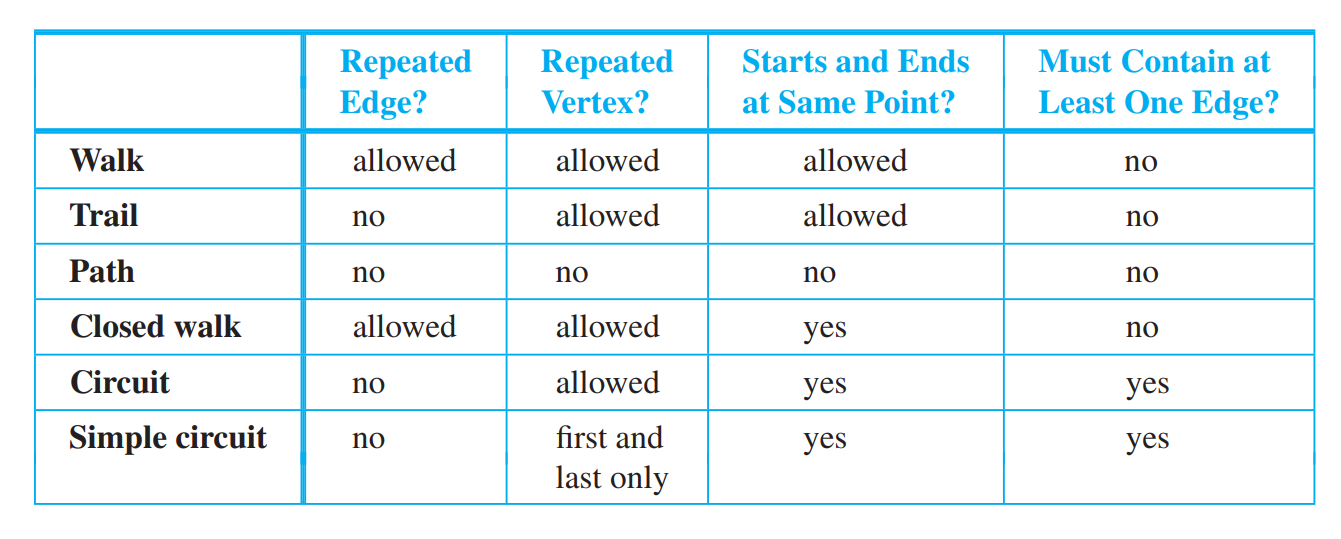
**Komplette** **grafer**: En komplett graf med **n** noder, Kn, er en simpel graf med **n** noder og nøyaktig en kant som kobler de sammen parvis.

**Subgraf**: en graf H er en subgraf av graf G hvis, og bare hvis, hver node i H også er en node i G, hver kant i H også er en kant i G, og hver kant i H har samme endepunkt som den har i G.

I enhver graf G er totalgraden det dobbelte av antall kanter. Totalgraden er alltid et partall. I enhver graf er antall odde grader et partall.

En graf er sammenkoblet dersom man kan gå fra en hvilken som helst node til en annen hvilken som helst node i grafen.



**Eulerkrets**: en krets i en graf G som inneholder alle noder og kanter i G. En Eulerkrets er en rekke noder og kanter i G som har minst en kant, starter og slutter i samme node, bruker hver node minst en gang og bruker hver kant nøyaktig en gang. **Kan gå gjennom samme node flere ganger.** Hvis minst en node i grafen har odd grad, har ikke grafen en Eulerkrets. Dersom en graf G er sammenhengende og enhver node i G har partall som grad, har G en Eulerkrets.

**Eulerspor:** Et Eulerspor fra **v** til **w** er en rekke noder og kanter som starter i v, slutter i w går gjennom hver node i G minst en gang og går gjennom hver kant nøyaktig en gang. Dersom en graf G er sammenhengende, v og w har odde grad og alle andre noder har partallsgrad finnes det et Eulerspor fra v til w.

**Hamiltonkrets:** simpel krets som inkluderer enhver node i G. Er altså en rekke noder og kanter slik at man går gjennom hver node nøyaktig en gang bortsett fra første og siste. Trenger **ikke** å inkludere alle kanter i grafen.

