D130: SAMMENDRAG FRA SEDGEWICK.

1. & 2. utelatt.

3. ELEMENTÆRE DATASTRUKTURER

- En datatype er en mengde av verdier og en mengde av operasjoner på disse verdiene.
- En **abstrakt datatype** er en datatype der konkret representasjon ikke kan aksesseres direkte fra utsiden, men kun via operasjonene.
- En **algoritme** er steg-for-steg beskrivelse av et problem som kan løses. Hvert steg tar endelig tid og algoritma må terminere.
- Rekursivt problem er definert ved hjelp av seg selv. Dette er motstykke til iterasjon, som løser et problem ved å gå i løkke. Det fins to typer problemer som egnes til rekursjon: **divide and conquer** og **branch and bound/backtracking**. Førstnevnte løses ved å splitte opp problemet i flere delproblemer, som igjen løses på samme måte. Sistnevnte genererer alle mulige måter å gjøre en ting på og utfører ei avskjæring med å se bort fra enkelte måter.
- Datastrukturer:
  - *** Arrays:** 1-, 2- og 3-dimensjonale.
  - *** Lenkede lister:** Enkeltlenkede, dobbeltlenkede både sirkulær og ikke-sirkulær.
  - *** Parallelle arrays:** Representasjon av lenkede lister ved arrays. Dette gjøres ved et dataarray og et (eller flere) indeks-arrays. F.eks. sortert på navn og sortert på alder. Plassbesparende. Kan benytte de 3 første elementene som hovedliste-, friliste- og stopp-peker. Så, sjekk friliste, slett elem. fra friliste, sett inn element og oppdater pekere. Raskt og effektivt.
  - *** Stack:** Last-in-first-out. Operasjoner: *push, pop, empty*. Representeres som array eller pekerliste. Array er mest effektivt, men må sette max. grense. Pekerlista har et annet problem - den er ineffektiv.
  - *** Kø:** First-in-first-out. Representeres som stack. Operasjoner: *put, get, empty*.

4. TRÆR

- Et tre er ikketom samling av noder og kanter.
- Definisjoner er
  - Sti eller vei: sekvens av noder som er knyttet sammen med kanter. Det fins *nøyaktig én sti* mellom rot og node i treet, og mellom alle par av noder.
  - Subtre: Alle noder er rota i et subtre.
  - Ordnet tre: Etterfølgerne er ordnet.
  - Nivå: Avstand fra rota. "Generasjoner". Antall noder fra rota (ikke inkluder med rota).
  - Høyden: Max. nivå til en node.
  - Nodenavn: Internnode og bladnode.
  - Multiveistre: Alle noder må ha et visst antall etterkommere, som må opptre i en spesiell rekkefølge. F.eks. binærtre.
  - Fullt binærtre: Alle nivåer er helt oppfyllte (balansert).
  - Komplett binærtre: Det siste nivået er ikke fullt. Det fylles opp fra venstre.
- Et tre med N noder har N-1 kanter, N+1 eksterne noder, høyden [$\log_2 N$]. Max. antall noder i binærtre med høyde h er 2^{h+1} - 1 om N = interne noder.
- Tre kan representeres ved pekerlister (vanlig) og arrays. Trær kan enten være rotrettete eller ikke. Om man ikke vet hvor mange etterfølgere en node skal ha, kan man benytte to pekere i hver node - en for å peke på neste etterfølgere mens en peker på første søsken.
- **Traversering:**
  - Preorden trav.: rekursivt. Besøk rota, traverser venstre subtre og deretter det høyre subtreet i preorden.
  - Inorden trav.: rekursivt. Traverser venstre subtre, besøk rota og traverser høyre subtre. (alfabetisk).
  - Postorden trav.: rekursivt. Traverser venstre subtre, traverser høyre og besøk rota (omvendt polsk notasjon).
  - Breddesøk: iterativt. Benytt kø. Nivå for nivå-traversering. Også kalt levelorder.

```
begin       put(node, Q);
            repeat
                        node := get(Q);
                        visit(node);
                        if node^.leftchild <> nil then put(node^.leftchild, Q);
                        if node^.rightchild <> nil then put(node^.rightchild, Q);
            until empty(Q);
end;
```

5. REKURSJON.

- **Fjerning av rekursjon:**

Det er forskjellige triks som kan benyttes til å fjerne rekursjonen. Først ut er *fjerning av enderekursjon*. Her benyttes "goto beginning" i slutten av prosedyra. Komplett fjerning av rekursjonen krever ellers en egen stack. Et eksempel er preordens traversering:

```
proc prefix(node : nodeptr);
begin   if node <> nil then
        begin
                visit(node);
                prefix(node^.leftchild);
                prefix(node^.rightchild);
        end;
end;
```

```
proc prefix;
begin
      push(node, S);
      repeat
            node := pop(S);
            visit(node);
            if node^.rightchild <> NIL then push(node^.leftchild);
            if node^.leftchild <> NIL then push(node^.rightchild);
      until empty(S);
end;
```

6. ALGORITME-ANALYSE.

- Dette uttrykker kjøretiden til en algoritme som funksjon av inputstørrelsen n. Problemet er å uttrykke lengda av input som funksjon av n. T(n) defineres som kjøretidsfunksjonen. Til hjelp for dette, kan det lønne seg å spille djevelens advokat (!).
- Ved å sette en øvre grense på kjøretiden T(n), får man O(f(n)) = O(...). Dette kalles ***big oh***-notasjonen og er mest brukt. T(n) <= c*f(n) når n>=n_0. T(n) er begrenset oppad av c*f(n).
- Reglene som gjelder er: sløyf alle konstante koeffisienter og ledd, sløyf alle ledd av lavere orden, $T_1(n)+T_2(n)$ = O(max(f(n), g(n))) og $T_1(n)*T_2(n)$=O(f(n)*g(n)).
- **Rekurrenser:**

Brukes til å finne tidskompleksiteten til rekursive algoritmer. Det viktigste er å kunne skrive opp rekurrensuttrykket. Eksempelvis er flettesortering: T(n) = 2T(n/2)+n når n>0, fordi den har to rekursive kall for hvert kall og flettinga tar n steg. Det interessante nå er løsning av rekurrensproblemet. Målet er å få høyresida fri for T(n). Dette gjøres ved å ta utgangspunktet i rekurrenser og løse den opp ved benyttelse av det man vet så langt (!). Tenkt rekursivt! Et triks er f.eks. ved flettesortering - for å unngå stygge tall settes n = 2^k. Dermed løses n ut etterpå med $\log_2$ funksjonen.

Viktige egenskaper å få med seg: n reduseres hele tiden, ikke-rekursiv del av orden O(1) eller O(n), gjerne ett eller to rekursive kall. Eksempel: Knapsack-problemet.

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2\,(2(T(n-1 + 1)) + 1 \\
&\text{....osv....} \\
&= 2^0 + 2^1 + \ldots + 2^n = \Sigma\, 2^i \text{ hvor } i=0..n.
\end{aligned}
$$

Effekten av konstantene kan reduseres. Ofte er det lurere å bruke ei algoritme som er treigere for lave tall, men har lave konstanter for denne mengde input. Man leter en øvre grense samt ei nedre grense når man skal finne tidskompleksiteten til ei algoritme. Eksempel: Max.subsett sum problem.

7. IMPLEMENTASJON AV ALGORITME.

- Valg av riktig algoritme for et gitt problem, er meget essesielt for prestasjonen av programmet. Det fins f.eks. ingen superbest søkealgoritme som virker mest effektivt i alle problemer. Man må altså velge riktig algoritme fra "det store vakuum" til bruk for problemløsninga. Dette valget er ikke alltid trivielt.
- Om algoritmen skal gå få ganger, trenger den kanskje ikke være så altfor effektiv. Det er bortkastet å sitte i flere timer å kode en 100 % optimalisert lur løsning, når man fint kan klare seg med f.eks. ei elementær sorteringsrutine. Dette gjelder fint når det f.eks.er mindre enn 500 elementer som skal sorteres.

8. ELEMENTÆRE SORTERINGSRUTINER.

- De elementære sorteringsrutinene har tidskompleksitet O(n^2), mens de "avanserte" er av O(n*log n). Når man studerer sorteringsrutiner, ser man også på mengden av ekstra lager (workspace, temp-arrays) og stabilitet. **Stabilitet** er når ei sorteringsrutine bevarer den relative rekkefølgen av like nøkler. De fleste elementære sorteringsrutinene er stabile, mens de mer avanserte er ustabile. Dette kan gi bivirkninger.
- Man snakker om 3 elementære sorteringsrutiner:

* **SELECTION SORT** (seleksjonssortering).

Ideen er at man i j'te runde velger det minste elementet blant A[j] og A[N] og bytter det med A[j]. Den bruker N*N/2 som er $N^2/2$ sammenlikninger og N byttinger. Algoritma er lite sensitiv for ordninga av input.

```
procedure Selection;
  var i,j,min,t : integer;
begin
   for i := 1 to N-1 do
   begin
      min := i;
      for j := i+1 to N do if a[j] < a[min] then min :=j;
      t := a[min]; a[min] := a[i]; a[i] := t;    end;
```

Selectionsort er metoden å bruke dersom man skal sortere store records og små nøkler.

