**Diskret Matematikk:**

**Refleksiv:**

* (a,a) ∈ R for alle elementer

**Symmetrisk:**

* if(a,b) ∈ R så (b,a) ∈ R

**Anti-symmetrisk:**

* if(a,b)&&(b,a) så a=b
* eks( <= || >=)
* Tom-sannhet: (< || >)

**Transitiv:**

* if(a,b)&&(b,c) så (a,c)

**Ekvivalensrelasjon:**

* refleksiv, symmetrisk og transitiv.
* Oppdelingen som svarer til ekvivalensrelasjonen er A0 = {0}, A1 = {−1,1},A2 = {−2,2},A3 = {3}.

**Partiell Ordning:**

* refleksiv, antisymmetrisk og transitiv.

**Euler-spor/vei:**

* Går gjennom alle kanter en gang
* Må ha to hjørner med odde grad

**Euler-krets:**

* eulerspor som slutter der den startet
* alle noder må ha partalls grad

**Hamilton-spor/vei:**

* Går gjennom alle noder en gang

**Hamiltonkrets:**

* Hamiltonspor som slutter på start

**Hamiltonsk:**

* sammenhengende graf med en hamiltonkrets.

**Totalgrad:**

* antall kanter ut/inn fra hver node plusset sammen.

**regulær graf:**

* av grad *n*, har alle nodene grad *n*

**Komplett graf:**

* alle nodene direkte forbundet med hverandre. Hvis grafen har *N* noder, er dette en regulær graf av grad *N*-1

**Delgraf:**

* undergraf, del av en annen graf.

**Nabomatrise:**

* matrise som viser hvilke noder en node hører til (0 og 1 for uvektet), ellers brukes vekten.

Veier fra a til b

**Tre:**

* uten sykler, sammenhengende
* n-1 kanter (sammenhengende)

**Skog:**

* uten sykler, ikke sammenhengende

**Isomorfi:**

1. Å ha x kanter/hjørner
2. Å ha x hjørner av grad y
3. Å ha x kretser av lengde y
4. Å ha en Euler-krets
5. Å ha en Hamiltonkrets.

**Simpel graf:**

* Ikke refleksiv
* Ingen til og fra med samme node

**Topologisk sortering:**

* Ingen sykluser
* Hvis a går til b, må a komme før b i sorteringen.

**Hasse diagram:**

* Relasjoner satt sammen slik at alt over står i relasjon til alt under.

**Leksikografisk ordning:**

* består av 0 eller flere symboler hentet fra et alfabet. Symbolene i alfabetet må ha en partiell ordning som vi kaller R. Da gjelder: For to strenger x og y sammenligner vi fra venstre mot høyre. Så lenge de er like, fortsetter vi mot høyre. Når de ikke er like lenger kommer enten det korteste først, eller det ordet men tegnet med lavest verdi først.

**Minimalt Spenntre:**

* *Pim’s*: start i tilfeldig node, koble sammen ved å følge laveste kantverdi tilgjengelig som er koblet til startnoden. Grafen må være sammenhengende
* *Kruskal’s*: start med globalt minste kant, fortsett uten å lage sykler til alle noder er koblet sammen (dersom grafen er sammenhengende)

Korteste vei tre:

* Dijkstra fra en til alle.

Sterkt sammenhengende komponenter:

* Noder i løkke blir ett komponent,

utenfor et annet komponent.

Automater:

* Alfabet: Σ
* Tilstander
* Aksepterende tilstander
* Tilstandstabell

Grammatikk:

* Terminalsymboler

**Datastrukturer:**

Kjøretid:

* Nedre:Ω Øvre:O Felles:Θ

Rekursive kall:

* Hvis (n-1) = vanlig loop.
* Gjelder hvis vi har

T (n) = aT (n/b) + cn^k

* a antall rekursive kall i metoden
* b brøkdelen av datasettet vi behandler i et rekursivt kall
* cn^k kompleksitet for vanlige løkker
* b^k < a, har vi T(n)∈ Θ(n^(logb(a)))
* b^k = a, har vi T(n)∈ Θ(n^k·log(n))
* b^k > a, har vi T(n)∈ Θ(n^k)
* logb(a) = log(a) / log(b)

Binært søketre: (liten/lik ventre, stor høyre)

* Balansert: O(log(n)) (alt)
* Ubalansert: O(n) (alt)
* Enten et tomt tre eller et binærtre der nodene har nøkler
* alle noder må tilfredstille at nodenes nøkkel er større enn nøklene til alle noder i venstre subtre og mindre enn alle nøklene i nodens høyre subtre

B-Tre:

* O(log(n)) (alt)
* Søketre, ikke binært
* Brukes for større datamengder som feks databaser

Heap:

* Fyll opp fra venstre
* indeks forelder = gulv((indeks-1)/2)
* hent rot: O(log(n)) (fiks\_heap)
* insert:
* delete:
* search:

Max: forelder størst. Min: forelder minst

Stakk:

* kø: sist-inn-først-ut.