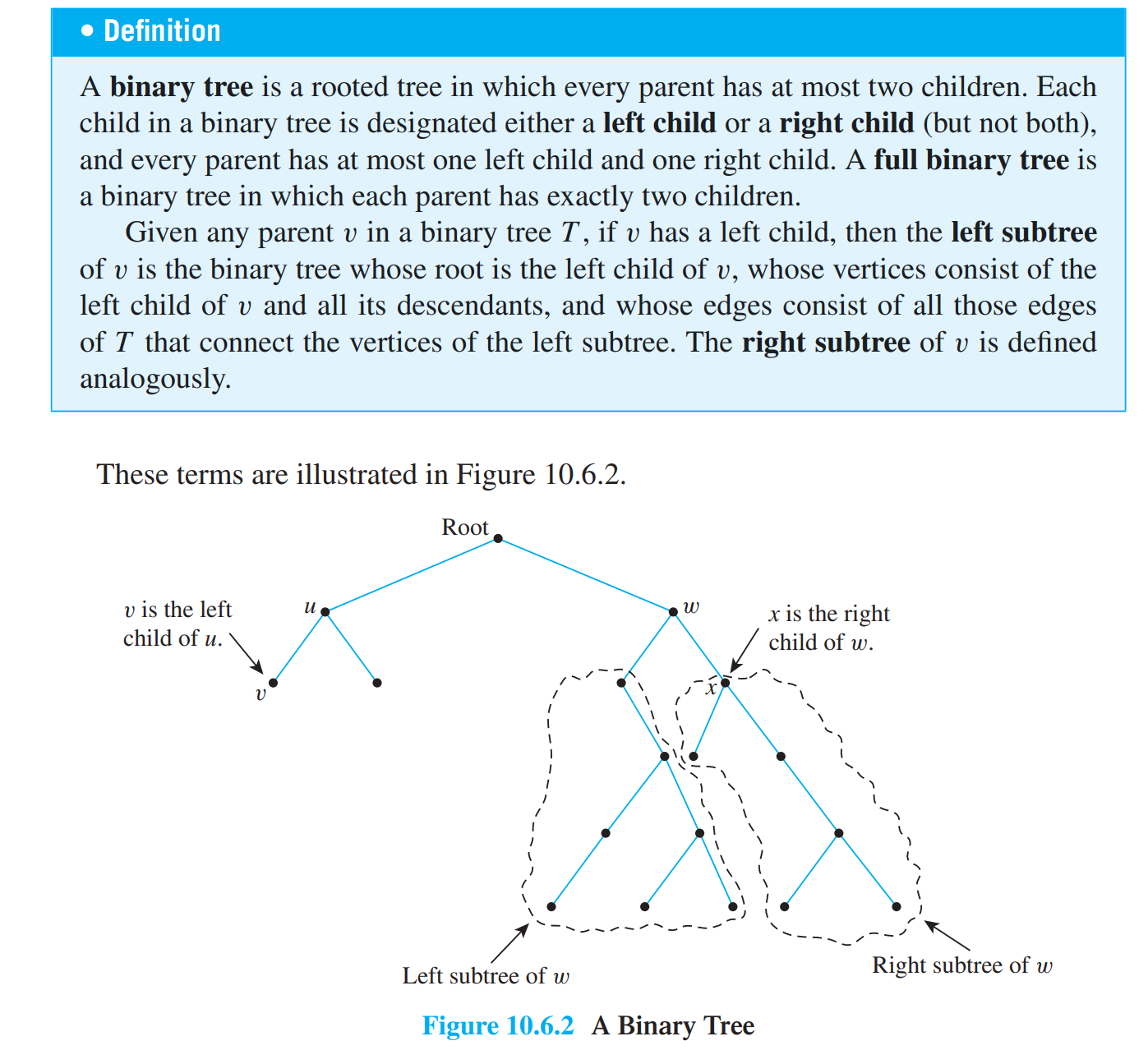
Dersom en graf G har en hamiltonkrets, så har G en subgraf H med følgende egenskaper: 1. H inneholder enhver node i G, H er sammenhengende, H har like mange kanter som noder, enhver node i H har grad 2.

**Isomorfe grafer:** To grafer som er like bortsett fra navngivingen av nodene og kantene er isomorfe.

**Invarianter for grafisomorfi:** Dersom graf G har egenskapen P og graf G’ er isomorf med G, så har også G’ egenskapen P

1. Har n kanter
2. Har m kanter
3. Har en node med grad k
4. Har m noder med grad k
5. Har en krets med lengde k
6. Har en simpel krets med lengde k
7. Har m simple kretser med lengde k
8. Er sammenhengende
9. Har en Eulerkrets
10. Har en Hamiltonkrets

**Trær:** En graf G er et tre dersom det er sammenhengende og ikke inneholder en krets. Ethvert tre med **n** noder har **n – 1** kanter. Enhver sammenhengende graf med **n** noder og **n – 1** kanter er et tre. Ethvert tre som har mer enn en node må ha minst en node med grad 1. Dersom en graf G er en sammenhengende graf med **n** noder og **n – 1** kanter, er G et tre.