*** INSERTION SORT** (innsettsortering).
Ideen er at man i j'te iterasjon setter j'te element A[j] på sin rette plass blant A[1] og A[N]. Den bruker $N^2/4$ sammenlikninger og $N^2/2$ byttinger gjennomsnittlig. Algoritma er sensitiv for hvordan input er ordnet. Den er lineær for nesten sortert input.

```
procedure Insertion;
var i,j,v : integer;
begin
   for i := 2 to N do
   begin
      v := a[i]; j := i;
      while a[j-1] > v do begin a[j] := a[j-1]; j := j-1; end;
      a[j] := v;
   end;
end;
```

*** SHELLSORT**.
Dette er en utvidelse av insertion sort. Shellsort er raskere, fordi den tillater byttinger mellom elementer med større innbyrdes mellomrom. En avstand som er meget velvalgt er 1093, 364, 121, 40, 13, 4 og 1. Disse tallene gis ved den rekursive formelen $tall_{n+1} = 3*tall_n + 1$. I det siste tilfellet, når avstanden er 1, har man insertion sort (men da på en nesten sortert array). Shellsort bruker aldri mer enn $N^{1.5}$ sammenlikninger for denne sekvensen. Dette kan man utnytte ved f.eks. sortering av et reversert sortert array (kjappere enn insertion sort). Implementasjonen blir som insertion sort men med en h := 3*h + 1 og en repeatløkke konstruksjon.

- **Distribusjonstelling:** Ideen er ikke spesielt vanskelig: Sorter ei fil med N records med distinkte nøkler som er mellom 1 og N. Dette vil gi ei O(n) rutine med t[a[i]] := a[i]. Mer realistisk kan man tenke seg sortering av samme fila med ikke-distinkte nøkler - til dette brukes distribusjonstelling. Her gjelder det å telle opp antall like nøkler og dermed bruke tallene til å flytte records'ne til rett posisjon på andre gjennomgang av fila.

9. QUICKSORT.

- Quicksort er en "divide and conquer" metode for sortering. Ideen er slik: Man partisjonerer elementene mellom start og stopp. Det man hovedsakelig gjør når man partisjonerer, er at man flytter de elementene mindre enn "piviot" på dens venstre side og motsatt for høyre side. Dermed kjører man Quicksort rekursivt på hver av disse delene.
- Prosessen er selvsagt ikke stabil pga. partisjoneringa. Implementasjonen blir som følger:

```
procedure Quicksort(l , r : integer);
   var v, t, i, j : integer;
begin
      if r > l then
      begin
            v := a[r]; i := l-1; j := r;
            repeat
                  repeat i := i+1 until a[i] >= v;
                  repeat j:= j-1 until a[j] <= v;
                  t := a[i]; a[i] := a[j]; a[j] := t;
            until j <= i;
            a[j]:= a[i]; a[i] := a[r]; a[r] := t; {undo last swap + select new piviot}
            quicksort(l, i-1);
            quicksort(i+1, r);
            end;
end;
```

- Quicksort fungerer best på filer med litt størrelse på. Dersom quicksort kalles på ei sortert fil, vil partisjonene degenereres og dermed får man $O(N^2)$. For større, randomiserte filer får man O(n*log n), fordi $C_N = N + 2C_{N/2} =$ N log N. Quicksort bruker gjennomsnittlig 2N ln N sammenlikninger. Ressursbruken er kun en stack på størrelse $\log_2 N$.
- Muligheter for flere valg av partisjoneringselementet for å unngå "worst-case". Man kan velge medianen av første, midterste og siste element. Dermed sortere disse elementene innbyrdes og plassere disse elementene først, nest sist og sist i delarrayet. Dette kalles ofte for *medianen av tre.* Ellers har man *select* - dette er ei partisjoneringsrutine som finner det k'te minste elementet i arrayet a[l..r]. Select er likt quicksort i programmeringsstruktur. Prosedyra stokker om slik at elementene <= a[k] havner først i arrayet, mens etter a[k] så følger elementene større enn a[k]. Quicksort basert på select kjører på lineær tid (gjennomsnittlig).

10. RADIXSORT.

- Med radixsort sammenlikner man ikke hele nøkler på en gang, men utnytter heller de "digitale identitetene" blant nøklene. Eksempelvis: En kontorrotte ønsker å sortere kort med xxx-sifre på dem. Kortene blir sortert i 10 hauger (0-100,101-200 osv...). Disse haugene sorteres to ganger innbyrdes slik at alt blir sortert til slutt. Nettopp dette er prinsippet bak radixsort. Genialt?
- Anta funksjon ***bits*** (x,0,1) som returnerer bit lengst til høyre.
- Det fins to type radixsortering, og disse er:

RADIX EXCHANGE
Denne sorterer fra venstre-til-høyre, og systematisk deler opp nøklene slik. Algoritma likner mye på quicksort - avviker faktisk kun i en spesiell test. "Partisjoneringa" starter fra venstre optimistisk søkende etter en 1 bit og likeledes for høyre side letende etter bit 0 til pekerne krysses. Radixexchange(1,N,arraylengde) blir det initielle kallet. Tidsbruken er N log N bitsammenlikninger.

STRAIGHT RADIXSORT
Alternativt kan man undersøke bits'ene fra høyre mot venstre (lsb mot msb). Et viktig poeng er at algoritma er stabil. Implementasjonen er en slags generalisering av distribusjonstelling. Eks.: tre elementer 19,45,6 skal sorteres. Først puttes 19 på plass 9, 45 på plass 5 og 6 på plass 6. Dermed har man 45,6,19, som igjen plasseres på sine riktige plasser 45 plasseres på 4., 6 plasseres på 0. og 19 plasseres på 1. plass ==> 6,19,45 (sortert). Straight radixsort kan sortere N elementer med b bitnøkler med b/m sammenlikninger. Om b/m er konstant ==> tid er b/m - N = O(n) som er lineært! Baksida av medaljen er selvsagt 2^m elementer i countarray+buffer (plass).

- Begge metodene bruker mindre enn N*B bitsammenlikninger for å sortere N elementer med b bits nøkler. Valget mellom quicksort og radixsort er vanskelig. Men en pekefingerregel er at Quicksort er raskest (tja... for skikkelig store filer kan det tenkes at radixsort vinner men....). Dessuten er det temmelig strenge krav på nøklene i radixsort. Man kan også øke farta ved kombinasjon av insertion sort. For maskinkodeprogrammører er dette løsninga.

11. PRIORITETSKØER.

- Dette er en passelig datastruktur som støtter operasjonene *sett_inn* og *slett_største_element* (i motsetning til stack og kø). En slik struktur kalles **prioritetskø**. Man snakker om to forskjellige implementasjoner for prioritetskøer. Den ene er å ha elementene i mengden representert som et (u-)sortert array. Med sortert array tar *Construct* og *Insert* O(n^2), *Remove* O(1) siden man tar ut det siste elementet, mens *Change* og *Delete* tar O(n). For usortert array-representasjon, tar *Construct* og *Insert* O(1) mens *Remove* tar O(n). Det er altså forskjellige fordeler.
- En annen implementasjon av prioritetskøer (mest brukt) er **heap**. Dette er et binærtre som er både balansert og komplett, men i tillegg har en delvis ordning slik at rota i et subtre har størst verdi.
- Merk at ved oversetting fra tre til array, så er foreldre til en node alltid lik indeks / 2.
- Implementasjonen av *contruct, insert og remove* benytter seg kun av sti fra rota til bunn. Dermed tar operasjonene maksimalt $\log_2$ N. Om man skal sette inn eller slette noe fra heapstrukturen, må man huske på å bevare heap-betingelsen. Til dette har man henholdsvis to rutiner som heter *upheap og downheap*. Heapbetingelsen "fikses" i upheap ved å bytte den nye noden med dens foreldre. Dette kan igjen føre til et nytt brudd på heapbetingelsen, som videre må "fikses" på samme måte som det forrige bruddet. Dette er en slags insertion sort langs stien mot rota, hvor rutina terminerer. Metoden for å slette minste element i en heap er enkelt: bytt plass med rota og siste element, slett siste element og kall downheap for å rydde opp. Downheap startes ved rota og flytter den "nye" elementet nedover i heapen til trav_node er større enn begge barna eller bunnen av treet er nådd. Ellers har man *replace* som er litt mer komplisert - den benytter a[0] på kunstig vis: dets barn er 0 og 1, så om v > største element i heap da berøres ikke heapen. Om ikke dette er tilfelle så er v satt inn i heap og a[1] returneres. *Delete* rutina kan lages ved at man kombinerer upheap og downheap sammen. Eksempelvis: om prioriteten til element på indeks k er øket ==> upheap(1), mens motsatt (den er minket) kalles downheap(k).
- **Heapsort** er enkel: Man konstruerer ei prioritetskø med N elementer. Om man tar *Remove* N ganger på denne heapen, så får man sortert liste i motsatt rekkefølge. Men motsatt rekkefølge kan "ordnes på" ved å benytte seg av en for-løkke som teller nedover mot 1 og lagre e.g. a[i] := remove. Orden til heapsort er O(n*log n).
- Egentlig er det bedre å bygge heapen nedenfra og opp, fordi downheap virker på små heaper som store. Ved å jobbe bakover opp i heapen, trenger man bare å kalle downheap(1) for å gjøre ferdig jobben (rota er problemet). Dette kalles for "**bottom-up**" heapbygging. Dette tar faktisk lineær tid, og grunnen til det er at de fleste heapene er små. Heapsort bruker < 2N lg N sammenlikninger med N elementer. Merk at heapsort er stabil.
- Som oftest orker man ikke å flytte rundt på store records ved prioritetskøer. Dette unngår man enkelt ved å benytte seg av *indirekte heaps*. Dette er liknende "pekersort". Det viktige er at man ønsker å vedlikeholde et annet array som holder heap-posisjonene til alle elementene i hovedarrayet. Slik får man "pgConstruct"- og "pgDownheap"-rutiner.

