# Litt om kompleksitet

## IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2024

# Oversikt

# Oversikt

- Så langt i kurset har vi snakket mye om ressursbruken av en gitt algoritme

# Oversikt

- Så langt i kurset har vi snakket mye om ressursbruken av en gitt algoritme
- Vi har snakket mye om kompleksiteten til løsninger på problemer

# Oversikt

- Så langt i kurset har vi snakket mye om ressursbruken av en gitt algoritme
- Vi har snakket mye om kompleksiteten til løsninger på problemer
- Denne uken skal vi snakke om kompleksiteten til problemene selv

# Oversikt

- Så langt i kurset har vi snakket mye om ressursbruken av en gitt algoritme
- Vi har snakket mye om kompleksiteten til løsninger på problemer
- Denne uken skal vi snakke om kompleksiteten til problemene selv
  - Alle algoritmer vi har sett har vært i polynomiell tid

# Kompleksiteten til algoritmer

- Kompleksitet viser ressursbruk: tid og minne

# Kompleksiteten til algoritmer

- Kompleksitet viser ressursbruk: tid og minne
- Vi vektlegger tid

# Kompleksiteten til algoritmer

- Kompleksitet viser ressursbruk: tid og minne
- Vi vektlegger tid
- Tid måles i antall steg i forhold til størrelsen på input i verste tilfelle

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
```

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
```

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
```

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
5             sum ← sum + 1
6     return sum
```

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
5             sum ← sum + 1
6     return sum
```

- Ytre løkke: $n$ iterasjoner

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
5             sum ← sum + 1
6     return sum
```

