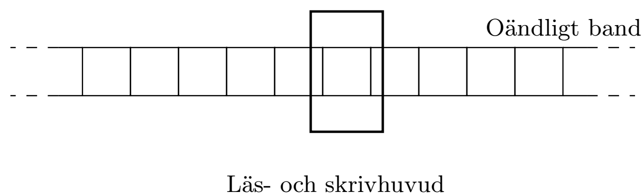
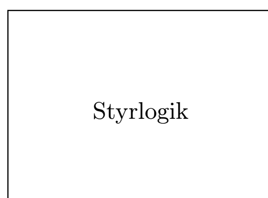
Föreläsning 9: Turingmaskiner och oavgörbarhet

Turingmaskinen

Den maximalt förenklade modell för beräkning vi kommer använda är *turingmaskinen.*
Data är ett oändligt långt band där nollor och ettor står skrivna:

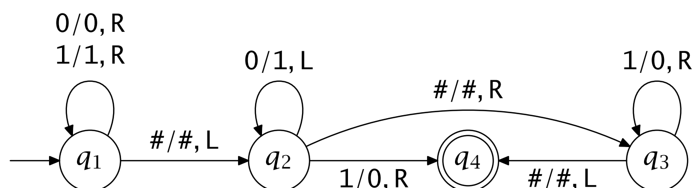




Läsning och skrivning kan bara ske en siffra i taget där huvudet står och huvudet kan bara flytta sig en position i taget.
Logiken för hur huvudet flyttas och vad det skriver består av ett ändligt antal tillstånd och övergångar (en *automat*).

Exempel på turingmaskin

Följande turingmaskin läser talet x på binär form från bandet och ändrar det till
$\max(x-1, 0)$.



Notation:
- Cirklar svarar mot tillstånd
- Dubbla cirklar är accepterande tillstånd
- Pilar betecknar övergångar:
- $a/b, L$ ska tolkas "om läshuvudet läser a, följ övergången, skriv b och flytta läshuvudet ett steg åt vänster"
  ($a/b, R$ analogt fast förflyttning åt höger)

Pilen som inte utgår från något tillstånd markerar vilket tillstånd turingmaskinen startar i.

Regler för turingmaskinen

- Automaten börjar i starttillståndet.
- Då står läshuvudet på första symbolen i indatasträngen. Indata omges av blanka positioner (betecknas #).
- Det får inte finnas flera övergångar med samma lässymbol från samma tillstånd.
- Om en TM hamnar i ett accepterande tillstånd avslutas beräkningen och "Ja" returneras.
- Om en TM hamnar i ett läge där det inte finns någon matchande övergång avbryts beräkningen och "Nej" returneras.

Formalisering av turingmaskinen

En turingmaskin beskrivs fullständigt av

- Alfabetet Σ (måste vara ändligt)
- Mängden Q av tillstånd (måste vara ändlig)
- Starttillståndet $q_0 \in Q$
- Mängden $F \subseteq Q$ av accepterande tillstånd
- Övergångsrelationen
  $\Delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{L, R, S\}$

Denna beskrivning motsvarar programkoden i ett program skrivet i ett vanligt programmeringsspråk.

Churchs tes

Churchs tes: *Varje algoritmiskt problem som kan lösas med något program skrivet i något språk kört på någon dator kan också lösas med en turingmaskin.*

- Stopp-problemet är oavgörbart även för turingmaskiner.
- Turingmaskinen kan användas som beräkningsmodell vid resonemang om oavgörbarhet.
- Stopp-problemet är oavgörbart för varje beräkningsmodell som är kraftfull nog att simulera en turingmaskin. (D.v.s. är *turingekvivalent*.)

Beräkningsmodellen RAM är turingekvivalent liksom alla moderna programspråk.

Lika kraftfulla varianter av turingmaskinen

- Annat (ändligt) alfabet.