**Tidskompleksitet**: O – øvre grense for kjøretid. Ω - nedre grense for kjøretid. Tetta – både øvre og nedre grense for kjøretid.

**Rekursjon:** Tidsforbruk, lineær rekursjon: T(1) = 1, T(2) = 2, T(3) = 3 🡪 T(n) = n 🡪 Tetta(n). Tidsforbruk to rekursive kall med størrelse (cirka) n – 1: T(1) = 1, T(2) = 3, T(n) = 2n – 1. Tidsforbruk, n rekursjoner av størrelse n – 1: T(1) = 1, T(2) = 2, T(n) = n!

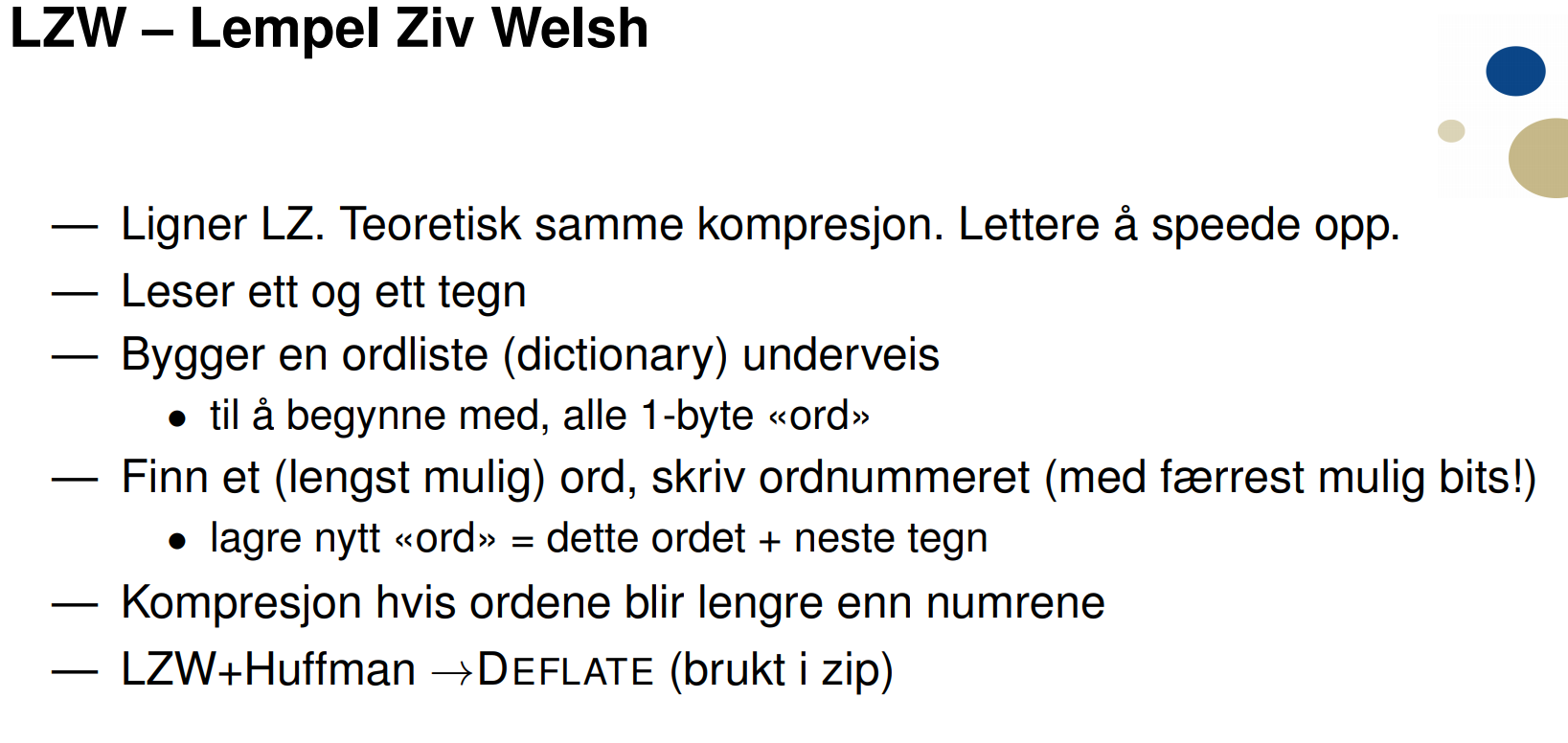
**Generell formel:** T(n) = 1 når n = 1, aT(n/b) + cnk når n > 1.   
bk < a 🡪 T(n) = Tetta(nlog(b)a)  
bk = a 🡪 T(n) = Tetta(nk \* log(n))  
bk > a 🡪 T(n) = nk.