- Ytre løkke: $n$ iterasjoner
- Indre løkke: Varierer med $i$

# En ineffektiv løsning på et enkelt problem

**ALGORITHM:** SUMMER TALLENE FRA 1 TIL $n$

**Input:** Et naturlig tall $n$
**Output:** Summen av alle tallene fra 1 til $n$

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
5             sum ← sum + 1
6     return sum
```

- Ytre løkke: $n$ iterasjoner
- Indre løkke: Varierer med $i$
- Totalt antall iterasjoner:

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

# Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

# Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

- Å summere første og siste tall gir $n + 1$

# Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

- Å summere første og siste tall gir $n + 1$
- Dersom vi summerer opp alt to ganger får vi:

$$\begin{array}{cccccccc} & 1 & + & 2 & + & \cdots & + & n \\ + & n & + & n-1 & + & \cdots & + & 1 \\ \hline = & n+1 & + & n+1 & + & \cdots & + & n+1 \end{array}$$

# Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

- Å summere første og siste tall gir $n + 1$
- Dersom vi summerer opp alt to ganger får vi:

$$\begin{array}{rccccccc} & 1 & + & 2 & + & \cdots & + & n \\ + & n & + & n-1 & + & \cdots & + & 1 \\ \hline = & n+1 & + & n+1 & + & \cdots & + & n+1 \end{array}$$

- Merk at $\overbrace{(n+1) + (n+1) + \cdots + (n+1)}^{n} = n \cdot (n+1)$

# Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

- Å summere første og siste tall gir $n + 1$
- Dersom vi summerer opp alt to ganger får vi:

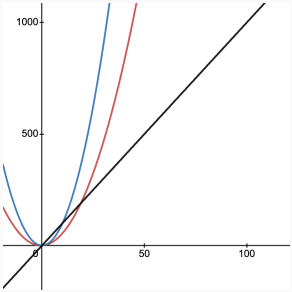
$$\begin{array}{rccccccc} & 1 & + & 2 & + & \cdots & + & n \\ + & n & + & n-1 & + & \cdots & + & 1 \\ \hline = & n+1 & + & n+1 & + & \cdots & + & n+1 \end{array}$$

- Merk at $\overbrace{(n+1) + (n+1) + \cdots + (n+1)}^{n} = n \cdot (n+1)$
- Hvis vi halverer resultatet får vi:

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

# Litt om kjøretidskompleksitet

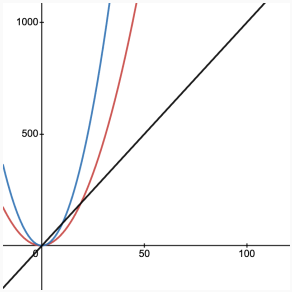
- Forstå hvordan kjøretiden vokser når input blir større




desmos

# Litt om kjøretidskompleksitet

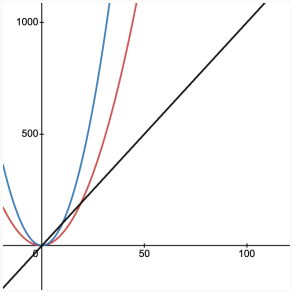
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$




desmos

# Litt om kjøretidskompleksitet

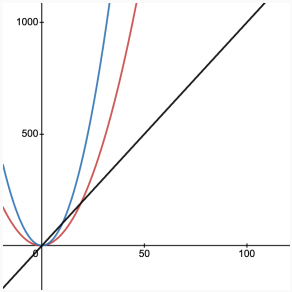
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer




desmos

# Litt om kjøretidskompleksitet

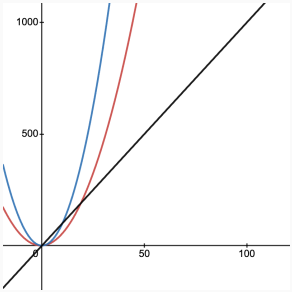
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk




desmos

# Litt om kjøretidskompleksitet

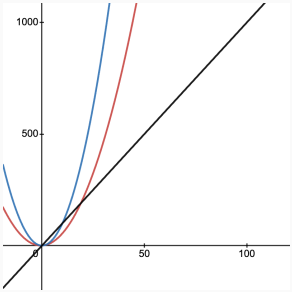
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært




desmos

# Litt om kjøretidskompleksitet

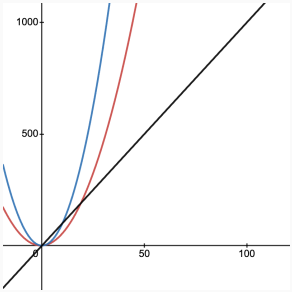
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Store-$\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer




desmos

# Litt om kjøretidskompleksitet

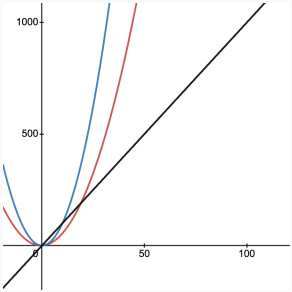
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Store-$\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer
- Det gir opphav til følgende perspektiv:




desmos

# Litt om kjøretidskompleksitet

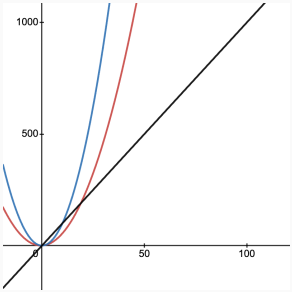
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Store-$\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer
- Det gir opphav til følgende perspektiv:
  - Vekst er viktigere enn konstantfaktorer




desmos

# Litt om kjøretidskompleksitet

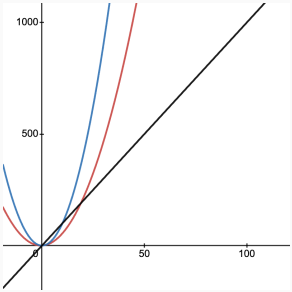
- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Store-$\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer
- Det gir opphav til følgende perspektiv:
  - Vekst er viktigere enn konstantfaktorer
  - Tregeste leddet styrer veksten




desmos

# Litt om kjøretidskompleksitet

- Forstå hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, $n^2$ og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - $n^2$ og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Store-$\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer
- Det gir opphav til følgende perspektiv:
  - Vekst er viktigere enn konstantfaktorer
  - Tregeste leddet styrer veksten
- Gjør det enklere å resonnere om kjøretid




desmos

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til *n* var *veldig* inneffektiv

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til *n* var *veldig* ineffektiv
- Den er ikke ineffektiv fordi den er kvadratisk

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til $n$ var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men siden det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2     sum ← 0
```

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til $n$ var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men siden det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2     sum ← 0
```

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til $n$ var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men siden det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
```

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til $n$ var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men siden det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         sum ← sum + i
5     return sum
```

  - Denne er i $\mathcal{O}(n)$

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til $n$ var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men siden det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2 |  sum ← 0
3 |  for i ← 1 to n do
4 |  |  sum ← sum + i
5 |  return sum
```

