Kompleksitet og Beregnbarhet

Sigurd Kittilsen

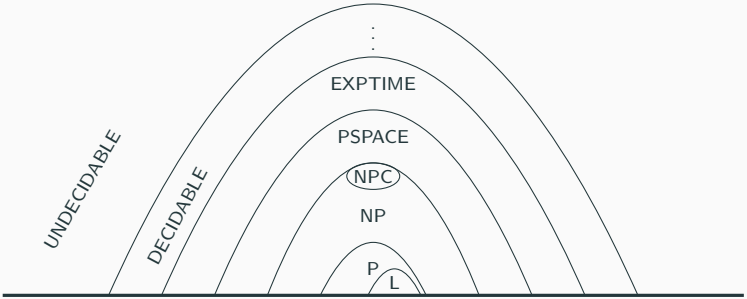
Høst 2021

Institutt for Informatikk

Dagens plan

Dagens plan

- Avgjørelsesproblemer
- P
- NP
- Reduksjoner
- NP-kompletthet
- Uavgjørbarhet



Avgjørelsesproblemer (Decision Problems)

Avgjørelsesproblemer (Decision Problems)

- Avgjørelsesproblemer er problemer der output er enten ja eller nei.

Avgjørelsesproblemer (Decision Problems)

- Avgjørelsesproblemer er problemer der output er enten ja eller nei.
- En algoritme som løser et slikt problem tar en instans som input og svarer ja/nei avhengig av hva den har regnet seg frem til.

Avgjørelsesproblemer (Decision Problems)

- Avgjørelsesproblemer er problemer der output er enten ja eller nei.
- En algoritme som løser et slikt problem tar en instans som input og svarer ja/nei avhengig av hva den har regnet seg frem til.
- Tidligere har vi sett blant annet *Eulerkrets* og *Hamiltonsykel*, der svaret var ja/nei.

Avgjørelsesproblemer (Decision Problems)

- Avgjørelsesproblemer er problemer der output er enten ja eller nei.
- En algoritme som løser et slikt problem tar en instans som input og svarer ja/nei avhengig av hva den har regnet seg frem til.
- Tidligere har vi sett blant annet *Eulerkrets* og *Hamiltonsykel*, der svaret var ja/nei.
- Hvordan kunne vi formulert spillene Sudoku og Sjakk som avgjørelses-problemer?

Eksempler på avgjørelsesproblemer

Eksempler på avgjørelsesproblemer

PRIMES	
Instans:	Et positivt heltall, n
Spørsmål:	Er n et primtall?

Eksempler på avgjørelsesproblemer

PRIMES	
Instans:	Et positivt heltall, n
Spørsmål:	Er n et primtall?

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

Eksempler på avgjørelsesproblemer

PRIMES	
Instans:	Et positivt heltall, n
Spørsmål:	Er n et primtall?

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

Eksempler på avgjørelsesproblemer

PRIMES	
Instans:	Et positivt heltall, n
Spørsmål:	Er n et primtall?

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Eksempler på avgjørelsesproblemer II

Eksempler på avgjørelsesproblemer II

SAT (oppfyllbarhet)	
Instans:	En utsagnslogisk formel F
Spørsmål:	Er F oppfyllbar?

Eksempler på avgjørelsesproblemer II

SAT (oppfyllbarhet)	
Instans:	En utsagnslogisk formel F
Spørsmål:	Er F oppfyllbar?

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

Eksempler på avgjørelsesproblemer II

SAT (oppfyllbarhet)	
Instans:	En utsagnslogisk formel F
Spørsmål:	Er F oppfyllbar?

CLIQUE	
Instans:	En graf G og et heltall k
Spørsmål:	Inneholder G en komplett delgraf (clique) av størrelse minst k?

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

Eksempler på avgjørelsesproblemer II

SAT (oppfyllbarhet)	
Instans:	En utsagnslogisk formel F
Spørsmål:	Er F oppfyllbar?

CLIQUE	
Instans:	En graf G og et heltall k
Spørsmål:	Inneholder G en komplett delgraf (clique) av størrelse minst k?

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

GRAF-ISOMORFI	
Instans:	To grafer G og H
Spørsmål:	Er G og H *isomorfe*?

Klassen P, polynomiell tid

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.
- Vi kan formulere dette som et avgjørelsesproblem (gitt G og k, stemmer det at G har minst k komponenter), og konkludere med at dette problemet er i P.

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.
- Vi kan formulere dette som et avgjørelsesproblem (gitt G og k, stemmer det at G har minst k komponenter), og konkludere med at dette problemet er i P.

Andre resultater:

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.
- Vi kan formulere dette som et avgjørelsesproblem (gitt G og k, stemmer det at G har minst k komponenter), og konkludere med at dette problemet er i P.

Andre resultater:

- *Eulerkrets* $\in P$

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.
- Vi kan formulere dette som et avgjørelsesproblem (gitt G og k, stemmer det at G har minst k komponenter), og konkludere med at dette problemet er i P.

Andre resultater:

- *Eulerkrets* $\in P$
- (Avgjørelsesproblemet) sjakk løses i konstant tid, så det er med i P

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.
- Vi kan formulere dette som et avgjørelsesproblem (gitt G og k, stemmer det at G har minst k komponenter), og konkludere med at dette problemet er i P.

Andre resultater:

- *Eulerkrets* $\in P$
- (Avgjørelsesproblemet) sjakk løses i konstant tid, så det er med i P
- *PRIMES* $\in P$, dette er ikke åpenbart og ble først vist i 2002

Klassen P, polynomiell tid

- Klassen P er en samling av avgjørelsesproblemer som kan løses i *polynomiell* tid.
- Vi har sett at en algoritme kjører i polynomiell tid hvis kjøretiden er $O(n^k)$, der n er størrelsen på input, og der k er en konstant.

Eksempler

- Vi vet at å telle antall komponenter i en graf kan gjøres i polynomiell tid ved hjelp av *DFS*.
- Vi kan formulere dette som et avgjørelsesproblem (gitt G og k, stemmer det at G har minst k komponenter), og konkludere med at dette problemet er i P.

Andre resultater:

- *Eulerkrets* $\in P$
- (Avgjørelsesproblemet) sjakk løses i konstant tid, så det er med i P
- *PRIMES* $\in P$, dette er ikke åpenbart og ble først vist i 2002

Klassen P inneholder alle "enkle" problemer.

Klassen *NP*, non-deterministisk polynomiell tid

Klassen *NP*, non-deterministisk polynomiell tid

Vi kan definere *NP* på to forskjellige, men fortsatt ekvivalente, måter. Vi skal fokusere mest på definisjon 1.

Klassen *NP*, non-deterministisk polynomiell tid

Vi kan definere *NP* på to forskjellige, men fortsatt ekvivalente, måter. Vi skal fokusere mest på definisjon 1.

1. *NP* er klassen av avgjørelsesproblemer der ja-instansene kan verifiseres i (deterministisk) polynomiell tid, gitt et *sertifikat* av polynomiell størrelse.

