Magnus Lie Hetland

Pensumhefte, 2019

Algoritmer og datastrukturer

2019-10-04 15:41:39

Innledning

Dette pensumheftet er en kombinasjon av tre ting:

(*i*) En forelesningsplan;

(*ii*) En pensumliste; og

(*iii*) Et mini-kompendium.

For hver forelesning er det listet opp hva som er hovedtema, og hvilken del av pensum forelesningen bygger på. I tillegg har jeg laget en detaljert opplisting av læringsmålene i faget, for å tydeliggjøre hva det er meningen at dere skal kunne. Bakerst i heftet er det noen appendiks med ekstra materiale, og det utgjør mini-kompendie-delen. Sammen med pensumboka, forelesningene og øvingene håper jeg at dette gir dere det dere trenger for å sette dere inn i stoffet – men det er viktig å huske på at det er dere som må gjøre jobben med å lære det!

> Her er det ikke bare faktakunnskap man skal tilegne seg; det er ideer som krever modning over litt tid, og det er derfor viktig at dere begynner å jobbe tidlig, og at dere jobber jevnt.

Det er ikke så taktisk å utsette pensumlesningen til eksamensperioden, og håpe at et skippertak skal ordne biffen.

Det er også viktig å merke seg at læreboka [1] er viktig i dette faget. Det er flere måter å fremstille algoritmestoff på, og man kan selvfølgelig bruke de ressursene man vil for å lære seg det. Det finnes mange videoforelesninger og animasjoner og lignende på nett, og mye er forklart ganske rett frem i Wikipedia og ellers. Men til syvende og sist så skal det jo være en eksamen i faget, og der må vi ha en eller annen form for standardisering, av rent praktiske hensyn.

> Derfor baserer vi oss på antagelsen om at dere faktisk leser læreboka, og at dere leser jevnt gjennom semesteret.

Forelesningene skal gi dere et første møte med stoffet, slik at det forhåpentligvis blir lettere for dere å forstå boka når dere setter dere ned for å lese selv. Det er ingen grunn til panikk om du ikke skjønner absolutt alt i en forelesning, eller om du ikke husker alt etterpå. Sannsynligvis er du bedre rustet til å jobbe med stoffet uansett.

Et hovedtips til studeringen kan være:

Test deg selv!

Det er et velkjent fenomen fra f.eks. forskning på hukommelse at om man leser en tekst flere ganger så kan man lures til å tro at man husker den, og kanskje til og med forstår den, bare fordi den virker kjent. Dette er en skummel felle å gå i; det er lett å tro at man har skjønt mer enn man har! Øvingene gir deg en god mulighet til å teste hva du faktisk har fått med deg, men det går jo an å finne andre måter å gjøre det på også.

Ikke bare er det å gjøre oppgaver en viktig måte å teste deg selv på, men det er også en måte å styrke hukommelsen på – såkalt *recall training*. Det er mulig stoffet ligger godt lagret, men om du ikke trener på å hente det frem, så kan det være du ikke husker det likevel. Dessuten er oppgaver en god trening i å anvende ferdighetene du tilegner deg i faget. Om du er interessert, så finnes det store mengder med algoritmeløsnings-oppgaver på nett, både knyttet til programmeringskonkurranser og til jobbintervjuer.

En siste, kanskje litt pussig gevinst ved å løse oppgaver er at det kan bidra til å bygge opp mentale strukturer som er mottakelige for relaterte ideer. Dette er det såkalte *pretesting*-fenomenet: Om du prøver deg på en oppgave fra en del av pensum du ikke har lært noe om ennå, selv om du *vet*, at du ikke kommer til å få den til, så kan det bidra til at du lærer stoffet bedre når du begynner å jobbe med det. Du trenger ikke se på løsningen til oppgaven engang. Her er det mange muligheter – det er for eksempel fullt av oppgaver i læreboka, og dem kan det være lurt å kikke igjennom *før* du begynner å lere; kanskje til og med før du går til forelesningen.

Dette gjelder gamle eksamensoppgaver også, naturligvis. Vi prøver å variere form og innhold litt, så man ikke skal kunne «pugge eksamen», men det kan likevel være lurt å øve seg på gamle eksamener. De nyeste vil trolig være mest relevante. Her også kan pretesting være en tanke.

Hva med å prøve deg på et par eksamenssett allerede den første uka?

