Modeller for beregning

DM573

Rolf Fagerberg

Mål

Målet for disse slides er at give overblik over studiet af modeller for beregning og deres styrker og begrænsninger.

Dette emne er et uddrag af kurset *DM553: Kompleksitet og beregnelighed* (6. semester).

De centrale spørgsmål

1. Hvad er beregning?
2. Hvad er et beregningsproblem?
3. Kan alle beregningsproblemer løses?
4. Hvor hurtigt kan de løses?

For at kunne studere spørgsmål 3 og 4 matematisk må man først lave præcise svar (definitioner) for 1 og 2.

De centrale spørgsmål

1. Hvad er beregning?
2. Hvad er et beregningsproblem?
3. Kan alle beregningsproblemer løses?
4. Hvor hurtigt kan de løses?

For at kunne studere spørgsmål 3 og 4 matematisk må man først lave præcise svar (definitioner) for 1 og 2.

I spørgsmål 3 er vi ligeglade med tidsforbruget. Beregningen skal bare stoppe (med korrekt svar) på et tidspunkt. Dette kaldes studiet af problemers **beregnelighed**.

I spørgsmål 4 er tidsforbruget i fokus. Hvilke problemer kan løses med et givet tidsforbrug? Dette kaldes studiet af problemers **kompleksitet**.

De centrale spørgsmål

1. Hvad er beregning?
2. Hvad er et beregningsproblem?
3. Kan alle beregningsproblemer løses?
4. Hvor hurtigt kan de løses?

For at kunne studere spørgsmål 3 og 4 matematisk må man først lave præcise svar (definitioner) for 1 og 2.

I spørgsmål 3 er vi ligeglade med tidsforbruget. Beregningen skal bare stoppe (med korrekt svar) på et tidspunkt. Dette kaldes studiet af problemers **beregnelighed**.

I spørgsmål 4 er tidsforbruget i fokus. Hvilke problemer kan løses med et givet tidsforbrug? Dette kaldes studiet af problemers **kompleksitet**.

Vi ser i dette sæt slides kun på beregnelighed.

De centrale spørgsmål

1. Hvad er beregning?
2. Hvad er et beregningsproblem?
3. Kan alle beregningsproblemer løses?
4. Hvor hurtigt kan de løses?

For at kunne studere spørgsmål 3 og 4 matematisk må man først lave præcise svar (definitioner) for 1 og 2.

I spørgsmål 3 er vi ligeglade med tidsforbruget. Beregningen skal bare stoppe (med korrekt svar) på et tidspunkt. Dette kaldes studiet af problemers **beregnelighed**.

I spørgsmål 4 er tidsforbruget i fokus. Hvilke problemer kan løses med et givet tidsforbrug? Dette kaldes studiet af problemers **kompleksitet**.

Vi ser i dette sæt slides kun på beregnelighed.

[Vi har tidligere set på asymptotisk analyse og notation, hvilket hører under kompleksitet.]

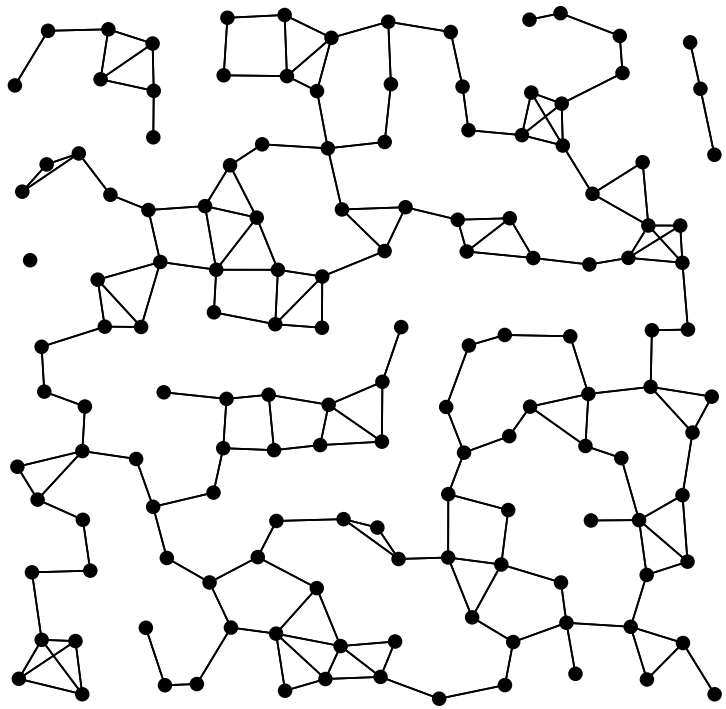
Hvad er et beregningsproblem?

