# INF 4130
# 8. oktober 2009

Stein Krogdahl

---

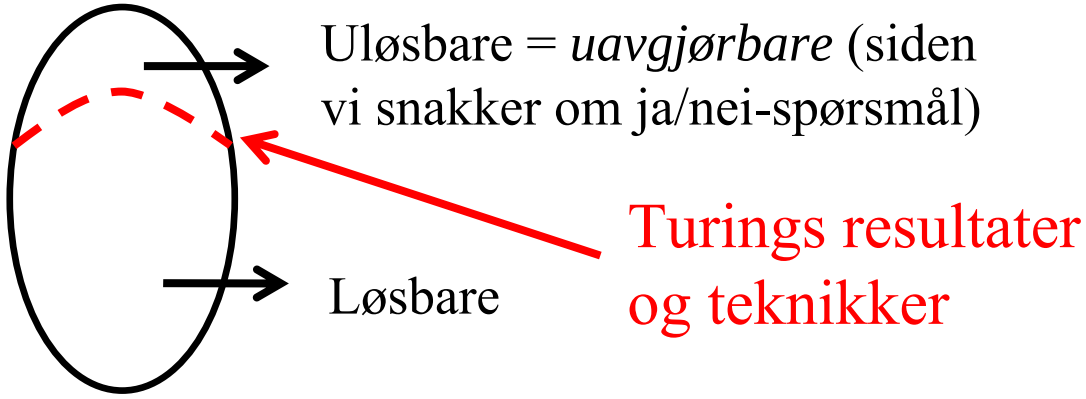
- Dagens tema: Uavgjørbarhet
  - Dette har blitt framstilt litt annerledes tidligere år
  - Se Dinos forelesninger fra i fjor.
  - I år: Vi tenker mer i programmer enn i Turing-maskiner
  - Pensum blir foilene, og antakeligvis litt fra kompendiet.
  - Morsom bok:  David Harel, ”Algorithmics”
- Neste uke: NP-kompletthet
  - Hvilke problemer kan løses, men ikke effektivt?

# Litt historie: Hva kan ikke gjøres

Fra 1900 og utover vokste feltet som kalles metamatematikk (matematiske studier av matematikken selv, dens muligheter og begrensninger).

På 1930-tallet kom en del resultater omkring eksistens/ikke-eksistens av algoritmer for ulike problemer.

- Kurt Gödel (1931): Ufullstendighetsresultater.
- Alan Turing (1936): «*On computable numbers, with an application to the Entscheidungsproblem*».