  - Denne er i $\mathcal{O}(n)$
- I analysen over utledet vi en formel som løser problemet direkte

```
1 Procedure SumTo(n)
2 |  return n·(n+1)/2
```

# Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til $n$ var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men siden det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         sum ← sum + i
5     return sum
```

  - Denne er i $\mathcal{O}(n)$
- I analysen over utledet vi en formel som løser problemet direkte

```
1 Procedure SumTo(n)
2     return n·(n+1)/2
```

  - Denne er i $\mathcal{O}(1)$

# Kompleksiteten til problemer

- Fokus i kompleksitetsteori er problemers *iboende* vanskelighetsgrad

# Kompleksiteten til problemer

- Fokus i kompleksitetsteori er problemers *iboende* vanskelighetsgrad
- Vanskelighetsgraden defineres av den *mest effektive* algoritmen som løser problemet

# Kompleksiteten til problemer

- Fokus i kompleksitetsteori er problemers *iboende* vanskelighetsgrad
- Vanskelighetsgraden defineres av den *mest effektive* algoritmen som løser problemet
- Å vise at det *ikke kan eksistere* en mer effektiv løsning kan ofte være veldig vanskelig

# Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*

# Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI

# Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI
- Tidligere har vi sett på problemer som har hatt rikere output

# Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI
- Tidligere har vi sett på problemer som har hatt rikere output
  - Som regel kan vi formulere et *lignende* avgjørelsesproblem

# Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI
- Tidligere har vi sett på problemer som har hatt rikere output
  - Som regel kan vi formulere et *lignende* avgjørelsesproblem
- Hvis en algoritme eksisterer for et problem, kan den brukes til å besvare det relaterte avgjørelsesproblemet

# Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI
- Tidligere har vi sett på problemer som har hatt rikere output
  - Som regel kan vi formulere et *lignende* avgjørelsesproblem
- Hvis en algoritme eksisterer for et problem, kan den brukes til å besvare det relaterte avgjørelsesproblemet
- Rikere varianter av problemer er *minst like vanskelig* som deres relaterte avgjørelsesproblemer

# Avgjørelsesproblemer (eksempler)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

# Avgjørelsesproblemer (eksempler)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**MST-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Finnes det et spenntrre over $G$ med total vekt mindre enn $k$?

# Avgjørelsesproblemer (eksempler)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**MST-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Finnes det et spenntrre over $G$ med total vekt mindre enn $k$?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

# Kompleksitetsklassene *P* og *NP*

- Avgjørelsesproblemer kan klassifiseres etter hvor vanskelige de er å løse

# Kompleksitetsklassene *P* og *NP*

- Avgjørelsesproblemer kan klassifiseres etter hvor vanskelige de er å løse
- De to mest kjente kompleksitetsklassene heter *P* og *NP*

# Kompleksitetsklassen $P$

- Et avgjørbarhetsproblem er i $P$ hvis det finnes en algoritme som løser det i polynomiell tid

# Kompleksitetsklassen $P$

- Et avgjørbarhetsproblem er i $P$ hvis det finnes en algoritme som løser det i polynomiell tid
- Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant $k$

# Kompleksitetsklassen $P$

- Et avgjørbarhetsproblem er i $P$ hvis det finnes en algoritme som løser det i polynomiell tid
- Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant $k$
- En intuitiv måte å se for seg polynomiell tid er et program som inneholder flere nøstede løkker

```
1 Procedure Polynomial(n)
2     for i_1 ← 0 to n − 1 do
3         for i_2 ← 0 to n − 1 do
              ⋮
4         for i_k ← 0 to n − 1 do
5             Constant(i)                                   // O(n^k)
6
```

# Kompleksitetsklassen $P$

- Et avgjørbarhetsproblem er i $P$ hvis det finnes en algoritme som løser det i polynomiell tid
- Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant $k$
- En intuitiv måte å se for seg polynomiell tid er et program som inneholder flere nøstede løkker