Påstand: Sprog (= mængder af strenge over et alfabet) er en simpel og meget generel ramme til at formalisere beregningsproblemer.

Hvad er et beregningsproblem?

Påstand: Sprog (= mængder af strenge over et alfabet) er en simpel og meget generel ramme til at formalisere beregningsproblemer.

Problem 1: Graph connectivity.



- Input: En graf (= et netværk) og to knuder s og t i det.
- Output: Er der en sti mellem s og t i grafen?

Graph connectivity set som et sprog

Formalisering:

- n = antal knuder (prikker).
- m = antal kanter (streger).
- Knude-ID'er er heltal mellem 1 og n.
- En kant (u, v) er to knude-ID'er.
- s og t er to knude-ID'er.
- Alfabet = $\{0, 1, \#\}$

Graph connectivity set som et sprog

Formalisering:

- n = antal knuder (prikker).
- m = antal kanter (streger).
- Knude-ID'er er heltal mellem 1 og n.
- En kant (u, v) er to knude-ID'er.
- s og t er to knude-ID'er.
- Alfabet $= \{0, 1, \#\}$

Velformet streng for et graph connectivity problem:

$$n\#s\#t\#u_1\#v_1\#u_2\#v_2\#\ldots\#u_m\#v_m = 111011\#101\#\ldots\#101011$$

hvor alle tal er angivet binært og hvor $1 \leq s, t, u_i, v_i \leq n$ (alle i).

Graph connectivity set som et sprog

Formalisering:

- n = antal knuder (prikker).
- m = antal kanter (streger).
- Knude-ID'er er heltal mellem 1 og n.
- En kant (u, v) er to knude-ID'er.
- s og t er to knude-ID'er.
- Alfabet $= \{0, 1, \#\}$

Velformet streng for et graph connectivity problem:

$$n\#s\#t\#u_1\#v_1\#u_2\#v_2\#\ldots\#u_m\#v_m = 111011\#101\#\ldots\#101011$$

hvor alle tal er angivet binært og hvor $1 \leq s, t, u_i, v_i \leq n$ (alle i).

> Sproget **GraphConnectivity** er alle sådanne velformede strenge, hvor der er en sti mellem s og t i den tilsvarende graf.

Graph connectivity set som et sprog

Formalisering:

- n = antal knuder (prikker).
- m = antal kanter (streger).
- Knude-ID'er er heltal mellem 1 og n.
- En kant (u, v) er to knude-ID'er.
- s og t er to knude-ID'er.
- Alfabet $= \{0, 1, \#\}$

Velformet streng for et graph connectivity problem:

$$n\#s\#t\#u_1\#v_1\#u_2\#v_2\#\ldots\#u_m\#v_m = 111011\#101\#\ldots\#101011$$

hvor alle tal er angivet binært og hvor $1 \leq s, t, u_i, v_i \leq n$ (alle i).