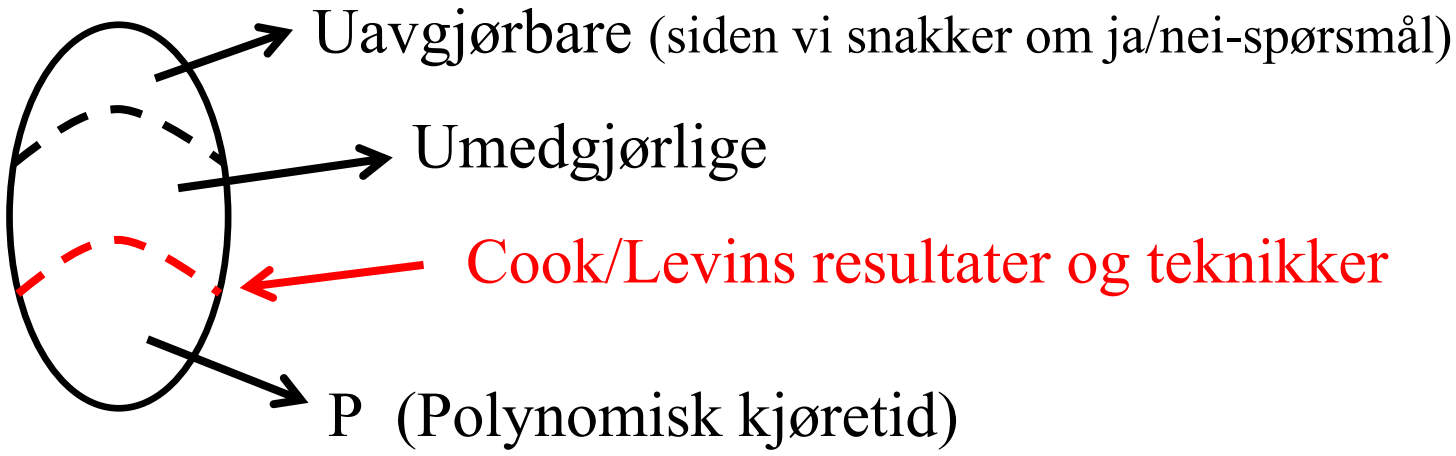


# Historie: Hva kan gjøres, men ikke raskt (neste forelesninger)

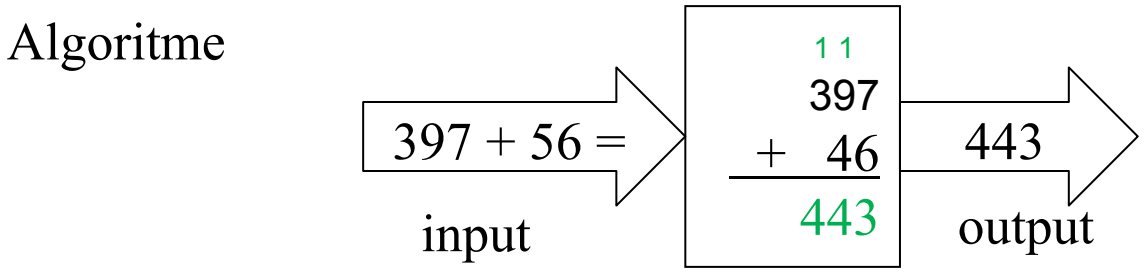
Edmonds: «*min algoritme har polynomisk kjøretid, den trivielle bruker eksponensiell tid!*»

Vi har altså nå resultater omkring problemer som har algoritmer med polynomisk kjøretid.

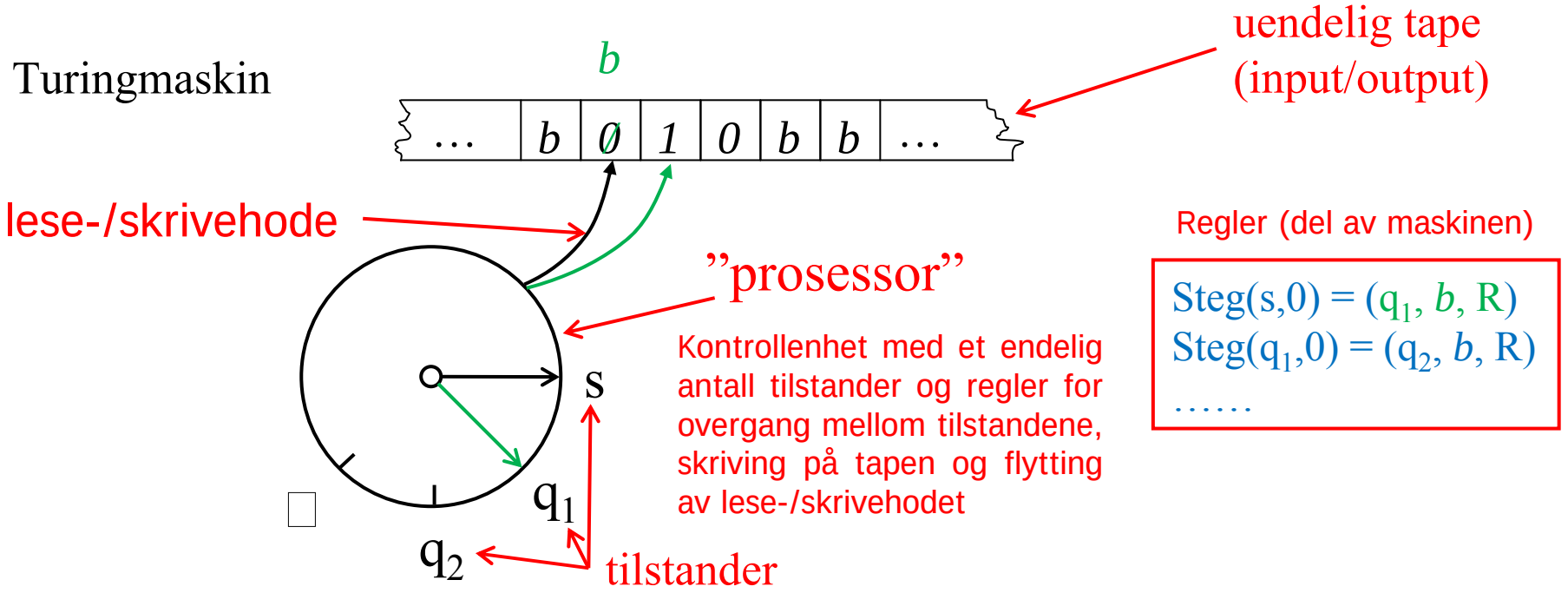
Cook / Levin (1972): NP-completeness.



