# Uvekta grafer:

**Nabotabell:** todimensjonal tabell Kant[N][N]. Kant[i][j] sier om det finnes en kant fra node i til j. Bruker N^2 plass, men lett å sløse plass. Lett å finne om to noder er naboer, men tar tid å finne alle kanter ut fra en node.

**Naboliste:** Tabell med noder, hver node er et listehode. Hver kant er et listeelement med referanse til neste kant, referanse til hvilken node kanten går til og evt. Vekt. Sparer plass siden man bare lagrer de som finnes, lett å finne alle veier ut av en node, vanskelig å finne alle veier inn. Kan utvides etter h vert.

**Kanttabell og nodetabell:** Hver node har en indeks til sin første kant i kanttabellen. Slutt der neste node har sin første kant. Dummynode for å vite hvor siste node slutter. Lett å programmere, kompakt i minne men må vite hvor mange kanter hver node har.

**BFS:** Eks: reiserute med færrest mulig togbytter. Ønsker korteste vei fra en startnode s, til alle andre. Finner først alle naboer til s (alle naboer legges i kø), deretter naboers naboer. Husker veien fra s og utover, ved at hver node har en forgjenger. Naboliste: Må sjekke alle kantene til de nodene vi finner. Hvis grafen henger sammen må vi sjekke alle N noder. O(N+K). Nabotabell: For hver node må vi finne kantene, ved å sjekke N mulige kanter. I verste fall, alle N nodene O(N^2).

**DFS:** Eks: Finne frem i en labyrint. Prøver å følge en vei så langt som mulig. Hvis det ikke går, gå tilbake til forrige node og prøv andre veier ut. Merker nodene som «besøkt», besøker ikke samme flere ganger. Opererer rekursivt: hvis en node ikke er besøkt, merk den og kjør DFS på dens naboer. Kompleksiteten har samme resonnement som for BFS: må innom alle noder og kanter. O(N+K) og O(N^2).

**Topologisk sortering:** Eks: planlegge produksjon, ønsker å finne en mulig rekkefølge. Grafen kan ikke være syklisk, og ofte er det mange muligheter. Samme tankegang som DFS: starter DFS i tilfeldig node, hver gang DFS er ferdig med en node, lenkes den inn først i resultatlista. Starter nye DFS så lenge det er urørte noder igjen. Kjører en serie med N DFS. Blir da lik kompleksitet som DFS: Naboliste: Θ(N + K), nabotabell: Θ(N^2). Bygge opp resultatlista tar Θ(N) tid og øker ikke kompleksiteten.

**Sterkt sammenhengende komponenter:** Eks: finne en samling nettsider som henger sammen via hyperlenker. Def: en del av en rettet graf, som er slik at alle nodene i den har en vei til alle de andre. Dette må gjelde begge veier. a 🡪 b og b 🡪 a. Brukes som en del av andre grafalgoritmer. Finnes slik: Kjør et eller flere DFS til alle noder er funnet (Θ(N+K) evt. Θ(N^2)). Bygg opp liste så de siste nodene kommer først (Θ(N)). Lag den omvendte grafen (samme noder, snu kantene) Θ(N+K) evt. Θ(1). Kjør en serie DFS på den omvendte grafen (Θ(N+K) evt. Θ(N^2)). De ulike DFS-trærne blir sterkt sammenhengende komponenter. Total: Θ(N+K) evt. Θ(N^2).

# Vektede grafer

**Floyd Warshall:** Alle til alle korteste vei. Θ(N^3) og trenger Θ(N^2) plass til resultat. Utgangspunktet er at alle distanser er uendelige.

**A\*:** En variant av Dijkstra hvor man bruker estimat for å rette søket mot mål. Et slikt estimat baseres gjerne på rettlinjet avstand fra noden til målet, og krever dermed informasjon om nodenes plassering på kartet. Gir et mer avlangt søk. Det får form som en ellipse som peker mot målet. Da behandles som oftest færre noder.

**Dijkstra:** ser bare på avstanden fra startnoden. Dermed sprer søket seg som en voksende sirkel, sentrert på startnoden. Forutsetter positiv vekt, greit når vektene er lengde eller tid. Tar grådige valg, velger den med lavest verdi først for å kunne sjekka alle og ikke få feil resultat.

**Kompleksitet:**

(1)Heap: O((K+N)logN) (1)

(2)Sortert tabell: O(KN)

(3)Usortert tabell: O(N^2)

Initialisering: Θ(N)

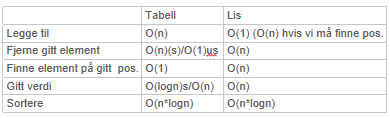
Lag kø: Θ(1)

Endre prioritet: (1):O(logN), (2): O(N), (3): O(1)

Hente minste: (1):O(logN), (2): O(1), (3): O(N)

**Bellman-Ford:** Minner om Dijkstra, men kan brukes med negative kanter. Gjenta N-1 ganger: går gjennom alle de K kantene. Sjekk om en node får kortere avstand ved å utnytte den aktuelle kanten. Hvis enda et gjennomløp, etter at korteste er funnet, gir bedre tid => negativ rundtur. Θ(NK). Sjekke alle kantene tar O(K) tid, gjør det N ganger.