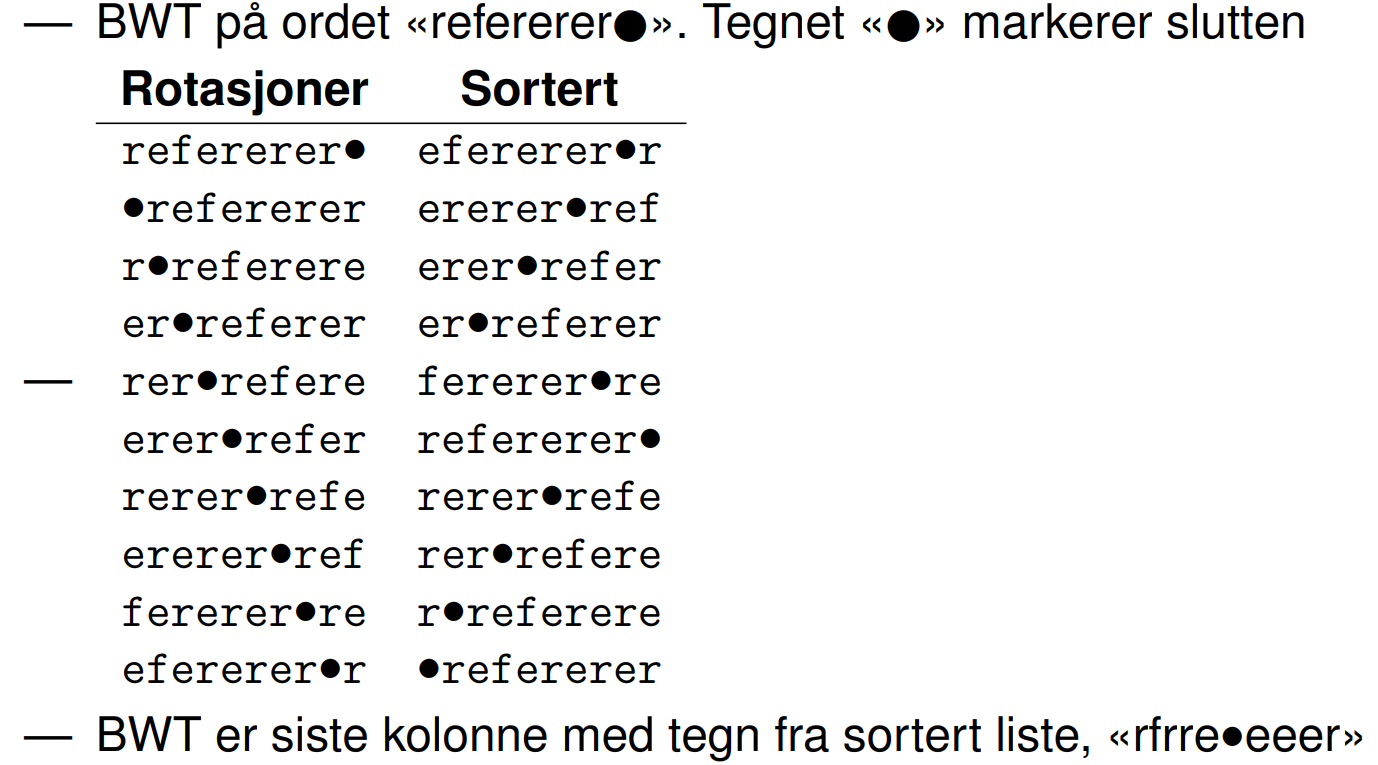
**Sortering**: Hvilken sorteringsalgoritme som er best avhenger av problemet: hvor stor datamengde som skal sorteres, og i hvilken grad den allerede er sortert. Fordel om algoritmen er effektiv på tall som står i tilfeldig orden. **Innsettingssortering:** enkel for små datamengder. Ω(n), O(n2). O(n2) for det generelle tilfellet. Raskere hvis datamengden nesten er sortert fra før, og kan da være bedre enn tradisjonelt sett raskere algoritmene. Rask på svært små datamengder. **Boblesortering:** enkel og lettfattelig. Lite brukt, innsetting er like enkel og oftest raskere i praksis. Tetta(n2). **Velgesortering:** velger det største tallet og plasserer dette rett, dvs. i enden av tabellen. Deretter blir nest største satt til rett, plassen bak osv. Tetta(n2). Kan være bedre hvis man arbeider med en form for minne hvor skriving tar mye lenger tid enn lesing. **Shellsort:** subkvadratisk forbedring av innsetting. O(n2). **Flettesortering:** splitt og hersk. Har en tabell med n tall. Denne deles på to. Behandler hver rekursivt med flettesortering. T(n) = O(nlog(n)). **Quicksort:** raskeste sorteringsalgoritmen for generelle data. Splitt og hersk. Tabellen deles i to deler hvor alle tall i den laveste delen har lavere verdi enn alle tall i den høyeste delen. Trenger ikke deles på midten. De to deltabellene sorteres hver for seg med rekursiv bruk av quicksort. Får deltabeller med bare et tall i. Tetta(n \* log(n)) dersom man får perfekt deling hver gang. Tetta(n2) dersom delingen blir verst hver gang. Gjennomsnittlig kjøretid O(n \* log(n)). I praksis raskere enn andre O(n \* log(n)) algoritmer. Finnes verste tilfelle O(n2). **Sortering generelt:** alle må bruke Ω(n \* log(n)). Kan aldri komme under O(n). **Tellesortering:** Tetta(n + k)