Og skulle det være noe du ikke forstår, noe du trenger hjelp til eller noe du vil slå av en prat om, så gjør gjerne det! Om det er spørsmål du tror flere kan ha nytte av å se svaret på, så spør gjerne i fagets forum på nett. Ellers går det fint an å sende e-post eller ta personlig kontakt med øvingsstaben eller meg.

Om det er uvant å skulle lære seg relativt teoretiske ting med høy grad av presisjon, så finnes det generelle råd og tips rundt dette. En bok jeg har anbefalt flere, og fått gode tilbakemeldinger på, er denne:

Bokanbefaling

A mind for numbers av Barbara Oakley [2]. Nært knyttet til boka er *Coursera-*

kurset *Learning how to learn: Powerful tools to help you master tough subjects.*

I de følgende lister med læringsmål er noen merket med *utropstegn*. Dette er en ganske uformell greie, men er ment å indikere temaer som kanskje er viktigere enn man skulle tro. Av og til betyr det de merkede målene er viktigere enn de andre, men iblant – for mer marginale ting – så er det mer av typen «Obs! Dette bør du ikke nedprioritere!» Men, som sagt, det er ganske uformelt, og ikke noe du bør legge altfor stor vekt på.

Overordnede læringsmål

De overordnede læringsmålene for faget er som følger.*

Dere skal ha kunnskap om:

[X_1] Et bredt spekter av etablerte algoritmer og datastrukturer

[X_2] Klassiske algoritmiske problemer med kjente effektive løsninger

[X_3] Komplekse problemer uten kjente effektive løsninger

Dere skal kunne:

[X_4] Analysere algoritmers korrekthet og effektivitet

[X_5] Formulere problemer så de kan løses av algoritmer

! [X_6] Konstruere nye effektive algoritmer

Dere skal være i stand til:

[X_7] Å bruke eksisterende algoritmer og programvare på nye problemer

[X_8] Å utvikle nye løsninger på praktiske algoritmiske problemstillinger

Mer spesifikke læringsmål er oppgitt for hver forelesning.

Punkter nedenfor merket med □ er pensum. Kapittelreferanser er til pensumboka *Introduction to Algorithms* [1].

Forkunnskaper

En viss bakgrunn er nødvendig for å få fullt utbytte av faget, som beskrevet i fagets anbefalte forkunnskaper.[1] Mye av dette kan man tilegne seg på egen hånd om man ikke har hatt relevante fag, men det kan da være lurt å gjøre det så tidlig som mulig i semesteret.

Dere bør:

* Se emnebeskrivelsen, `https:/ntnu.no/studier/emner/TDT4120`.

[Y_1] Kjenne til begreper rundt *monotone funksjoner*

[Y_2] Kjenne til notasjonene $\lceil x \rceil$, $\lfloor x \rfloor$, $n!$ og $a \bmod n$

[Y_3] Vite hva *polynomer* er

[Y_4] Kjenne grunnleggende potensregning

[Y_5] Ha noe kunnskap om grenseverdier

! [Y_6] Være godt kjent med logaritmer med ulike grunntall

[Y_7] Kjenne enkel sannsynlighetsregning, indikatorvariable og forventning

[Y_8] Ha noe kjennskap til rekkesummer (se også side 15 i dette heftet)

[Y_9] Beherske helt grunnleggende mengdelære

[Y_{10}] Kjenne grunnleggende terminologi for relasjoner, ordninger og funksjoner

[Y_{11}] Kjenne grunnleggende terminologi for og egenskaper ved grafer og trær

[Y_{12}] Kjenne til enkel kombinatorikk, som permutasjoner og kombinasjoner

De følgende punktene er ment å dekke de anbefalte forkunnskapene. De er pensum, men antas i hovedsak kjent (bortsett fra bruken av asymptotisk notasjon). De vil derfor i liten grad dekkes av forelesningene. De er også i hovedsak ment som bakgrunnsstoff, til hjelp for å tilegne seg resten av stoffet, og vil dermed i mindre grad bli spurt om direkte på eksamen.

☐ Kap. 3. Growth of functions: 3.2

☐ Kap. 5. Probabilistic analysis . . . : s. 118–120

☐ App. A. Summations: s. 1145–1146 og ligning (A.5)

☐ App. B. Sets, etc.

☐ App. C. Counting and probability: s. 1183–1185, C.2 og s. 1196–1198

Vi vil generelt prøve å friske opp relevant kunnskap, men om dette stoffet er ukjent, så regnes det altså stort sett som selvstudium. Dersom det er noe dere finner spesielt vanskelig, så ta kontakt.

