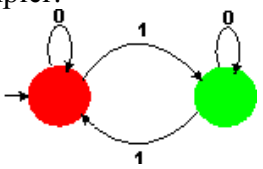
NP-kompletthet

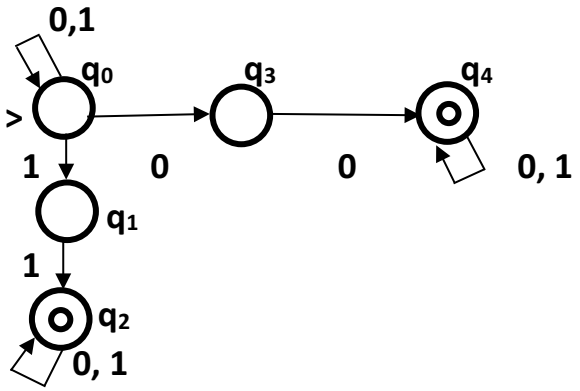
Endelig tilstandsmaskin

- **FSM** Finite State Machine (Endelig Tilstandsmaskin)
- **Endelig** Tilstandsmaskinen består av et begrenset/endelig antall tilstander.
- **Uendelig** Tilstandsmaskinen består av et ubegrenset/uendelig antall tilstander.
- **Deterministisk** Entydig bestemt hva som skjer (hvor man havner/hva man videre skal gjøre) ved en input.
- **Ikke-deterministisk** *Ikke entydig* bestemt hva som skjer (hvor man havner/hva gjøre) ved en input, dvs. har mulighet for å gå til flere ulike tilstander. Men, fører *en* av veiene gjennom FSM'en fram til en sluttilstand (svar), så betyr det at FSM'en godtar input (har funnet svaret).

  Enhver ikke-deterministisk FSM kan skrives om til en deterministisk FSM !!!
- **Turing-maskin** En utvidelse av FSM, som kan lese *og* skrive (ut) *både* forover og bak-over i en uendelig input-stream. Har ingen spesielle sluttilstander.
- To *meget enkle* eksempler:



1. Deterministisk – godtar odde antall 1'ere:
2. *Ikke*-deterministisk, som aksepterer et vilkårlig antall 0'er og 1'ere, der det *minst ett sted* forekommer to etterfølgende 0'er *eller* 1'ere.



Orden

Har i dette emnet sett at mange algoritmer er av orden noe med: n^2 eller log N.
De siste finner ofte fort/raskt svaret. De som tar mye/«uendelig» med tid er av orden:

1. $\mathbf{n^x}$ **polynomiell / polynomisk**
2. $\mathbf{x^n}$ **eksponensiell**
3. **n!** **faktoriell**

De polynomielle kalles ofte «lette problemer» (i hvert fall om x er *meget* liten (2, 3, 4, …)), da det er innenfor rimelighetens grenser rent tids- og ressursmessig å løse dem.
De eksponensielle/faktorielle kalles «harde/vanskelige problemer»

Kompleksitetsklasser

Ut fra hvordan problemer er å løse, tilhører de ulike kompleksitetsklasser:

P Klasse som består av *alle* problemer som kan løses i polynomiell tid

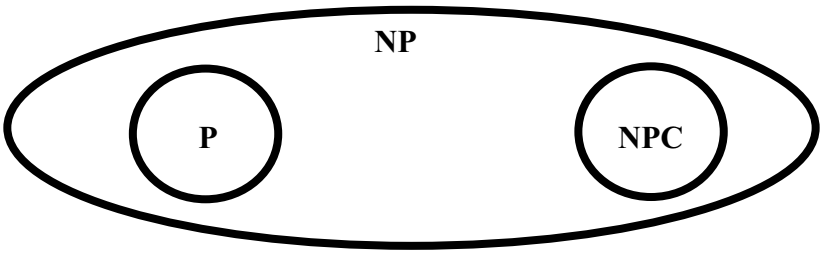
NP (= Non-deterministic Polynomial)
Klasse som består av *alle* problemer som kan løses i polynomiell tid på en ikke-deterministisk FSM. For: 1) dersom maskinen «gjetter»/kommer frem til svaret, uten

nødvendigvis å funnet/generert alle løsninger, så 2) kan det i polynomisk tid sjekkes/verifiseres om svaret er rett. Men, vi har selvsagt ingen garanti for at det «gjettes» riktig. Skulle problemet ha vært løst systematisk/fullstendig trengs det eksponensiell tid.