**Minimale spenntrær:** Trenger bare nok forbindelser til at grafen akkurat henger sammen. Trenger ikke rundturer. Summen av vektene er minst mulig. Eks: planlegge graving, rør, strømledninger, kretskort, switching.



**Kruskals:** Ser på hver node som et frittstående tre. Velger laveste vektede kanter og slår sammen trærne. Bruker minimalt spenntre. O(KlogK).

**Prims:** Nesten identisk med Dijkstras. Bruker minimalt spenntre, ved å koble på en og en node basert på prioritetskø. Ser på avstand fra nærmeste node i spenntreet i stedet for avstand fra start. Kompleksitet som Dijkstra.

**Maksimal flyt:** finne den maksimale flyten vi kan ha gjennom et nettverk, fra kilde (k) til sluk (s), uten å gå over kapasitetsgrensen. Noder kan bare sende videre, ikke lagre noe. Eks: samlebånd, jernbaner, elektriske kretser, høyspentnettet. Total flyt: summen av alt som flyter ut av kilden, eller summen av alt som flyter inn i kilder, og summen av alle.

**Flytøkende veier:** En vei fra kilde til sluk der kapasiteten ikke er fullt utnyttet. Kan øke flyt langs veien tilsvarende den kanten som har minst restkapasitet.

**Restnett:** en graf som viser uutnyttet kapasitet i flytnettverket.

**Flytkansellering:** kan øke flyten ved å «snu» flyten gjennom en kant som har uhensiktsmessig flyt. Rosa er 10, svart 30, går vi motsatt vei blir den rosa f.eks. 5 om vi går 5 feil vei og den svarte bli 35 (har brukt 5 ikke 10)

**Ford Fulkerson:** Finn en hvilken som helst flytøkende vei, finn kanten med minst restkapasitet i denne veien. Øk flyten langs hele veien tilsvarende denne restkapasiteten. Gjenta til det ikke fins flere flytøkende veier. TAR LANG TID.

**Edmond Karp:** Samme som Fulkerson men bruker BFS for å finne flytøkende vei. Starter i kildenoden og bruker opp de flytøkende veiene som har færrest kanter først. O(NK^2)

# Rekursjon:

Metoden kaller seg selv.

a er antall rekursive kall i metoden, b er brøkdelen av datasett vi behandler i et rekursivt kall, cn^k er kompleksiteten for metoden (vanlige løkker).

Hvis bk < a, har vi at

Hvis bk = a, har vi at

Hvis bk > a, har vi at

Sortering

**Sortering:** ikke mulig å lage en sorteringsalgoritme for heltall asymptotisk raskere enn O(n) siden den da ikke vil gå gjennom alle tallene. Vil ikke alltid gi korrekt sortering.

**Boblesortering:** Lett å forstå, lett å programmere. Går gjennom n-1 ganger. Bytter om naboer som står feil. Små tall «bobler opp», store tall synker ned. 2 løkker gir Θ(n^2).

**Velgesortering:** finn største verdi, sett den på plass n-1, nest største på n-2, fortsetter sånn til nest minste er på plass 1. God hvis data er nesten sortert fra før. O(n^2), Ω(n).

**Sortere ved å bytte om naboer:** Ω(n^2)

**Flettesortering:** deler tabellen inn i to halvparter, disse sorteres rekursivt med flettesortering. Så flettes de sorterte deltabellene sammen. Ulempe: trenger en ekstra tabell som tar plass.

**Quicksort:** velger et av tallene som «pivot». Deretter gjøres en delvis sortering ved å ta mindre tall på venstre og større på høyre del. Pivot står på rett plass mellom mengdene. Høyre og venstre side sorteres videre med rekursive kall til quicksort. Rekursjonen forsetter til vi har fått del-tabeller med str. 1. O(nlogn), men O(n^2) hvis bare et tall på en side. Best på generelle data, raskere enn andre med samme tid. Skjevdeling. Svakhet: verste kjøretid på O(n^2), men svært sjeldent.



Gjennomsnitt:

**Forbedring av quicksort:** god på store datasett. Så når den produserer små datasett kan man bruke innsettningsmetoden på mellomresultater slik at sorteringen blir raskere.

Kan ikke lage en som er raskere enn O(n) => vil ikke gå gjennom alle elementene.

**Dual pivot:** mer arbeid (cache-effekter skjuler ekstraarbeidet), praksis 10-20% raskere. Færre sammenligninger, mindre rekursjon.