Vi forventer også at dere behersker grunnleggende programmering, at dere har noe erfaring med rekursjon og grunnleggende strukturer som tabeller (*arrays*) og lenkede lister. Dere vil få noe repetisjon av dette.

Gjennom semesteret

Det følgende er pensum knyttet til flere forelesninger gjennom semesteret:

☐ Lysark fra ordinære forelesninger

☐ Appendiks til dette dokumentet

Læringsmål for hver algoritme:

[Z_1] Kjenne den formelle definisjonen av det generelle problemet den løser

[Z_2] Kjenne til eventuelle tilleggskrav den stiller for å være korrekt

[Z_3] Vite hvordan den oppfører seg; kunne utføre algoritmen, trinn for trinn

! [Z_4] Forstå korrekthetsbeviset; hvordan og hvorfor virker algoritmen egentlig?

[Z_5] Kjenne til eventuelle styrker eller svakheter, sammenlignet med andre

[Z_6] Kjenne kjøretidene under ulike omstendigheter, og forstå utregningen

Læringsmål for hver datastruktur:

[Z_7] Forstå algoritmene (jf. mål Z_{01}–Z_{06}) for de ulike operasjonene på strukturen

[Z_8] Forstå hvordan strukturen representeres i minnet

Læringsmål for hvert problem:

[Z_9] Kunne angi presist hva input er

[Z_{10}] Kunne angi presist hva output er og hvilke egenskaper det må ha

Pensumoversikt

Delkapitler er også listet opp under tilhørende forelesning, sammen med læringsmål. Kapittel- og oppgavereferanser er til *Introduction to Algorithms* av Cormen mfl. [1].

☐ Kap. 1. The role of algorithms in computing

☐ Kap. 2. Getting started

☐ Kap. 3. Growth of functions: Innledning og 3.1

☐ Kap. 4. Divide-and-conquer: Innledning, 4.1 og 4.3–4.5

☐ Kap. 6. Heapsort

☐ Kap. 7. Quicksort

☐ Kap. 8. Sorting in linear time

☐ Kap. 9. Medians and order statistics

☐ Kap. 10. Elementary data structures

☐ Kap. 11. Hash tables: s. 253–264

☐ Kap. 12. Binary search trees: Innledning og 12.1–12.3

☐ Kap. 15. Dynamic programming: Innledning og 15.1, 15.3–15.4

☐ Kap. 16. Greedy algorithms: Innledning og 16.1–16.3

☐ Kap. 17. Amortized analysis: Innledning og s. 463–465 (tom. «at most 3»)

☐ Kap. 21. Data structures for disjoint sets: Innledning, 21.1 og 21.3

☐ Kap. 22. Elementary graph algorithms: Innledning og 22.1–22.4

☐ Kap. 23. Minimum spanning trees

☐ Kap. 24. Single-source shortest paths: Innledning og 24.1–24.3

☐ Kap. 25. All-pairs shortest paths: Innledning, 25.2 og 25.3

☐ Kap. 26. Maximum flow: Innledning og 26.1–26.3

☐ Kap. 34. NP-completeness

☐ Oppgaver 2.3-5, 4.5-3, 16.2-2 og 34.1-4, med løsning

☐ Appendiks A–F i dette heftet

Forelesning 1

Problemer og algoritmer

> Vi starter med fagfeltets grunnleggende byggesteiner, og skisserer et rammeverk for å tilegne seg resten av stoffet. Spesielt viktig er ideen bak induksjon og rekursjon: Vi trenger bare se på *siste trinn*, og kan *anta* at resten er på plass.

Pensum:

☐ Kap. 1. The role of algorithms in computing

☐ Kap. 2. Getting started: Innledning, 2.1–2.2

☐ Kap. 3. Growth of functions: Innledning og 3.1

Læringsmål:

[A_1] Forstå bokas *pseudokode*-konvensjoner

[A_2] Kjenne egenskapene til *random-access machine*-modellen (RAM)

[A_3] Kunne definere *problem*, *instans* og *problemstørrelse*

! [A_4] Kunne definere *asymptotisk notasjon*, O, Ω, Θ, o og ω.