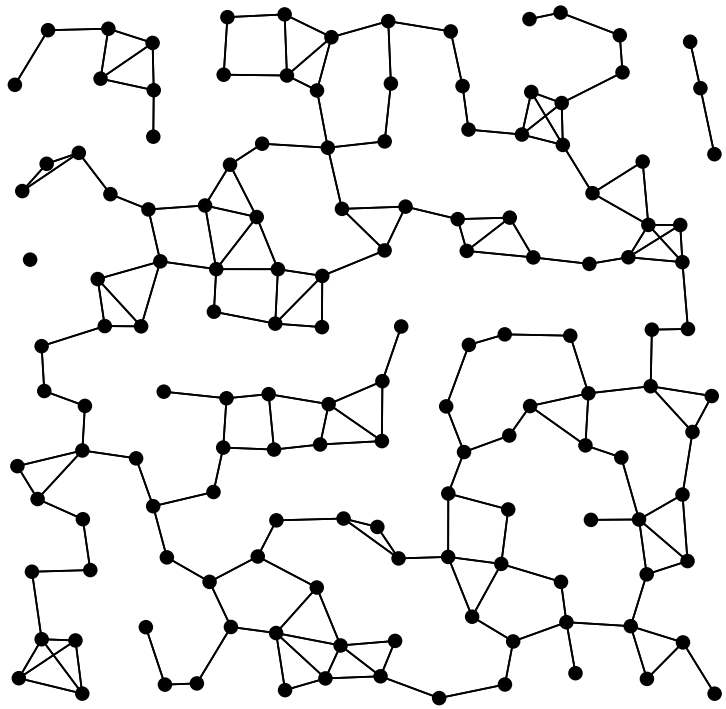
> Sproget **GraphConnectivity** er alle sådanne velformede strenge, hvor der er en sti mellem s og t i den tilsvarende graf.

Dvs. sproget består af alle problem-instanser, hvor svaret er ja.

Hvad er et beregningsproblem?

Påstand: Sprog er en simpel og meget generel ramme til at formalisere beregningsproblemer.

Problem 2: Shortest paths in graphs.



- Input: En graf (= et netværk) og to knuder s og t i det.
- Output: Hvor mange kanter (streger) er der på en korteste sti mellem s og t i grafen?

Shortest paths set som et sprog

Formalisering:

- n = antal knuder (prikker).
- m = antal kanter (streger).
- Knude-ID'er er heltal mellem 1 og n.
- En kant (u, v) er to knude-ID'er.
- s og t er to knude-ID'er.
- Stilængde k (et ikke-negativt heltal)
- Alfabet $= \{0, 1, \#\}$

Velformet streng for et shortest path problem:

$$n\#s\#t\#k\#u_1\#v_1\#u_2\#v_2\#\ldots\#u_m\#v_m = 111011\#\ldots\#101011$$

hvor alle tal er angivet binært og hvor $1 \leq s, t, u_i, v_i \leq n$ (alle i).

> Sproget **ShortestPaths** er alle sådanne velformede strenge, hvor der er en sti mellem s og t med højst k kanter i den tilsvarende graf.

Hvad er et beregningsproblem?

Påstand: Sprog er en simpel og meget generel ramme til at formalisere beregningsproblemer.

Problem 3: Sortering.

- Input: en række heltal $x_1, x_2, x_3, \ldots, x_n$
- Output: en permutation $p_1, p_2, p_3, \ldots, p_n$ af tallene $1, 2, 3, \ldots, n$ som opfylder at

$$x_{p_1} \leq x_{p_2} \leq x_{p_3} \leq \cdots \leq x_{p_n}.$$

[Permutation betyder ombytning. F.eks. er $2, 4, 1, 3, 5$ en permutation tallene $1, 2, 3, 4, 5$.]

Dvs. output angiver sorteret rækkefølge af input ved at p_1 er index på det tal i input, som skal stå først, p_2 er index på det tal i input, som skal stå derefter, og så videre.

Sortering set som et sprog

Formalisering:

- n = antal elementer.
- x_i = input element (heltal).
- p_i = index (heltal mellem 1 og n).
- Alfabet = $\{0, 1, \#\}$

Sortering set som et sprog

Formalisering:

- n = antal elementer.
- x_i = input element (heltal).
- p_i = index (heltal mellem 1 og n).
- Alfabet = $\{0, 1, \#\}$

Velformet streng for et sorteringsproblem:

$$n\#x_1\#x_2\#\ldots\#x_n\#p_1\#p_2\#\ldots\#p_n = 111011\#\ldots\#101011$$

hvor alle tal er angivet binært, og hvor $p_1, p_2, p_3, \ldots, p_n$ er en permutation af tallene $1, 2, 3, \ldots, n$.

Sproget **Sorting** er alle sådanne velformede strenge, hvor

$$x_{p_1} \leq x_{p_2} \leq x_{p_3} \leq \cdots \leq x_{p_n}.$$

Sprog formaliserer beregningsproblemer

Sprog formaliserer beregningsproblemer

- Graph connectivity er et eksempel på et decision problem. Her kan strenges medlemskab af sproget **GraphConnectivity** bruges direkte til at løse problemet for et givet input.