12. MERGESORT.

- Mergesort består hovedsakelig av to rekursive kall etterfulgt av ei fletterutine (mergerutine). Orden til mergesort er $C_N = 2C_{N/2} + N = O(n*\log n)$. I motsetning til quicksort, så er mergesort stabil, fordi den relative posisjonen til elementene er uberørt av mergerutina.
- Hovedulempen med mergesort er at den bruker mye plass proporsjonalt med N. Ideen til sorteringsrutina er som følger: del fila i to, sorter de to halvdelene rekursivt og flett disse sammen etterpå.
- Måten mergesort i array-implementasjonen gjøres på er som følger: kall rekursivt (splitt) for hver halvdel, og i "herskedelen" så kopieres alle elementene i venstre del over i temperært array b og videre kopieres den høyre halvdelen av originalarrayet over i b-arrayet i motsatt rekkefølge. På denne måten slipper man å bruke "sentinel-nøkler". Dermed flettes disse to delarrayene av b over i originalarrayet. Dermed er området sortert. Implementasjonen blir slik:

```
procedure mergesort(l, r : integer);
  var i, j, k, m : integer;
begin
    if r-l > 0 then
    begin
        m := (r+l) div 2; mergesort(l,m); mergesort(m+1, r);
        for i := m downto l do b[i] := a[i];
        for j := m+1 to r do b[r+m-1-j] := a[j]; {reverse order}
        for k := l to r do
            if b[i] < b[j] then begin a[k] := b[i]; i := i+1; end
            else begin a[k] := b[j]; j := j+1; end;
    end;
end;
```

- Mergesort kan aksessere dataelementene i sekvensiell orden. Det vil si at mergesort er egnet til å sortere lister. Prinsippet her er som følger: benytt et dummyelement z på slutten av lista, ha en funksjon som returnerer peker til sortert liste - denne funksjonen finner midten av innlista ved å gå dobbelt så langt frem med ett element som det andre. Dermed kuttes disse listene i to, hvor mergesort kalles individuelt på hver av delene, som merges sammen til slutt.
- **Bottom-Up mergesort** er en ikkerekursiv variant. Den går igjennom lista hvor den merger sammen 1 og 1 elementer i sublister, etterpå går den igjennom lista produserende sublister med størrelse 4, dermed sublister med 8 i størrelser osv. Dette foregår inntil hele lista er sortert. Selve strukturen på denne prosedyra blir to repeat-løkker hvor den ytre terminerer om a=head^.next, mens den indre når to_do=z. Arrayimplementasjonen av problemet er liknende.
- Merk at mergesort gjør mesteparten av arbeidet etter de rekursive kallene, i motsetning til quicksort.

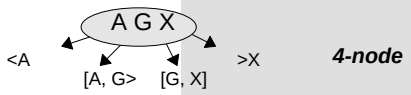
14. ELEMENTÆRE SØKEMETODER.

- Man har en mengde av elementer med nøkkelverdier. For å finne elementer med gitt nøkkel, fins det flere elementære måter å søke etter det på. Måter å representere mengden kan være array eller lenket liste.
- **Sekvensiell søk** leter gjennom alle elementene i mengden, én for én. For en usortert mengde er det lurt å sette det søkende elementet ved slutten av lista eller arrayet, fordi da blir testen i den indre løkka kort. Suksess markeres ved at man e.g. sjekker at arrayindeksen er <= N (antall elementer i mengden). Mislykket søk trenger N+1 steg, mens vellykket søk trenger gjennomsnittlig N/2 steg. Dette var for usorterte mengder.

  For sorterte mengder tar sekvensiell søk gjennomsnittlig N/2 uavhengig av suksess eller failure. Den store fordelen med sorterte mengder er at man kan benytte:
- **Binærsøk:** Benytter en "divide & conquer" metode som deler mengden inn i to deler. Om "x" er mindre enn midterste element, så let i venstre del. Hvis ikke, gjelder det motsatte. Algoritma er meget fristende rekursiv av natur, men det er faktisk meget trivielt å programmere iterativt (og raskt). Binærsøk bruker aldri mer enn $\log_2$ N+1 sammenlikninger for suksess eller failure, fordi rekurrensen er $C_N = C_{N/2} + 1$. Et viktig poeng er at *insert* operasjon i ei sortert liste er krevende, så dette er ikke helt "himmelsk" algoritme. En forbedring av binærsøk kan være å "gjette" på hvilket intervall søket bør startes på, likeledes som når man leter etter et telefonnummer i telefonkatalogen. Man starter der det er mest sannsynlig å finne elementet. Denne metoden krever helst at nøklene er uniformt (likt) fordelt. Metoden kalles **interpolasjon søk**. Ordenen til denne algoritma er *O(log log N)*. *Blyraskt!*
- **Binærtre** er den mest simple, effektive, dynamiske datastrukturen for rask søking. Implementasjonen kan løses ved bruk av dummy-element som peker på seg selv (eller uten). Binærsøk anvendes direkte på denne representasjonen. Insertoperasjonen leter seg ned i treet til "rett" plass og setter nytt element inn der. Lengden av stiene i treet er gjennomsnittlig log n. Derfor er dette effektivt. Søk og insert operasjonene krever gjennomsnittlig 2log N steg med randomiserte nøkler. Merk at treet fort kan degenereres til ei lineær liste om elementene som settes inn er "sortert", slik at ordenen blir O(N) i worstcase.
- **Sletting i binærtre** er ikke trivielt. Anta at t skal slettes i et tre. Let normalt til man kommer til t i treet og sett endelig posisjon p=t. Dermed er det 3 tilfeller: i) om t^.right=nil da skal p's barn settes lik t^.left; ii) om t^.right<>nil og t^.left=nil så skal t^.right settes lik p's barn etter sletting med dens venstre link kopiert fra t. iii) ellers settes x=minste element i t's høyre subtre. x^.right settes lik venstre link fra dens foreldre, og begge av dens links er satt fra t. Dette er som sagt ikke enkelt. Det fins en annen metode, *lazy delete* - som markerer en node for "slettet".

15. BALANSERTE TRÆR.

- For å unngå worstcase med binærsøk, benyttes en spesiell type datastruktur - trær som forblir balanserte hele tiden.
- **Top-down 2-3-4-trær** tillater 3 type noder: 2-node (vanlig), 3-node (<A, [A,G], >G) og 4-node (<A, [A,G>, [G,X], >X).



4-node

- For å sette inn ny node i 2-3-4-tre må man gjøre et mislykket søk som tidligere og hekte på den nye noden. Om man kommer til en 2-node, kan man selvsagt terminere og forandre noden til en 3-node. Likeledes for 3-node. Problemet melder seg når man kommer til en 4-node. Det greieste er å splitte 4-noden opp i to 2-noder og overføre den midterste av dens nøkler, slik at det blir plass for den nye noden nederst. For å unngå at foreldrenoder er 4-noder, splitter man alltid 4-noder på vei ned i treet. Merk at slike transformasjoner er helt lokale og påvirker ikke hele treet. Straks rota blir 4-node og splittes, økes treets nivå med en (og kun da!). Siden 4-noder splittes ovenfra og ned, kalles trærne top-down 2-3-4-trær og disse er helt *balanserte!*

- Søk i slike trær tar aldri (!) mer enn log N+1 steg. Innsetting krever max. log N+1 steg, men gjennomsnittlig færre steg.
- En "grei" måte å representere 2-3-4-trær i datamaskinen, er å ta i bruk **rød-svarte trær**. Disse trærne er vanlige binære trær som bruker en ekstra bit for hver node. 3- og 4-noder representeres som små binære trær bundet sammen med "røde" kanter mellom. Implementasjon her er ikke enkel. Problemet er at i en sti langs treet skal det alltid være en kant mellom to røde kanter. Til dette trenger man å fargeflipp, rotasjon og splitt prosedyrer. Roteringa er også en lokal operasjon som bringer nodene til venstre for N et steg nærmere mot rota, mens de nodene til høyre for N er senket et steg. For å unngå enkelte komplikasjoner kan dobbelrotasjon være lurt. Implementasjonen og pseudoen til dette er ikke eksamensrelevant (kan man jo håpe på ja... :-).
- Søk i rød-svarte trær krever rundt log N sammenlikninger, mens innsetting krever gjennomsnittlig mindre enn en rotasjon. En øvre grense for søk er 2log N+2 og innsetting er ¼ så mange rotasjoner som sammenlikninger.

16. HASHING.

- Hashingprinsippet er en metode for å oppnå direkte aksess til et søkt element i en mengde ved hjelp av aritmetiske identiteter. Elementene (eller indirekte med pekere) er altså lagret i en tabell [0, ... , M-1]. En hashfunksjon omformer nøkkelen (som kan være string, tall etc.) til en tabellindeks, slik at man får direkte aksess. På denne måten tar oppslaget konstant tid O(1). Hashing er en trade-off handel - den forbruker mye ekstra plass men er uhyre effektivt.
- Hashfunksjonen må være enkel og rask å beregne, distribuere godt over tabellen og utnytte hele nøkkelen godt. Et eksempel på en hashfunksjon på et array på størrelse M, kan være (nøkkel) mod M. Her er det viktig at M er primtall, fordi ellers vil hashfunksjonen være "gjentakende", med at den treffer tidligere elementer ofte slik at man får store "hopp" i arrayet (dårlig utnyttelse).
- Man snakker om to typer hashing - **åpen og lukket hashing**.
- **Åpen hashing** kalles **seperate chaining**, og det er i prinsippet et array med listehoder, som er første element i ei enkeltlenket liste hvor alle "kollisjons"-elementer havner hen. Anta at dette arrayet med pekere er M stort, da er tidsbruket for insert og search lik O(N/M) hvor N=antall elementer i mengden. En ide kan være å benytte sorterte lister under array-recordene eller benytte binærtre-struktur her, dog er det sannsynligvis ikke verdt det. En god regel er å la M=N/10.
- **Lukket hashing** er når elementene lagres direkte i tabellen, hvor M > N og hvor kollisjoner behandles på forskjellig vis. Mulig måte å unngå "kollisjoner" på er:

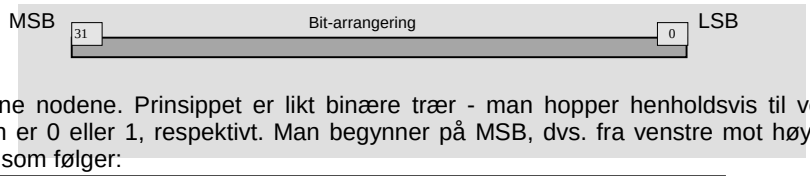
LINEÆR PROBING
Lineær probing er når man søker på påfølgende element når den indeksen som hashfunksjon først ga, allerede er brukt. Om det påfølgende elementet er fullt, søker man videre på neste element. Om det ikke er ledig, går man videre. Dette foregår helt til en ledig plass er funnet. Om hashtabellen er mindre enn 2/3 full, tar det omtrent 5 "probinger".