```
1 Procedure Polynomial(n)
2     for i_1 ← 0 to n − 1 do
3         for i_2 ← 0 to n − 1 do
              ⋮
4         for i_k ← 0 to n − 1 do
5             Constant(i)                                  // O(n^k)
6
```

- Alle algoritmer vi har sett i kurset så langt har vært polynomielle

# Kompleksitetsklassen $P$

- Et avgjørbarhetsproblem er i $P$ hvis det finnes en algoritme som løser det i polynomiell tid
- Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant $k$
- En intuitiv måte å se for seg polynomiell tid er et program som inneholder flere nøstede løkker

```
1 Procedure Polynomial(n)
2     for i_1 ← 0 to n − 1 do
3         for i_2 ← 0 to n − 1 do
              ⋮
4             for i_k ← 0 to n − 1 do
5                 Constant(i)                                  // O(n^k)
6
```

- Alle algoritmer vi har sett i kurset så langt har vært polynomielle
- Husk at $\mathcal{O}$ kun gir en øvre grense

# Kompleksitetsklassen $P$

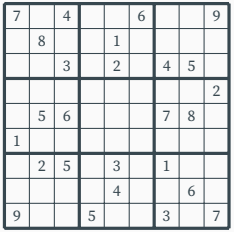
- Et avgjørbarhetsproblem er i $P$ hvis det finnes en algoritme som løser det i polynomiell tid
- Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant $k$
- En intuitiv måte å se for seg polynomiell tid er et program som inneholder flere nøstede løkker