Sprog formaliserer beregningsproblemer

- Graph connectivity er et eksempel på et decision problem. Her kan strenges medlemskab af sproget **GraphConnectivity** bruges direkte til at løse problemet for et givet input.
- Shortest paths er et eksempel på et optimization problem. Her kan strenges medlemskab af sproget **ShortestPaths** bruges sammen med binær søgning over værdien af k til at løse problemet for et givet input.

Sprog formaliserer beregningsproblemer

- Graph connectivity er et eksempel på et decision problem. Her kan strenges medlemskab af sproget **GraphConnectivity** bruges direkte til at løse problemet for et givet input.
- Shortest paths er et eksempel på et optimization problem. Her kan strenges medlemskab af sproget **ShortestPaths** bruges sammen med binær søgning over værdien af k til at løse problemet for et givet input.
- Sorting er et eksempel på et construction problem. Her kan strenges medlemskab af sproget **Sorting** bruges sammen med generering af alle mulige permutationer $p_1, p_2, p_3, \ldots, p_n$ til at løse problemet for et givet input.

Sprog formaliserer beregningsproblemer

- Graph connectivity er et eksempel på et decision problem. Her kan strenges medlemskab af sproget **GraphConnectivity** bruges direkte til at løse problemet for et givet input.
- Shortest paths er et eksempel på et optimization problem. Her kan strenges medlemskab af sproget **ShortestPaths** bruges sammen med binær søgning over værdien af k til at løse problemet for et givet input.
- Sorting er et eksempel på et construction problem. Her kan strenges medlemskab af sproget **Sorting** bruges sammen med generering af alle mulige permutationer $p_1, p_2, p_3, \ldots, p_n$ til at løse problemet for et givet input.

Dette viser, at sprog er en simpel og generel ramme til at formalisere beregningsproblemer.

Sprog formaliserer beregningsproblemer

- Graph connectivity er et eksempel på et decision problem. Her kan strenges medlemskab af sproget **GraphConnectivity** bruges direkte til at løse problemet for et givet input.
- Shortest paths er et eksempel på et optimization problem. Her kan strenges medlemskab af sproget **ShortestPaths** bruges sammen med binær søgning over værdien af k til at løse problemet for et givet input.
- Sorting er et eksempel på et construction problem. Her kan strenges medlemskab af sproget **Sorting** bruges sammen med generering af alle mulige permutationer $p_1, p_2, p_3, \ldots, p_n$ til at løse problemet for et givet input.

Dette viser, at sprog er en simpel og generel ramme til at formalisere beregningsproblemer.

Dermed har vi et godt svar på spørgsmål 2:

Beregningsproblemer = medlemskab af sprog.

Hvad er beregning?

Dermed har vi også et godt svar på spørgsmål 1:

> Beregning = afgørelse af medlemskab af sprog.

Hvad er beregning?

Dermed har vi også et godt svar på spørgsmål 1:

> Beregning = afgørelse af medlemskab af sprog.

Studiet af modeller for beregning kan derfor sættes til at være studiet af modeller til at afgøre strenges medlemskab af sprog.

Hvad er beregning?

Dermed har vi også et godt svar på spørgsmål 1:

> Beregning = afgørelse af medlemskab af sprog.

Studiet af modeller for beregning kan derfor sættes til at være studiet af modeller til at afgøre strenges medlemskab af sprog.

Mange sådanne modeller er udviklet (1930-1970):

- DFA'er
- CFG'er
- RegEx'er
- ⋮
- Turing-maskiner

Modeller for beregning

Modellers beregningsstyrker kan siges at være, hvor mange sprog, de kan genkende.

Modeller for beregning

Modellers beregningsstyrker kan siges at være, hvor mange sprog, de kan genkende.

Man har f.eks. vist følgende om klasserne af sprog, som kan genkendes:

$$\text{RegEx} = \text{DFA} = \varepsilon\text{-NFA} \subsetneq \text{CFG'er} = \text{DFA-med-stak} \subsetneq \text{Turing-maskiner}$$

Modeller for beregning

Modellers beregningsstyrker kan siges at være, hvor mange sprog, de kan genkende.