Klassen *NP*, non-deterministisk polynomiell tid

Vi kan definere *NP* på to forskjellige, men fortsatt ekvivalente, måter. Vi skal fokusere mest på definisjon 1.

1. *NP* er klassen av avgjørelsesproblemer der ja-instansene kan verifiseres i (deterministisk) polynomiell tid, gitt et *sertifikat* av polynomiell størrelse.
2. *NP* er klassen av avgjørelsesproblemer som kan løses av en *ikke-deterministisk* algoritme i polynomiell tid.

Klassen *NP*, non-deterministisk polynomiell tid

Vi kan definere *NP* på to forskjellige, men fortsatt ekvivalente, måter. Vi skal fokusere mest på definisjon 1.

1. *NP* er klassen av avgjørelsesproblemer der ja-instansene kan verifiseres i (deterministisk) polynomiell tid, gitt et *sertifikat* av polynomiell størrelse.
2. *NP* er klassen av avgjørelsesproblemer som kan løses av en *ikke-deterministisk* algoritme i polynomiell tid. En ikke-deterministisk algoritme kan "gjette" blant mange alternavtiver og gjetter alltid "riktig" i konstant tid.

Problemer i *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  $\rightarrow$ Sertifikat: Det løste brettet

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen
- *CLIQUE* $\in$ *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen
- *CLIQUE* $\in$ *NP*
  - $\rightarrow$ Sertifikat: k noder som utgjør en clique

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen
- *CLIQUE* $\in$ *NP*
  - $\rightarrow$ Sertifikat: k noder som utgjør en clique
- *GRAF* $-$ *ISOMORFI* $\in$ *NP*

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen
- *CLIQUE* $\in$ *NP*
  - $\rightarrow$ Sertifikat: k noder som utgjør en clique
- *GRAF* $-$ *ISOMORFI* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En mapping av nodene i G til nodene i H

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen
- *CLIQUE* $\in$ *NP*
  - $\rightarrow$ Sertifikat: k noder som utgjør en clique
- *GRAF* $-$ *ISOMORFI* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En mapping av nodene i G til nodene i H
- $P \subseteq NP$

Problemer i *NP*

- (Avgjørelsesversjonen av) Sudoku er i *NP*
  - $\rightarrow$ Sertifikat: Det løste brettet
- *Hamiltonsykel* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Den hamiltonske sykelen
- *PARTITION* $\in$ *NP*
  - $\rightarrow$ Sertifikat: Delmengden av riktig sum
- *SAT* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En tilordning av variablene som oppfyller formelen
- *CLIQUE* $\in$ *NP*
  - $\rightarrow$ Sertifikat: k noder som utgjør en clique
- *GRAF* $-$ *ISOMORFI* $\in$ *NP*
  - $\rightarrow$ Sertifikat: En mapping av nodene i G til nodene i H
- $P \subseteq NP$
  - $\rightarrow$ En deterministisk algoritme er en ikke-deterministisk algoritme som ikke benytter seg av ikke-determinisme

Reduksjoner

Reduksjoner



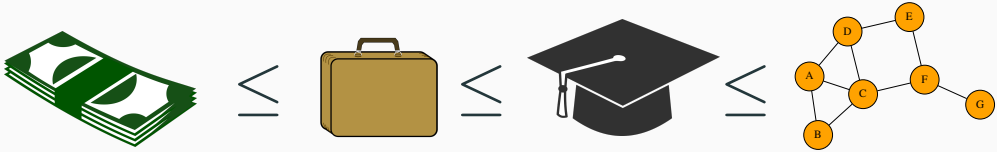
Reduksjoner



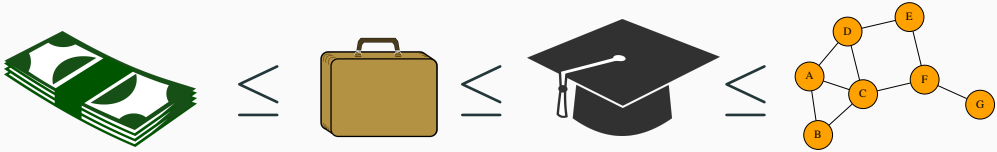
Reduksjoner



Reduksjoner

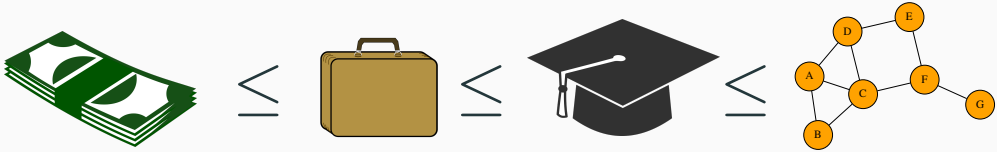


Reduksjoner





Reduksjoner





Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.

Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.
- En *polynomtidsreduksjon* fra et problem A til et problem B er en oversettelse av instanser i A til instanser i B i polynomiell tid.

Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.
- En *polynomtidsreduksjon* fra et problem A til et problem B er en oversettelse av instanser i A til instanser i B i polynomiell tid.
- Vi kan se på reduksjonen som en algoritme som tar inn A-instanser og har output i form av B-instanser.

Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.
- En *polynomtidsreduksjon* fra et problem A til et problem B er en oversettelse av instanser i A til instanser i B i polynomiell tid.
- Vi kan se på reduksjonen som en algoritme som tar inn A-instanser og har output i form av B-instanser.
- Vi krever at en slik algoritme kjører i polynomiell tid.

Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.
- En *polynomtidsreduksjon* fra et problem A til et problem B er en oversettelse av instanser i A til instanser i B i polynomiell tid.
- Vi kan se på reduksjonen som en algoritme som tar inn A-instanser og har output i form av B-instanser.
- Vi krever at en slik algoritme kjører i polynomiell tid.
- I tillegg krever vi at reduksjonen bevarer ja/nei-egenskapene til instansene

Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.
- En *polynomtidsreduksjon* fra et problem A til et problem B er en oversettelse av instanser i A til instanser i B i polynomiell tid.
- Vi kan se på reduksjonen som en algoritme som tar inn A-instanser og har output i form av B-instanser.
- Vi krever at en slik algoritme kjører i polynomiell tid.
- I tillegg krever vi at reduksjonen bevarer ja/nei-egenskapene til instansene, det vil si at hvis den får inn en ja-instans av A så gir den en ja-instans av B og tilsvarende for nei-instanser. Det er dette vi mener når vi sier at reduksjonen er korrekt.

Polynomtidsreduksjoner

- Reduksjoner er verktøyet vi bruker for å relatere/oversette problemer til andre problemer.
- En *polynomtidsreduksjon* fra et problem A til et problem B er en oversettelse av instanser i A til instanser i B i polynomiell tid.
- Vi kan se på reduksjonen som en algoritme som tar inn A-instanser og har output i form av B-instanser.
- Vi krever at en slik algoritme kjører i polynomiell tid.
- I tillegg krever vi at reduksjonen bevarer ja/nei-egenskapene til instansene, det vil si at hvis den får inn en ja-instans av A så gir den en ja-instans av B og tilsvarende for nei-instanser. Det er dette vi mener når vi sier at reduksjonen er korrekt.
- Hvis det finnes en polynomtidsreduksjon fra A til B, sier vi at A kan (polynomtid)reduseres til B, skrevet $A \leq_P B$.