# Algoritme = en Turing-maskin




beregning ↔ regler



# Church's tese

**Algoritme** tilsvarer **Turing-maskin**

Turing-maskiner kan beregne enhver funksjon som kan beregnes av en algoritme, et dataprogram eller en datamaskin.

**Programmeringsspråks beregningskraft**

Et programmeringsspråk (Java, C, Lisp, …) er Turing-komplett om det kan gjøre de samme beregninger som Turing-maskiner. Kan vi implementere en Turing-maskin i språket vårt, er det det.

**Universelle datamaskinmodeller**

En datamaskinmodell er Turing-komplett om den kan gjøre de samme beregninger som en Turing-maskin. McCullok og Pitts har vist at nevrale nett kan simulere Turing-maskiner.

**Uberegnbarhet**

Hvis en funksjon *f* ikke kan beregnes av en Turing-maskin, så finnes det ingen datamaskin som kan beregne *f*.

# Sammenlikning mellom Turing-maskin og program/algoritme

- Vi skal ikke knytte oss like mye til Turing-maskiner som tidligere år, men i stedet tenke mer i programmer/algoritmer.
- Men merk: Det fine med Turing-maskiner er at det er
  - opplagt hva et ”steg” er at det er opplagt at
  - et steg kan utføres i endelig tid
  - den arbeider med en endelig datamengde i hvert skritt
  - om det er tvil om detaljer kan man alltid gå tilbake å studere den fullt definerte Turing-maskinen.
- Fordelen med å bruke programmer (= algoritmer) er
  - vi er på vårt ”hjemmeområde”,
  - at vi allerede har en grei terminologi
  - at vi veldig lett ser hva enkle grep fører til

# Turing-maskiner og programmer/algoritmer

- Imidlertid:
  - For at et program skal ha samme ”styrke” som en Turing-maskin må programmeringsspråket og utførelsesdelen ha litt spesielle egenskaper
  - Må ha noe som tilsvarer Turing-maskinens ”tape” som alltid kan utvides etter behov, men som alltid bare har data på endelig del.
- Derfor:
  - Språket må ha variable som kan ha så stor verdi man bare vil
    - og en eveig løkke kan derfor sette ”i := i + 1” inne i løkka uten at det blir noe overflow-tull eller liknende
  - Bør kutte ut reelle tall, siden de gjør det uklart hva et steg er!
  - Om OO-språk: Må kunne lage så mange ”objekter” e.l. man ønsker.
  - Alle programmer skal starte med en viss endelig input-fil
  - Leverer svar på en eller flere output-filer (som er tomme fra starten)
  - Vi må altså velge en passelig steg-enhet i prog.-språket: F.eks. hente opp en operand, gjøre en aritmetisk operasjon, gjøre et assignment, …
  - Vi vil vanligvis velge steget slik at output/input av ett tegn er ett steg.