Man har f.eks. vist følgende om klasserne af sprog, som kan genkendes:

$$\text{RegEx} = \text{DFA} = \varepsilon\text{-NFA} \subsetneq \text{CFG'er} = \text{DFA-med-stak} \subsetneq \text{Turing-maskiner}$$

Church-Turing Formodningen ("Thesis"):

> Vi kan ikke opfinde en beregningsmodel stærkere end Turing-maskiner.

Modeller for beregning

Modellers beregningsstyrker kan siges at være, hvor mange sprog, de kan genkende.

Man har f.eks. vist følgende om klasserne af sprog, som kan genkendes:

$$\text{RegEx} = \text{DFA} = \varepsilon\text{-NFA} \subsetneq \text{CFG'er} = \text{DFA-med-stak} \subsetneq \text{Turing-maskiner}$$

Church-Turing Formodningen ("Thesis"):

> Vi kan ikke opfinde en beregningsmodel stærkere end Turing-maskiner.

Støtte for formodningen: man har opfundet utallige modeller med samme styrke som Turing-maskiner, men ingen stærkere.

Hvorfor hedder det “Turing maskine”?

Hvorfor hedder det "Turing maskine"?

Alan Turing (1912–1954)

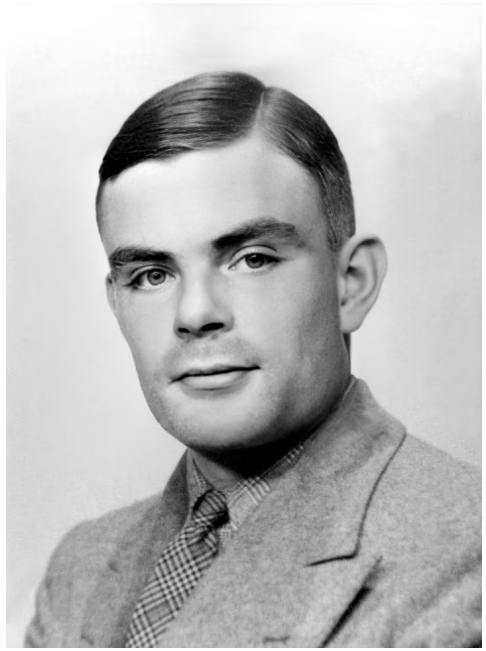


Foto: ARCHIVIO GBB, Redux

Engelsk matematiker. Studerede beregning og beregnelighed, logik, AI, philosofi og teoretisk biologi.

Kaldes grundlæggeren af datalogi. "Nobel-prisen" i datalogi hedder Turing-prisen.

Definerede Turing maskiner i 1937.

Hvorfor hedder det "Turing maskine"?

Alan Turing (1912–1954)

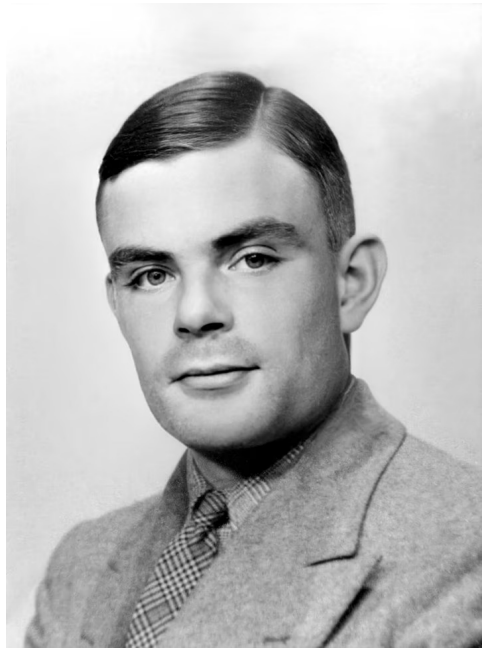


Foto: ARCHIVIO GBB, Redux

Engelsk matematiker. Studerede beregning og beregnelighed, logik, AI, philosofi og teoretisk biologi.