Grådige algoritmer:

* tar kun i betrakting det den vet i øyeblikket,
* går ikke tilbake og sjekker om en løsning er riktig.
* Tenker ikke fremover

eks:

* Huffmankoding
* Kruskal's og Prim's algorithm for minimum spenntrær
* Dijkstras algoritme for korteste vei

**Datakompresjon**

Huffmankoding:

* Antall tegn A=50 B=23 C=12 D=5 E=5

(1bit x 50) + (2bits x 23) + (3bits x 12) + (4bits x 5) + (5bits x 5) = 162 bits

1. Tall opp antall forekomster av hvert symbol
2. Sorter symbolene etter antall forekomster
3. Slå sammen de to symbolene med minst forekomster i en gruppe, og sorter igjen
4. Gjenta punkt 3 til det bare er to grupper igjen
5. Representer denne grupperingen ved hjelp av et binært tre. Hver gren/forgrening blir tildelt 0-bit eller 1-bit
6. Sekvensen av biter fra roten til hver løvnode i treet, gir Huffman-koden.

Lempel-Ziv:

* Buffer med tidligere tekst
* Erstatt tekst som kommer igjen med en referanse til et tidligere punkt i teksten. (Fortell hvor mye ukomprimert som kommer)

Burrows Wheeler:

Komprimering:

* Sett (\*) tegn bakerst, Roter ordet mot høyre.
* Når ferdig, sorter disse radene leksikografisk(alfabetisk)
* Resultatet er siste kolonne fra den sorterte tabellen.

Dekomprimering:

* Sett komprimert i kolonne til høyre, sorter alfabetisk fra toppen.
* Sett in resultatet på nytt til venstre for forrige, sorter på nytt,
* Fortsett helt til riktig lengde på ordet. Raden med tegnet bakerst er riktig.

Move-to-front:

* Les ett tegn, finn tegnet i tabellen, skriv index til output
* flytt tegnet vi fant til første plass i tabellen (move to front)

start: [0]=A,[1]=B,[2]=C

CABB BBB BAC

2120 000 012

Run-length:

* Eks., bruker negativ byte for ukomprimerte sekvenser: ABIIIIIIIIIIIIBBBCDEFFFGH
* [-2]AB[12]I[3]B[-3]CDE[3]F[-2]GH

**Dynamisk Programmering:**

* når vi må prøve mange kombinasjoner av løsninger på delproblemer
* Å gjøre samme jobb flere ganger, kaster bort tid
* husker del-løsninger i tabell
* Sørg for at hvert element enkelt kan beregnes fra noen av de som allerede er beregnet

**NP Definisjoner**

P(polynomialtime):

* Mengden av problemer som kan løses i polynomisk tid

(n^2, n\*k, n + k^2)

* Sortering, korteste vei, max flyt

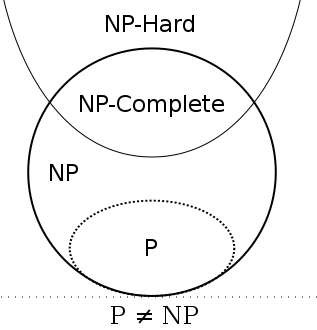
NP(Nondeterminisitc-Polynomialtime)

* Mengden av problemer hvis løsningsforslag kan *sjekkes* i polynomisk tid.

NPC – NP-kompletthet:

* NP-complete is the intersection of NP-hard and NP. Equivalently, NP-complete is the class of decision problems in NP to which all problems in NP can be reduced to in polynomial time by a deterministic Turing machine.

NP-Hard:

* Trenger ikke nødvendivigvis være løsbare (Halting), ikke nødvendigivs mulig å sjekke løsning i Ptid.
* NPC-problemr kan gjøres om til NP hard varianter:

Et bilde som inneholder skjermbilde

Automatisk generert beskrivelse

**Sorteringsalgoritmer**

Boblesortering:

* Går gjennom tabellen n − 1 ganger. Bytter om naboer som står feil. Små tall «bobler opp». Store tall synker ned
* Lite brukt siden innsettingssortering er like enkelt og mer effektiv i praksis
* Ɵ(n²)

Velgesortering(selection):

* Finn største verdi, sett den på plass n-1. Sett neststørste på plass n-2.
* Fortsett slik, til vi har satt nestminste på plass 1.
* *Hva egner den seg for?*
* Om man arbeider med en uvanlig form for minne hvor skriving tar lengre tid enn lesning
* Ɵ(n²)

Insettingssortering:

* Setter ett og ett element på plass blant de sorterte. Enkel, effektiv på små datasett. Spesielt god hvis data er delvis sortert fra før
* *Hva egner den seg for?*
* Små datasett
* Om datamengden nesten er sortert fra før av (kan komme bedre ut enn de vanligvis raskere sublinære algoritmer)
* Om tallene er korrekt sortert fra før: Ω(n)
* Forventet kjøretid (gjelder for store datasett ikke små): O(n²)