- Separat utmatningsband.
- Flera arbetsband.
- Flera läshuvuden.
- Halvoändligt arbetsband.

Alla dessa varianter är ekvivalenta med den vanliga turingmaskinen i följande mening:
Körtiden skiljer sig bara åt med högst en
polynomisk faktor.

Icke-deterministiska turingmaskiner

- Det kan finnas flera övergångar med samma lässymbol. Då får turingmaskinen göra ett *icke-deterministiskt* val.
- Om någon följd av val leder till ett accepterande tillstånd säger vi att maskinen
accepterar.
- Om det inte finns någon följd av val som leder fram till ett accepterande tillstånd säger vi att maskinen *inte accepterar.*

Icke-determinism, forts.

Icke-deterministiska turingmaskiner kan användas för att definiera **NP**:
Denna klass innehåller precis de problem där det finns någon accepterande beräkning som tar polynomisk tid.
NP = **N**on-deterministic **P**olynomial time
Man tror att icke-deterministiska turingmaskiner är kraftfullare än deterministiska, d.v.s. att $\mathbf{P} \neq \mathbf{NP}$.

Cooks sats

Cooks sats säger att problemet SAT är **NP**-fullständigt.
Indata är en logisk formel Φ med booleska variabler och problemet är att avgöra om formeln är satisfierba eller inte.
Bevis av Cooks sats:
SAT $\in$ **NP** eftersom det givet en variabeltilldelning är lätt att kolla om formeln är satisfierad.
Vi måste visa att SAT är **NP**-svårt, d.v.s. att om $Q' \in \mathbf{NP}$ så $Q' \leq_P$ SAT.
Då $Q' \in \mathbf{NP}$ finns det en icke-det. turingmaskin M som accepterar språket Q' i tid högst kn^c där n är indatas längd (k och c är konstanter).
Bevisidé:
Konstruera en logisk formel som är satisfierbar precis då M accepterar indatasträngen.
Vi antar att M har ett halvoändligt band och alfabetet $\{0, 1, \#\}$.
Numrera M:s tidssteg från 1 till kn^c. Vid varje tidpunkt t beskrivs beräkningen av

- läshuvudets position
- aktuellt tillstånd q
- bandets innehåll på positionerna $1 - kn^c$

I den logiska formeln inför vi variabler med följande betydelse:

x_{qt}	$q \in Q$, $1 \le t \le kn^c$
y_{ijt}	$i \in \{0, 1, \#\}, 1 \le j \le kn^c, 1 \le t \le kn^c$
z_{jt}	$1 \le j \le kn^c, 1 \le t \le kn^c$

Tolkning:

$x_{qt} = 1$	omm då M är i tillstånd q vid tid t
$y_{ijt} = 1$	omm tecken i står i ruta j vid tid t
$z_{jt} = 1$	omm huvudet står på ruta j vid tid t

Att det existerar någon beräkning i tid kn^c som gör så att $M(a_1, \dots, a_n)$ accepterar svarar mot att

1. Beräkningen börjar med $a_1, \dots, a_n$
2. x, y, z beskriver en giltig beräkning
3. Beräkningen accepterar, d.v.s. den slutar i ett accepterande tillstånd

Alla dessa villkor kan man beskriva som en enda SAT-formel vars storlek är polynomisk i n.
Detta visar att det finns en reduktion $Q' \le_P SAT$ för varje **NP**-fullständigt problem Q'. Ur detta följer att SAT är **NP**-fullständigt.

Oavgörbarhet

Ett ja/nej-problem är *avgörbart* om det finns någon algoritm som korrekt löser det i ändlig tid för varje instans.
Motsatsen är om problemet är *oavgörbart*. Då kan det av logiska skäl inte finnas någon algoritm som alltid löser det.
Vanligen finns det algoritmer som löser problemet för en del indata, men det finns alltid indata som de inte klarar av.
Om utdata är något annat än ja/nej pratar man on *beräkningsbara* och *inte beräkningsbara* problem.