```
1 Procedure Polynomial(n)
2     for i_1 ← 0 to n − 1 do
3         for i_2 ← 0 to n − 1 do
              ⋮
4
5         for i_k ← 0 to n − 1 do
6             Constant(i)                                   // O(n^k)
```

- Alle algoritmer vi har sett i kurset så langt har vært polynomielle
- Husk at $\mathcal{O}$ kun gir en øvre grense
  - For eksempel er en algoritme i $\mathcal{O}(\log n)$ også i $\mathcal{O}(n^2)$

# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid



# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 4 | | | 6 | | | 9 |
| | 8 | | | 1 | | | | |
| | | 3 | | 2 | | 4 | 5 | |
| | | | | | | | | 2 |
| | 5 | 6 | | | | 7 | 8 | |
| 1 | | | | | | | | |
| | 2 | 5 | | 3 | | 1 | | |
| | | | | 4 | | | 6 | |
| 9 | | | 5 | | | 3 | | 7 |

# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 4 | | | 6 | | | 9 |
| | 8 | | | 1 | | | | |
| | | 3 | | 2 | | 4 | 5 | |
| | | | | | | | | 2 |
| | 5 | 6 | | | | 7 | 8 | |
| 1 | | | | | | | | |
| | 2 | 5 | | 3 | | 1 | | |
| | | | | 4 | | | 6 | |
| 9 | | | 5 | | | 3 | | 7 |

# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?
- Ingen kjent polynomielltidsløsning for dette problemet

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 4 | | | 6 | | | 9 |
| | 8 | | | 1 | | | | |
| | | 3 | | 2 | | 4 | 5 | |
| | | | | | | | | 2 |
| | 5 | 6 | | | | 7 | 8 | |
| 1 | | | | | | | | |
| | 2 | 5 | | 3 | | 1 | | |
| | | | | 4 | | | 6 | |
| 9 | | | 5 | | | 3 | | 7 |

# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?
- Ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 4 | | | 6 | | | 9 |
| | 8 | | | 1 | | | | |
| | | 3 | | 2 | | 4 | 5 | |
| | | | | | | | | 2 |
| | 5 | 6 | | | | 7 | 8 | |
| 1 | | | | | | | | |
| | 2 | 5 | | 3 | | 1 | | |
| | | | | 4 | | | 6 | |
| 9 | | | 5 | | | 3 | | 7 |

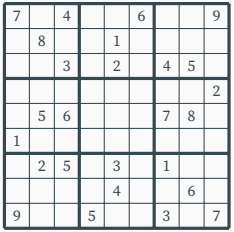
# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?
- Ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid
- Dette problemet vokser fort:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 4 | | | 6 | | | 9 |
| | 8 | | | 1 | | | | |
| | | 3 | | 2 | | 4 | 5 | |
| | | | | | | | | 2 |
| | 5 | 6 | | | | 7 | 8 | |
| 1 | | | | | | | | |
| | 2 | 5 | | 3 | | 1 | | |
| | | | | 4 | | | 6 | |
| 9 | | | 5 | | | 3 | | 7 |

# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?
- Ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid
- Dette problemet vokser fort:
  - $n = 4$: 288 gyldige brett



# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?
- Ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid
- Dette problemet vokser fort:
  - $n = 4$: 288 gyldige brett
  - $n = 9$: 6 670 903 752 021 072 936 960 gyldige brett

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 4 | | | 6 | | | 9 |
| | 8 | | | 1 | | | | |
| | | 3 | | 2 | | 4 | 5 | |
| | | | | | | | | 2 |
| | 5 | 6 | | | | 7 | 8 | |
| 1 | | | | | | | | |
| | 2 | 5 | | 3 | | 1 | | |
| | | | | 4 | | | 6 | |
| 9 | | | 5 | | | 3 | | 7 |

# Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt $9 \times 9$ grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  **SUDOKU** $n \times n$
  **Instans:** Et delvis utfylt $n \times n$ Sudoku-brett.
  **Spørsmål:** Har brettet en gyldig løsning?
- Ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid
- Dette problemet vokser fort:
  - $n = 4$: 288 gyldige brett
  - $n = 9$: 6 670 903 752 021 072 936 960 gyldige brett
- Det er allikevel enkelt å *sjekke* at en gitt løsning er gyldig

| 7 | 1 | 4 | 3 | 5 | 6 | 8 | 2 | 9 |
|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 2 | 4 | 1 | 9 | 6 | 7 | 3 |
| 6 | 9 | 3 | 7 | 2 | 8 | 4 | 5 | 1 |
| 3 | 7 | 9 | 8 | 6 | 4 | 5 | 1 | 2 |
| 2 | 5 | 6 | 1 | 9 | 3 | 7 | 8 | 4 |
| 1 | 4 | 8 | 2 | 7 | 5 | 9 | 3 | 6 |
| 4 | 2 | 5 | 6 | 3 | 7 | 1 | 9 | 8 |
| 8 | 3 | 7 | 9 | 4 | 1 | 2 | 6 | 5 |
| 9 | 6 | 1 | 5 | 8 | 2 | 3 | 4 | 7 |

# Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI

# Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Vi kan verifisere løsningen for et avgjørelsesproblem ved hjelp av et *sertifikat*

# Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Vi kan verifisere løsningen for et avgjørelsesproblem ved hjelp av et *sertifikat*
- For Sudoku er spørsmålet om et gitt brett har en løsning

# Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Vi kan verifisere løsningen for et avgjørelsesproblem ved hjelp av et *sertifikat*
- For Sudoku er spørsmålet om et gitt brett har en løsning