NP-kompletthet, de vanskeligste problemene i *NP*

Et problem er L er *NP*-komplett hvis

NP-kompletthet, de vanskeligste problemene i *NP*

Et problem er L er *NP*-komplett hvis

1. $L \in NP$, og

NP-kompletthet, de vanskeligste problemene i NP

Et problem er L er NP-komplett hvis

1. $L \in NP$, og
2. Ethvert problem A i NP kan polynomtidreduseres til L, altså $A \leq_P L$ for alle $A \in NP$

NP-kompletthet, de vanskeligste problemene i NP

Et problem er L er NP-komplett hvis

1. $L \in NP$, og
2. Ethvert problem A i NP kan polynomtidreduseres til L, altså $A \leq_P L$ for alle $A \in NP$

- Hvis L oppfyller 2, sier vi at L er NP-hardt.

NP-kompletthet, de vanskeligste problemene i *NP*

Et problem er L er *NP*-komplett hvis

1. $L \in NP$, og
2. Ethvert problem A i *NP* kan polynomtidreduseres til L, altså $A \leq_P L$ for alle $A \in NP$

- Hvis L oppfyller 2, sier vi at L er *NP*-hardt.
- *PARTITION* er et eksempel på et *NP*-komplett problem.

NP-kompletthet, de vanskeligste problemene i *NP*

Et problem er L er *NP*-komplett hvis

1. $L \in NP$, og
2. Ethvert problem A i *NP* kan polynomtidreduseres til L, altså $A \leq_P L$ for alle $A \in NP$

- Hvis L oppfyller 2, sier vi at L er *NP*-hardt.
- *PARTITION* er et eksempel på et *NP*-komplett problem.
- Siden vi krever at alle problemer i *NP* skal kunne reduseres til *PARTITION* vet vi at *PARTITION* er minst like vanskelig som *alle* problemene i *NP*.

NP-kompletthet, de vanskeligste problemene i *NP*

Et problem er L er *NP*-komplett hvis

1. $L \in NP$, og
2. Ethvert problem A i *NP* kan polynomtidreduseres til L, altså $A \leq_P L$ for alle $A \in NP$

- Hvis L oppfyller 2, sier vi at L er *NP*-hardt.
- *PARTITION* er et eksempel på et *NP*-komplett problem.
- Siden vi krever at alle problemer i *NP* skal kunne reduseres til *PARTITION* vet vi at *PARTITION* er minst like vanskelig som *alle* problemene i *NP*.
- I tillegg krever vi at de *NP*-komplette problemene skal være i *NP*, som fører til at de *NP*-komplette problemene er de vanskeligste problemene i *NP*.

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.
- Siden alle problemer i *NP* kan polynomtidreduseres til A, og A kan polynomtidreduseres til X, vet vi at alle problemer i *NP* kan polynomtidreduseres til X.

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.
- Siden alle problemer i *NP* kan polynomtidreduseres til A, og A kan polynomtidreduseres til X, vet vi at alle problemer i *NP* kan polynomtidreduseres til X.

Hva gjorde man før man visste om noen *NP*-komplette problemer?

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.
- Siden alle problemer i *NP* kan polynomtidreduseres til A, og A kan polynomtidreduseres til X, vet vi at alle problemer i *NP* kan polynomtidreduseres til X.

Hva gjorde man før man visste om noen *NP*-komplette problemer?

- Tidlig på 1970-tallet ble det vist at *SAT* er *NP*-komplett (Cook-Levin Theorem).

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.
- Siden alle problemer i *NP* kan polynomtidreduseres til A, og A kan polynomtidreduseres til X, vet vi at alle problemer i *NP* kan polynomtidreduseres til X.

Hva gjorde man før man visste om noen *NP*-komplette problemer?

- Tidlig på 1970-tallet ble det vist at *SAT* er *NP*-komplett (Cook-Levin Theorem).
- Før det visste man ikke sikkert at det fantes (naturlige) *NP*-komplette problemer.

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.
- Siden alle problemer i *NP* kan polynomtidreduseres til A, og A kan polynomtidreduseres til X, vet vi at alle problemer i *NP* kan polynomtidreduseres til X.

Hva gjorde man før man visste om noen *NP*-komplette problemer?

- Tidlig på 1970-tallet ble det vist at *SAT* er *NP*-komplett (Cook-Levin Theorem).
- Før det visste man ikke sikkert at det fantes (naturlige) *NP*-komplette problemer.
- Resultatet ble vist med en generell reduksjon fra alle problemer i *NP* til *SAT*

NP-kompletthet, de vanskeligste problemene i *NP* II

- Den vanligste måten å vise at et problem X er *NP*-hardt, er ved å redusere et kjent *NP*-komplett problem, A, til X.
- Siden alle problemer i *NP* kan polynomtidreduseres til A, og A kan polynomtidreduseres til X, vet vi at alle problemer i *NP* kan polynomtidreduseres til X.

Hva gjorde man før man visste om noen *NP*-komplette problemer?

- Tidlig på 1970-tallet ble det vist at *SAT* er *NP*-komplett (Cook-Levin Theorem).
- Før det visste man ikke sikkert at det fantes (naturlige) *NP*-komplette problemer.
- Resultatet ble vist med en generell reduksjon fra alle problemer i *NP* til *SAT*
- Teoremet gjorde det mye enklere å vise at mange andre interessante problemer er *NP*-komplette ved å redusere *SAT* til dem.

Noen *NP*-komplette problemer

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

- *SAT*

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

- *SAT*
- *PARTITION*

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

- *SAT*
- *PARTITION*
- *Hamiltonsykel*

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

- *SAT*
- *PARTITION*
- *Hamiltonsykel*
- *CLIQUE*

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

- *SAT*
- *PARTITION*
- *Hamiltonsykel*
- *CLIQUE*
- *SUDOKU*
- *Super − Mario − Bros.*
- *Battleships*
- *Minesveiper*

Noen *NP*-komplette problemer

Etter at man hadde funnet ut at *SAT* er *NP*-komplett, dukket det opp *NP*-komplette problemer i mange forskjellige domener. Her er noen problemer vi vet at er *NP*-komplette:

- *SAT*
- *PARTITION*
- *Hamiltonsykel*
- *CLIQUE*
- *SUDOKU*
- *Super − Mario − Bros.*
- *Battleships*
- *Minesveiper*

Vi skal i tillegg vise at *SUBSET − SUM*, *HAMPATH* og *TSP* alle er *NP*-komplette.

SUBSET − SUM **er** *NP*-**komplett**

SUBSET − SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

SUBSET−SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at *SUBSET−SUM* er i *NP*.

SUBSET−SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at *SUBSET−SUM* er i *NP*.
- Det gjør vi ved å vise at det finnes korte (polynomielt store) sertifikater for ja-instansene.