P ⊆ NP, da alle problemer som kan løses i polynomisk tid på en deterministisk FSM også kan løses i polynomisk tid på en ikke-deterministisk FSM.

NP-komplett

NP-komplette (NPC) problemer er en *samling av «beslektede» (kan på polynomisk tid reduseres/transformeres til hverandre – derfor «komplett») NP problemer* som er særdeles interessante. *De regnes for å være av de verre NP problemene å løse.* Det er slik at også NPC ⊆ NP (men det betyr ikke at P = NPC).



Det finnes altså ikke effektive algoritmer for garantert å *løse* dem i polynomiell tid. De er også spesielle (pga. slektskapet) fordi, om det blir funnet en effektiv algoritme for å løse *ett* av dem, så finnes det en effektiv algoritme for *alle*! Det motsatt er også tilfelle: kan det bevises at *ett* av problemene ikke har noen effektiv løsning, så gjelder det for *alle* de andre også! Men, vi har ennå ikke noe svar på dette! Og mange mener vi aldri vil få svar på det heller.
Selv om problemene i NP er vanskeligere enn de i P (og NPC av de verste i NP), så er det hverken bevist at P = NP eller at P ≠ NP (Clay Mathematics Institute har utlovet en premie på $1.000.000 til den som finner svaret på dette! Mye innen datasikkerhet, bl.a. digitale signaturer forutsetter at de er ulike (som mange antar) – siden de bl.a. baserer seg på at saker tar for lange tid å beregne.)
Det er derfor teorien for NPC problemer er utviklet. Det første var SAT (se eget punkt nedenfor). Den ble primært laget for å kunne besvare «Ja/Nei»-problemer («true/false» - decision problems), f.eks: er et tall et primtall eller ei. NPC problemer (ja/nei) må kunne løses i polynomiell tid på en ikke-deterministisk Turingmaskin. Siden ikke-deterministiske FSM er en imaginær modell av en datamaskin, må man i praksis datamessig kode dette som en deterministisk maskin. Da brukes ofte teknikken «*backtracking*», og som for disse NPC problemene ender ut i eksponensiell vekst i tid.

Hvordan vise at et problem er NPC?

Å bevise at et problem er NPC kan være en vrien oppgave, men det finnes muligheter. Når vi først har bevist at et problem er NPC blir arbeidet med å bevise at andre problem også er NPC mye lettere. Gitt et problem X som vi vet er i NP, så kan vi vise at det er NPC ved å velge et allerede kjent NPC problem Y, og vise at *Y* kan reduseres til X i polynomiell tid. Fremgangsmåte:

1. Vis at X er i NP
2. Velg et kjent NPC problem Y
3. Konstruer en transformasjon T fra Y til X
4. Bevis at T er en polynomiell transformasjon.

NB: Legg merke til at vi må vise at et eller annet kjent NPC problem (Y) kan transformeres i polynomiell tid *til* vårt nye problem (X) og ikke motsatt.