DOBBEL-HASHING
Istedenfor å hoppe ett steg bortover i arrayet, får man hashet en fiksert avstand ut fra nøkkelen ved bruk av en andre hashfunksjon. Dette minker sannsynligheten for lange sekvensielle søk. Man øker indeksen med $u=h_2(i)$. Kravene til denne hashfunksjonen er nokså harde: For å unngå opphopninger er det viktig at funksjonen aldri returnerer 0 (evig løkke) og at den og M ikke er delelig på hverandre (slik at tabellen brukes overalt). Av dette blir hashfunksjonene forskjellige fra hverandre. Dobbelhashing er gjennomsnittlig raskere enn lineær probing, men da må tabellen være mindre enn 80 % full for mislykket søk og 99 % full for suksess søk.

- Ved lukket hashing har man følgende problem - sletting og innsetting er ikke enkelt og bestemmelse av M er ikke simpelt. Om tabellen formodentlig skulle bli full, kan man jo alltids lage en ny hashtabell. En generell regel er at man aldri burde la tabellen bli mer enn 90 % full. For at insert, delete, delete_min skal ta O(log N) og find skal ta O(1), så kan man kombinere balanserte søketrær og hashing (f.eks. binærtrær og hashing).

17. RADIX SØKING.

- Radixsøking ser på enkeltbit i nøkkelverdiene. Man antar at Pascal har en funksjon bit(nøkkel, i) som returnerer verdien på i'te bit fra høyre i nøkkelen. Det fins 3 metoder å gjøre dette på - *digitale søketrær, radix søke tries og patricia.* Hele tiden er det forutsatt at nøklene er distinkte.



DIGITALE SØKETRÆR
Nøklene ligger i de interne nodene. Prinsippet er likt binære trær - man hopper henholdsvis til venstre eller høyre avhengig av om søkebit'n er 0 eller 1, respektivt. Man begynner på MSB, dvs. fra venstre mot høyre i bitsekvensen. Algoritma for søkinga blir som følger:

```
denne := rot;  bpos := maxb-1;
while <ikke har passert forbi en bladnode> and (nøkkel <> denne^.nøkkel) do
begin        if bit(nøkkel, bpos)=0 then denne:= denne^.vsub else denne := denne^.hsub;
             bpos := bpos-1;
end;
funnet := (nøkkel=denne^.nøkkel);
```

Når man går ned til nivå i, skiller man nøkler med hensyn på i'te bit fra venstre. Eksempelvis skal man lete etter bokstaven "H" - 01000. Da går man først til venstre, høyre, venstre og venstre helt til man kommer til bokstaven "H". Rekkefølgen på innsetting av elementene i treet har noe å si for representasjonen av selve treet. Søk og innsetting i digitale søketrær tar gjennomsnittlig log N og b sammenlikninger i worst case, hvor b=antall bits i nøkkelen. Problemet er at i hver iterasjon, så må man gjøre en full nøkkelsammenlikning. Løsninga er å benytte seg av ei algoritme som krever kun en full nøkkelsammenlikning pr. søk - dette løser radix søke-tries.

RADIX SØKE-TRIES.
Ideen er å plassere nøklene i bladnodene (eksterne noder). De interne nodene representerer bit-for-bit søkeveier ned til bladnodene. Om en nøkkel fins i treet, så ligger den langs stien som representerer venstre bitsekvens i nøkkelen. Implementasjonen er helt identisk med digitale søketrær, foruten av while-test blir mer effektiv ==>
... while not bladnode(denne) do ... Hvert søk krever da i snitt $\log_2$ N og i verste fall maxb bitsammenlikninger. Problemet med denne representasjonen er at man får 1.44*N noder (N/log N), og 44 % mer unødvendig lagringsplass er betenkelig. Løsninga er å benytte en annen algoritme som bruker kun én full nøkkelsammenlikning pr. søk og N noder - patricia.
Men det er verdt å merke seg at radix søketries ikke tar hensyn til innsettingsrekkefølgen av elementene - representasjonen blir den samme. Man kan få merkbar økning i ytelse ved å sammenlikne mer enn én bit om gangen. Man kan f.eks. ha multiveis radix søketries med 4-noder, dog er implementasjonen av noe slikt usannsynlig vanskelig!

PATRICIA
Fordelen er, som sagt, kun én sammenlikning av hele nøkler med n noder og elementer. Ideen er å plassere nøklene i de interne nodene. "Bladnodene" markeres ved at de peker tilbake til de interne nodene oppover i treet. Langs veien ned til en "bladnode" gjør vi bare bit-for-bit sammenlikninger. Et meget viktig poeng videre er at hver node inneholder en variabel *bpos* som angir hvilket bit som er det neste interessante å sammenlikne. Som før går man til venstre hvis denne bit'n er 0 og høyre motsatt.
Alle søkene terminerer når man har nådd en "uplink". Da trengs en full nøkkelsammenlikning for å avgjøre suksess. Innsettinga i patriciatrær skjer som følger... Anta at v skal settes inn i treet. Sett t=noden hvor v ville ha måttet være, og i =MSB (<--venstre) hvor v og t ikke matcher. Anta så at b=siste bitposisjon som førte søket til t. Observer om t og v må matche på b, slik at man får 2 tilfeller (ja, nei). Søk ordinært, dvs. flytte p og x nedover langs "matchende sti". Etter termineringa får man to tilfeller (ikke illustrert). Avhengig av situasjonen, lager man ny node t for v, og la skillebit'et være i. Plassér t eller p også før x. (Dette er ikke eksamensrelevant og dermed ikke nøye diskutert.)
Patricia har altså N noder og krever $\log_2$ N bitsammenlikninger for en gjennomsnittlig søk.

18. EKSTERN SØKING.

- Når elementene man skal lete etter ligger på sekundærlageret, er det aktuelt å lese hele den aktuelle siden i primærlageret. Å lese inn en hele side, er nokså tidskrevende og en metode for å unngå unødvendige innlesninger er ønsket. Her er det hovedsakelig 3 metoder:

INGEN INDEKS.
Man bruker k sider og risikerer k side-innlesninger for én post-aksess. Fordelen er riktignok maksimal utnyttelse. Selv om elementene ligger sortert, risikerer man å lese inn k sider, uansett.

INDEKSERT SEKVENSIELL AKSESS.
Her bruker man den første sida på hver "disk" som indeks for hvor dataene ligger. Ulempene er at man må sette av ekstra plass for indeks og innsetting/sletting krever omstrukturering av hele fila. Fordelen er selvsagt at søkinga krever konstant antall sideinnlesninger.
Men: anta at en liste har 5 sider. Dette krever at nøkkelandelen av en post er <= 1/5. Om man har mange disker, kan indeksen spres over flere sider likevel. Dette er med andre ord temmelig *relativt.*

B-TRÆR.
Bedre metode å behandle slik søk er å bruke balanserte trær med den egenskapen at antall nøkler pr. node er B stor (B-trær). Man snakker altså om en generalisering av 2-3-4-trær. Alle "interne noder" har mellom [m/2..m] barn, og alle bladnoder har samme dybde. m er ordenen til B-treet. Det fins to varianter av B-trær.
Ved første varianten inneholder alle noder både hele dataelementer og barnepekere. Fordelene med dette er at søk og innsetting krever mindre enn $\log_{m/2}$ N sideinnlesninger fra disk, hvor N=antall nøkler.
Den andre varianten av B-trær lar kun bladnoder inneholde hele dataelementer. De andre nodene inneholder kun nøkler og barnepekere. Om nøklene er små, kan de interne nodene ha større grad. Fordelene er at det blir færre side-innlesninger $\log_m$ N og separering av indeks og dataelementene. Ulempen er at man får omtrent 1.44N/M noder i treet, hvor N=antall elementer i mengden og M=antall orden av treet.
Et triks for hastighet er å holde indeksen i primærminnet under søking. B-trær vil først lønne seg ved store filer.