SUBSET−SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at *SUBSET−SUM* er i *NP*.
- Det gjør vi ved å vise at det finnes korte (polynomielt store) sertifikater for ja-instansene.
- Vi blir bedt om å svare på om en delmengde av S summerer til k.

SUBSET − SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at *SUBSET − SUM* er i *NP*.
- Det gjør vi ved å vise at det finnes korte (polynomielt store) sertifikater for ja-instansene.
- Vi blir bedt om å svare på om en delmengde av S summerer til k.
- Sertifikatet kan være et utvalg av elementene i S.

SUBSET−SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at *SUBSET−SUM* er i *NP*.
- Det gjør vi ved å vise at det finnes korte (polynomielt store) sertifikater for ja-instansene.
- Vi blir bedt om å svare på om en delmengde av S summerer til k.
- Sertifikatet kan være et utvalg av elementene i S.
- Med dette som sertifikat er det lett å sjekke at elementene summerer til k.

SUBSET−SUM er *NP*-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at *SUBSET−SUM* er i *NP*.
- Det gjør vi ved å vise at det finnes korte (polynomielt store) sertifikater for ja-instansene.
- Vi blir bedt om å svare på om en delmengde av S summerer til k.
- Sertifikatet kan være et utvalg av elementene i S.
- Med dette som sertifikat er det lett å sjekke at elementene summerer til k.
- Dette stemmer godt med intuisjonen vår: hvis du påstår at du har en ja-instans av *SUBSET−SUM*, og noen ikke tror på deg, kan du bare fortelle dem hvilke elementer de skal velge og så klarer de enkelt å sjekke at du har rett.

$SUBSET-SUM$ er NP-komplett

SUBSET-SUM	
Instans:	En multimengde av heltall S og et heltall k
Spørsmål:	Finnes det en delmengde av S som summerer til k?

- Først må vi vise at $SUBSET-SUM$ er i NP.
- Det gjør vi ved å vise at det finnes korte (polynomielt store) sertifikater for ja-instansene.
- Vi blir bedt om å svare på om en delmengde av S summerer til k.
- Sertifikatet kan være et utvalg av elementene i S.
- Med dette som sertifikat er det lett å sjekke at elementene summerer til k.
- Dette stemmer godt med intuisjonen vår: hvis du påstår at du har en ja-instans av $SUBSET-SUM$, og noen ikke tror på deg, kan du bare fortelle dem hvilke elementer de skal velge og så klarer de enkelt å sjekke at du har rett.

Nå som vi vet at $SUBSET-SUM \in NP$, gjenstår kun å vise at problemet er NP-hardt.

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.
- En instans av *PARTITION* er en mengde S_P av heltall, og spørsmålet er om S_P kan partisjoneres inn i to delmengder av samme sum.

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.
- En instans av *PARTITION* er en mengde S_P av heltall, og spørsmålet er om S_P kan partisjoneres inn i to delmengder av samme sum.
- Hvis summen av hele S_P er et oddetall, vet vi at vi har en nei-instans,

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.
- En instans av *PARTITION* er en mengde S_P av heltall, og spørsmålet er om S_P kan partisjoneres inn i to delmengder av samme sum.
- Hvis summen av hele S_P er et oddetall, vet vi at vi har en nei-instans,
- så vi designer en nei-instans av *SUBSET−SUM*, for eksempel $\{1\}$ og $k = 2$.

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.
- En instans av *PARTITION* er en mengde S_P av heltall, og spørsmålet er om S_P kan partisjoneres inn i to delmengder av samme sum.
- Hvis summen av hele S_P er et oddetall, vet vi at vi har en nei-instans,
- så vi designer en nei-instans av *SUBSET−SUM*, for eksempel $\{1\}$ og $k = 2$.
- Hvis summen av S_P er et partall lurer vi jo egentlig på om det går an å finne en delmengde av S_P som summerer til $\frac{\Sigma S_P}{2}$.

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.
- En instans av *PARTITION* er en mengde S_P av heltall, og spørsmålet er om S_P kan partisjoneres inn i to delmengder av samme sum.
- Hvis summen av hele S_P er et oddetall, vet vi at vi har en nei-instans,
- så vi designer en nei-instans av *SUBSET−SUM*, for eksempel $\{1\}$ og $k = 2$.
- Hvis summen av S_P er et partall lurer vi jo egentlig på om det går an å finne en delmengde av S_P som summerer til $\frac{\Sigma S_P}{2}$.
- I dette tilfellet returnerer vi $\langle S = S_P,\ k = \frac{\Sigma S_P}{2} \rangle$.

SUBSET−SUM er *NP*-komplett II

- Vi viser at *SUBSET−SUM* er *NP*-hardt ved å redusere *PARTITION* til *SUBSET−SUM*.
- En instans av *PARTITION* er en mengde S_P av heltall, og spørsmålet er om S_P kan partisjoneres inn i to delmengder av samme sum.
- Hvis summen av hele S_P er et oddetall, vet vi at vi har en nei-instans,
- så vi designer en nei-instans av *SUBSET−SUM*, for eksempel $\{1\}$ og $k = 2$.
- Hvis summen av S_P er et partall lurer vi jo egentlig på om det går an å finne en delmengde av S_P som summerer til $\frac{\Sigma S_P}{2}$.
- I dette tilfellet returnerer vi $\langle S = S_P,\ k = \frac{\Sigma S_P}{2} \rangle$.
- Dette er en ja-instans av *SUBSET−SUM* hvis og bare hvis S_P var en ja-instans av *PARTITION*, så reduksjonen er korrekt.

SUBSET − SUM er *NP*-komplett III

- Det eneste reelle arbeidet reduksjonen vår gjør, er å beregne summen av S_P som kan gjøres i polynomiell tid.

SUBSET−SUM er *NP*-komplett III

- Det eneste reelle arbeidet reduksjonen vår gjør, er å beregne summen av S_P som kan gjøres i polynomiell tid.
- Reduksjonen vi lagde var kun et eksempel på hvordan man kan redusere *PARTITION* til *SUBSET−SUM*, og vi kunne også valgt et annet *NP*-komplett problem å redusere fra.

SUBSET−SUM er *NP*-komplett III

- Det eneste reelle arbeidet reduksjonen vår gjør, er å beregne summen av S_P som kan gjøres i polynomiell tid.
- Reduksjonen vi lagde var kun et eksempel på hvordan man kan redusere *PARTITION* til *SUBSET−SUM*, og vi kunne også valgt et annet *NP*-komplett problem å redusere fra.
- Det viktige er at vi nå vet at *SUBSET−SUM* er *NP*-hardt, som var det siste vi manglet.

SUBSET − SUM er *NP*-komplett III

- Det eneste reelle arbeidet reduksjonen vår gjør, er å beregne summen av S_P som kan gjøres i polynomiell tid.
- Reduksjonen vi lagde var kun et eksempel på hvordan man kan redusere *PARTITION* til *SUBSET − SUM*, og vi kunne også valgt et annet *NP*-komplett problem å redusere fra.
- Det viktige er at vi nå vet at *SUBSET − SUM* er *NP*-hardt, som var det siste vi manglet.
- Vi kan konkludere med at *SUBSET − SUM* er *NP*-komplett.