Noen mye brukte/omtalte eksempler på NPC problemer (det er noen tusen av dem)

Se: https://en.wikipedia.org/wiki/List_of_NP-complete_problems

1. Subset Sum (spesialtilfelle av **Knapsack Problem)**

Vi har en mengde A = { x_1, x_2, ...x_n} med n positive heltall og et annet positivt heltall G, som vi kaller målet. Problemet er å finne en delmengde av mengden A med den største summen som ikke overskrider G. En praktisk versjon av problemet kan være hvis A er en mengde med n forskjellige gullklumper og G er maks vekt som ryggsekken vår tåler. Spørsmålet er nå hvor mye gull kan vi maksimalt ta med oss? En ja/nei versjon av problemet kan være om vi kan fylle sekken nøyaktig til bristepunktet. Vi har 2^n forskjellige delmengder av en mengde med n elementer, og skal vi sjekke alle brukes $O(2^n)$ tid.
(Dette er NPC fordi:
Vi lager en transformasjon fra Partition-problemet (som forutsettes å være NPC). Partition-problemet er å avgjøre om en gitt en mengde med heltall kan *deles opp i to delmengder, der summen av tallene er lik.* Det må vises at alle tilfeller av Partition kan transformeres til et tilfelle av Subset Sum. La A = { x_1, x_2, ...x_n} være en mengde positive heltall som vi ønsker å dele i to delmengder med samme sum. Anta at summen av elementene i A er S, slik at hver av delmengdene får summen S/2. Vi kan konstruere et tilfelle av Subset Sum med de samme heltallene og mål G = S/2. Derfor kan heltallene x_1, x_2, ...x_n deles inn i to mengder med samme sum hvis og bare hvis en eller annen delmengde av tallene summerer seg til S/2. Dermed ser vi at Subset Sum er NPC.)

2 . Multiprocessor Scheduling

Multiprocessor Scheduling er følgende problemstilling: Gitt flere mikroprosessorer og prosesser som ikke nødvendigvis har samme tidsforbruk. Prosessene kan ikke deles, men vi kan fordele prosessene som vi vil på prosessorene. Kan alle prosessene være ferdig innen en gitt tidsfrist?
(For å vise at dette er NPC kan vi videre prøve å lage en transformasjon fra Subset Sum. Gitt et tilfelle fra Subset sum. La x_1, x_2, ...x_n være positive heltall med mål G. La S være summen av tallene. Vi konstruerer så et Scheduling problem med n+2 prosesser som krever henholdsvis x_1, x_2, ...x_n, G+l, S-G+1 tid. Den totale prosessortiden som trengs er mao. 2S+2. Vi har to mikroprosessorer til disposisjon og deadline er S+1. Vi må vise at alle n+2 prosessene kan gjøres ferdig av to prosessorer på S+1 tid hvis og bare hvis summen til en eller annen delmengde av x_1, x_2, ...x_n er G. Anta først at en delmengde U av x_1, x_2, ...x_n summeres til G. Da kan prosessor nr.1 tildeles prosessene med tidsforbruk som tilsvarer delmengden U i tillegg til prosessen som krever S- G+1 tid. Resten tildeles prosessor nr.2 og arbeidet blir ferdig på S+1 tid.
Anta nå at mikroprosessorene gjør jobben ferdig på S+1 tid. Prosessene som tar G+l og S-G+l tid ikke kan tildeles samme prosessor siden den da ville være opptatt minst S+2 tid. Tidsforbruket for de andre prosessene som er tildelt prosessoren som utfører S-G+1 prosessen er G, og derfor må summen av en delmengde av x_1, x_2, ...x_n være G.)

3 .Satisfiability (SAT)

Teorien om NPC problemer begynte i 1971 da Stephen Cook viste at Satisfiability (SAT)-problemet er komplett for klassen NP. Dette betyr at hvis SAT kan løses i polynomiell tid, så kan alle andre problem i NPC det også, og motsatt: kan man bevise at et annet NPC problem er eksponensielt, så gjelder det også for SAT. Problemet omhandler om boolske uttrykk på konjunktiv normalform (CNF) kan oppfylles. En variant er: **Circuit-SAT**: Vi har en krets med logiske porter og én utverdi. Kan denne verdien bli 1 (en)?