20. PATTERN-MATCHING.

- Stringsøking er fundamentalt for datavitenskap. Algoritmer som "matcher" kompliserte M-bokstavers mønstre i N-bokstavers tekststrenger har tidskompleksitet MN i worst case. Et spesielt språk brukes til å beskrive slike mønstre. Symboler er bundet sammen med 3 operasjoner: **sammensetting** (AB, A etterfulgt av B), **valg** (A+B, valg A eller B) og **tillukning** (A*, A forekommer nå 0 eller flere ganger etter hverandre). Eksempelvis kan C(AC+B)D* være CACDD eller CBD osv. Uttrykk som kan beskrives slik kalles *regulære uttrykk*. Presedensen til operasjonene er *, AB og + i synkende rekkefølge.
- Til å avgjøre om en tekst matcher et gitt regulært uttrykk, trengs en *endelig tilstandsautomat*. Man har en "match" hvis og bare hvis det fins en måte å spasere gjennom automaten som tilslutt fører oss til stopp-tilstanden. Siden automaten ikke kan avgjøre om man har en match ved å undersøke hvert tegn, tilegnes automaten egenskapen ikke-deterministisk - dvs. at automaten "gjetter" riktig. Man får mange tilstander. I tillegg fins det såkalte *nulltilstander,* som peker på (flere) tidligere tilstander. Eksempelvis innføres nulltilstander ved tillukning - en tilstand peker syklisk til det symbolet som ønskes gjentatt. Stopptilstanden er også en nulltilstand.
- Selvsagt kan man ikke lage ei algoritme som er ikke-deterministisk. I virkeligheten må man teste alle muligheter.
- Implementasjonen av den ikke-deterministiske maskinen, benytter seg ikke-rekursivt av en algoritme som bl.a. krever en **deque**. Sistnevnte er en slags kloning av en stack og kø - elementer kan settes inn i starten eller slutten av lista. Operasjonene er altså push, put og pop.
- Hoveddelen av programmet fjerner en tilstand fra deque'n og gjør det som deretter trengs. Om en bokstav skal matches, sjekkes det for ønsket symbol. Dersom det fins, puttes en ny tilstand på slutten av deque'n. Hvis tilstanden er nulltilstand, puttes de to mulige tilstandene i begynnelsen av deque'n. Dette medfører at input-symbolene og de symbolene som er resultat av neste tilstandspeker, må separeres. Dette gjør man ved å putte et "scan-symbol" i deque'n. Når man treffer på scan-symbolet igjen, betyr det at man kan gå videre i tekststrengen. Løkka terminerer når deque'n er tom, når en nulltilstand er nådd (match funnet) eller hvis tekststringen er "exhausted" (target er nådd). Ingen match funnet er hvis man kun står igjen med scan-symbolet i deque'n. Koden er rett frem fra dette, og den kan være en funksjon som returnerer siste matchede bokstav i stringen. Ordenen blir O(MN), hvis høyst antall iterasjoner mellom hver gang man leser et tegn er M.
- Datastrukturen brukt i algoritma er helst 3 parallelle arrays - et for nåværende tilstand mens de to andre er for de to andre mulige tilstandene (next1, next2).
- Implementasjonen kan også gjøres rekursivt. Jo har et eksempel på dette.

21. PARSING.

- Problemstillingen "parsing" er å undersøke om en tekst er grammatisk korrekt. Dette er en nødvendig del av en kompilator. Man skal se på grammatikker for en spesiell type språk, nemlig *kontekst-frie språk.* Dette er språk som kan beskrives av BNF-notasjon.
- Det fins to generelle metoder for dette: *top down* (rekursiv) og *bottom up* (iterativ). Top down rekursiv parsing studeres i dette kapitlet. En parser "forstår" en string ved å bygge et **parsetre** for uttrykket. Parsetreet kan bygges hvis og bare hvis uttrykket er grammatisk korrekt. Når treet er bygget, kan treet traverseres på forskjellig vis, slik at man lett kan *oversette* fra et språk til et annet. Man får altså to faser i prosessen.
- Implementasjonsdetaljer ved bygging av parsetrær ved **top down**: Lag først ei prosedyre for hvert symbol på venstre side av BNF-grammatikken (f.eks. sum, ledd). I starten av hver prosedyre, forventer man å finne spesielle symboler. Hvis man støter på noe uønsket, skrives det ut ei feilmelding om feil grammatikk. Disse prosedyrene bør helst være rekursive, for å tillate flere ledd på samme nivå. Eksempelvis er uttrykk, ledd og faktor - disse må kompileres med forward. Rutinene inneholder gjerne flere gjensidige rekursive kall på hverandre for å tillate flere nivåer. Top-down descent fungerer ikke på alle kontekst-frie språk. F.eks. <uttrykk> ::= v | <uttrykk>+<ledd>. Dette gir ei uendelig løkke, fordi man vet ikke hva som skal velges fra starten av.
- **Bottom up** implementasjonen er værre. Man bør nok helst benytte seg av en stack og herme Pascals rekursjon. I tillegg kan man sjekke for paranteser ved å øke og minke en teller. Om telleren er null til slutt, er parantesene ok. Riktignok er dette så innviklet at det regnes som irrelevant til eksamen.
- Det bør presiseres at parsing er et bredt forskningsfelt i dataverdenen. Om ytterligere lesestoff er nødvendig, så refererer jeg til obligatoriske oppgave II, alternativ 2 (parsetre).

29. ELEMENTÆRE GRAF-ALGORITMER.

- **Definisjoner:** En graf er en samling av noder og kanter. Den kan være urettet eller rettet. En ***enkel vei*** i en graf er en sekvens med *unike* noder forbundet med kanter (=sti, bare vei=gange). En ***sykel*** er en enkel vei hvor det går en kant mellom første og siste. Grafer uten sykler (løkker) kalles ***asykliske***, f.eks. et trær. Nodemengden betegnes ved V og kantmengden ved E. En ***subgraf*** av en graf G=(V,E) har inneholdt alle sine hjørner i V, hvor kantene til subgrafen selvsagt bare går ut til subgrafens hjørner henholdsvis. Man sier gjerne at en graf har en subgraf, hvis man eksempelvis fjerner et hjørne og "tar" tak i et gjenværende hjørne, så følger en subgraf med. I en **forbundet** graf går det minst én vei mellom hvert par noder. Hvis grafen ikke er forbundet, består den av to eller flere forbundne ***komponenter***, som hver for seg er forbundede grafer. En asyklisk forbundet (sammenhengende) graf kalles et ***fritt tre***. Et ***spenntre*** er en subgraf inneholdende alle hjørnene men bare kantene tilstrekkelig for å danne et tre. En viktig egenskap er hvis grafen har mindre enn V-1 kanter, så er den ikke sammenhengende. Hvis grafen har mer enn V-1

kanter, har man en sykel. En ***vektet graf*** er hvor hver kant har en vekt. Et ***minimalt spenntre*** er hvor vektsummen i en vektet graf er minimal. Slike MST er ikke distinkte for hver graf. ***Rettete*** grafer har kanter som går "ei vei". Rettete vektede grafer benyttes oftest i aktivitetsnettverk.
- Representasjon av grafer i datamaskinen er på 3 måter:
  - **** nabomatrise:*** Her benyttes et to-dimensjonalt array med true eller false. Hoveddiagonalen er uvesentlig. Om det går en kant mellom A og B, så er $A_{rad}*B_{kol}$ = true og $B_{rad}*A_{kol}$ = true.
  - **** naboliste:*** Her benyttes et 1-dimensjonalt array med listehoder for hver node. Listehodene peker på nabolister til gitt node. Fordelen er at nabolista tar mindre plass ved få kanter.
  - **** direkte pekerstruktur:*** Gitt ved dummyhoder hvor noder virkelig "peker" på hverandre - som på tegninga.

- For å avgjøre om grafen er sammenhengende, hvilke komponenter den består av og om den har noe sykler, kan man benytte **dybde-først søk**. Ideen er å besøke alle ubesøkte "barn" (noder) rekursivt i hvert søk. Man tilegner hver node en **`id`** som er økende hele tiden. Dette kan lagres i et array eller i selve noden. Dette brukes til å sjekke om en node er besøkt, før man kaller dybde-først søk på den. For nabolister blir dfs-strukturen en "while"-løkke som går i gjennom alle utkantene til noden og sjekker om noen av disse ikke er besøkt - de ubesøkte besøkes deretter rekursivt. Tidsbruken for naboliste-representasjon er *O(V+E)*, fordi man besøker V noder og undersøker hver kant to ganger = E.
For nabomatrise-representasjon blir implementasjonen temmelig lik - man har et array **`besøkt`** som tilegnes en **`id`** etterhvert som noder besøkes. Tidsbruken for nabomatrise-representasjon er $O(V^2)$, fordi hver bit i matrisa er undersøkt.
- Dybde-først søk kan utføres ikke-rekursivt for nabolister. Implementasjonen bruker verdiene i besøk-arrayet som følger: 0=ikke besøkt, <positivt tall>=besøkt og -1=på stacken. Visit-prosedyra blir da slik:

```
procedure visit(k : integer;
  var t : link;
begin     push(k);
          repeat
                k := pop; id := id+1; besøkt[k] := id; t := adj[k];
                while t <> nil do
                begin     if besøkt[t^.nr] = 0 then
                          begin     push(t^.nr);
                                    besøkt[t^.nr] := -1;
                          end;
                          t := t^.neste;
                end;
          until stackempty;
end;
```

- **Bredde-først søk** besøker alle "barna", så alle "barne-barne" osv. Algoritma benytter seg av en typisk kø. Implementasjonen er nesten lik den over for dybde-først søk. De eneste forskjellene er at man bytter ut stack-operasjonene med kø-operasjoner. Dette tilsvarer nivå for nivå traversering av et tre!