**Innsetningssortering:** Rask på «små» datasett, pga. kompakt programkode. Egnet for å sortere generelle data hvis vi vet at datasettet ikke kan bli for «stort». Egnet for store hvis vi vet at få elementer står feil. Ulempen er gjennomsnitt kjøretid på O(n^2), som utelukker store datasett. (Lite datasett avhenger av datamaskinen, gjerne i området 10-100). Setter et og et på plass blant de sorterte.

**Tellesortering:** Raskere enn andre pga. kjøretid på O(n), men stiller krav til dataen. Nøklene må være heltall i et intervall som ikke er bredere enn O(n). Teller hvor mange det finnes av hvert tall, regner hvor mange som er mindre eller lik hvert tall, bruker det til å sette hvert tall på plass. Sammenligner ikke tallene, bruker tallene som indeks i hjelpetabell.

**Shellsort:** Mellom O(n) og O(n^2), lab forsøk har målt ca. O(n^1,1).

**Radixsortering:** Funnet opp for å sortere hullkort, brukes nå for å oppnå rask sortering. Deler nøklene opp i «sifre», sorterer på minst signifikante siffer først, deretter nestminste osv. bruker en stabil hjelpealgoritme.

**Heap:** bruker hent maks og setter nyeste inn på toppen. (den nederst til høyre). Ω (n), Θ (n logn).

**Shellsort:** dette er lignende innsettningsmetoden, bare at man bruker en variabel som går fra n/2 til 1, og når den kommer til 1 bruker den innsetningssortering. (Omega=n, O=n^2)

# Liste

Plass til mange elementer, de står i rekkefølge, elementer kan settes inn og fjernes. Går an å lete etter elementer. Kan implementeres som en tabell (bruker ikke plass på forrige neste, har direkte oppslag) og lenket liste (ref. til forrige og neste element, når inntak/uttak skjer andre steder en bakerst, når antall elementer varierer mye, når vi har bruk for å flytte elementer/hele).

**Kø:** lineær datastruktur, kan bare sette inn bakerst og ta ut først. Tabell/lenka liste.

**Stakk:** lineær. Innsetning og uttak på toppen.

# Heap

**Def:** komplett binærtre, med en nøkkel i hver node. Løvnoder bare på nederste og evt. Nest nederste nivå. Evt. «tomme plasser» nederst står til høyre. **Max:** en nodes nøkkel er ≥ barnenodenes nøkler. **Min:** nodes nøkkel er ≤ barnas nøkler.

**Hente maks/hente min:** O(n log n).

**Endre prio:** O(logn) siden den går fra topp til bunn eller omvendt. Omega(1).

**Tabell:** tenker binærtre, men programmerer tabell. Tab[0] er rot. Rotas barn på 1 og 2. Node k har barn i 2k+1 og 2k+2, og foreldre i (k-1)/2.

**Fibonacci-heap:** kjøretiden som en heap, bortsett fra at en kan endre prioriteten i O(1) tid. Ulempen er at strukturen er mer komplisert å programmere.

# Trær

**Indre node:** Noder med barn. **Ytre node/løvnode:** noder uten barn.

**Binærtrær:** maks to barn pr. node. Er ordnet, barnas rekkefølge er vesentlig. Snakker om venstre og høyre barn. **Venstre subtre:** er det treet som har rot i nodens venstre barn. T (fullt binærtre) har 2k +1 hjørner og k+1 løvnoder. Hvis T har t løvnoder og høyde h er t ≤ 2^h. F.eks. høyde 5 gir 2^5 = 32 løvnoder.

**Dybde:** antall lenger mellom noden og rota.Rota har dybde 0.

**Høyde:** antall lenger mellom noden og den noden som er lengst unna i nodens subtrær. Treets høyde = rotas høyde.

**Generelt:** kretsfri og sammenhengende. Minst et hjørne av grad 1. Har n-1 kanter.

**Skog:** kretsfri og ikke sammenhengende (flere trær)

**Traversering:** Ønsker å gå gjennom å gjøre noe med alle nodene. Eks. finne høyden/skrive ut alt.

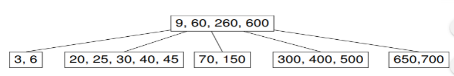
**Preordentraversering:** først noden vi har funnet, venstre subtre, høyre subtre. Eks: skrive ut familietreet så foreldre kommer før barna.

**Postordentraversering:** venstre subtre, høyre subtre, så noden selv. Eks: finne høyden på treet: nodens høyde er 1 + største av venstre og høyre. Nødvendig hvis vi prøver å finne noe som avhenger av subtrærne.

**Inordentraversering:** venstre subtre, noden, høyre subtre. Eks: få ut verdiene i et binært søketre, i sortert rekkefølge.