4. **Hamiltonian Circuit**
   Gitt en graf. Er det mulig å konstruere en vei gjennom grafen som går innom alle nodene og ender opp der vi startet, og slik at alle nodene er besøkt nøyaktig en gang?

5. **Traveling Salesman**
   Et særtilfelle av den forrige. Finnes det en vei med lengde mindre enn et gitt tall/hva er kortest vei?

6. **Longest Path**
   Hva er den lengste veien gjennom en graf, uten at noder besøkes flere ganger?

7. **Colorability**
   Fargelegging av en graf slik at ingen av nabonodene har samme farge. Kan grafen fargelegges med mindre enn k farger?

8. **Clique**
   En Clique er en mengde av noder der det er en kant mellom ethvert par av noder. Finnes det en sammenhengede/komplett delgraf av Clique med k noder?

9. **Vertex Cover**
   Er en delmengde av alle nodene i grafen slik at alle kantene har minst et endepunkt i mengden. Har vi en slik delmengde med størrelse k?

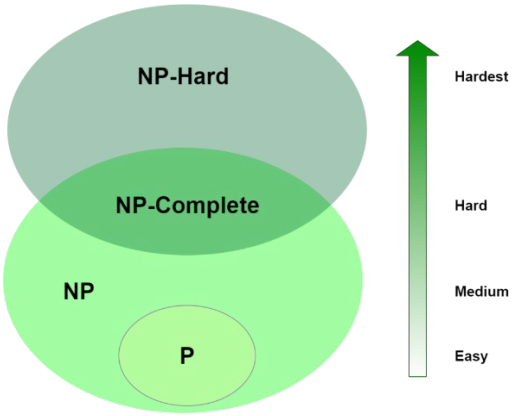
10. **Set Cover and exact Cover**
    Et set cover er flere mengder som tilsammen dekker universet og exact cover er flere disjunkte mengder som tilsammen dekker universet. Spørsmålet er: Kan universet dekkes med unionen av k mengder?

11. **3-D Matching**
    Det klassiske gifteproblemet er et typisk 2-D matching problem. 3-D blir f.eks: Gitt tre like lange lister med mennesker, arbeidsplasser og bosted og regler for akseptable tildelinger. Kan vi tildele et passende arbeid og bosted for hver person?

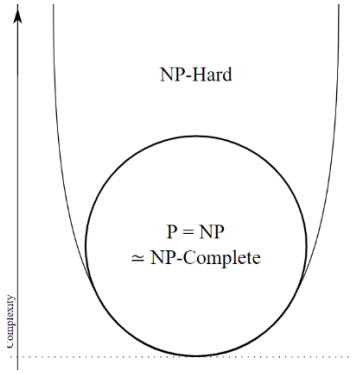
Og enda litt til ……:

CoNP Klasse som består av alle problemer som er komplementet til et problem i NP (de med «ja»-svar). Dvs. *vis* at det *ikke* finnes en løsning («nei»-svar). Man *vet* ikke om NP = CoNP, men man *antar* at de er ulike (NP ≠ CoNP).

NP-Hard (NPH) Klasse av problemer som er enda vanskeligere å løse enn NPC problemer (og da også generelle NP problemer, som jo er «enklere» enn de i NPC).



(Figur fra: https://www.baeldung.com/cs/p-np-np-complete-np-hard)



(Om P = NP. Figur fra: https://en.wikipedia.org/wiki/NP-hardness)

Eksempler på *totalt uløselige/uavgjørbare* problemer:

- Halting Problem: Vil et *tilfeldig* dataprogram noen gang stoppe?
- Det *perfekte* sjakkspill (mer kombinasjoner enn molekyler i universet!(?))

Kilder:

https://munin.uit.no/bitstream/handle/10037/2341/thesis.pdf (Del 2, side 11-24)
(digital versjon: https://docplayer.me/27012075-Hovedfagsoppgave-matematikk-ore-tromso-veileder-ben-johnsen.html)