30. FORBUNDEDE GRAFER.

- Å skrive ut komponentene i en graf er simpelt - kjør dybde-først søk på alle nodene og skriv ut hver node kun én gang.
- En **biforbundet** graf har minst to forskjellige stier som binder sammen to noder. Om grafen ikke er biforbundet, er det minst et punkt som er "flaskehalsen" - all kommunikasjon over grafen foregår gjennom dette punktet. Dersom dette punktet fjernes, står man igjen med to komponenter. Et slikt punkt kalles **artikuleringspunkt**. For å finne artikuleringspunkter, trengs det en utvidelse av dybde-først algoritma - man må finne laveste **`id`** som kan nås fra noden. Dette gjøres som følger: Man lar *min* være minimum av nodens egen id (x) og laveste id som kan nås fra de ubesøkte nodene. Det rekursive kallet returnerer laveste id som kunne nås fra denne naboen. Denne verdien er mindre eller lik x. Hvis verdien er lik x, har man et artikuleringspunkt, fordi naboen er en del av en forbundet graf ut fra x som kun kan nå øvre mengden via x. Dette kan gjøres i lineær tid, O(V+E).
- **Union-find algoritmer** brukes til å finne ut om node X og Y er i samme mengde (med i samme forbundne komponent). Prinsippet er at man snur pekerne i trærne slik at man får **rot-rettede trær (unions)**. Fordelene er at søkeveiene som man skal følge, blir kortere. Mengdene må være disjunkte. Gitt to noder. For hver av dem, begynner man å tusle mot rota med en peker. Om disse pekerne er like når rota er nådd hos begge, er X og Y med i samme mengde.
- Datastrukturen for union-find er et *foreldre-array* (eller om man vil - et etterfølger array). Bygging av slike trær er proporsjonalt med V^2, dvs. $O(V^2)$. Et ønske for rotrettede trær med høyde mindre eller lik 2 utlyses. Fordelen er at da vil avgjørelsen om X og Y er i samme mengde, være O(1) !
- Den nye datastrukturen som trengs til dette er basert på **vektbalansering og sti-komprimering**. Førstnevnte implementeres ved at man vedlikeholder størrelsen av hvert tre i etterfølger-arrayet med et positivt tall. Sti-komprimering baserer seg på at man lar rota være etterfølger til flere noder. Man ønsker altså at helst alle noder peker på rota. Dette er implementert ved å gå en annen gang gjennom treet etter rota er funnet og sette etterfølger-arrayet for hvert forekommet hjørne langs veien, til å peke på rota. Dette gir find-algoritma høy hastighet, og i tillegg tar bygginga nesten lineær tid.

31. GRAFER MED VEKTER.

- Minimale spenntrær og korteste vei problemer er essensielt for grafteorien. Skal først se på 3 metoder for å finne minimale spenntrær:

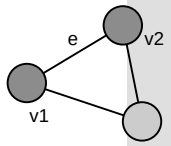
PRIORITET-FØRST SØK (MST).
Man ønsker jo hele tiden å ta ut kanter som fører til minimal vekt av grafen. For å ta ut den kanten som fører til minst vektøkning, kan prioritetskøer benyttes. Implementasjonen av dette er bortimot lik dybdeførst og breddeførst algoritmene, med to unntak. For det første - en kant som allerede er med i MST'et, må undersøkes om den nye kanten senker prioriteten. For det andre trengs et array for å vedlikeholde foreldrenoden til en node. Med andre ord, trenger man en representasjon av selve MST'et.
Prioritet-først søk krever O($\log_2$ V*(V+E)) steg, fordi prioritetskø-operasjonene tar hver $\log_2$ V steg og disse må utføres for hvert hjørne og hver kant (V+E).

PRIMS ALGORITME (MST).
Strategien er å la MST'et "vokse" ut fra en vilkårlig node ved hele tiden å inkludere den korteste kanten som forbinder det foreløpige MST'et med de andre nodene. Dette er analogt med det man lærte i Ma-141. Pseudokoden for dette blir slik:

```
proc Prim( G : graf; var T : kantsett );
  var U : kantsett; u, v : kant;
begin     T := Ø;
          U := {1};
          while U<>V do
          begin     la (u, v) være korteste kant slik at u ⊂ U og v ⊂ (V-U);
                    T := T ∪ { (u,v) } ;
                    U := U ∪ {v} ;
          end;
end;
```

-Tidskompleksiteten til Prims algoritme er O(n^2), fordi man n ganger går gjennom n-1 noder. Om antall kanter i en graf er omtrentlig lik n^2, faller valget av algoritme på Prims. Ellers brukes neste metode ....

KRUSKALS ALGORITME (MST).
Stragegien her er å la mange små MST'er "smelte" sammen til et stort MST, ved hele tiden å inkludere den korteste kanten som ikke gir en sykel. Om man antar at grafen er forbundet, blir pseudokoden slik:




```
proc Kruskal;
begin
    repeat
        e := < neste kant i stigende vektrekkefølge >;
        if < e.v1 og e.v2 ikke allerede er forbundet av inkluderte kanter > then <inkluder e i MST>
    until < antall kanter = (V-1) >;
end;
```

Problemet er først hvordan man skal aksessere kantene i stigende vektrekkefølge. En mulighet er å sortere alle kantene etter vekt, men da kan man risikere sortering av flere kanter enn man får bruk for. Fordelen er at aksessen blir O(1). En annen måte er å bruke prioritetskø. Da tar oppbygginga O(E), mens ulempen blir at aksessen blir O($\log_2$ E).
Videre må man kunne avgjøre om e.v1 og e.v2 er forbundet. Til dette lønner union-find seg med O(V).
Kruskals algoritme krever O(E*log E) steg for å finne et minimalt utspenningstre, for hver kant kjøres union-find på (union-find tar gjennomsnittlig log E).
- Å finne **korteste vei** er interessant. Det fins to algoritmer for å løse dette problemet: prioritet-først søk og Dijkstras algoritme.

PRIORITET-FØRST SØK (korteste vei).
Dette er en generalisering av dybde- og bredde-først søking. Gå hele tiden til den kanten som gjør slik at stilengda fra A til B blir kortest mulig, ved å velge første element (minste) fra prioritetskøen. For dybde-først søk lar man prioritetskøen være en stack, slik at yngste kant har størst prioritet. For bredde-først søk blir prioritetskøen en FIFO-kø, hvor eldste kant har størst prioritet. Prioritet-først søk på spredte grafer for å finne korteste vei mellom A og B, trenger O(log V*(V+E)) steg.

DIJKSTRAS ALGORITME (korteste vei).
For "tette" grafer lønner denne algoritma seg. Den er den samme som benyttes for korteste vei i Ma-141. Strategien: Anta at node 1 er startnoden, V=total nodemengde og S=mengden av noder hvis korteste vei allerede er bestemt (*permanente* noder). La mengden av de veiene fra 1 som dannes ved å bruke noder i S, og så la den siste kanten gå ut til en node i V-S. Dermed er w $\in$ (V-S) endenoden i den korteste veien. Pseudokoden for algoritma blir da:

```
proc Dijkstra ( var D : korteste vei array; C : kantvekt array );
begin     S := {1} ;
          for i := 2 to n do D[i] := C[1, i] ; {init D}
          for i := 1 to n-1 do
          begin     < velg en node w ∈ (V-S) slik at D[w] er minimal >
                    < inkluder w i S >
                    < oppdater veilengdene i L >
                    for < hver node v ∈ (V-S) > do D[v] := min ( D[w], D[w] + C[w, v] );
          end;
end;
```

Om man vil ha med hvilken sti som er korteste vei, kan man sette inn F[v] := w i den indre løkka. F[v] er forgjenger-noden til v langs den korteste veien man hittil har sett fra 1 til v !

32. RETTEDE GRAFER.

- Man ønsker å finne ut om det går en rettet vei fra A til B. Om dette er mulig, lager man en direkte kant fra A til B. Dette kan selvsagt gjøres ved å kalle dybde-først søk for alle nodene, men dette er ei rekursiv rutine som krever O(V*(V+E)) steg for spredte grafer og $O(V^3)$ steg for tette grafer. En ikke-rekursiv variant løser også problemet, og den algoritma heter Warshalls algoritme. Problemet defineres som **transitiv tillukning**.

WARSHALLS ALGORITME (TRANSITIVITET).
Ideen er om det fins en vei mellom X og Y ved å bruke node Z med mindre indeks enn Y og om det fins en vei fra Y til Z, da fins det en vei fra X til Z ved å bruke indekser mindre enn Y+1. Implementasjonen blir slik:

```
for y := 1 to V do
   for x := 1 to V do
      if A[x, y] then
         for z := 1 to V do
            if A[y, z] then A[x, z] := true;
```

Tidsbruken til Warshalls algoritme for transitiv tillukning til en graf, er $O(V^3)$ ikke-rekursive steg.
- For å finne korteste vei mellom alle par av noder i en graf, benyttes Floyds algoritme.

FLOYDS ALGORITME (ALL PAIR SHORTEST PROBLEM).
Antar rettede, positive kanter i nabomatrisa A. Om det ikke er noen kant, så får den uendelig stor vekt. Målet er å omarbeide A slik at A[i, j] er lengden av korteste vei mellom i og j. Hvilken vei som er kortest, er ikke relevant - kun lengda. Hovedideen er at hvis det går en vei av lengde ik mellom i og k, og en vei med lengde kj mellom k og j, så må det gå en vei av lengde ik+kj mellom i og j. Gjentar man dette systematisk for alle tripler (i, j, k) får man Floyds algoritme og en orden $O(V^3)$. Implementasjonen blir slik:

```
for  y := 1 to V do
   for x := 1 to V do
      if A[x, y] > 0 then
         for j := 1 to V do
            if A[y, j] > 0 then
               if ( A[x, j]=0 ) or ( A[x, y]+A[y, j] < A[x, j] ) then  A[x, j] := A[x, y]+A[y, j] ;
```

- For rettede asykliske grafer (DAG) kan **topologisk sortering** foretas. Det fundamentale her er at intet hjørne v kan besøkes før alle forgjengerne til v er besøkt. Dette er traverseringa av grafen i obligatoriske oppgave III. Bemerk at den topologiske sorteringa ikke er unik. Det fins mange slike sorteringer (revers topologisk sortering etc.). Man har to metoder å finne slik topologisk sortering - den ene er basert på dybde-først søk mens den andre bredde-først søk.
Strategien ved dybde-først søk er at når alle forgjengerne til node x er plassert i ts_liste, så kan x plasseres først i lista. For bredde-først søk plasseres x bakerst i lista, dersom alle forgjengerne til x er plassert i ts_liste. Koden blir:

Dybde-først søk.

```
procedure besøk ( x : node ) ;
var    ef : hode ;
begin  < marker x som besøkt >
       for ef := < alle etterfølgere av x > do besøk(ef);
       < sett inn x først i ts_liste >
end;
```

```
procedure TopSort_Breddeførst;
begin    < putt de initielle frontene inn i FrontEndQue FNQ >
         while not empty(FNQ) do
         begin     x := get(FNQ);
                   for ef := < x's etterfølgere > do
                   begin     ef^.nfg := ef^.nfg -1;
                             if ef^.nfg=0 then put(ef, FNQ);
                   end;
                  <plasser x bakerst i lista >
          end;
end;
```

- **Sterkt forbundede komponenter** i en rettet graf er en delmengde av noder hvor det går en rettet vei mellom hvert par av noder. En algoritme for å finne disse komponentene har følgende strategi: i) gjør et dfs i G, og nummerer nodene i stigende rekkefølge rett før besøks-prosedyra returnerer. ii) Konstruer en ny graf G' ved å snu alle kantene i G. iii) Gjør dfs i G' ved å starte i noden med høyest nummer. Hvis søket ikke besøkte alle nodene, gjenta søket fra den høyest nummererte av de gjenværende ubesøkte nodene. iv) Hvert tre i det resulterende dfs-spenntreet, er en sterkt forbundet komponent. Metoden ble oppfunnet av Tarjan, og er ufattelig smart. Den krever kun O(V+E). Beviset for at dette virker: Anta at A er noden med høyest id og P er mengden av noder som kan nås fra A. $P \cup A$ er da en sterkt forbundet komponent, fordi nodene i P kan nås fra A etter at kantene i grafen snus!

33. NETTVERKSFLYT.

- Anta at man har et nettverk for å lede væske (olje, vann) fra en kilde til et utløp. I hvert skillepunkt er det en *regulator* (switch) som bestemmer hvor mye vann (liter pr. sekund) som skal flyte i hvert av de utgående rørene. Rørene kan ha ulik kapasitet (tykkelse). I hver regulator må inflow=outflow og følgelig må inflow=outflow for hele nettverket. Problemet er hvordan man skal stille regulatorene slik at man får maxflow. En klassisk algoritme for dette er:

FORD-FULKERSON ALGORITMEN (MAXFLYT).

Først lar man vannet være avskrudd (flyten er overalt lik 0). Så lar man M være mengden av urettede veier fra kilde til utløp, hvor veien verken har en full-foroverkant eller en tom bakover-kant (betingelse). Velg en vilkårlig vei V i M. Øk total gjennomstrømming ved å velge verdien F slik at betingelsen over er oppfylt. Flyten til V's fremoverkant økes med F, mens den reduseres for V's bakover-kanter. Om M nå er tom, har man funnet maksimal flyt. Hvis ikke, gjenta prosessen fra andre steg.

Når man velger veier i M, kan man velge veien med færrest kanter. Dette gjøres ved algoritme for korteste vei (bredde-først) hvor alle vekter er lik 1. Antall veier som må velges, blir da færre enn V*E. En annen strategi for valg av vei, er å velge den veien som gir størst økning i flyten. Dette gjør man ved å bruke f.eks. Dijkstras for å finne kortere vei i en vektet graf med nødvendige modifikasjoner. Første vei som har nådd utløpet har "vunnet". Implementasjonen blir slik:

```
function ny_vei : boolean;
begin     put(Q, kilde);
          while not empty(Q) do
          begin     x := get(Q);
                    < marker x som besøkt >; ny_vei := true;
                    if x=utløp then <ferdig>
                    for y := <alle ubesøkte noder i starten av alle inngående kanter som ikke er tomme > do
                    begin     forgjenger[y] := x;
              *                put(y, Q);
                    end;
                    for y := < alle ubesøkte noder i slutten av alle utgående kanter som ikke er fulle> do *
          end;
     ny_vei := false;
end;
```

Hovedprogrammet kaller opp ny_vei og øker flyten med et tall, helt til ny_vei returnerer false (while).

34. MATCHINGER.

- En matching er et subsett av kanter der ingen hjørner forekommer mer enn en gang. **Maksimal matching** er når flest mulig noder dekkes. Problemet er å finne denne maksimale matchinga. Det fins to tolkninger av problemet, og disse er *maksimal matching mot maksimal vektet matching*. Måten å løse problemet på er genial.
- Anta at man har en todelt graf (biparite graph), som er en inndeling av nodene i 2 mengder hvor en kant ikke kan gå mellom to noder i samme mengde. En omformulering av problemet gir oss tilfellet med maksimal flyt i nettverk: *Finn største delmengde av kanter som tilfredstiller at ett symbol forekommer høyst en gang*. Alle kanter får vekt lik 1, og man har maksimal matching når maks. flyt er oppnådd. Dette kalles en reduksjon av problemet. De åpne rørene (kantene) i maks. flyt, tilsvarer nå maksimal matching i det midterste nivået mellom kilden og utløpet. Maksimal flyt oppnås når *så mange som mulig av nodene i nederste mengde* er representert med åpen kant.
- En maksimal matching i todelt graf oppnås etter $O(V^3)$ steg for tette grafer og O(V*(V+E)*log V) steg for spredte grafer. Maks. matching kan selvsagt løses ved "brute force", dog er ikke dette særlig lurt => $O(2^n)$.
- **Stabile marrige problemet** er å foreta en stabil kopling mellom mennesker slik at begge deres krav blir oppfylte, dvs. er lykkelig gifte! Anta at man har n kvinner og menn, hvor hver av disse har sine personlige rankingliste for hvem de vil gifte seg med. Et ustabilt ekteskap er hvis det fins to par hvor mannen i det ene og kvinnen i det andre, heller vil ha hverandre enn den de er sammen med for øyeblikket.

En løsning på problemet er "brute force", men man er ute etter en mer effektiv algoritme - man simulerer virkeligheten istedenfor. Strategien blir slik: i) La mennene skifte på å være frier. ii) En mann frir til kvinnene i samme rekkefølge som rankinglista si. iii) Ved frieri får man to tilfeller. Om hun er ledig, blir det forlovelse med en gang (!). Hvis hun er forlovet, kan to ting skje: 1. Hun foretrekker frieren fremfor forloveden, og dermed må forloveden ut på gata slik at hun og frieren kan forlove seg. 2. Hun avslår tilbudet, og da må frieren prøve på neste kvinne i lista si.

Datastrukturen blir slik: **next[m]** = array som forteller rangeringa av neste kvinne mannen prøver seg på, **prefer[m, i]** = array som forteller at kvinnens rangering som nr. i hos m, **rank[w, m]** = array som forteller kvinnen w's rangering av mann m og **fiancee[w]** = kvinnen mannen er forlovet med nå. Dette gir implementasjonen:

```
for m := 1 to n do
begin     s := m;
          repeat
               next[s] := next[s] +1;
               w := prefer[s, next[s] ];
               if rank[w, s] < rank[w, fiancee[w] ] then begin swap(s, fiancee[w] ); s := fiancee[w]; end;
          until s=0 {dummy-mannen er nådd}
end;
```

Problemet løses i lineær tid. Beviset for dette er noe komplisert og dermed utelatt. Verdt å få med seg er riktignok at det ikke eksisterer ustabile forlovelser etter algoritmas terminering. Dermed er antall forlovede lik n.

42. DYNAMISK PROGRAMMERING.

- Dynamisk programmering er når man løser mange (alle!) små delproblemer og lagrer svarene på disse, slik at disse kan benyttes til å løse større problemer seinere. Altså løses problemet ved å **kombinere** forskjellige **"småløsninger".** Metoden brukes også når en splitt-og-hersk løsning er vanskelig å formulere eller har stort tidsforbruk. Eksempler på slike tilfeller er ved *flettesortering, binære trær algoritmer, Warshalls algoritme for transitiv tillukning og Floyds algoritme for å finne alle korteste veier.*
- **Digresjon:** Insertion sort er et spesialtilfelle av flettesortering. Eneste forskjellen for å lage insertion sort rekursivt i forhold til flettesortering, er at midten hos insertion sort = høyre-1. Rekurrensen i begge tilfeller blir T(n)=T(n-1)+T(1)+n som er $O(n^2)$.
- Eksempel på dynamisk programmering ved korteste vei. Begynn i motsatt ende av en rettet asyklisk graf (DAG), og løs delproblemene å finne korteste veilengde nærmest B. Tilslutt vil man finne korteste vei fra B til A, dvs. fra A til B. Dette kan gis i ei algoritme:

```
K := Ø;
F := Ø;
for x := < noder uten etterfølgere > do L[x] := ∞ ;
L[B] := 0; < sett B inn i F >; <oppdater K>;
while < F ikke inneholder A > do
begin
      x := < ta ut vilkårlig node fra K >;
      L[x] := min( vekt[x, x.ef1] + L[x.ef1] , vekt[x, x.ef2] + L[x.ef2] );
      < sett x inn i F >; < oppdater K >;
end;
```

Kommentar:

Å oppdatere K vil si å oppdatere mengden av alle noder som kun har etterfølgere til noder i F (og dermed er klare for å få satt sin korteste vei).