! [A_5] Kunne definere *best-case*, *average-case* og *worst-case*

! [A_6] Forstå *løkkeinvarianter* og *induksjon*

! [A_7] Forstå *rekursiv dekomponering* og *induksjon over delproblemer*

[A_8] Forstå Insertion-Sort

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 2

Datastrukturer

> For å unngå grunnleggende kjøretidsfeller er det viktig å kunne organisere og strukturere data fornuftig. Her skal vi se på hvordan enkle strukturer kan implementeres i praksis, og hva vi vinner på å bruke dem i algoritmene våre.

Pensum:

☐ Kap. 10. Elementary data structures: Innledning og 10.1–10.3

☐ Kap. 11. Hash tables: s. 253–264

☐ Kap. 17. Amortized analysis: Innledning og s. 463–465 (tom. «at most 3»)

Læringsmål:

[B_1] Forstå hvordan *stakker* og *køer* fungerer
(Stack-Empty, Push, Pop, Enqueue, Dequeue)

[B_2] Forstå hvordan *lenkede lister* fungerer
(List-Search, List-Insert, List-Delete, List-Delete′, List-Search′, List-Insert′)

[B_3] Forstå hvordan *pekere* og *objekter* kan implementeres

! [B_4] Forstå hvordan *direkte adressering* og *hashtabeller* fungerer
(Hash-Insert, Hash-Search)

[B_5] Forstå *konfliktløsing ved kjeding* (*chaining*)
(Chained-Hash-Insert, Chained-Hash-Search, Chained-Hash-Delete)

[B_6] Kjenne til grunnleggende *hashfunksjoner*

[B_7] Vite at man for *statiske datasett* kan ha *worst-case* O(1) for søk

[B_8] Kunne definere *amortisert analyse*

[B_9] Forstå hvordan *dynamiske tabeller* fungerer
(Table-Insert)

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 3

Splitt og hersk

> Rekursiv dekomponering er kanskje den viktigste ideen i hele faget, og designmetoden *splitt og hersk* er en grunnleggende utgave av det: Del instansen i mindre biter, løs problemet rekursivt for disse, og kombinér løsningene.

Pensum:

☐ Kap. 2. Getting started: 2.3

☐ Kap. 4. Divide-and-conquer: Innledning, 4.1 og 4.3–4.5

☐ Kap. 7. Quicksort

☐ Oppgaver 2.3-5 og 4.5-3 med løsning (binærsøk)

☐ Appendiks B og C i dette heftet

Læringsmål:

! [C_1] Forstå designmetoden *divide-and-conquer* (*splitt og hersk*)

[C_2] Forstå *maximum-subarray*-problemet med løsninger

[C_3] Forstå BISECT og BISECT′ (se appendiks C i dette heftet)

[C_4] Forstå MERGE-SORT

[C_5] Forstå QUICKSORT og RANDOMIZED-QUICKSORT

! [C_6] Kunne løse rekurrenser med *substitusjon*, *rekursjonstrær* og *masterteoremet*

! [C_7] Kunne løse rekurrenser med *iterasjonsmetoden* (se appendiks B i dette heftet)

[C_8] Forstå hvordan *variabelskifte* fungerer

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 4

Rangering i lineær tid

> Vi kan ofte få bedre løsninger ved å styrke kravene til input eller ved å svekke kravene til output. Sortering basert på sammenligninger ($x \leqslant y$) er et klassisk eksempel: I verste tilfelle må vi bruke $\lg n!$ sammenligninger,men om vi *antar mer* om elementene eller *bare sorterer noen* av dem så kan vi gjøre det bedre.

Pensum:

☐ Kap. 8. Sorting in linear time

☐ Kap. 9. Medians and order statistics

Læringsmål:

! [D_1] Forstå hvorfor *sammenligningsbasert sortering* har en *worst-case* på $\Omega(n \lg n)$

[D_2] Vite hva en *stabil sorteringsalgoritme* er

[D_3] Forstå Counting-Sort, og hvorfor den er stabil

! [D_4] Forstå Radix-Sort, og hvorfor den trenger en stabil subrutine

[D_5] Forstå Bucket-Sort

[D_6] Forstå Randomized-Select

[D_7] Kjenne til Select

Merk: Det kreves ikke grundig forståelse av virkemåten til Select.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 5

Rotfaste trestrukturer

> Rotfaste trær gjenspeiler rekursiv dekomponering. I binære søketrær er alt i venstre deltre mindre enn rota, mens alt i høyre deltre er større, og det gjelder rekursivt for alle deltrær! Hauger er enklere: Alt er mindre enn rota. Det begrenser funksjonaliteten, men gjør dem billigere å bygge og balansere.