  - Dersom svaret er JA, kan vi skrive en algoritme som verifiserer det i *polynomiell tid*

# Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Vi kan verifisere løsningen for et avgjørelsesproblem ved hjelp av et *sertifikat*
- For Sudoku er spørsmålet om et gitt brett har en løsning
  - Dersom svaret er JA, kan vi skrive en algoritme som verifiserer det i *polynomiell tid*
  - En slik algoritme tar instansen og det løste brettet og sjekker at alle regler er fulgt

# Verifisering av løsninger med sertifikater

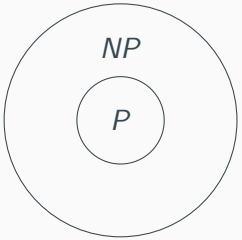
- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Vi kan verifisere løsningen for et avgjørelsesproblem ved hjelp av et *sertifikat*
- For Sudoku er spørsmålet om et gitt brett har en løsning
  - Dersom svaret er JA, kan vi skrive en algoritme som verifiserer det i *polynomiell tid*
  - En slik algoritme tar instansen og det løste brettet og sjekker at alle regler er fulgt
  - Det løste brettet fungerer her som et *sertifikat*.

# Kompleksitetsklassen *NP*

- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid

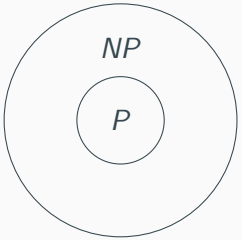
# Kompleksitetsklassen *NP*

- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i *P* er også i *NP*



# Kompleksitetsklassen *NP*

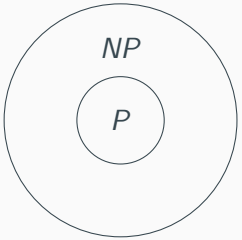
- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i *P* er også i *NP*




- *NP* er en forkortelse for «Nondeterministic Polynomial time»

# Kompleksitetsklassen *NP*

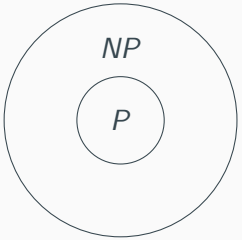
- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i *P* er også i *NP*




- *NP* er en forkortelse for «Nondeterministic Polynomial time»
- Alle problemer i *NP* er *minst like vanskelige* som de i *P*

# Kompleksitetsklassen *NP*

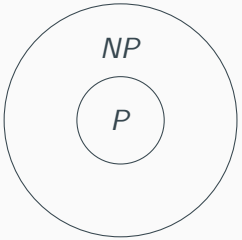
- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i *P* er også i *NP*




- *NP* er en forkortelse for «Nondeterministic Polynomial time»
- Alle problemer i *NP* er *minst like vanskelige* som de i *P*
- Usikkert om $P = NP$ eller $P \neq NP$

# Kompleksitetsklassen *NP*

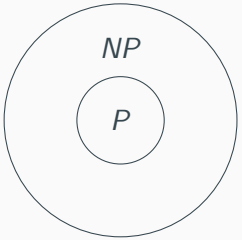
- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i *P* er også i *NP*




- *NP* er en forkortelse for «Nondeterministic Polynomial time»
- Alle problemer i *NP* er *minst like vanskelige* som de i *P*
- Usikkert om $P = NP$ eller $P \neq NP$
  - Et av de største uløste problemene i matematikk og informatikk

# Kompleksitetsklassen *NP*

- Et avgjørbarhetsproblem er i *NP* hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i *P* er også i *NP*




- *NP* er en forkortelse for «Nondeterministic Polynomial time»
- Alle problemer i *NP* er *minst like vanskelige* som de i *P*
- Usikkert om $P = NP$ eller $P \neq NP$
  - Et av de største uløste problemene i matematikk og informatikk
  - Løsning gir en premie på én million dollar

# Oversettelse mellom problemer

- Problemløsing involverer ofte å oversette problemene til andre problemer som er lettere å løse

# Oversettelse mellom problemer

- Problemløsing involverer ofte å oversette problemene til andre problemer som er lettere å løse
- I kompleksitetsteori gjør vi dette systematisk gjennom reduksjoner

# Oversettelse mellom problemer

- Problemløsing involverer ofte å oversette problemene til andre problemer som er lettere å løse
- I kompleksitetsteori gjør vi dette systematisk gjennom reduksjoner
- Reduksjoner gir oss en måte å sammenligne vanskelighetsgraden av ulike problemer

# Polynomtidsreduksjoner

- En polynomtidsreduksjon fra problem $A$ til problem $B$ transformerer instanser av $A$ til instanser av $B$ i polynomiell tid

# Polynomtidsreduksjoner

- En polynomtidsreduksjon fra problem $A$ til problem $B$ transformerer instanser av $A$ til instanser av $B$ i polynomiell tid
- Korrekt reduksjon krever at JA-instanser av $A$ blir JA-instanser av $B$, og tilsvarende for NEI-instanser

# Polynomtidsreduksjoner

- En polynomtidsreduksjon fra problem $A$ til problem $B$ transformerer instanser av $A$ til instanser av $B$ i polynomiell tid
- Korrekt reduksjon krever at JA-instanser av $A$ blir JA-instanser av $B$, og tilsvarende for NEI-instanser
- Alle avgjørelsesproblemer i $P$ kan reduseres til hverandre

# Polynomtidsreduksjoner (eksempel)

**EVEN**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et partall?

**ODD**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et oddetall?

- En instans av **EVEN** er et heltall $n$