Kaldes grundlæggeren af datalogi. "Nobel-prisen" i datalogi hedder Turing-prisen.

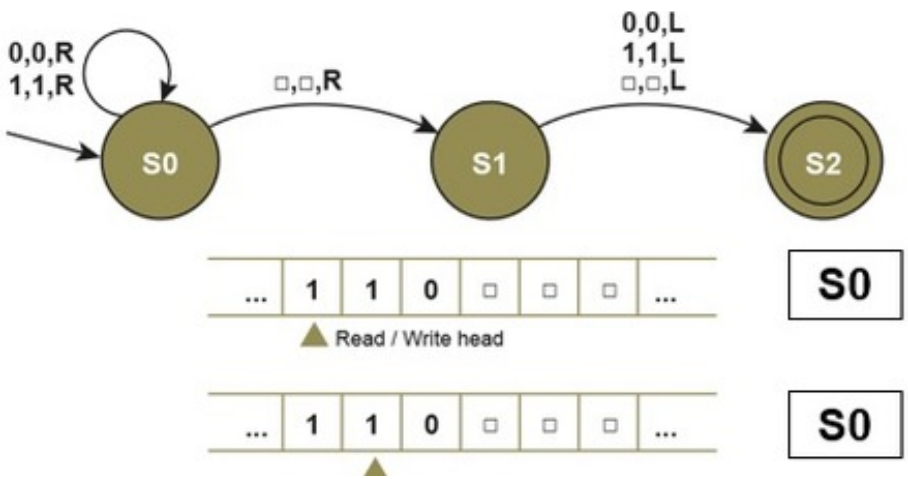
Definerede Turing maskiner i 1937. Og udrettede imponerende meget andet.

Turing maskiner

1) En read/write hukommelse: et bånd med tegn ($\sim$ en uendelig, skrivbar streng).

2) En generaliseret version af en DFA:

- Læser tegnet på nuværende position og vælger mellem kanterne ud af nuværende state (lige som en DFA).
- En kant har også en handling: **skriver** tegn på bånd og flytter position 1/-**1** (**nyt** ift. en DFA).



[Figure by hazza488 at quizlet.com]

Turing maskiner og sprog

Turing maskine **genkender** en streng, hvis:

1. Dens DFA når frem til en accept state, når den startes med strengen på bånd.

Turing maskiner og sprog

Turing maskine **genkender** en streng, hvis:

1. Dens DFA når frem til en accept state, når den startes med strengen på bånd.

To andre muligheder:

2. Maskinen går i stå i en ikke-accept state.
3. Maskinen kører evigt.

Turing maskiner og sprog

Turing maskine **genkender** en streng, hvis:

1. Dens DFA når frem til en accept state, når den startes med strengen på bånd.

To andre muligheder:

2. Maskinen går i stå i en ikke-accept state.
3. Maskinen kører evigt.

En Turing maskine M **bestemmer** et sprog L hvis:

- M stopper på alle input strenge (dvs. 3 ovenfor opstår aldrig).
- M genkender $s \Leftrightarrow s \in L$.

Turing maskiner og sprog

Turing maskine **genkender** en streng, hvis:

1. Dens DFA når frem til en accept state, når den startes med strengen på bånd.

To andre muligheder:

2. Maskinen går i stå i en ikke-accept state.
3. Maskinen kører evigt.

En Turing maskine M **bestemmer** et sprog L hvis:

- M stopper på alle input strenge (dvs. 3 ovenfor opstår aldrig).
- M genkender $s \Leftrightarrow s \in L$.

Et sprog (dvs. et problem) kaldes **beregneligt**, hvis der er en Turing maskine, som bestemmer det.

Universal Turing maskine

Turing viste, at man kan lave én, fast Turing maskine T_U, som kan udføre det samme som **enhver** anden Turing maskine T, hvis blot den som input (udover input strengen s) også får en passende beskrivelse af T.

Universal Turing maskine

Turing viste, at man kan lave én, fast Turing maskine T_U, som kan udføre det samme som **enhver** anden Turing maskine T, hvis blot den som input (udover input strengen s) også får en passende beskrivelse af T.

> Bemærk ligheden til vores dages computere: én, fast CPU, som læser både et program og et input fra hukommelsen.