Pensum:

☐ Kap. 6. Heapsort

☐ Kap. 10. Elementary data structures: 10.4

☐ Kap. 12. Binary search trees: Innledning og 12.1–12.3

Læringsmål:

! [E_1] Forstå hvordan *heaps* fungerer, og hvordan de kan brukes som *prioritetskøer*

(Parent, Left, Right, Max-Heapify, Build-Max-Heap, Heapsort, Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, Heap-Maximum. Også tilsvarende for *min-heaps*, f.eks., Build-Min-Heap og Heap-Extract-Min.)

[E_2] Forstå Heapsort

[E_3] Forstå hvordan *rotfaste trær* kan implementeres

! [E_4] Forstå hvordan *binære søketrær* fungerer

(Inorder-Tree-Walk, Tree-Search, Iterative-Tree-Search, Tree-Minimum, Tree-Maximum, Tree-Successor, Tree-Predecessor, Tree-Insert, Transplant, Tree-Delete)

[E_5] Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$

[E_6] Vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

Merk: Det kreves ikke grundig forståelse av Transplant og Tree-Delete.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 6

Dynamisk programmering

> På sett og vis en generalisering av *splitt og hersk*, der delprobler kan overlappe: I stedet for et *tre* av delproblem-avhengigheter har vi en *rettet asyklisk graf*. Vi finner og lagrer del-løsninger i en rekkefølge som stemmer med avhengighetene.

Pensum:

☐ Kap. 15. Dynamic programming: Innledning og 15.1, 15.3–15.4

☐ Oppgave 16.2-2 med løsning (0-1 knapsack)

☐ Appendiks D i dette heftet

Delkapitler 15.2 og 15.5 kan være nyttige som støttelitteratur.

Læringsmål:

! [F_1] Forstå ideen om en *delproblemgraf*

! [F_2] Forstå designmetoden *dynamisk programmering*

! [F_3] Forstå løsning ved *memoisering* (*top-down*)

[F_4] Forstå løsning ved *iterasjon* (*bottom-up*)

[F_5] Forstå hvordan man *rekonstruerer* en løsning fra lagrede beslutninger

[F_6] Forstå hva *optimal delstruktur* er

[F_7] Forstå hva *overlappende delproblemer* er

[F_8] Forstå eksemplene *stavkutting* og *LCS*

[F_9] Forstå løsningen på *0-1-ryggsekkproblemet* (se appendiks D i dette heftet) (Knapsack, Knapsack′)

Merk: Noe av stoffet vil kanskje forskyves til forelesning 7.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 7

Grådige algoritmer

> Grådige algoritmer består av en serie med valg, og hvert valg tas *lokalt*: Algoritmen gjør alltid det som ser best ut her og nå, uten noe større perspektiv. Slike algoritmer er ofte enkle; utfordringen ligger i å finne ut om de gir rett svar.

Pensum:

☐ Kap. 16. Greedy algorithms: Innledning og 16.1–16.3

Læringsmål:

! [G_1] Forstå designmetoden *grådighet*

! [G_2] Forstå *grådighetsegenskapen* (*the greedy-choice property*)

[G_3] Forstå eksemplene *aktivitet-utvelgelse* og *det fraksjonelle ryggsekkproblemet*

[G_4] Forstå HUFFMAN og *Huffman-koder*

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 8

Traversering av grafer

> Vi traverserer en graf ved å besøke noder vi vet om. Vi vet i utgangspunktet bare om startnoden, men oppdager naboene til dem vi besøker. Traversering er viktig i seg selv, men danner også ryggraden til flere mer avanserte algoritmer.

Pensum:

☐ Kap. 22. Elementary graph algorithms: Innledning og 22.1–22.4

☐ Appendiks E i dette heftet

Læringsmål:

[H_1] Forstå hvordan grafer kan implementeres

[H_2] Forstå BFS, også for å finne *korteste vei uten vekter*

[H_3] Forstå DFS og *parentesteoremet*

[H_4] Forstå hvordan DFS *klassifiserer kanter*

[H_5] Forstå Topological-Sort

[H_6] Forstå hvordan DFS kan *implementeres med en stakk*

[H_7] Forstå hva *traverseringstrær* (som *bredde-først-* og *dybde-først*-trær) er

! [H_8] Forstå *traversering med vilkårlig prioritetskø*

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 9

Minimale spenntrær

> Her har vi en graf med vekter på kantene, og ønsker å bare beholde akkurat de kantene vi må for å koble sammen alle nodene, med en så lav vektsum som mulig. Erke-eksempel på grådighet: Velg én og én kant, alltid den billigste lovlige.