# Polynomtidsreduksjoner (eksempel)

**EVEN**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et partall?

**ODD**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et oddetall?

- En instans av **EVEN** er et heltall $n$
- Instansen kan oversettes til en instans av **ODD** ved å transformere $n$ til $n + 1$

# Polynomtidsreduksjoner (eksempel)

**EVEN**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et partall?

**ODD**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et oddetall?

- En instans av **EVEN** er et heltall $n$
- Instansen kan oversettes til en instans av **ODD** ved å transformere $n$ til $n + 1$
- Denne er korrekt, fordi

# Polynomtidsreduksjoner (eksempel)

**EVEN**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et partall?

**ODD**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et oddetall?

- En instans av **EVEN** er et heltall $n$
- Instansen kan oversettes til en instans av **ODD** ved å transformere $n$ til $n + 1$
- Denne er korrekt, fordi
  - en JA-instans av **EVEN** blir en JA-instans av **ODD**

# Polynomtidsreduksjoner (eksempel)

**EVEN**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et partall?

**ODD**
**Instans:** Et heltall $n$.
**Spørsmål:** Er $n$ et oddetall?

- En instans av **EVEN** er et heltall $n$
- Instansen kan oversettes til en instans av **ODD** ved å transformere $n$ til $n + 1$
- Denne er korrekt, fordi
  - en JA-instans av **EVEN** blir en JA-instans av **ODD**
  - en NEI-instans av **EVEN** blir en NEI-instans av **ODD**

# Polynomtidsreduksjoner (eksempel)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i $P$ kan polynomtidsreduseres

# Polynomtidsreduksjoner (eksempel)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i $P$ kan polynomtidsreduseres
- Vi rekker å *løse problemene* i polynomiell tid

# Polynomtidsreduksjoner (eksempel)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i $P$ kan polynomtidsreduseres
- Vi rekker å *løse problemene* i polynomiell tid
- **SCC-$k$** kan reduseres til sort

# Polynomtidsreduksjoner (eksempel)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i $P$ kan polynomtidsreduseres
- Vi rekker å *løse problemene* i polynomiell tid
- **SCC-$k$** kan reduseres til sort
- Løs **SCC-$k$**$(G, k)$

# Polynomtidsreduksjoner (eksempel)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i $P$ kan polynomtidsreduseres
- Vi rekker å *løse problemene* i polynomiell tid
- **SCC-$k$** kan reduseres til sort
- Løs **SCC-$k$**$(G, k)$
  - Hvis løsningen er JA, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [1, 2, 3]$

# Polynomtidsreduksjoner (eksempel)

**SORT**
**Instans:** To arrayer $A_1$ og $A_2$.
**Spørsmål:** Består $A_2$ av de samme elementene som $A_1$ i sortert rekkefølge?

**SCC-$k$**
**Instans:** En graf $G$ og et tall $k$.
**Spørsmål:** Har $G$ minst $k$ sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i $P$ kan polynomtidsreduseres
- Vi rekker å *løse problemene* i polynomiell tid
- **SCC-$k$** kan reduseres til sort
- Løs **SCC-$k$**$(G, k)$
  - Hvis løsningen er JA, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [1, 2, 3]$
  - Hvis løsningen er NEI, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [2, 1, 3]$

# De vanskeligste problemene i *NP*

- *NP*-harde problemer er minst like vanskelige som hvert problem i *NP*

# De vanskeligste problemene i *NP*

- *NP*-harde problemer er minst like vanskelige som hvert problem i *NP*
- SAT var det første problemet som ble bevist å være *NP*-hardt

# De vanskeligste problemene i *NP*

- *NP*-harde problemer er minst like vanskelige som hvert problem i *NP*
- SAT var det første problemet som ble bevist å være *NP*-hardt
- For å vise at et problem *A* er *NP*-hardt, kan man redusere et kjent *NP*-hardt problem, som SAT, til *A*

# *NP*-komplette problemer

- Et problem *A* er *NP*-komplett hvis det både

# *NP*-komplette problemer

- Et problem *A* er *NP*-komplett hvis det både
  1. er i *NP* (kan verifiseres i polynomiell tid)

# *NP*-komplette problemer

- Et problem *A* er *NP*-komplett hvis det både
  1. er i *NP* (kan verifiseres i polynomiell tid)
  2. er *NP*-hardt (hvert problem i *NP* kan polynomtidsreduseres til *A*)

# *NP*-komplette problemer

- Et problem *A* er *NP*-komplett hvis det både
  1. er i *NP* (kan verifiseres i polynomiell tid)