Mal for å vise at et problem er *NP*-komplett

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$

Mal for å vise at et problem er NP-komplett

Anta at vi har blitt bedt om å vise at problem X er NP-komplett.

1. Vis at $X \in NP$
   1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$
   1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$
   1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid
2. Vis at X er *NP*-hardt

Mal for å vise at et problem er NP-komplett

Anta at vi har blitt bedt om å vise at problem X er NP-komplett.

1. Vis at $X \in NP$
   1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid
2. Vis at X er NP-hardt
   2.1 Velg et kjent NP-komplett problem A, og reduser det til X

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$
   1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid
2. Vis at X er *NP*-hardt
   2.1 Velg et kjent *NP*-komplett problem A, og reduser det til X
   2.2 Tips: velg det problemet som ligner mest på X, siden dette som regel gjør reduksjonen enklere (både å lage og å forstå)

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$
   1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid
2. Vis at X er *NP*-hardt
   2.1 Velg et kjent *NP*-komplett problem A, og reduser det til X
   2.2 Tips: velg det problemet som ligner mest på X, siden dette som regel gjør reduksjonen enklere (både å lage og å forstå)
   2.3 Argumenter for at reduksjonen er korrekt. Dette kan for eksempel gjøres ved å anta at man får inn en ja-instans og at man da produserer en ja-instans, og tilsvarende for nei-instanser

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$
   - 1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   - 1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid
2. Vis at X er *NP*-hardt
   - 2.1 Velg et kjent *NP*-komplett problem A, og reduser det til X
   - 2.2 Tips: velg det problemet som ligner mest på X, siden dette som regel gjør reduksjonen enklere (både å lage og å forstå)
   - 2.3 Argumenter for at reduksjonen er korrekt. Dette kan for eksempel gjøres ved å anta at man får inn en ja-instans og at man da produserer en ja-instans, og tilsvarende for nei-instanser
   - 2.4 Argumenter for at reduksjonen kjører i polynomiell tid (typisk maks en til to linjer)

Mal for å vise at et problem er *NP*-komplett

Anta at vi har blitt bedt om å vise at problem X er *NP*-komplett.

1. Vis at $X \in NP$
   - 1.1 Dette gjør man typisk ved å vise at problemet har korte sertifikater for ja-instansene
   - 1.2 Forklar hvordan man, gitt et sertifikat, kan verifisere ja-instanser i polynomiell tid
2. Vis at X er *NP*-hardt
   - 2.1 Velg et kjent *NP*-komplett problem A, og reduser det til X
   - 2.2 Tips: velg det problemet som ligner mest på X, siden dette som regel gjør reduksjonen enklere (både å lage og å forstå)
   - 2.3 Argumenter for at reduksjonen er korrekt. Dette kan for eksempel gjøres ved å anta at man får inn en ja-instans og at man da produserer en ja-instans, og tilsvarende for nei-instanser
   - 2.4 Argumenter for at reduksjonen kjører i polynomiell tid (typisk maks en til to linjer)
3. Konkluder med at X må være *NP*-komplett, siden vi har vist at $X \in NP$ og at X er *NP*-hardt

HAMPATH er *NP*-komplett

HAMPATH er *NP*-komplett

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

HAMPATH er *NP*-komplett

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

1. *HAMPATH* $\in$ *NP*, siden gitt en riktig rekkefølge av nodene, kan vi enkelt verifisere at de utgjør en Hamiltonsk sti i G

HAMPATH er *NP*-komplett

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

1. *HAMPATH* $\in$ *NP*, siden gitt en riktig rekkefølge av nodene, kan vi enkelt verifisere at de utgjør en Hamiltonsk sti i G
2. Vi viser at *HAMPATH* er *NP*-hardt ved å redusere *Hamiltonsykel* til *HAMPATH*

HAMPATH er *NP*-komplett

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

1. *HAMPATH* $\in$ *NP*, siden gitt en riktig rekkefølge av nodene, kan vi enkelt verifisere at de utgjør en Hamiltonsk sti i G
2. Vi viser at *HAMPATH* er *NP*-hardt ved å redusere *Hamiltonsykel* til *HAMPATH*
3. *Hamiltonsykel* $\leq_P$ *HAMPATH* (neste slide)

HAMPATH er *NP*-komplett

HAMPATH	
Instans:	En graf G
Spørsmål:	Har G en Hamiltonsti?

1. *HAMPATH* $\in$ *NP*, siden gitt en riktig rekkefølge av nodene, kan vi enkelt verifisere at de utgjør en Hamiltonsk sti i G
2. Vi viser at *HAMPATH* er *NP*-hardt ved å redusere *Hamiltonsykel* til *HAMPATH*
3. *Hamiltonsykel* $\leq_P$ *HAMPATH* (neste slide)
4. Vis at reduksjonen kjører i polynomiell tid
5. Hvis vi klarer dette, har vi vist at *HAMPATH* er *NP*-komplett

HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel

HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel
- Vi observerer at sykler ikke egentlig har en start eller en slutt

HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel
- Vi observerer at sykler ikke egentlig har en start eller en slutt
- Lag G_{HP} på følgende måte: Velg en vilkårlig node i G_{HS} og lag er kopi av den

HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel
- Vi observerer at sykler ikke egentlig har en start eller en slutt
- Lag G_{HP} på følgende måte: Velg en vilkårlig node i G_{HS} og lag er kopi av den
- Kopien skal ha samme naboer som den originale, og de skal ikke være naboer med hverandre

HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel
- Vi observerer at sykler ikke egentlig har en start eller en slutt
- Lag G_{HP} på følgende måte: Velg en vilkårlig node i G_{HS} og lag er kopi av den
- Kopien skal ha samme naboer som den originale, og de skal ikke være naboer med hverandre
- Lag to nye noder og koble den ene til originalen og den andre til kopien

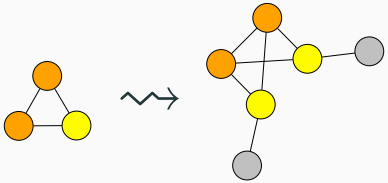
HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel
- Vi observerer at sykler ikke egentlig har en start eller en slutt
- Lag G_{HP} på følgende måte: Velg en vilkårlig node i G_{HS} og lag er kopi av den
- Kopien skal ha samme naboer som den originale, og de skal ikke være naboer med hverandre
- Lag to nye noder og koble den ene til originalen og den andre til kopien

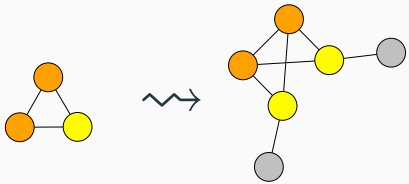


HAMPATH er *NP*-komplett II (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