**Trær:** Noder uten barn kalles **løvnoder**, mens noder med barn er **indre noder**. **Binærtre:** tre der hver node kan ha maksimalt to barn. Et binærtre er **perfekt** dersom alle løvnoder (noder uten barn) befinner seg på nederste nivå. Et binærtre er **komplett** dersom det er perfekt bortsett fra at det kan mangle noen noder lengst til høyre i nederste nivå. Et binærtre er **fullt** dersom alle indre noder har akkurat to barn.

**Heap:** finnes to slags heaper: min og max. En heap er et komplett binærtre hvor hver node inneholder en nøkkelverdi. Må også ha **heapegenskapen:** I en max-heap har hver node en nøkkelverdi som er større enn eller lik begge barnenodenes. Rota har den største nøkkelen. I en min-heap har hver node en nøkkelverdi som er mindre enn eller lik barnenodenes, dermed har rota den minste verdien. **Anvendelser:** sortering og prioritetskø.

**Avanserte programmeringsteknikker:** Rå kraft, prøver alle muligheter og velger den beste. Tar mye tid og er siste utvei. **Splitt og hersk:** deler et stort problem opp i mindre problem. Løses separat og settes sammen. To varianter er **dynamisk programmering** og **grådige algoritmer**. Dynamisk programmering brukes når vi må prøve mange kombinasjoner av løsninger på delproblemer. Kan ende opp med å løse samme delproblemer flere ganger. Lagrer derfor løsninger på delproblemer i en tabell. Sjekker tabellen først, ligger løsninger der brukes resultatet. Eksempel: ryggsekkproblemet. **Grådige algoritmer:** gjør grådige valg, det som ser best ut der og da. Går aldri tilbake på disse, og tenker ikke fremover. Raske og effektive, hvis problemet lar seg løse. Eksempler: prim og kruskal, dijkstras og huffmannkoding.