  2. er *NP*-hardt (hvert problem i *NP* kan polynomtidsreduseres til *A*)
- Disse problemene er antatt vanskelige å løse, men løsningene er enkle å verifisere

# *NP*-komplette problemer

- Et problem *A* er *NP*-komplett hvis det både
  1. er i *NP* (kan verifiseres i polynomiell tid)
  2. er *NP*-hardt (hvert problem i *NP* kan polynomtidsreduseres til *A*)
- Disse problemene er antatt vanskelige å løse, men løsningene er enkle å verifisere
- *NP*-komplette problemer spiller en viktig rolle i kryptografi

# Men er $P = NP$?

- Om ett $NP$-komplett problem kan løses i polynomiell tid, kan alle slike problemer løses, noe som ville bevise at $P = NP$

# Men er $P = NP$?

- Om ett $NP$-komplett problem kan løses i polynomiell tid, kan alle slike problemer løses, noe som ville bevise at $P = NP$
- Dersom en slik løsning er rask nok til å kunne beregnes på moderne datamaskiner, så ville det hatt enorme konsekvenser

# Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet

# Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar

# Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar
  - Finnes det problemer som er grunnleggende uløselige?

# Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar
  - Finnes det problemer som er grunnleggende uløselige?
- Dette var spørsmålet David Hilbert stilte i begynnelsen av 1900-tallet

# Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar
  - Finnes det problemer som er grunnleggende uløselige?
- Dette var spørsmålet David Hilbert stilte i begynnelsen av 1900-tallet
- Svaret kom fra Alonzo Church og senere Alan Turing i 1936

# Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar
  - Finnes det problemer som er grunnleggende uløselige?
- Dette var spørsmålet David Hilbert stilte i begynnelsen av 1900-tallet
- Svaret kom fra Alonzo Church og senere Alan Turing i 1936
  - Det finnes uløselige problemer!

# Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*

# Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*
- Det presenteres som et beslutningsproblem:

# Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*
- Det presenteres som et beslutningsproblem:
  - Instans: En algoritme $A$, og en input til $A$, $x$

# Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*
- Det presenteres som et beslutningsproblem:
  - Instans: En algoritme $A$, og en input til $A$, $x$
  - Spørsmål: Terminerer $A$ på input $x$?

# Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*
- Det presenteres som et beslutningsproblem:
  - Instans: En algoritme $A$, og en input til $A$, $x$
  - Spørsmål: Terminerer $A$ på input $x$?
- Stoppeproblemet er uløselig, og dette kan bevises ved et motsigelsesbevis

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Ved å kjøre $D(D)$ oppstår en motsigelse:

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Ved å kjøre $D(D)$ oppstår en motsigelse:
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Ved å kjøre $D(D)$ oppstår en motsigelse:
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke
  - Dersom $H(D, D)$ svarer at $D(D)$ ikke terminerer, så svarer $D(D)$ JA

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Ved å kjøre $D(D)$ oppstår en motsigelse:
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke
  - Dersom $H(D, D)$ svarer at $D(D)$ ikke terminerer, så svarer $D(D)$ JA
- I begge tilfeller leder antagelsen om at $H$ finnes til en motsigelse

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Ved å kjøre $D(D)$ oppstår en motsigelse:
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke
  - Dersom $H(D, D)$ svarer at $D(D)$ ikke terminerer, så svarer $D(D)$ JA
- I begge tilfeller leder antagelsen om at $H$ finnes til en motsigelse
- Derfor kan ikke en slik algoritme $H$ eksistere

# Motsigelsesbevis for stoppeproblemet

- Anta at det finnes en algoritme $H$ som gir riktig svar på alle instanser av HALT
- Konstruerer algoritme $D$ som tar en algoritme $A$ som input:
  - Sjekker om $H(A, A)$, terminerer $A$ med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Ved å kjøre $D(D)$ oppstår en motsigelse:
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke
  - Dersom $H(D, D)$ svarer at $D(D)$ ikke terminerer, så svarer $D(D)$ JA
- I begge tilfeller leder antagelsen om at $H$ finnes til en motsigelse
- Derfor kan ikke en slik algoritme $H$ eksistere
- Med andre ord: ingen algoritme kan løse stoppeproblemet