**Nivåordentraversering:** rota, så rotas barn, så disse nodenes barn osv. For å få til dette legger vi rota i en kø, deretter løkke: hente en node ut av køen, behandle den, legg dens barn inn i køa, fortsetter slik til køen er tom. Eks: Skrive ut stamtavle, generasjon for generasjon. Tegne opp tre grafisk.

**Binært søketre:** brukes på: databaseindeks med søk på «større, mindre, lik». Hver node har en nøkkel, alle noder i venstre subtre har mindre nøkkel, alle i høyre har større. Operasjoner: sette inn, søke, slette. **Slette:** ingen barn: fjernes, et barn: erstattes med barnet, to barn: erstattes med neste i sorteringsrekkefølgen (det blir minste noden i høyre subtre). O(logn).

**B-trær:** brukes på strukturerer som (kanskje) er for store til å ha i minnet (databaseindekser, filsystemer). Er et søketre. Hver node har mange barn, typisk begrenset av blokkstørrelsen. Færre nivåer, ofte bare to. utnytter minnet effektivt, få diskaksesser. Hver node (utenom rota) har mellom n-1 og 2n-1 nøkler. I praksis velges n slik at en node fyller opp en blokk. En intern node har ett barn mer enn den har nøkler. Eks: n=3 og to nivåer.

# Hashtabeller

**Virkemåte:** bruker en hashfunksjon som tar nøkkelen som parameter, og produserer en tabellindeks => tabell[hashfunksjon(nøkkel)].

**Anvendelser:** finne igjen elementer Θ(1). Indeks i databaser, gir raskere oppslag. Oppslag i symboltabeller i kompilatorer.

**Lastfaktor:** tabell med størrelse m og n elementer: α = n/m. Full tabell α = 1, dårlig utnyttet: α nesten 0, overfylt α > 1.

**Gode hashfunksjoner:** må alltid gi en verdi mellom 0 og tabellstørrelsen, rask å beregne, gir god spredning. Gir få kollisjoner (ulike nøkler hasher til samme pos).

**Restdivisjon:** h(k) = k mod m. Best hvis m er primtall. Restdivisjon med heltall går fort. Dårlig hvis m er en toerpotens. For m = 16 vil h bare avhenge av de fire siste bitene i k. Dårlig hvis m er 10, 100… h avhenger bare av de siste sifrene i nøkkelen. K er nøkkel og m tabellstørrelse.

**Multiplikativ hash:** har en tabell der størrelsen m = 2^x, og A = (sqrt5-1)/2, og tar da f.eks. 2^8 \* A, og ganger dette med nøkkelen vår. Deretter tar en de første bitsene i de 8 siste bitsene, fordi vi ganger med 2^8 bits

**Lenka lister:** hver tabellposisjon er et listehode. Når flere elementer hasher til samme plass, lenker vi dem inn. Innsetning er fortsatt Θ(1). Fordelen med dette er at tabellen aldri vil bli full. Ulempen er at det går med plass til referanser(nest).

**Lineær probing:** enkleste form for åpen adressering, ved kræsj tar vi neste ledige plass. Probe(h,i,m) = (h+1) mod m. h=hashfunksjonen, m = tabellstørrelse, i = teller opp hver forsøk. Ulempen er at vi kan få lange kjeder med kollisjoner. Alle verdiene som havner o samme område må gå gjennom samme lange kollisjonskjede.

**Kvadratisk probing:** prøver å unngå de lange kjedene i lineær. Probe(h,i,m) = (h+ kii + k2i) mod m. K1 og K2 kan ikke ha felles faktorer, ellers kan ikke probesekvensen prøve alle tabellområder. Risikerer å mislykkes selv om det er plass.

**Dobbel hashing:** denne hashmetoden bruker to hashfunksjoner i probesekvensen, en som angir den første posisjonen vi skal prøve på, og en annen som angir hvor langt vi skal hoppe hvis det blir kollisjon. Probe(h1, h2,i, m) = (h1 + ih2) mod m. Er vanlig å bruke h1(k) = k mod m, og h2 = (k mod m-1) +1. Det er viktig at hoppesekvensen aldri blir 0, fordi da kan man få en evig løkke.

# Prioritetskøer

**Usortert tabell:** Enkel å implementere, prioritetsendring O(1). Ulempe: å finne eller hente ut minimum tar O(n). **Heap:** Kjapp, men noe mer komplisert enn

usortert tabell. Å finne minimum tar O(1) tid. Å hente min eller endre pri tar O(logn) tid.

**Fibonacci-heap:** Kjøretider som heap, bortsett fra at pri endres i O(1) tid. Ulempen er at strukturen er mer komplisert å programmere.

# Hashtabeller

**Hashfunksjon:** må kunne bli 0. (h(k) = kmodm er god på primtall)

**Hash hopp lengde ved kollisjon:** kan ikke bli null (kmod(m-1)+1 primtall)

# Relasjoner

**Funksjon:** en og bare en strek fra hvert element i x til y.

**Relasjon:** R fra A til B er en delmengde av A x B. Mengden av alle par som står i relasjon til hverandre under R. En relasjon er altså selv en mengde.