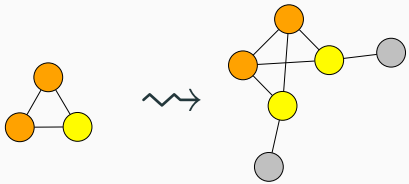
- *Hamiltonsykel* tar inn en graf G_{HS} og spørsmålet er om grafen inneholder en Hamiltonsk sykel
- Vi observerer at sykler ikke egentlig har en start eller en slutt
- Lag G_{HP} på følgende måte: Velg en vilkårlig node i G_{HS} og lag er kopi av den
- Kopien skal ha samme naboer som den originale, og de skal ikke være naboer med hverandre
- Lag to nye noder og koble den ene til originalen og den andre til kopien



HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)

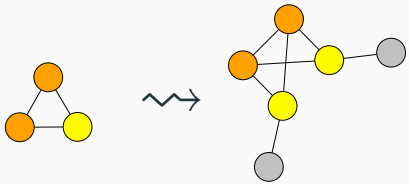


HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



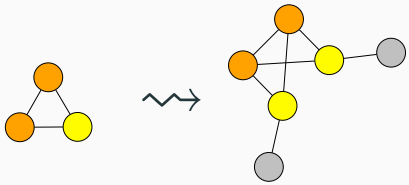
- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti

HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak

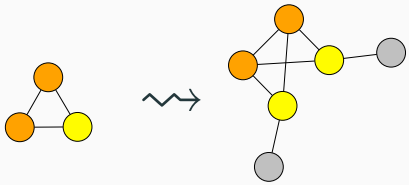
HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel

- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak

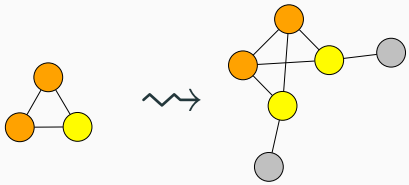
HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel
  - En eventuell hamiltonsk sti må begynne (og slutte) i en grå node

- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak

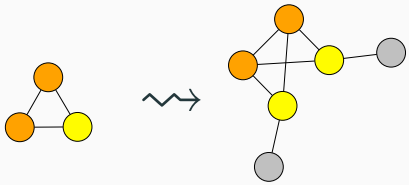
HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak

- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel
  - En eventuell hamiltonsk sti må begynne (og slutte) i en grå node
  - Rett etter (før) en grå node må det være en gul node

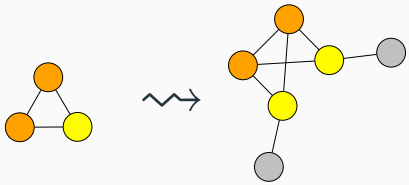
HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak

- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel
  - En eventuell hamiltonsk sti må begynne (og slutte) i en grå node
  - Rett etter (før) en grå node må det være en gul node
  - Nodene i stien mellom de gule må utgjøre "innmaten" av en sykel i G_{HS}

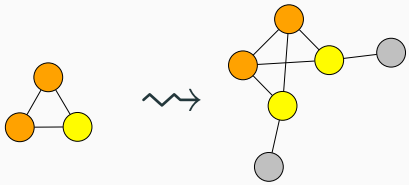
HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak

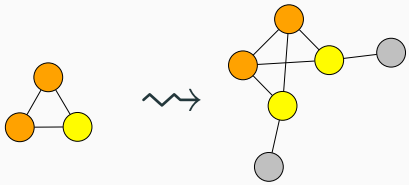
- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel
  - En eventuell hamiltonsk sti må begynne (og slutte) i en grå node
  - Rett etter (før) en grå node må det være en gul node
  - Nodene i stien mellom de gule må utgjøre "innmaten" av en sykel i G_{HS}
- Reduksjonen bruker polynomiell tid

HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak
- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel
  - En eventuell hamiltonsk sti må begynne (og slutte) i en grå node
  - Rett etter (før) en grå node må det være en gul node
  - Nodene i stien mellom de gule må utgjøre "innmaten" av en sykel i G_{HS}
- Reduksjonen bruker polynomiell tid
  - Reduksjonen lager 3 nye noder og potensielt $(|V| - 1) + 2$ nye kanter

HAMPATH er *NP*-komplett III (*Hamiltonsykel* $\leq_P$ *HAMPATH*)



- Hvis G_{HS} hadde en Hamiltonsk sykel, må G_{HP} ha en Hamiltonsk sti
  - Stien blir "innmaten" til sykelen, pluss to ekstra noder foran og bak
- Hvis G_{HP} har en hamiltonsk sti, må G_{HS} ha hatt en Hamiltonsk sykel
  - En eventuell hamiltonsk sti må begynne (og slutte) i en grå node
  - Rett etter (før) en grå node må det være en gul node
  - Nodene i stien mellom de gule må utgjøre "innmaten" av en sykel i G_{HS}
- Reduksjonen bruker polynomiell tid
  - Reduksjonen lager 3 nye noder og potensielt $(|V| - 1) + 2$ nye kanter

TSP er *NP*-komplett

TSP er *NP*-komplett

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

TSP er *NP*-komplett

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

- *TSP* er et klassisk *optimaliseringsproblem*, her formulert som et avgjørelsesproblem

TSP er *NP*-komplett

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

- *TSP* er et klassisk *optimaliseringsproblem*, her formulert som et avgjørelsesproblem
- Nodene representerer for eksempel byer, og vekten til kantene representerer hva det koster å reise mellom byene

TSP er *NP*-komplett

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

- *TSP* er et klassisk *optimaliseringsproblem*, her formulert som et avgjørelsesproblem
- Nodene representerer for eksempel byer, og vekten til kantene representerer hva det koster å reise mellom byene
- Vi blir bedt om å finne en Hamiltonsk sykel, av kost mindre enn k

TSP er *NP*-komplett

TSP (Traveling Salesman Problem)	
Instans:	En vektet komplett graf G og et heltall k
Spørsmål:	Finnes det en rekkefølge man kan traversere hele G, og ende opp der du startet, slik at summen av kosten mellom nodene er mindre enn k?

- *TSP* er et klassisk *optimaliseringsproblem*, her formulert som et avgjørelsesproblem
- Nodene representerer for eksempel byer, og vekten til kantene representerer hva det koster å reise mellom byene
- Vi blir bedt om å finne en Hamiltonsk sykel, av kost mindre enn k
- I *Hamiltonsykel* jobber vi med uvektede grafer, en reduksjon må derfor gjøre grafen vektet

TSP er *NP*-komplett II

TSP er *NP*-komplett II

- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.

TSP er *NP*-komplett II

- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.

TSP er *NP*-komplett II

- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.
- Gitt en graf G, sett vekten til alle kantene til 1.

TSP er *NP*-komplett II

- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.
- Gitt en graf G, sett vekten til alle kantene til 1.
- For alle par av noder i G som det ikke går en kant mellom, legg til en kant med vekt 2. Dette sørger for at grafen blir komplett. Vi setter $k = |V| + 1$.

TSP er *NP*-komplett II

- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.
- Gitt en graf G, sett vekten til alle kantene til 1.
- For alle par av noder i G som det ikke går en kant mellom, legg til en kant med vekt 2. Dette sørger for at grafen blir komplett. Vi setter $k = |V| + 1$.
- En hamiltonsk sykel i den originale grafen tilsvarer en reiserute av kost k i den nye grafen og vice versa.