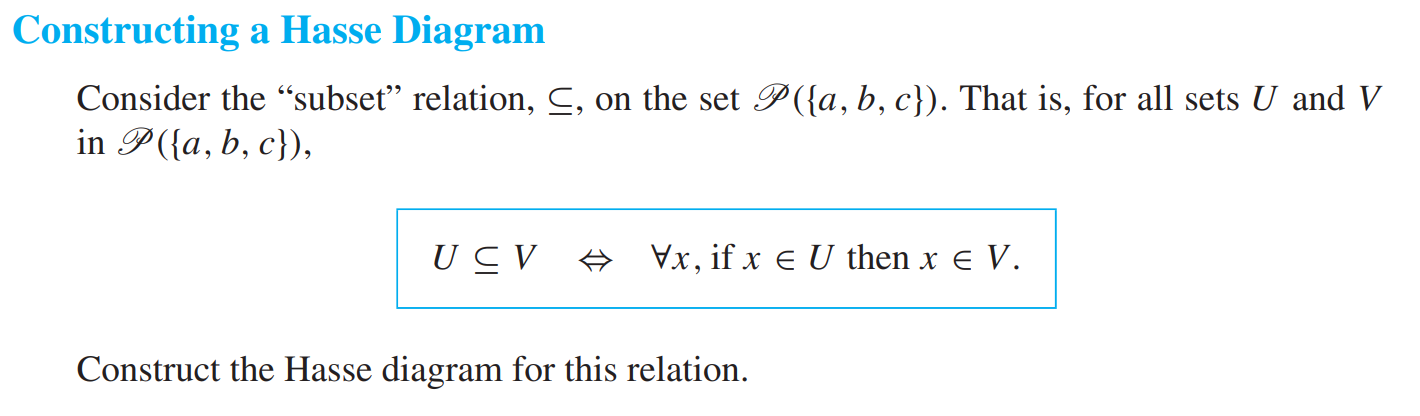
**Datakompresjon: Run length coding:** enkleste form for datakompresjon. En serie repetisjoner erstattes med et antall: ABIIIIIIIBBCD 🡪 AB12I2BCD. **Lempel-Ziv:** leser gjennom fila, input kopieres til output, hvis en lang nok sekvens kommer om igjen: dropp den, skriv heller en referanse til output. Hjelper hvis sekvensen er lenger enn en slik referanse. Må også ha en måte å fortelle at det kommer en rekke tegn som ikke skal komprimeres. [12] Problemer, p[-11,8][8]. Output kan komprimeres videre med Huffman. Bakover-referanse må være kompakt. Å se langt bakover i datastrømmen gir større sjanse for å finne repetisjoner, men også lenger kjøretid. Det man komprimerer må være lenger enn samlet lengde en bakover-referanse og header for en ukomprimert blokk. Ikke noe vits å komprimere svært korte strenger.  **BZip2:** komprimerer mer enn LZ algoritmene: 1) run length coding, 2) Burrows-Wheeler, 3) Move to Front transformasjon, 4) run length coding igjen, 5) Huffmankoding

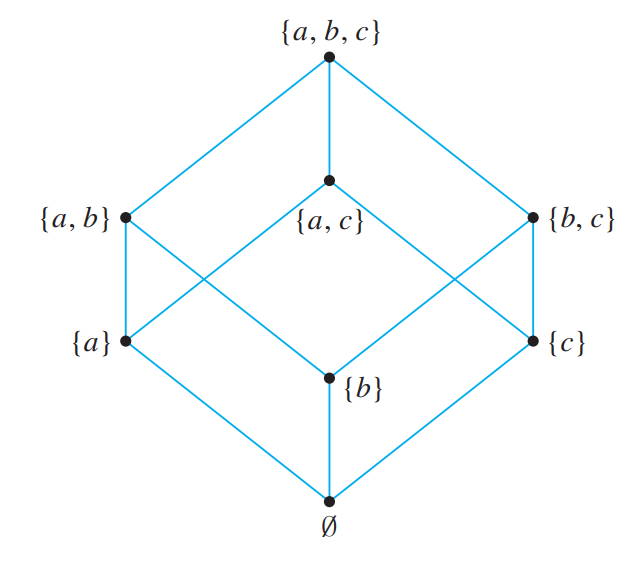


MFT komprimerer ikke data, men forbedrer. Leser inn et og et tall tegn fra input. Alle repeterte tegn blir til nuller

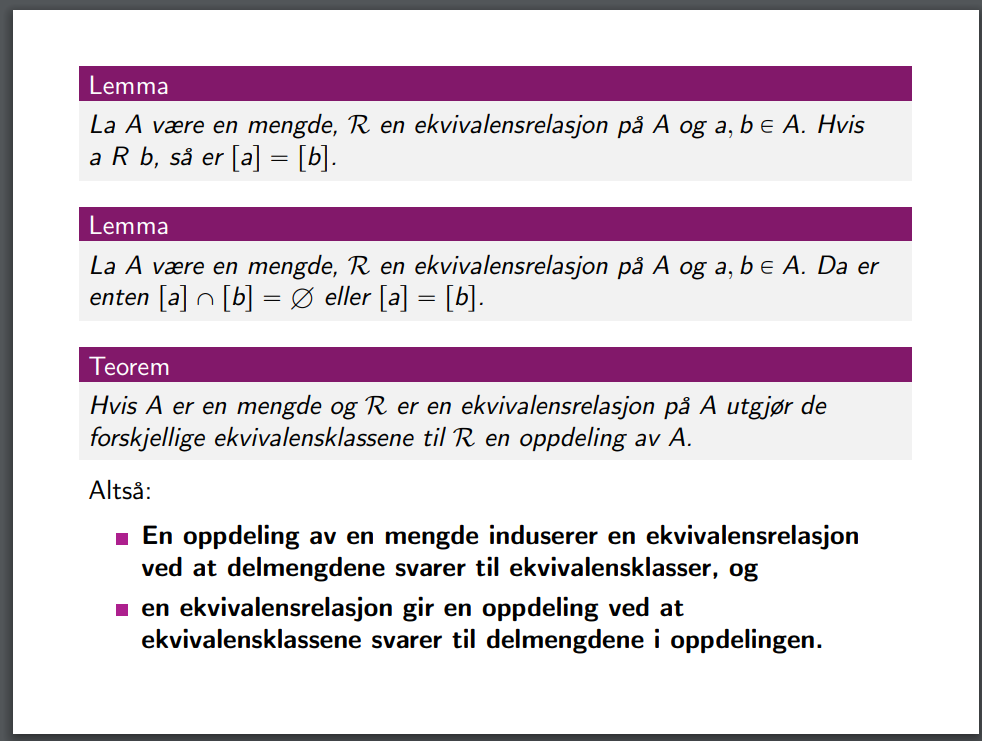
**BZip2:** BW sorterer slik at man får mange repetisjoner, move to front gjør repetisjoner om til nuller, da fungerer run-length coding veldig bra.