Dette kan også gjøres rekursivt med splitt-og-hersk. Dermed gjør man utregning på samme ting flere ganger.
- En tyv med en sekk med kapasitet M raner en safe med N forskjellige ting i. **Knapsack problemet** er hvordan tyven skal få flest verdifulle ting med seg i den lille sekken. Problemet er også kalt **binge-problemet**. En algoritme for å løse problemet er å generere alle mulige sekker med noen avskjæringer, slik:

```
for j := 1 to N do
begin     for i := 1 to M do
          if (i-size[j]) >= 0 then
          if cost[i] < (cost[ i-size[j] ] + val[j] ) then
          begin          cost[i] := cost[ i-size[j] ] + val[j]; best[i] := j;
          end; { best brukes til å finne ut hva som er i sekken etterpå }
end;
```

Denne algoritma løser knapsack problemet med O(NM).
Det fins 3 ytterligere algoritmer for bingepakking: **first-fit**, **best-fit** og **first-fit-decreasing & best-fit-decreasing**. First-fit plasserer elementer i den bingen med lavest indeks hvor det får plass. Dette er ikke veldig optimalt. Prinsippet i Best-fit er at elementet plasseres i den første bingen med nok kapasitet. Denne har lik ytelse med First-fit. For bedre ytelse må man rette oppmerksomheten mot siste mulighet: Elementene sorteres etter avtakende størrelse før de plasseres i bingene (...-decreasing). En implementasjon av en ***optimal*** løsning med O(n!) er:

```
for < alle permutasjoner av tallene i A > do
begin     < gå til første binge >
          for < alle elementene > do
          begin     if <elementet ikke får plass i bingen>  then < gå til neste binge >
                    < putt elementet i bingen >
          end;
end;
```

For å generere samtlige permutasjoner av A[i..n], brukes rekursiv algoritme:

```
proc gen_perm( i : integer );
begin     if i=n then < funnet en permutasjon >
          else
          begin     for j := i to n do
                    begin     gen_perm( i+1 );
                              roter_venstre( i );
                    end;
          end;
end;
```

- Dynamisk programmering kan også benyttes til å finne ut hvor mange utregninger som trengs for multiplikasjon av to matriser. Videre kan binære søketrær optimaliseres ved at ofte aksesserte elementer plasseres på toppen.
- **Grådighetsalgoritmer:** Dette er algoritmer som bringer deg raskest mot målet på kort sikt. Disse gjør den beste lokale forandringen - men ikke nødvendigvis den beste på lang sikt. Prims og Kruskals algoritmer er eksempler på hvor

denne strategien førte frem. Eksempel: Om man skal hjem - er det lureste å sykle til studenthuset (5 min) og videre hjem (30 min) eller sykle til Langnes (20 min) for å ta helikopter hjem (30 sek) ? Dette fungerer heller ikke på traveling salesman problemet (se neste kapittel).

44. UTMATTENDE SØK - EXHAUSTIVE SØK.

- Søkerutiner som tar eksponensiell tid $O(2^n)$ må undersøke alle muligheter. Det mest kjente problemet av en slik type er **traveling salesman problemet** - gitt N byer, finn korteste vei gjennom alle byene uten å besøke en by to ganger. Her må man sjekke alle turene. Det er nettopp sjekkinga av disse turene som er "exhaustive søk". En implementasjon av ei algoritme som søker alle mulige veier, bygger på prinsippet at den *rydder opp etter seg* slik:

```
proc visit ( k : integer );
begin       id := id+1; val[k] := id;
            for t := 1 to V do
            begin   if A[k, t] then   if val[t]=0 then visit(t);
                    id := id -1; val[k] := 0;
            end;
end;
```

- **Backtracking** er når man ved "exhaustive søk" reduserer mulighetene. Sagt annerledes: Straks man innser at veien man "kjører på" umulig kan føre oss til målet, rygger man tilbake til forrige punkt hvor det var uprøvede muligheter. Orden blir likevel $O(2^n)$.
- **Avskjæring** (pruning) kan foretas for å redusere graden av backtracking. Dette gjøres ved å oppdage på nåværende stadium at et gitt veivalg umulig kan føre til målet. Eksempel på en slik avskjæring kan være en if-setning i knapsack problemet - om sekken blir full ved å plassere gitt gjenstand i den, så unngå å putt den i sekken! (smålogisk...) Huskeregel er Jo's gamle Ford Granada med slitte sommerdekk.

45. NP-KOMPLETTE PROBLEMER.

- Man kan klassifisere algoritmeproblemer i 3 klasser: medgjørlige ($O(n^k)$), umedgjørlige ($O(n!)$, $O(2^n)$) og ikke-avgjørbare problemer. Det fins altså problemer som ikke lar seg løse ved et programmeringsspråk. Eksempel på det er flisene til Jo.
- Rekapituler deterministisk og ikke-deterministiske algoritmer ("gjetter seg frem"). Man sier at **mengden NP** er problemer som kan løses i polynomisk tid av en ikke-deterministisk algoritme. **Mengden P** er delmengde av NP og problemene i P kan løses polynomisk av deterministisk algoritme. Sagt med andre ord: i P er alle medgjørlige algoritmer. **Mengden NPC** inneholder de umedgjørlige problemene, siden ingen har funnet polynomiske algoritmer for dem. Slike problemer kalles **NP-komplette**.
- Disse problemene er gjensidig avhengige, fordi eksistensen av en polynomisk algoritme for noen NPC-elementer, betyr at det fins en polynomisk algoritme for hver av de andre (relasjon innenfor mengden). For å kalle et problem NP-komplett, må det kunne *reduseres* fra et kjent NP-komplett problem til et liknende et i polynomisk tid.
- Måten man ***reduserer*** et problem fra NP til NPC-klassen, er at man rett og slett legger begrensninger på input til den polynomiske algoritmen (som har gjenkjent slående lik problemstilling). Dermed får man at NP-problemet er et spesialtilfelle av den polynomiske algoritma (dessuten får man nobelprisen). Problemet vil være å finne ut hvilke begrensninger som P skal pålegges.
- Et eksempel: Gitt traveling salesman problemet og *hamilton sykel* algoritma. Sistnevnte forteller om det fins en enkel vei som inkluderer alle hjørnene. Man ønsker å finne ut om traveling salesman problemet er NP-komplett. Avstandene mellom byene, blir kantene i hamilton sykel-algoritma, og problemet blir nå å finne en tur med avstand mindre eller lik N - dvs. mindre eller lik antall hjørner i grafen. Dette impliserer at traveling salesman problemet er NP-komplett.
- Reduksjonen er som regel meget komplisert. Flere tusen problemer har vist sin NP-kompletthet på denne måten. Eksempler er multiprosessores jobbplaner og partisjonering (fins det to partisjoner med lik sum?).

Ordensanalyse til algoritmer.

$T(n) = T(n-1) + 1 \quad = O(n)$
$T(n) = 2T(n-1) + 1 \quad = \sum 2^i = O(2^n)$
$T(n) = T(n-1) + n \quad = O(n^2)$
$T(n) = 2T(n-1) + n \quad = \sum 2^i(n-i) = O(2^n)$?
$T(n) = T(n/2) +1 \quad = O(\log_2 n)$
$T(n) = 2T(n/2) + 1 \quad = 2n = O(n)$
$T(n) = T(n/2) + n \quad = O(n*\log_2 n)$?
$T(n) = 2T(n/2) + n \quad = O(n*\log_2 n)$

Alle de elementære sorteringsrutinene har orden $O(n^2)$.
Quicksort har orden $O(n*\log_2 n)$.
Radix exchange har orden $O(n*\log_2 n)$.
Straight radixsort har $O(n)$ om b/m sammenlikninger er konstant. } gir $O(N*B)$,

Heapoperasjonene har hver $O(\log_2 n)$.
Heapsort har $O(n*\log_2 n)$.
Mergesort har $O(n*\log_2 n)$. Det samme gjelder for bottom-up-mergesort.
Binærsøk har $O(n)$, men gjennomsnittlig $O(\log_2 n)$.
2-3-4-trær har $O(\log_2 n)$.

Åpen hashing har $O(N/M)$, hvor M=antall pekere i arrayet og N=antall elementer i mengden.
Lukket hashing har spesialtilfeller for lineær probing og dobbelhashing.

Digitale søketrær tar gjennomsnittlig $\log_2 n$ steg. $\Rightarrow O(b)$, dvs. antall bits i nøkkelen.
Innunder samme finner man radix søketries og patricia.

Ekstern søking ved å bruk av 2-3-4-trær har $O(\log_{m/2} n)$, hvor m=antall nøkler pr. node.
Patternmatching tar $O(MN)$, hvor M=søkestreng-lengde og N=tekststreng-lengde.

Dybde-først søk for nabolister tar $O(V+E)$, mens for nabomatrise $O(V^2)$.
Samme for breddeførst søk.

Å finne artikuleringspunkt i graf for nabolister tar $O(V+E)$.

Prioritetskø MST tar $O(\log_2 n*(V+E))$.
Prims MST tar $O(V^2)$.
Kruskals MST tar $O(E*\log_2 E)$.
Prioritetskø korteste vei tar $O(\log_2 V*(V+E))$.
Dijkstras korteste vei tar $O(V^2)$.
Warshalls transititivitet tar $O(V^3)$.
Floyds alle-par korteste vei tar $O(V^3)$.
Maksimal matching i tette grafer tar $O(V^3)$, eller $O(V*\log_2 V*(V+E))$.

Stabile marrige problemet kan løses med $O(n)$.
Knapsack problemet tar $O(NM)$, hvor M=kapasitet i sekken og N=antall forskjellige ting.
Backtracking og avskjæringsproblemer har $O(2^n)$.

UNIVERSITY OF TROMSØ

SAMMENDRAG: SEDGEWICK

COPYRIGHT (c) 1995

Eksamenforberedelser

D-130
Våren 1995

Sveinar Rasmussen
sveinarr@stud.cs.uit.no

Revidert utgave v1.5
Korrigert 25. november 1996

Institutt for matematiske realfag.