Universal Turing maskine

Turing viste, at man kan lave én, fast Turing maskine T_U, som kan udføre det samme som **enhver** anden Turing maskine T, hvis blot den som input (udover input strengen s) også får en passende beskrivelse af T.

> Bemærk ligheden til vores dages computere: én, fast CPU, som læser både et program og et input fra hukommelsen.

Dvs. i stedet for at argumentere om, hvad Turing maskiner kan, er det lige så godt at argumentere om, hvad **programmer** (beskrivelser af T) kan, når de udføres på T_U.

Kan alle beregningsproblemer løses?

Kan alle beregningsproblemer løses?

Nej.

Kan alle beregningsproblemer løses?

Nej.

Halting problemet:

Input: Et program (en streng) P og en streng I.

Output: True, hvis P stopper på input I, ellers False.

Kan alle beregningsproblemer løses?

Nej.

Halting problemet:

Input: Et program (en streng) P og en streng I.

Output: True, hvis P stopper på input I, ellers False.

Turing viste, at intet program kan løse halting problemet.

Kan alle beregningsproblemer løses?

Nej.

Halting problemet:

Input: Et program (en streng) P og en streng I.

Output: True, hvis P stopper på input I, ellers False.

Turing viste, at intet program kan løse halting problemet.

Beviset kan siges at være en variant af "liars paradox":

Denne sætning er falsk.

(Sætningen ovenfor kan hverken være sand eller falsk.)

Kan alle beregningsproblemer løses?

Antag, at der findes et program Halting(P, I), som løser halting problemet. Dvs. Halting(P, I) returnerer True hvis P stopper på input I, ellers returneres False.

Kan alle beregningsproblemer løses?

Antag, at der findes et program Halting(P, I), som løser halting problemet. Dvs. Halting(P, I) returnerer True hvis P stopper på input I, ellers returneres False.

Vi kan så lave følgende program Z:

```
ProgramZ(String x)
  if Halting(x, x):
    while True: # loop forever
      pass
  else:
    stop
```

Kan alle beregningsproblemer løses?

Antag, at der findes et program Halting(P, I), som løser halting problemet. Dvs. Halting(P, I) returnerer True hvis P stopper på input I, ellers returneres False.

Vi kan så lave følgende program Z:

```
ProgramZ(String x)
  if Halting(x, x):
    while True: # loop forever
      pass
  else:
    stop
```

Hvad sker der, når programmet Z gives input Z? Der er to muligheder:

Case 1: Program Z stopper på input Z. Så vil Halting returnere Sandt på input (Z, Z). Derfor looper Z evigt (se if-delen). Modstrid.

Case 2: Program Z looper evigt på input Z. Så vil Halting returnere Falsk på input (Z, Z). Derfor stopper program Z på input Z (se else-delen). Modstrid.

Kan alle beregningsproblemer løses?

Mange andre problemer kan ikke løses af programmer (dvs. ved beregning på Turing maskiner). For eksempel:

Refactoring problemet:

Input: To programmer (strenge) P_1 og P_2.

Output: True, hvis P_1 og P_2 opfører sig ens på alle input.

Kan alle beregningsproblemer løses?

Mange andre problemer kan ikke løses af programmer (dvs. ved beregning på Turing maskiner). For eksempel:

Refactoring problemet:

Input: To programmer (strenge) P_1 og P_2.

Output: True, hvis P_1 og P_2 opfører sig ens på alle input.

Faktisk er antallet af programmer tælleligt, men antallet af sprog er overtælleligt. Så der er (mange) flere sprog end programmer, så mange sprog er ikke beregnelige (kan ikke bestemmes af Turing maskiner).

Filosofi, teori, praktik

Bemærk, at under arbejdet med disse filosofiske og teoretiske centrale spørgsmål (hvad er beregning, kan alt beregnes) er der opstået mange praktisk anvendelige modeller:

- RegEx'er bruges til at formulere søgning i strenge (og DFA'er til at udføre disse søgninger).
- Grammatikker bruges til at definere programmeringssprog og lave tilhørende parsningsalgoritmer og compilere.
- State machines (generaliserede DFA'er) bruges til at modellere processer i mange sammenhænge.