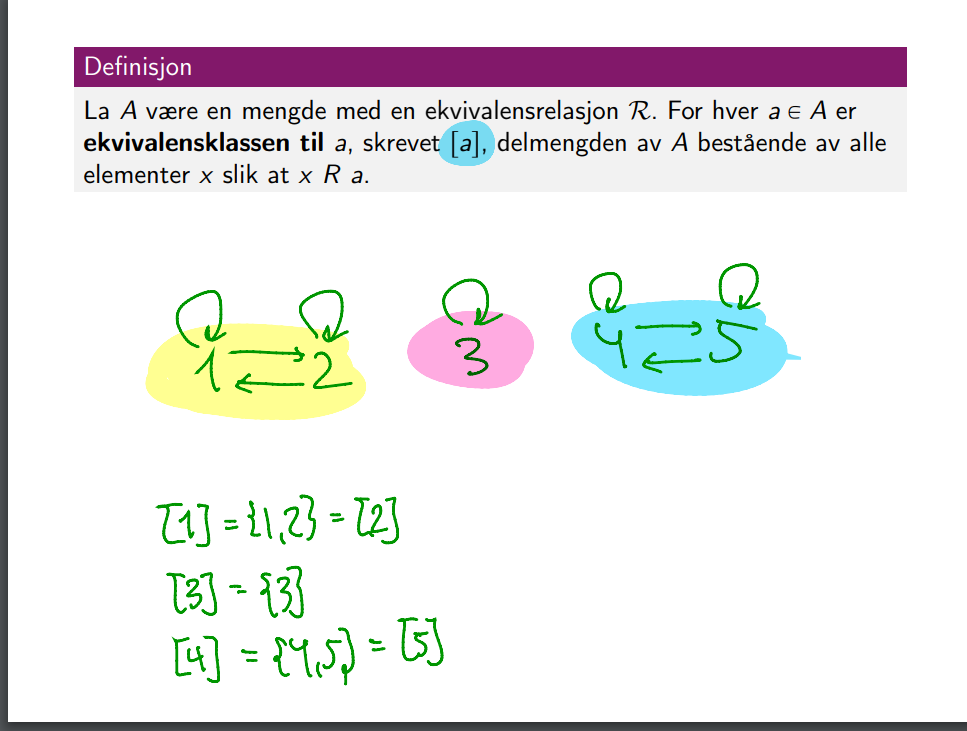
**Huffmannkoder:** Bruker vanligvis koder med et fast antall bits per tegn, f.eks. 8 bit = 1 byte per bokstav. Dermed er det lett å vite hvor hvert tegn begynner. Med huffmannkoder teller man opp hvor mange ganger hvert tegn forekommer: tegn med høy frekvens får kort kode med få btis, sjeldne tegn får lenger kode med mange bits. Dermed sparer man plass, om frekvensene varierer. Kodingen må også gjøres slik at dekoding blir mulig.