TSP er *NP*-komplett II

- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.
- Gitt en graf G, sett vekten til alle kantene til 1.
- For alle par av noder i G som det ikke går en kant mellom, legg til en kant med vekt 2. Dette sørger for at grafen blir komplett. Vi setter $k = |V| + 1$.
- En hamiltonsk sykel i den originale grafen tilsvarer en reiserute av kost k i den nye grafen og vice versa.
- Kravet om at kosten til traverseringen skal være mindre enn $|V| + 1$ gjør at man aldri kan benytte kantene med vekt 2.

TSP er *NP*-komplett II

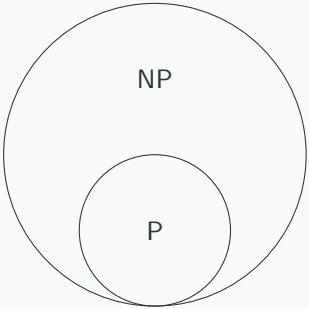
- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.
- Gitt en graf G, sett vekten til alle kantene til 1.
- For alle par av noder i G som det ikke går en kant mellom, legg til en kant med vekt 2. Dette sørger for at grafen blir komplett. Vi setter $k = |V| + 1$.
- En hamiltonsk sykel i den originale grafen tilsvarer en reiserute av kost k i den nye grafen og vice versa.
- Kravet om at kosten til traverseringen skal være mindre enn $|V| + 1$ gjør at man aldri kan benytte kantene med vekt 2.
- Vi legger til maksimalt $O(|V|^2)$ kanter, som gjør at reduksjonen kjører i polynomiell tid.

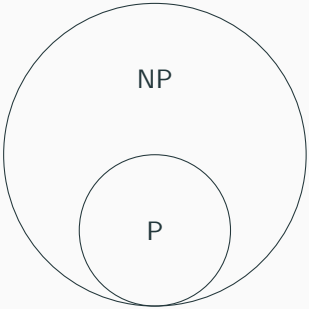
TSP er *NP*-komplett II

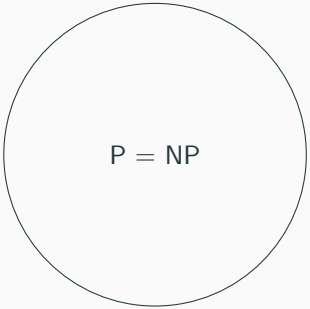
- $TSP \in NP$ fordi vi kan bruke rekkefølgen som sertifikat.
- Vi viser at *TSP* er *NP*-hardt ved å redusere *Hamiltonsykel* til *TSP*.
- Gitt en graf G, sett vekten til alle kantene til 1.
- For alle par av noder i G som det ikke går en kant mellom, legg til en kant med vekt 2. Dette sørger for at grafen blir komplett. Vi setter $k = |V| + 1$.
- En hamiltonsk sykel i den originale grafen tilsvarer en reiserute av kost k i den nye grafen og vice versa.
- Kravet om at kosten til traverseringen skal være mindre enn $|V| + 1$ gjør at man aldri kan benytte kantene med vekt 2.
- Vi legger til maksimalt $O(|V|^2)$ kanter, som gjør at reduksjonen kjører i polynomiell tid.
- Siden vi har vist at *Hamiltonsykel* $\leq_P$ *TSP* og at $TSP \in NP$, har vi vist at *TSP* er *NP*-komplett.

P **vs** *NP*

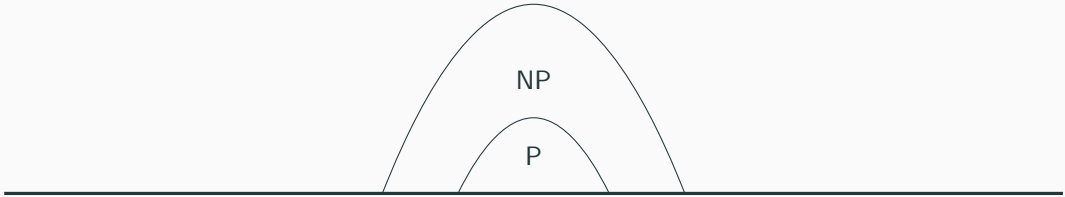
P **vs** *NP*



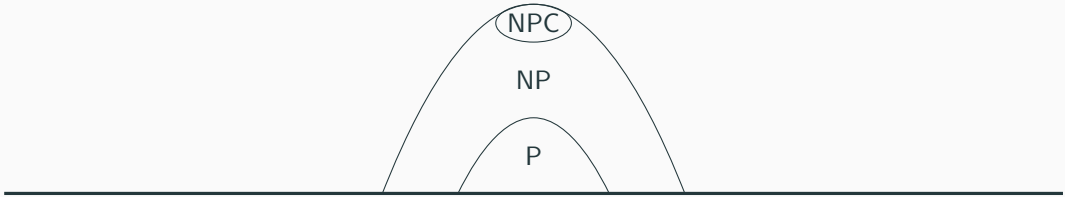

NP
P


P = NP

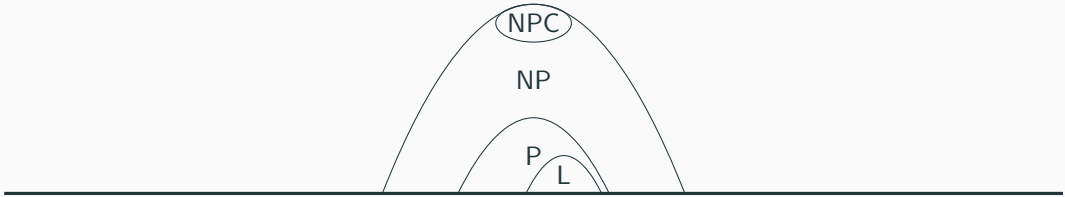
Noen få klasser og hvordan vi tror verden ser ut



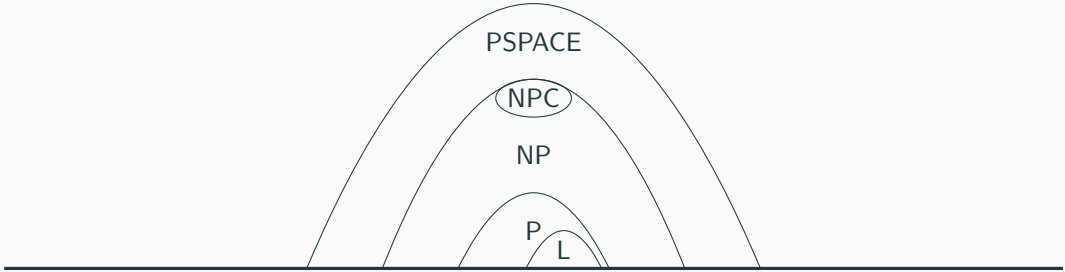
Noen få klasser og hvordan vi tror verden ser ut