# Universell Turing-maskin

- En viktig ting med Turing-maskiner (i bevis etc.) er at vi kan lage en ”universell Turing-maskin”.
  - Dette er en Turing-maskin som kan ta *beskrivelsen* av en hvilken som helst Turing-maskin T som input (først på tapen)
  - og så kan utføre (simulere) T, der input til T ligger bak baskrivelsen av T på tapen
- Dette tilsvarer i programmerings-verdenen at vi kan skrive en kompilator/interpretator som et vanlig program
  - Dette er selvfølgelig for oss, og vi behøver derfor ikke mase mye med det, og kan derfor tenke lettere.
  - Men i 1936 måtte Turing mase mye med dette!

# Hva er et ”problem”?

- Et problem *P* består av en mengde *instanser* $\{P_1, P_2, \ldots\}$ som er de egentlige konkrete problemene, som hver har et svar
- Vi skal her først og fremst snakke om ”desisjons-problemer”, altså problemer der hver enkelt instans har et svar *Ja* eller *Nei*
- En *koding* av et problem P, *K(P)*, er en tekstlig sekvensiell (og en-en-tydig) representasjon av hver av instansene til problemet.
- En algoritme A *som løser (eller avgjør) P* er en algoritme som ved input $K(P_i)$ *etter endelig tid* leverer riktig svar på $P_i$, for alle $P_i$.
- Et problem P er *avgjørbart* om det *finnes* en algoritme som løser P.
  - Merk her at det her *ikke* forlanges at vi skal kunne *vise fram* denne algoritmen, men bare at en algoritme slik finnes.

# Observasjon:

## Et problem som har et endelig antall instanser er avgjørbart

Bevis:

- Vi antar at instansene for P er $P_1$, $P_2$, … , $P_N$
- Følgende algoritme vil da løse problemet:

```
if  <input = P1> then <levér svaret på P1> else
if  <input = P2> then <levér svaret på P2> else
…
if  <input = PN> then <levér svaret på PN> ;
```

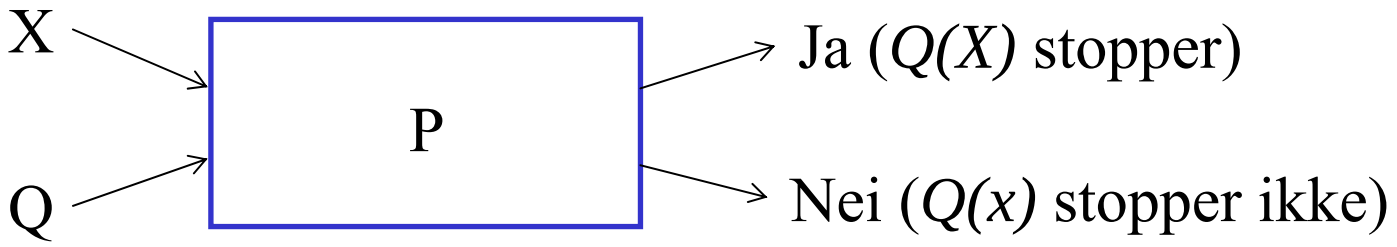
Merk at vi vet at denne algoritmen *finnes*, selv om vi ikke nødvendigvis vet alle de riktige svarene.

# Et problem som *ikke* er avgjørbart

- Stoppeproblemet ”STP” (eller HALT) er som følger:
  - Instans: En algoritme representert som et program, og en (endelig) input-streng X til dette.
  - Spørsmål: Vil programmet med input X stoppe, eller gå i evig løkke?
- Vi påstår at STP ikke er avgjørbart!
- Beviset er overraskende enkelt
  - Egentlig så enkelt at man lett får følelsen av at det hele er litt ”hokus-pokus-filiokus”.
- Vi skal bevise dette ved å anta at STP *er* avgjørbart
  - og vise at *denne antakelsen fører til en selvmotsigelse*.