Ex. 1: Ordbildningsproblemet

Givet en mängd med ordpar $\{(x_i, y_i)\}$.
Finns det någon talföljd $a_1, a_2, \dots, a_k$ så att $x_{a_1} x_{a_2} \cdots x_{a_k} = y_{a_1} y_{a_2} \cdots y_{a_k}$?
Exempel:

$$\{\underbrace{(abb, bbab)}_1, \underbrace{(a, aa)}_2, \underbrace{(bab, ab)}_3, \underbrace{(baba, aa)}_4, \underbrace{(aba, a)}_5\}$$

har lösningen $a = [2, 1, 1, 4, 1, 5]$:

$$\underbrace{a}_{x_2}\underbrace{abb}_{x_1}\underbrace{abb}_{x_1}\underbrace{baba}_{x_4}\underbrace{abb}_{x_1}\underbrace{aba}_{x_5} = \underbrace{aa}_{y_2}\underbrace{bbab}_{y_1}\underbrace{bbab}_{y_1}\underbrace{aa}_{y_4}\underbrace{bbab}_{y_1}\underbrace{a}_{y_5}$$

medan
$\{(bb, bab), (a, aa), (bab, ab), (baba, aa), (aba, a)\}$
saknar lösning.

Ex. 2: Stopp-problemet

Givet ett program P och indata X till P.
Stannar programmet P när det körs på indata X?
Vilket programspråk P är skrivet i spelar ingen roll; problemet är oavgörbart för alla tillräckligt kraftfulla språk.

Ex. 3: Problem från tillämpningar

Programverifikation
: Givet ett program P och en specifikation S för vad programmet ska göra på olika indata, uppfyller programmet specifikationen?

Minimering av program
: Kan en viss rad i programmet P nås för något indata?

Alla dessa problem är oavgörbara p.g.a. nära släktskap med stopp-problemet. I tillämpningar får man nöja sig med att lösa förenklade versioner av problemen ovan.

Bevis för avgörbarhet / oavgörbarhet

Tekniker för bevis av oavgörbarhet:

Direkt bevis
: Ge ett direkt logiskt bevis för varför ditt problem är oavgörbart.

Reduktion
: Reducera ett känt oavgörbart problem till ditt problem. Om reduktionen är beräkningsbar måste ditt problem vara oavgörbart.

Bevis för avgörbarhet:

- Ge en algoritm som avgör problemet, visa att den är korrekt och att körtiden är ändlig.

Stopp-problemet är oavgörbart

Antag att det finns en algoritm $\mathit{Stopp}(P, X)$ som avgör stopp-problemet. Betrakta följande program:

```
M(P)
(1)     if Stopp(P, P)
(2)         gå in i oändlig loop
(3)     else
(4)         return
```

Var händer när $M(M)$ körs?
$M(M)$ stannar: Då måste $\mathit{Stopp}(M, M)$ vara falsk för att return ska nås — orimligt.
$M(M)$ stannar inte: Då är $\mathit{Stopp}(M, M)$ sann och programmet kommer aldrig lämna den oändliga loopen — orimligt.
Motsägelse; algoritmen $\mathit{Stopp}(P, X)$ kan alltså inte finnas $\Rightarrow$ stopp-problemet är oavgörbart.

Exempel på reduktion

Nästan alla varianter av stopp-problemet är
oavgörbara, t.ex. följande:
Stannar programmet $\boldsymbol{P}$ på alla indata?
Att det inte kan finnas en algoritm $\mathit{StoppAlla}(P)$ som avgör detta problem visar följande reduktion:

```
STOPP(P, X)
(1)        Konstruera programmet Q enligt
               Q(Y)
                   if X = Y
                       P(X)
                   else
                       stanna direkt
(2)        return StoppAlla(Q)
```

Om $\mathit{StoppAlla}(\cdot)$ fanns skulle stopp-problemet vara avgörbart; orimligt.