**Binærrelasjon:** på en mengde A er en binærrelasjon fra A til A.

**n-relasjon:** Gitt mengder A1, A2, …, AN er en n-relasjon R på A1 x A2 x … AN en delmengde av A1 x A2 x A3. Eks. databaser: STUDNR x NAVN x HJEMSTED.

**Refleksiv:** har loop: (a, a) € R.

**Symmetrisk:** hvis (a, b) € R, så må (b, a) € R.

**Antisymmetrisk:** hvis (a, b) € R og (b, a) € R så er a = b.

**Transitiv:** hvis (a, b) € R og (b, a) € R, så må også (a, c) € R.

**Transitiv tillukning:** Noen ganger har vi en relasjon som ikke er transitiv, som vi ønsker at skal være transitiv. Den transitive tillukningen R^t er transitiv, R € R^t, og for hver transitiv relasjon slik at R € S, så er R^t € S.

**Viktige relasjoner:** ekte mindre enn: ikke ref eller sym, en transitiv.

**A = b(mod n) ⬄ a -b = kn:** ekvivalensrelasjon. **A | b ⬄ b=k a**

**Ekvivalensrelasjon:** refleksiv, symmetrisk og transitiv. Hvis en relasjon er indusert av oppdelingen av A, er R en ekvivalensrelasjon.

**Ekvivalensklasse:** [a]: delmengden av A bestående av alle elementer x slik at xRa. [1] = {1, 2} = [2].

**Partiell ordning:** refleksiv, antisymmetrisk og transitiv.

**Leksikografisk ordning:** brukes når det som skal ordnes består av 0 eller flere symboler hentet fra et alfabet, dvs. er strender over dette alfabetet. Symbolene i alfabetet må ha en partiell ordning som vi kaller R. Da gjelder: for to vilkårlige strenger x og y sammenligner vi symbolene fra venstre mot høyre. Så lenge de er like fortsetter vi mot høyre. Hvis vi kommer til en posisjon hvor de er ulike så er x < y hvis x[i] R y[i], og x > y hvis y[i] R x[i]. Hvis vi kommer til en posisjon hvor x tar slutt, men ikke y er x < y, og motsatt.

**Hassediagram:** knyttet til partielle ordninger. Forenkling av grafen: fjerne looper, plassere hjørner slik at alle piler peker oppover, fjerne piler fra transitivitet, fjern retningen på pilene.

**Total ordning:** Gitt en partiell ordning R over A. Hvis alle elementer i A er sammenlignbare, sier vi at R er en total ordning. A og b er sammenlignbare hvis a ≤(krøll) b eller b ≤ a. ≤ med sånn krøll.

* a kalles maksimalt hvis for alle b i A så er b ≤ a eller ikke sammenligne.
* a kalles største hvis for alle b i A så er b ≤ a.
* a kalles minimalt hvis for alle b i A så er a ≤ b, eller ikke sammenligne.
* a kalles minste hvis for alle b i A så er a ≤ b.
* om a står helt fritt: både minimalt og maksimalt.

**Isomorfi-invarianter:** har n noder, har m kanter, har en node med grad k, har m noder med grad k, har en krets med lengde k. har en enkel krets med grad k, har m enkle kretser med lengde k, er sammenhengende, har en Eulerkrets, har en Hamilton-krets.

Grafteori

**V(G):** alle hjørnene til G

**E(G):** alle kantene til G

**Kant-hjørne-funksjonen:** beskriver hvilke hjørner en kant ligger inntil.

**Naboer:** to hjørner som har en kant mellom seg. Nabokanter: to kanter som møtes i et hjørne. Parallelle kanter: to eller flere kanter som går mellom de to samme hjørnene. Tom graf: graf uten hjørner.

**Tom graf:** graf uten hjørner.

**Simpel graf:** har ikke parallelle kanter eller looper. I en simpel graf brukes notasjonen {v1 , v2} for kanten mellom v1 og v2.

**Undergraf/delgraf:** H er undergraf av G hvis hvert hjørne i H også er et hjørne i G, og enhver kant i H også er en kant i G.

**Komplett graf:** Kn er en graf med n hjørner (n er positivt heltall), er en simpel grad med nøyaktig n hjørner og nøyaktig en kant mellom hvert par av hjørner. Hvert hjørne har grad n-1.

**Todelt:** Km, n: simpel graf med nøyaktig en kant mellom hvert par av hjørner i oppdelingen, og ingen flere enn disse. (delt i to, streker mellom delene, ikke innad i delene).

**Grad:** total = 2 \* antall kanter i G. I enhver graf er det et partalls antall hjørner med odde grad. 1, 3, 5 => kan ikke finnes.

