inte uppräkningsbar. Det skall vi visa snart, men innebörden är alltså att det inte går att skapa en Turingmaskin för varje språk – vissa problem är omöjliga att lösa algoritmiskt!

Bevisidé

Tanken är att vi visar att $\mathcal{L}$ är av samma storlek som $\mathcal{B}$, mängden av alla oändliga binära (= bestående av ettor och nollor) sekvenser. Om $\mathcal{B}$ inte är uppräkningsbar, kan heller inte $\mathcal{L}$ vara det.

Det går att bevisa att $\mathcal{B}$ inte är uppräkningsbart genom att följa exakt samma resonemang som vi gjorde när vi visade att $\mathcal{R}$ inte är uppräkningsbart – nu när vi skapar x ser vi till att n:te positionen i x är motsatsen till n:te positionen i $f(n)$. Klart!

Kvar är då att visa att $\mathcal{L}$ är lika stort som $\mathcal{B}$. Om vi kan hitta en mappning mellan mängderna är vi klara.

Mappning mellan $\mathcal{L}$ och $\mathcal{B}$

Låt $\Sigma^* = \{s_1, s_2, \ldots\}$. Varje språk $A \in \mathcal{L}$ har en unik sekvens i $\mathcal{B}$ som skapas på följande sätt:

Den i:te biten i sekvensen är 1 om $s_i \in A$, 0 om $s_i \notin A$. Denna sekvens kallas för den *karaktäristiska sekvensen* för A.

Vi har nu hittat vår mappning mellan $\mathcal{L}$ och $\mathcal{B}$! $\mathcal{L}$ är alltså inte uppräkningsbart, men Turingmaskinerna är det.

Sammanfattning

Idag har vi gått igenom varför vi inte kan avgöra vilka program som kommer att avslutas och att det finns vissa problem som inte kan lösas av någon beräkningsmodell, någonsin.

De som har hängt med i nördnyheterna, vet att ett företag påstår sig ha skapat en kvantdator som de skall presentera den här veckan. Även om det vore otroligt häftigt om den fungerar, så gäller tyvärr fortfarande att det finns problem den inte kan lösa – vår kära Turingmaskin kan nämligen simulera även en kvantdator...