**Relasjoner:** La R være en relasjon på et sett A. 1. R er refleksiv hvis, og bare hvis, for alle x i A, x R x. 2. R er **symmetrisk**, hvis og bare hvis, for alle x og y i A, dersom x R y, så y Rx. 3. R er **transitiv**, hvis og bare hvis, for alle x, y, z i A, dersom x R y og y R z så x R z. **Ekvivalensrelasjon:** relasjon som er refleksiv, symmetrisk og transitiv. Dersom A er et sett og R er en ekvivalensrelasjon på A, for hvert element a i A, så er **ekvivalensklassen** til a, [a], settet av alle elementer x i A slik at x R a. Alle piler som går inn i punktet x i et diagram. Dersom a R b, så [a] = [b]. De distinkte ekvivalensklassene til A danner oppdelingen av A, dvs. unionen av klassene danner A. En relasjon R er **antisymmetrisk** hvis, og bare hvis, for alle a og b i A, dersom a R b og b R a så er a = b. En **partiell orden relasjon** er en relasjon som er refleksiv, antisymmetrisk og transitiv. 



Anta R er en partiell orden relasjon på et sett A. Elementene a og b er sammenlignbare dersom a R b eller b R a. Hvis dette ikke er tilfelle, er a og b **ikke sammenlignbare**. Dersom R er en partiell orden relasjon på et sett A, og for enhver element a og b i A, dersom a R b eller b R a, så er R en **total orden relasjon** på A. Et element i A er et maksimalt element dersom det er det eneste elementet i Hassediagrammet som ikke har en pil ut fra seg og oppover. Dersom dette er tilfelle er dette elementet både det største og det eneste maksimale elementet. Dersom det er flere elementer som ikke har piler ut av seg som går oppover har man ingen største, men flere maksimale. **Tre typer partielle orden relasjoner:** dele, mindre enn eller lik, subset.





**Dynamisk programmering:** Noen ganger kan løsningen på et stort problem bygges opp av løsninger på flere delproblemer. På forhånd vet vi ikke hvilke delproblemer vi trenger å løse. For å slippe å beregne samme delproblem fler ganger, noterer vi slike del-løsninger i en tabell. Når vi trenger en del-løsning, slår vi opp i tabellen for å se om delen allerede er beregnet. Dermed blir hver del-løsning bare beregnet én gang. Dette kan redusere kompleksiteten, og kalles dynamisk programmering.

Ryggsekkproblemet er et eksempel som kan løses med dynamisk programmering.

**Grådige algoritmer:** Algoritmer som gjør grådige valg; de tar det valget som ser best ut i øyeblikket, og går aldri tilbake på slike valg. Dette er raskt og effektivt, men bare hvis en slik strategi gir korrekt løsning.

**NP-kompletthet:** P: mengden av problemer som kan løses i polynomisk tid. Sortering, korteste vei, max flyt osv. **NP:** problemer hvis løsning kan **sjekkes**  i polynomisk tid. Å løse problemet kan være verre. P inngår i NP. **NPC:** En gruppe vanskelige problemer i NP. Beste kjente løsninger O(2n), ikke polynomisk. NPC inngår i NP. Hvert problem i NP kan omformes til et hvilket som helst problem i NPC, på polynomisk tid. Har man ett NPC-problem kan man finne en tilnærmet løsning, men ikke eksakt. **NPC-problemer:** Traveling salesman, er G1 isomorf med en subgraf av G2? Ryggsekkproblemet, har grafen en Hamiltonkrets? 3SAT, komplett subgraf. **Traveling salesman:** Har n byer og kostnader for å reise mellom dem. Ønsker en reiserute: innom alle byene med kostnad under x. Kan prøve alle muligheter O(n!). Kan sjekkes på O(n) tid. **Ryggsekkproblemet:** Gitt n udelelige varer, hver med pris og vekt, gitt en bærekapasitet, kan vi få med varer for verdien V? Vanskelig å finne, lett å sjekke. **NP-hardt:** Et problem er NP-hardt hvis og bare hvis det finnes et NPC-problem som kan reduseres til dette problemet i polynomisk tid. Svaret trenger ikke kunne sjekkes i polynomisk tid. Alle NP-komplette problemer er NP-harde. Noen av dem har hardere varianter hvor svaret ikke kan sjekkes på enkelt vis. **Traveling salesman:** NP-variant: Finne billigste av alle mulige reiseruter. Dette er ikke like lett å sjekke. **Ryggsekkproblemet:** NP-hard variant: få med maksimal verdi, vanskelig å vite at vi ikke kan få med enda mer. **Komplett subgraf:** np-hard variant: finne den største subgrafen. **Halting-problemet:** Gitt et problem P og en inndata D, vil programmet noen gang bli ferdig med å prosessere D, eller vil det gå i uendelig løkke? NP-hardt fordi ja/nei svaret ikke kan sjekkes i polynomisk tid. Hvis programmet kjører i O(2n) tid må vi vente så lenge på en sjekk. 