**Vei:** en vei fra v til w er en endelig alternerende rekke av kanter og hjørner.

**Veilengde:** antall kanter i veien. <a, b, c> har lengde 2

**Spor:** fra v til w er en vei som ikke repeterer noen kant.

**Sti:** repeterer ikke kant eller hjørne.

**Rundtur:** lukket vei: starter og slutter i samme hjørne.

**Krets:** rundtur som ikke repeterer noen kant. Trivielle: består bare at et hjørne.

**Simpel krets:** repeterer kun start hjørnet, ingen andre.

**Sammenhengende:** v og w er sammenhengende hvis det finnes en vei fra v til w, samme for alle par. Hvis v og w er en del av en krets kan en kant fjernes og det vil fortsatt finnes et spor fra v til w. Er den

sammenhengende og har en ikke-triviell krets kan en kant fjernes og den vil fortsatt være sammenhengende.

**Sammenhengende komponent:** krav: H er en undergraf av G, H er sammenhengende og H er «størst mulig»/finnes ingen andre undergraf som har flere kanter enn H slik at H blir undergraf til den.

**Eulerkrets:** for G er en krets som inneholder ethvert hjørne og enhver kant i G. Har Eulerkrets hvis ethvert hjørne har partallsgrad og sammenhengende.

**Eulerspor:** fra v til w, inneholder alle hjørner og kanter. Alle har partallsgrad utenom v og w som har odde grad.

**Hamiltonkrets:** gitt en graf G er en Hamiltonkrets for G en simpel krets som inneholder alle hjørnene. G har da en undergraf H som: inneholder alle hjørner i G, er sammenhengende, har samme antall hjørner som kanter og ethvert hjørne i H har grad 2. Ikke Hamilton hvis ikke H!

**Nabomatrise:** En matrise som representerer hvor mange piler som går fra v til w.

**Veilengde:** antall kanter i veien. <a, b, c> har lengde 2. Nedover fra, bortover til. Hvis ikke piler: tar man på begge muligheter, får 2\* så mange tall i matrisen. A^n gir oss antall veier med lengde n fra v til w.

**Isomorf:** to grafer som kan gjøres om slik at de er «like» og bevarer alle hjørne-kant-funksjonene. **1.** Å ha n kanter. **2.** ha m hjørner. **3:** å ha et hjørne av grad k. **4:** å ha m hjørner av grad k. **5:** å ha en krets av lengde k. **6:** å ha en simpel krets av lengde k. **7:** å ha m simple kretser av lengde k. **8:** å være sammenhengende. **9:** å ha en Eulerkrets. **10:** å ha en Hamilton.

Språk

**∑n:** Mengden av alle strenger over ∑ med lengde n.

**∑+:** mengden av alle strenger over ∑ med lengde minst 1.

**∑\*:** mengden av alle strenger over ∑. Har med tomme strengen.

**L1 = {a, aa}, L2 = {e, bb, bbb}.**

**L1L2 =** {a (=ae), abb, abbb, aa, aabb, aabbb}

**L1 U L2 =** {a, aa, e, bb, bbb}

**(L1 U L2)\* =** {e, aabbb, abbb, a, abb, … osv.}

**Rekkefølge:** \*, så sammensetning, sist: | (eller).

**Endelige alfabet:** funksjon L som assosierer ethvert regulært uttrykk r med et språk over ∑. Da kaller vi L( r) for språket definert av r: L(Ø) = Ø, L(E ) = {e}, L(a) = {a} for hver a E ∑. Hvis L( r) og L(s) er språkene definert av de regulære uttrykkene r og s over ∑: L(rs) = L( r)L(s), L(r|s) = L(r) U L(s), L(r\*) = (L(r))\*

Endelige automater

**Inputalfabet:** {0, 1}

**Starttilstand/initial:** S0

**Tilstander =** {S0, S1, S2}

**Aksepttilstand:** S2.

**Neste tilstandsfunksjon:** tabell som viser hva som sender til hva. Her vil S0 sende 0 til S1 og 1 til S0. Pil på start og dobbel runding på aksepttilstand.

Et språk **aksepteres** av A hvis alle strenger ender i akseptansetilstand.

**Grammatikk:** en kontekstfri grammatikk består av et alfabet V, en delmengde ∑(delmengde av V) kalt **terminalsymbolene.** En mengde på formen x 🡪 y, der x E (V - ∑) og y E V\*, og et **startsymbol** S som er i V-∑.

**V-**∑: midlertidige symboler.

**Kontekstfritt språk:** språk som genereres av en kontekstfri grammatikk. Alle regulære språk er også kontekstfrie, motsatte er ikke alltid tilfelle.

Avanserte programmeringsteknikker

**Optimaliseringsproblemer:** Har problem med mange mulige løsninger. Ønsker den beste: alt sett på korteste vei og minimalt spenntre.