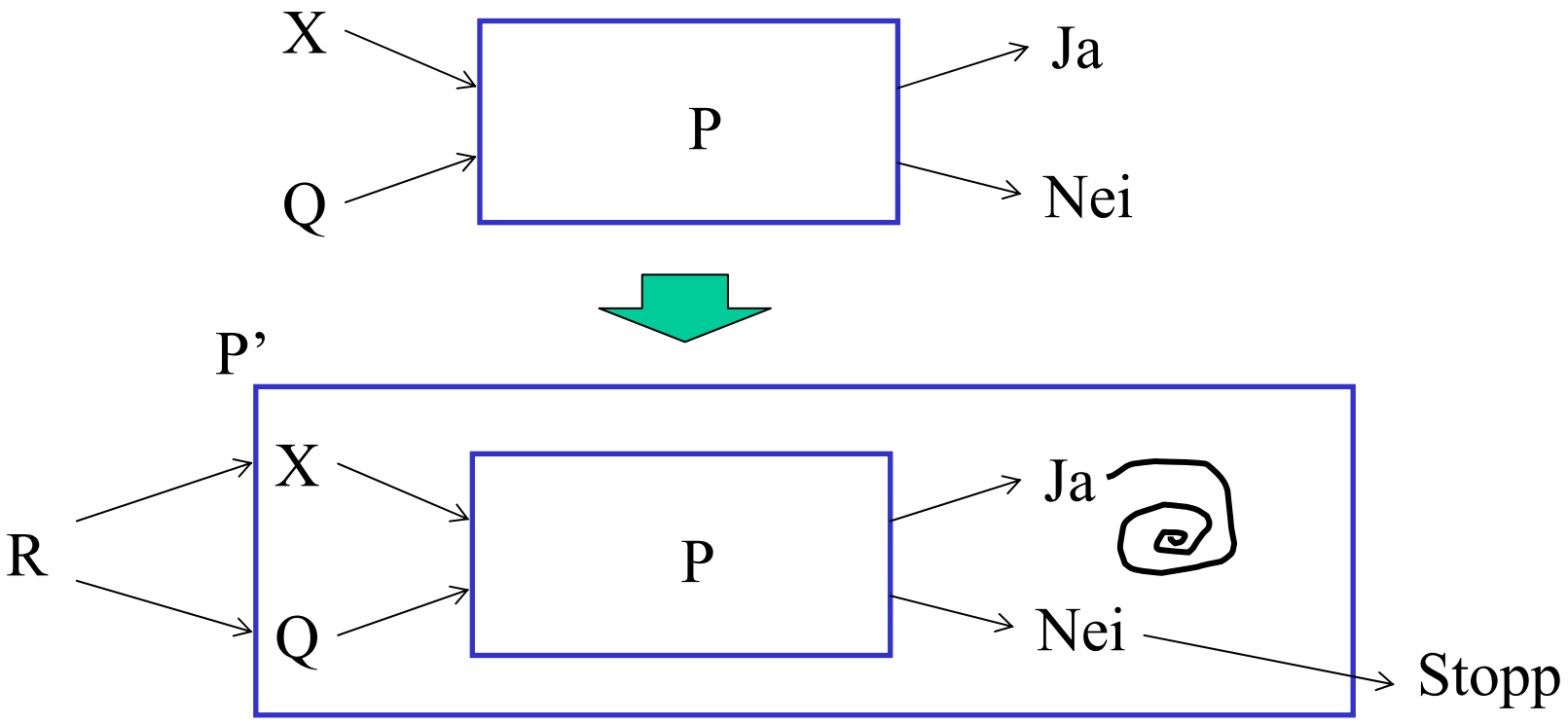
# Hokus:
## Bevis for at STP er uavgjørbart

- Vi *antar* at STP er avgjørbart
  - Altså at det finnes et program P som løser STP
  - Det vil si at om P får som input et vilkårlig program Q og input X til dette, så vil P (etter "endelig tid") komme med svaret *Ja* (altså, *Q(X)* vil stoppe) eller *Nei* (*Q(X)* vil gå i evig løkke)