Flettesorteing:

* Deler tabellen inn i to halvparter. (del-tabeller). Disse sorteres rekursivt med flettesortering • rekursjon stopper når del-tabell har størrelse 1. Så «flettes» de sorterte deltabellene sammen til én sortert tabell
* T (n) = 2T ( n/2) + n
* *Hva egner den seg for?*
* funker greit på uansett størrelse på lista
* For sortering av lenket liste
* Ulemper: krever dobbelt så mye minne som vi har data pga hjelpe tabellen
* Splitt og hersk algoritme for sortering
* Ɵ(n log n)

Heapsort:

* Start med usortert tabell, sorter den med lag\_heap (som om tabellen var en heap), Fjern rotnoden, og fiks\_heap til tom
* O(n log(n))

Tellesortering:

* Tell hvor mange det fins av hvert tall (hjelpetabell)
* Regn om til hvor mange som er <= hvert tall
* Bruk informasjonen til å sette hvert tall på rett plass

Legg merke til:

* Sammenligner ikke tallene
* Bruker tallene som index i hjelpetabellen
* Det er bare heltall som kan brukes som index
* Forutsetter at vi sorterer heltall i intervallet 0...K der K er et passende heltall.
* Er en stabil algoritme, bytter ikke om på elementer som har samme sorteringsnøkkel.
* *Når egner den seg:*
* når tallene ligger tett, ev med mange like verdier
* Om tallene er spredd utover et større område enn n bør vi heller bruke en av de generelle O(n log n) algoritmene
* Ulempe: bruker over dobbelt så mye minne som det er data til, i motsetning til algoritmer som sorterer en tabell direkte uten å ta særlig plass
* Ɵ(n + K)
* O(n) når K ∈ O(n)

Quicksort:

* Plukker ut en «delingsverdi»
* mindre plasseres nedenfor, større ovenfor, Har nå tre deler.
* Delingsverdien(på rett plass), små til venstre, større på høyre.
* De to sidene sorteres rekursivt med quicksort
* stopper når en side er størrelse 1
* *Hva egner den seg for:*
* Sorterer store datamengder raskt
* Ulemper: ikke like effektiv på små datasett
* Lønner seg å bryte rekusjonen før datasettet blir for lite og bruke en annen sorteringalgoritme for å sortere det (feks innsettingssortering)
* Dette blir raskere enn vanlig quicksort
* Splitt og hersk algoritme for sortering
* Raskeste kjente sorteringsalgoritmen for generell data
* O(n²): om datamengden blir skjevfordelt
* Dette skjer svært sjeldent og vi har i gjennomsnitt O(n log n)
* Forbedringer:
* Om vi finner en bra delingsverdi (eks median)
* Sortere små deltabeller med en raskere algoritme for mindre datasett.

*Feller:*

* I verste fall får vi skjevdeling, bare ett tall på ene siden (n^2)
* Skjevdeling med 0 elementer på ene siden (heng)
* For dyp rekursjon (kræsj)
* Skjevdeling når nøklene er like (n^2)
* Skjevdeling hvis tabellen er sortert fra før (n^2)
* Tidkrevende tester (tregt)

Quicksort med to delingstall (dual pivots)

* Velger to delingstall, p1 og p2 i stedet for ett. Må ha p1 <= p2
* Fordeler tallene i tre grupper heller enn to:
* først, de som er mindre enn p1
* så de som er >= p1 og <= p2
* deretter de som er større enn p2
* De tre delene sorteres rekursivt med denne formen for quicksort
* Hvis p1 = p2 trenger vi ikke sortere midterste intervallet (alle er like)
* Mer arbeid, men10–20% raskere.
* færre sammenligninger, mindre rekursjon

shellsort:

* Subkvadratisk sorteringsalgoritme, altså lavere kompleksitet enn tidligere algoritmer
* En forbedring på innsettingssortering
* Avhenger veldig på hvordan s senkes fra n/2 til 1
* O(n²) (worst case)
* Perfekt binærtre: alle løvnodene befinner seg på nederste nivå og alle indre noder har 2 barn
* Komplett binærtre: perfekt bortsett fra at det kan mangle noen noder på nederste nivå
* Fullt binærtre: alle indre noder har akkurat to barn det vil si at det kan finnes løvnoder som ikke er på nederste nivå
* **Preorden**
* Enhver node behandles før sine barn
* 1. Rota
* 2. venstre subtre
* 3. høyre subtre
* **Postorden**
* Enhver node behandles etter sine barn
* 1. Venstre subtre
* 2. Høyre subtre
* 3. Rota
* **Inorder**
* 1.Venstre subtre
* 2. Rota
* 3. Høyre subtre
* **Nivåordnet**
* 1.Rota
* 2. Rotas barn
* 3. disse nodens barn
* Bruker en kø for å få til
* hente en node ut av køa
* behandle den
* legge til barn i kø
* repeat til køa er tom