**Rå kraft:** prøv alle muligheter, velg den beste! Eks: generere alle mulige spenntrær, velge det minimale, det er O(2^k) kombinasjoner. Tar mye tid, er siste valg hvis vi ikke finner noe smartere.

**Splitt og hersk:** dele opp et stort problem i flere mindre. Løses separat og settes sammen til en total løsning. For optimaliseringsproblemer forutsetter dette at delløsningene faktisk er deler av totalløsningen. Quicksort fungerer slik. Andre varianter: dynamisk programmering og grådige algoritmer.

**Dynamisk programmering:** brukes når vi må prøve mange kombinasjoner av løsninger på delproblemer. Hvis vi naivt prøver alle ender vi lett opp med å kombinere de samme delproblemene flere ganger. Husker del-løsninger i tabell. Hvis løsningen ligger der slipper vi å løse den. Ryggsekk.

**Grådige algoritmer:** algoritmer som gjør «grådige valg»: hva ser best ut akkurat nå? Går aldri tilbake på valg, tenker ikke fremover. Raske og effektive, hvis problemet lar seg løse. Eks: prims og Kruskal, Dijkstras algoritme, Huffman.

Tekstsøk og datakompresjon

**Anvendelser:** fritekstsøk i dokumenter, nettsider, databaser, søkemotorer, søke etter repeterende strenger for datakompresjon, DNA-matching.

**Boyer-Moore:** Se på siste tegn i søketeksten først, hvis det ikke passer så flytt søketeksten så langt vi kan. Hvis det passer, se på nest siste osv.

**Regelen om upassende tegn:** hvis tegnet ikke fins i søketeksten, kan vi flytte m steg frem. Hvis det fins til venstre i søkeordet, kan vi flytte ordet så det passer med teksten. Vi har en tabell for hvor mye vi kan flytte, i praksis er denne med hele alfabetet, hvor de fleste gir flytt på m. tegn som finnes i søketeksten gir kortere flytt (s i siste gir flytt på m-1) Omega(n/m) for søket. Mye bedre. Todimensjonal tabell, i = upassende tegnet, j = posisjonen i søketeksten, verdien er hvor lang vi kan flytte.

**Galil sin regel:** hvis vi søker etter aa i aaaa har vi O(nm), søkeordet passer overalt så de samme a-ene sjekkes flere ganger. Hvis vi flytter søkeordet kortere enn den delen av søkeordet vi alt har sjekket trenger vi ikke sjekke det overlappende området igjen => O(n) og omega(n/m). Korte flytt skjer fordi søkeordet delvis matcher seg selv. Hvis det ikke hadde passet hadde vi flyttet lenger.

**Run-length coding:** ABIIIIBBBCD 🡪 AB4I3BCD. Kan ha sifre i det vi komprimerer: ABIIIIBBBCD 🡪 [-2]AB[4]I[3]B[-2]CD, negativ indeks for ukomprimert del. Enkleste formen for komprimering, kan ikke komprimere for eksempel ABABABA.

**Lempel-Ziv:** Kompresjon oppnås ved å erstatte gjentatte strenger med referanse til den forrige strengen. Når man komprimerer, leser man gjennom teksten. Strenger man ikke har sett før, skriver man opp som vanlig. Strenger man har sett, erstattes med en referanse som forteller hvor langt bakover man må gå for å finne kopien, og hvor lang kopien skal være. Dermed er det mulig å pakke ut igjen senere. Hvis referansene er kortere enn strengene de erstatter, oppnår vi kompresjon. 1 byte kan peke 255 tegn bakover, 2 byte kan peke 65536 tegn bakover. Problemer, problemer 🡪 [12]Problemer, p[-11,8][8]

**Lempel-Ziv Welch:** Ligner LZ, men lettere å speede opp. Leser et og et tegn, bygger en ordliste underveis (til å begynne med, alle 1-byte ord). Finne et (lengst mulig) ord, skriv ordnummeret. Lagre nutt ord = dette ordet + neste tegn. Kompresjon hvis ordene blir lengre enn numrene.

**DEFLATE:** brukt i zip. LSW + Huffman.

**BZip2 blokk komprimering:** komprimerer mer enn LZ-algoritmene. 1. run-length coding, 2. Burrows-Wheeler transformasjon (hoveddel). 3. Move-To-Front transformasjon (MFT). 4. Run-length coding igjen. 5. Huffman.

**BWT:** først bruker man å høyrerotere ordet helt til man har kommet helt rundt minus 1 (prikk nest bakerst). Deretter sorterer man ordene leksiografisk. BWT er siste kolonne med tegn fra sortert liste. For å få dette tilbake til vanlig så setter man inn denne kolonnen på nytt og sorterer, og setter inn på nytt. Dette gjør man like mange ganger som antall ord en har i ordet som en skal transformere. Den med prikk sist er riktig ord.