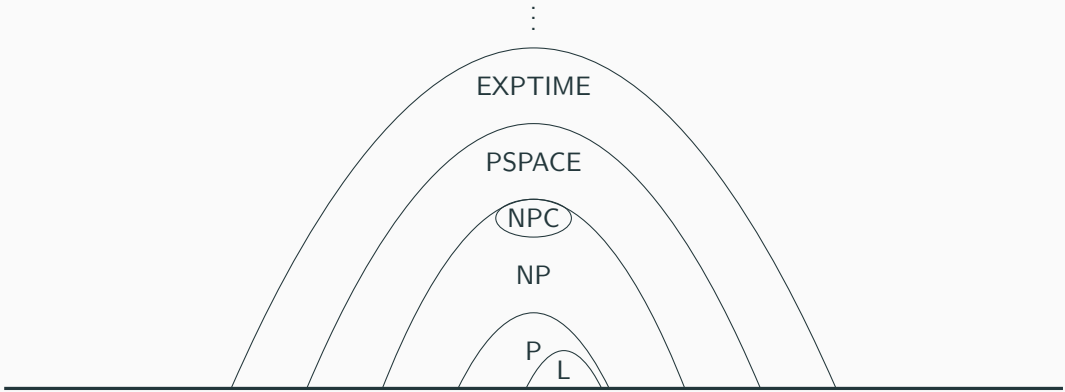
Noen få klasser og hvordan vi tror verden ser ut



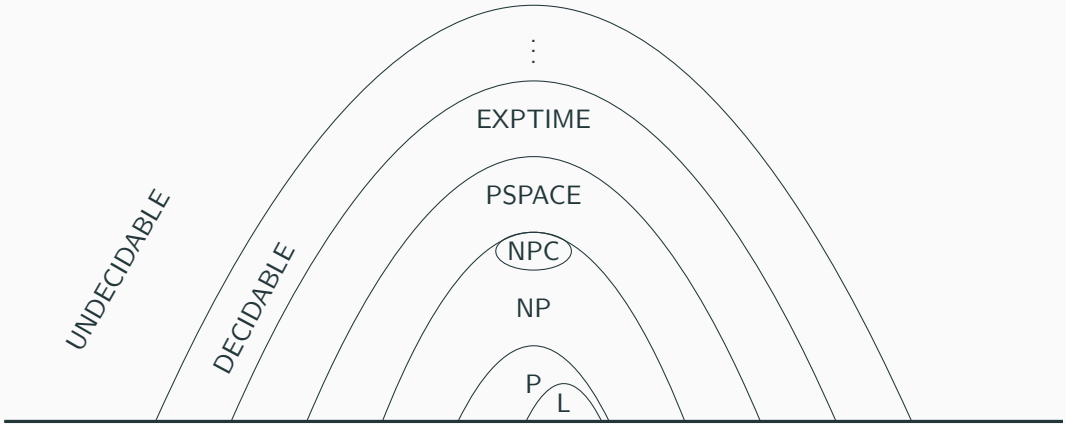
Noen få klasser og hvordan vi tror verden ser ut



Noen få klasser og hvordan vi tror verden ser ut



Noen få klasser og hvordan vi tror verden ser ut



HALT er uavgjørbart

HALT er uavgjørbart

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Vi viser at *HALT* er uavgjørbart ved et *motsigelsesbevis*.

HALT er uavgjørbart

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Vi viser at *HALT* er uavgjørbart ved et *motsigelsesbevis*.

- Anta at *HALT* er avgjørbart.

HALT er uavgjørbart

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Vi viser at *HALT* er uavgjørbart ved et *motsigelsesbevis*.

- Anta at *HALT* er avgjørbart.
- Det vil si at det finnes en algoritme, H, som i endelig tid svarer riktig (ja/nei) på alle instanser av *HALT*.

HALT er uavgjørbart

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Vi viser at *HALT* er uavgjørbart ved et *motsigelsesbevis*.

- Anta at *HALT* er avgjørbart.
- Det vil si at det finnes en algoritme, H, som i endelig tid svarer riktig (ja/nei) på alle instanser av *HALT*.
- Vi lager en ny algoritme D, som bruker H som *subrutine*.

HALT er uavgjørbart

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Vi viser at *HALT* er uavgjørbart ved et *motsigelsesbevis*.

- Anta at *HALT* er avgjørbart.
- Det vil si at det finnes en algoritme, H, som i endelig tid svarer riktig (ja/nei) på alle instanser av *HALT*.
- Vi lager en ny algoritme D, som bruker H som *subrutine*.
- D tar en algoritme A som input:

HALT er uavgjørbart

HALT (stoppeproblemet)	
Instans:	En algoritme/et program A, og en input til A, w
Spørsmål:	Terminerer A på input w?

Vi viser at *HALT* er uavgjørbart ved et *motsigelsesbevis*.

- Anta at *HALT* er avgjørbart.
- Det vil si at det finnes en algoritme, H, som i endelig tid svarer riktig (ja/nei) på alle instanser av *HALT*.
- Vi lager en ny algoritme D, som bruker H som *subrutine*.
- D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   1.2 Hvis $H(A, A) = NEI$: output JA

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.
- Hva skjer om vi kjører D med D som input?

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.
- Hva skjer om vi kjører D med D som input?
- Hvis $H(D, D) = JA$, terminerer *ikke* D på input D,

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.
- Hva skjer om vi kjører D med D som input?
- Hvis $H(D, D) = JA$, terminerer *ikke* D på input D,
- Hvis $H(D, D) = NEI$, terminerer jo D på input D.

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.
- Hva skjer om vi kjører D med D som input?
- Hvis $H(D, D) = JA$, terminerer *ikke* D på input D,
- Hvis $H(D, D) = NEI$, terminerer jo D på input D.
- Altså får vi en selvimotsigelse i begge tilfeller.

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.
- Hva skjer om vi kjører D med D som input?
- Hvis $H(D, D) = JA$, terminerer *ikke* D på input D,
- Hvis $H(D, D) = NEI$, terminerer jo D på input D.
- Altså får vi en selvimotsigelse i begge tilfeller.
- Siden D kan konstrueres, gitt H, konkluderer vi med at H ikke kan finnes.

HALT er uavgjørbart II

D tar en algoritme A som input:

1. På input A: Sjekk $H(A, A)$, (Sjekk om A terminerer med seg selv som input)
   - 1.1 Hvis $H(A, A) = JA$: gå inn i en evig løkke
   - 1.2 Hvis $H(A, A) = NEI$: output JA

- D går altså i en evig løkke hvis A terminerer med seg selv som input.
- Hvis A *ikke* terminerer (med seg selv som input), terminerer D.
- Hva skjer om vi kjører D med D som input?
- Hvis $H(D, D) = JA$, terminerer *ikke* D på input D,
- Hvis $H(D, D) = NEI$, terminerer jo D på input D.
- Altså får vi en selvimotsigelse i begge tilfeller.
- Siden D kan konstrueres, gitt H, konkluderer vi med at H ikke kan finnes.
- Siden H var en vilkårlig algoritme som løste *HALT* og H ikke finnes, kan vi konkludere med at *ingen* algoritme løser *HALT*, altså er *HALT* *uavgjørbart*.