Pensum:

☐ Kap. 21. Data structures for disjoint sets: Innledning, 21.1 og 21.3

☐ Kap. 23. Minimum spanning trees

Læringsmål:

[I_1] Forstå *skog*-implementasjonen av *disjunkte mengder*
(Connected-Components, Same-Component, Make-Set, Union, Link, Find-Set)

[I_2] Vite hva *spenntrær* og *minimale spenntrær* er

! [I_3] Forstå Generic-MST

[I_4] Forstå hvorfor *lette kanter* er *trygge kanter*

[I_5] Forstå MST-Kruskal

[I_6] Forstå MST-Prim

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 10

Korteste vei fra én til alle

> Bredde-først-søk kan finne stier med færrest mulig kanter, men hva om kantene har ulik lengde? Det generelle problemet er uløst, men vi kan løse problemet med gradvis bedre kjøretid for grafer (1) uten negative sykler; (2) uten negative kanter; og (3) uten sykler. Og vi bruker samme prinsipp for alle tre!

Pensum:

☐ Kap. 24. Single-source shortest paths: Innledning og 24.1–24.3

Læringsmål:

[J_1] Forstå ulike varianter av *korteste-vei-* eller *korteste-sti*-problemet (*Single-source*, *single-destination*, *single-pair*, *all-pairs*)

[J_2] Forstå strukturen til *korteste-vei*-problemet

[J_3] Forstå at negative sykler gir mening for korteste *enkle vei* (*simple path*)

[J_4] Forstå at *korteste enkle vei* kan løses vha. *lengste enkle vei* og omvendt

[J_5] Forstå hvordan man kan representere et *korteste-vei-tre*

! [J_6] Forstå *kant-slakking* (*edge relaxation*) og RELAX

[J_7] Forstå ulike egenskaper ved korteste veier og slakking (*Triangle inequality*, *upper-bound property*, *no-path property*, *convergence property*, *path-relaxation property*, *predecessor-subgraph property*)

[J_8] Forstå BELLMAN-FORD

[J_9] Forstå DAG-SHORTEST-PATH

! [J_{10}] Forstå kobling mellom DAG-SHORTEST-PATH og dynamisk programmering

[J_{11}] Forstå DIJKSTRA

Merk: Noe av stoffet vil kanskje forskyves til forelesning 11.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 11

Korteste vei fra alle til alle

> Vi kan finne de korteste veiene fra hver node etter tur, men mange av delproblemene vil overlappe – om vi har mange nok kanter lønner det seg å bruke dynamisk programmering med dekomponeringen «Skal vi innom k eller ikke?»

Pensum:

☐ Kap. 25. All-pairs shortest paths: Innledning, 25.2 og 25.3

Læringsmål:

[K_1] Forstå *forgjengerstrukturen* for *alle-til-alle*-varianten av korteste vei-problemet (Print-All-Pairs-Shortest-Path)

[K_2] Forstå Floyd-Warshall

[K_3] Forstå Transitive-Closure

[K_4] Forstå Johnson

Merk: Noe stoff fra forelesning 12 vil kanskje dekkes her.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 12

Maksimal flyt

> Et stort skritt i retning av generell lineær optimering (såkalt lineær programmering). Her ser vi på to tilsynelatende forskjellige problemer, som viser seg å være *duale* av hverandre, noe som hjelper oss med å finne en løsning.

Pensum:

☐ Kap. 26. Maximum flow: Innledning og 26.1–26.3

Læringsmål:

[L_1] Kunne definere *flytnett*, *flyt* og *maks-flyt-problemet*

[L_2] Kunne håndtere *antiparallelle kanter* og *flere kilder og sluk*

! [L_3] Kunne definere *restnettet* til et flytnett med en gitt flyt

[L_4] Forstå hvordan man kan *oppheve* (*cancel*) flyt

[L_5] Forstå hva en *forøkende sti* (*augmenting path*) er

[L_6] Forstå hva *snitt*, *snitt-kapasitet* og *minimalt snitt* er

! [L_7] Forstå *maks-flyt/min-snitt-teoremet*

[L_8] Forstå FORD-FULKERSON-METHOD og FORD-FULKERSON

[L_9] Vite at FORD-FULKERSON med BFS kalles *Edmonds-Karp*-algoritmen

[L_{10}] Forstå hvordan maks-flyt kan finne en *maksimum bipartitt matching*

! [L_{11}] Forstå *heltallsteoremet* (*integrality theorem*)