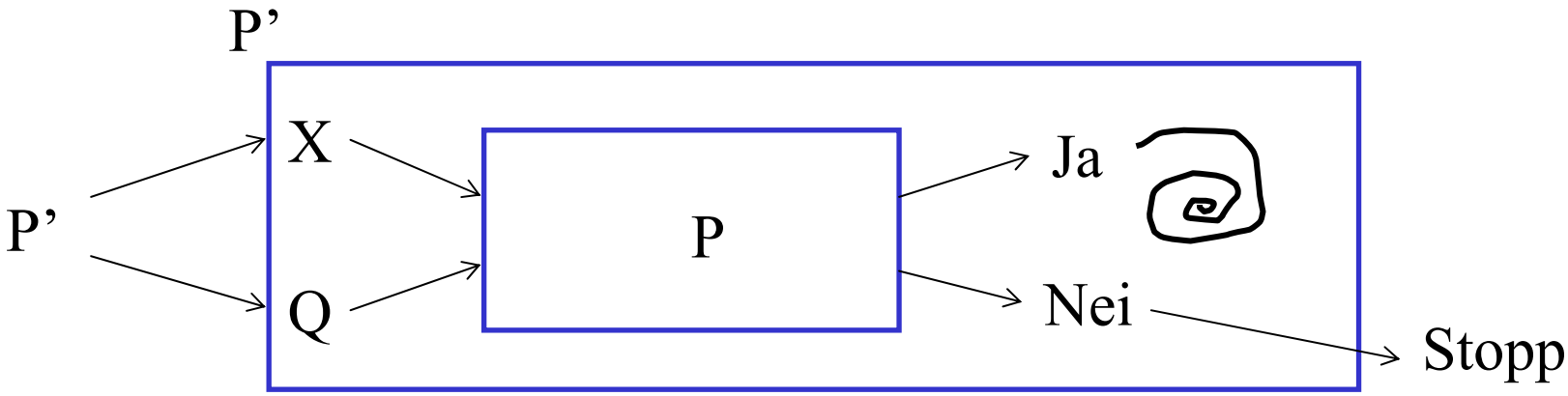


# Pokus:
## Mer bevis for at STP er uavgjørbart

- Ut fra P lager vi så et nytt program P', som har ett program R som input, og som lages som følger:



# Filiokus: Vi ser så på utførelsen P'(P')




- Det viser seg at denne utførelsen fører til en selvmotsigelse, slik:
  - Anta først P'(P') stopper. Da skulle P her svare *Ja*, men det vil jo si at P'(P') faktisk går i evig løkke, altså en motsigelse.
  - Anta så at P'(P') *ikke* stopper. Da skulle P her svare *Nei*, men det vil jo igjen bety at P'(P') faktisk *vil* stoppe, altså også her en motsigelse.
- Det betyr altså at P'(P') verken stopper eller ikke stopper.
- Denne motsigelsen betyr at vi må ha gjort en eller annen gal antakelse, og den eneste kandidaten er at *det finnes en algoritme som løser STP.*
- Altså er *den* antakelsen gal! Stoppeproblemet er uavgjørbart.

# Dette beviset kan også sees som et ”diagonaliserings-bevis”

- En viss type bevis kalles ”diagonalisering-bevis”
- Et slikt bevis brukes f.eks. til å vise at det er ”flere” reelle tall enn naturlige tall.
- Interesserte kan slå opp i Dino Karabegs foiler fra 24. september i fjor, eller i kompendiet.