**Huffman:** Venstre: 0, høyre: 0. Maks høyde: alle indre noder har (minst) ett barn som er en løvnode: n noder gir n-1 kanter. Min høyde er 2^x = n, for eksempel. 2^4=16 => gir mulighet for 16 ulike barn. Teller hvor mange ganger et tegn forekommer, lager frekvenstabell 🡪prioritetskø 🡪 tar to og to minste og lager en ny node med frekvensen til de to sammenlagt som legges inn osv. Høy frekvens kortere koder. M

# NP og P

**P**: Mengden av alle problemer som kan løses i polynomisk tid.

**NP**: Mengden av alle problemer der svaret kan sjekkes i polynomisk tid.

**NP-komplett:** Mengden av alle problemer som er i NP og minst like vanskelig som et hvilket som helst

**NP-problem:** Alle NP-problem kan omformes (reduseres) til et NPC-problem i polynomisk tid. Lett å se om man har fått riktig svar.

**NP-hard:** Mengden av alle problemer som er slik at det finnes et NPC-problem som kan omformes (reduseres) til dette problemet i polynomisk tid. Vanskelig å sjekke svaret.

**Finne sum = 0:** Med n (positive og negative) tall finnes det 2n mulig delsummer som må sjekkes. Når svaret er funnet kan det sjekkes på O(n) tid. Tilhører da NP-komplett.

**Forskjellen:** NP-komplette problemer er løsbare, og løsninger kan sjekkes i polynomisk tid. For et NP-hardt problem er det ikke nødvendigvis mulig å sjekke løsninger i polynomisk tid heller. Noen, som halting-problemet, er ikke generelt løsbare heller

**Traveling salesman:** har n-byer og kostnaden for å reise mellom dem, ønsker en reiserute: innom alle byene, koster under x. Prøve alle muligheter (n!). Løsningen kan sjekkes på O(n) tid. NPH-V: finne den billigste av alle, ikke så lett å vite om det finnes billigere rute, så vanskelig å sjekke svar.

**Ryggsekkproblemet:** gitt n udelelige varer, hver med pris og vekt (evt. Volum/vekt i stedet for vekt). Gitt en bærekapasitet, kan vi få med varer for verdien V? Vanskelig å finne ut, legg å sjekke svar. Eks. praktiske anvendelser innen automatisk planlegging. NPH-V: finne maksimal verdi vi kan ha med, vanskelig å vite om vi ikke kan få med mer.

**Komplett subgraf:** finne en komplett subgraf med x noder (til en gitt graf). Lett å se om løsningen er komplett. NPH-V: finne den største komplette subgrafen, tungt å sjekke at det ikke fins en som er større.

**Halting-problemet:** Er ikke generelt løsbart. Gitt et program P og inndata D. Vil programmet noen gang bli ferdig med å prosessere D, eller uendelig løkke? NPH fordi ja/nei-svaret ikke kan sjekkes i polynomisk tid, må vente til programmet faktisk stopper. Kjøretid O(2^n) så må vi vente så lenge på svar. Hvis vi kunne løst dette problemet kunne vi løst alle med ja/nei svar. F.eks. **Goldbachs uløste problem:** er alle partall summen av to primtall? Program som tester fra 2 til uendelig, stopper kun hvis den finner feil. Halting kunne da visst om programmet stoppet eller ikke, og vi hadde fått svar uten å kjøre programmet.

**Bruceforce:** prøve alle muligheter, som er større enn polynomisk (t:2^n eller n^n.

**Bitoperasjoner** er veldig raske fordi datamaskiner bruker binære tall. Bit 0 tilsvarer 0 volt, 1 typisk litt over 1 volt.  
**Høyreskift**:- int a = 154, a>>=k; // a = a >> k; 🡪 tilsvarer a = a/2k; a>>=1; // a = a >> 1; 🡪 tilsvarer a = a/2 = 77; 10011010 (154) >> 🡪 01001101 (77) – alle flyttes en til høyre  
**Venstreskift**:- int a = 5, a <<= k; // a = a <<k; 🡪 tilsvarer a = a\*2k a <<= 3; // a = a <<3; 🡪 tilsvarer a = a\*23 = a\*8 = 40; 000101 (5) <<3 🡪 101000 (40)  
**bitvis xor ^(exclusive or):** 1 om de er ulike, 0 om de er like  
**bitvis and &:** 1 om begge er 1, 0 om de er ulike eller begge 0  
**bitvis or |:** 1 om de er ulike eller begge er 1, 0 om begge er 0  
bitvis not ~: alle snus/inverteres  
Rotasjoner: Som skift, men det som går ut kommer inn på andre siden, brukes i grafikk og for scrambling i kryptografi (sammen med xor)  
  
Nettmasker, gjøres med and-operasjon mot en nettmaske