Merk: Noe av stoffet vil kanskje dekkes i forelesning 11.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Forelesning 13

NP-kompletthet

> NP er den enorme klassen av *ja-nei*-problemer der ethvert ja-svar har et bevis som kan sjekkes i polynomisk tid. Alle problemer i NP kan i polynomisk tid reduseres til de såkalt *komplette* problemene i NP. Dermed kan ikke disse løses i polynomisk tid, med mindre *alt* i NP kan det. Ingen har klart det så langt ...

Pensum:

☐ Kap. 34. NP-completeness

☐ Oppgave 34.1-4 med løsning (0-1 knapsack)

Se også appendiks D i dette heftet.

Læringsmål:

[M_1] Forstå sammenhengen mellom *optimerings-* og *beslutnings-problemer*

[M_2] Forstå *koding* (*encoding*) av en instans

[M_3] Forstå hvorfor løsningen vår på 0-1-ryggsekkproblemet *ikke er polynomisk*

[M_4] Forstå forskjellen på *konkrete* og *abstrakte* problemer

[M_5] Forstå representasjonen av beslutningsproblemer som *formelle språk*

[M_6] Forstå definisjonen av klassene P, NP og co-NP

[M_7] Forstå *redusibilitets-relasjonen* $\leqslant_P$

! [M_8] Forstå definisjonen av NP-*hardhet* og NP-*kompletthet*

[M_9] Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC

! [M_{10}] Forstå hvordan NP-kompletthet kan bevises ved én reduksjon

! [M_{11}] Kjenne de NP-komplette problemene CIRCUIT-SAT, SAT, 3-CNF-SAT, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP og SUBSET-SUM

[M_{12}] Forstå at *0-1-ryggsekkproblemet* er NP-hardt

[M_{13}] Forstå at *lengste enkle-vei*-problemet er NP-hardt

[M_{14}] Være i stand til å konstruere enkle NP-kompletthetsbevis

Merk: Det kreves ikke grundig forståelse av de ulike NP-kompletthetsbevisene.

Husk også læringsmål Z_1–Z_{10} på side iv, som gjelder hver forelesning.

Appendiks A

Noen gjengangere

Enkelte tema dukker opp flere ganger i faget, uten at de nødvendigvis er knyttet til én bestemt forelesning eller ett bestemt pensumkapittel. Her beskriver jeg noen slike tema som det kan være greit å få litt i fingrene fra tidlig av, siden det kan gjøre det lettere å forstå en god del andre ting i faget.

Håndtrykksformelen

Dette er én av to enkle men viktige formler som stadig dukker opp når man skal analysere algoritmer. Formelen er:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Den kalles gjerne *håndtrykksformelen*, fordi den beskriver antall håndtrykk som utføres om n personer skal hilse på hverandre. Vi kan vise dette ved å telle antall håndtrykk på to ulike måter. De to resultatene må da være like.

Telling 1: La oss se på personene én etter én. Første person hilser på alle de andre $n-1$ personene. Andre person har allerede hilst på første person, men bidrar med $n-2$ nye håndtrykk ved å hilse på de gjenværende. Generelt vil person i bidra med $n-i$ nye håndtrykk, så totalen blir

$$(n-1) + (n-2) + \cdots + 2 + 1 + 0\,.$$

Om vi snur summen, så vi får $0 + 1 + \cdots + (n-1)$, så er dette den venstre siden av ligningen. Dette oppstår typisk i algoritmer de vi utfører en løkke gjentatte ganger, og antall iterasjoner i løkken øker eller synker med 1 for hver gang, slik:

```
1  for i = 1 to n − 1
2      for j = i + 1 to n
3          i tar j i hånden
```

Telling 2: Vi kan også telle på en mer rett frem måte: Hver person tar alle de andre i hånden, og inngår dermed i $n-1$ håndtrykk. Hvis vi bare teller hver enkelt person sin halvdel av håndtrykket, får vi altså $n(n-1)$ *halve* håndtrykk. To slike halve håndtrykk utgjør jo ett helt, så det totale antallet håndtrykk blir den høyre siden av ligningen, nemlig $n(n-1)/2$.