# Finnes andre uavgjørbare problemer?

- Ja, i masse vis! En liste ligger f.eks. på:

  en.wikipedia.org/wiki/List_of_undecidable_problems

- Vi kan nevne:
  - Om BNF-grammatikker:
    - Gitt to BNF-grammatikker. Generer de samme språk?
    - Gitt en BNF-grammatikk. Er den flertydig?
  - Nesten "ethvert" spørsmål om programmer:
    - Vil en gitt variabel i et gitt program noen gang få verdien 2?
    - Vil et gitt program, ved input $x$ alltid levere verdien $x^2$
  - Diofantiske likninger
    - Gitt et polynom i variablene x, y, z, … med heltallige koeffisienter. Kan dette bli null for heltallige x, y, z, …?
  - En mengde såkalte "flisleggingsproblemer"

# Hvordan vise at et problem er uavgjørbart?

- Dette er greit nok når vi først har *ett* uavgjørbart problem
- Om vi ønsker å vise at R er uavgjørbart, gjør vi som følger:
  - Anta at vi *har* en algoritme P som løser (alle instanser av) R.
  - Forsøk så, med denne antakelsen, å løse stoppeproblemet (STP), ved å bruke P som en slags ”subrutine” e.l.
  - Om du klarer det, og siden STP fakstisk *er* uavgjørbart, vet vi at antakelsen var gal. *Altså er problemet R uavgjørbart.*
- Vi sier da at stoppeproblemet er ”reduserbart” til R
  - Det er jo vel og bra, men det er ”umulig” å huske hvilken vei reduksjonen skal gå for å få vist det man ønsker.
- Derfor anbefaler jeg heller å tenke slik:
  - Vis at ”dersom R kan avgjøres så kan også stoppeproblemet avgjøres”.
  - Men stoppeproblemet er uavgjørbart, og dermed er også R uavgjørbart.
  - må gjerne forsøke *begge veier* før man finner den riktige!

# Eksempel på uavgjørbarhets-bevis

- Vi vil vise at følgende problem R (forenklet STP) er uavgjørbart:
  - Instans: Et program
  - Spørsmål: Vil dette programmet stoppe om det får en tom input-fil?
- Etter ”skjemaet” antar vi at det finnes et program P som avgjør R.
- Vi må så vise at vi kan løse en ”generell” instans av STP (vi betegner den (Q, X)) ved hjelp av P. Vi omarbeider da Q til Q’ slik:
  - Legg til en første del som legger konstant-strengen X inn i en ny array A.
  - Omgjør alle input-setninger  i Q, slik at de i stedet leser fra arrayen A.
  - Vi ser da at Q’ med tomt input, etter initialiseringen av A vil kjøre og lage output akkurat som Q ville med input X.
- På Q’ bruker vi så programmet P. Vi kjører altså P(Q’)
  - Dette skulle (etter antakelsen om P) svare *Ja* om Q’ med tomt input vil stoppe, og ellers svare *Nei*.
  - Men derved ville vi altså kunne avgjøre om Q(X) vil stoppe, og vi ville altså kunne avgjøre stoppeproblemet.  Dette er feil, altså er R uavgjørbart.

# Konklusjon på forrige foil

- Stoppeproblemet er uavgjørbart enten instansene er et program P med input X, eller om instansene bare er et program som ikke tar input.

- Det kan ofte i bevis være like greit å bruke denne spesielle versjonen av stoppeproblemet, i stedet for den generelle versjonen.

- Og husk generelt: Ethvert problem som er vist å være uavgjørbart kan brukes i slike bevis i stedet for stoppeproblemet.

# Det finnes *enda* vanskeligere problemer enn de uavgjørbare

- For de uavgjørbare kan vi ha en algoritme som helt sikkert stopper og sier ”ja” om svaret er ja, men som går i løkke om svaret er nei:
  - Slik algoritme for stoppeproblemet:
  - Kjør algoritmen, og om den stopper skriv ut ”ja”.
- Det finnes enda vanskeligere problemer P:
  - P er uavgjørbart, men det finnes heller ingen algoritme som på alle ja-instanser innen endelig tid vil svare ja.
- Disse kan kalles ”sterkt uavgjørbare”. Et slikt er:
  - Instans: Et program
  - Spørsmål: Vil dette stoppe for *alle lovlige input*?