Man kan også gjøre om en sum av denne typen ved å brette den på midten, og legge sammen første og siste element, nest første og nest siste, etc. Vi får da

$$(n-1+0)+(n-2+1)+(n-3+2)+\dots$$

Hvert ledd summerer til $n-1$, og det er $n/2$ ledd. Mer generelt er summen av en aritmetisk rekke (der vi øker med en konstant fra ledd til ledd) lik gjennomsnittet av første og siste ledd, multiplisert med antallet ledd i rekken.

Utslagsturneringer

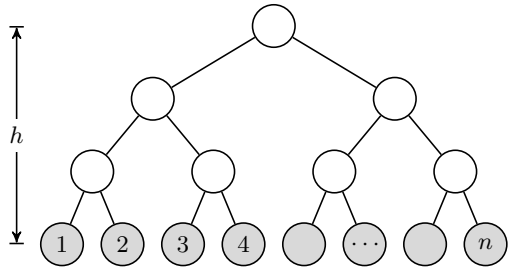
Dette er den *andre* av de to sentrale formlene:

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Det vil si, de første toerpotensene summerer til én mindre enn den neste. En måte å se dette på er av totallssystemet, der et tall $a = 11\cdots1$ med h ettall etterfølges av tallet $b = 100\cdots0$, som består av ett ettall, og h nuller. Her er $a = 2^0 + 2^1 + \cdots + 2^{h-1}$ og $b = 2^h$, så $a = b - 1$.

Et grundigere bevis kan vi få ved å bruke samme teknikk som før, og telle samme ting på to ulike måter. Det vi vil telle er antall matcher i en utslagsturnering (*knockout*-turnering), det vil si, en turnering der taperen i en match er ute av spillet. Dette blir altså annerledes enn såkalte *round robin*-turneringer, der alle møter alle – for dem kan vi bruke håndtrykksformelen for å finne antall matcher, siden hver match tilsvarer ett håndtrykk.

Telling 1: Vi begynner med å sette opp et *turneringstre*:



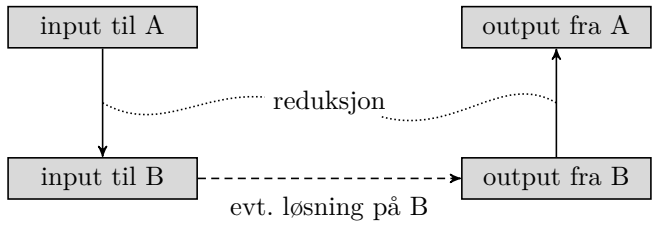
Her er deltakerne plassert nederst, i løvnodene; hver av de tomme, hvite nodene representerer en match, og vil etterhvert fylles med vinneren av de to i nodene under, helt til vi står igjen med én vinner på toppen. Vi går altså gjennom h runder, og i hver runde organiseres de spillerne som gjenstår i par som skal møtes. Om vi nummererer rundene baklengs, etter hvor mange runder det er igjen til finalen, så vil antall matcher i runde i være 2^i. Det totale antall matcher er altså $2^0 + 2^1 + \cdots + 2^{h-1}$, som er venstresiden av ligningen vår.

Telling 2: Antall deltakere er $n = 2^h$. I hver match blir én deltaker slått ut, helt til vi sitter igjen med én vinner etter finalen. Antall matcher trenger vi for å slå ut alle bortsett fra én er $n - 1$, eller $2^h - 1$, som er høyresiden i ligningen.

Det er forøvrig også viktig å merke seg *høyden til treet*: $h = \log_2 n$. Dette er altså antall doblinger fra 1 til n, eller antall halveringer fra n til 1. Det er også en sammenheng som vil dukke opp ofte.

Reduksjoner

Ofte jobber vi med å bryte ned problemer i sine enkelte bestanddeler, og deretter bygge opp en løsning fra bunnen av, trinn for trinn. Men av og til har vi kanskje allerede en løsning på et problem som *ligner* på det vi prøver å løse. Eller kanskje vi vet at et lignende problem er spesielt *vanskelig* å løse – at ingen har klart det ennå? I slike sammenhenger kan vi benytte oss av det som kalles en *reduksjonsalgoritme*, eller *reduksjon*. En reduksjon *transformerer input* fra ett problem til et annet, slik at vi kan konsentrere oss om å løse problemet vi har redusert til. Transformasjonen må være slik at om vi løser det nye problemet, så vil løsningen også være riktig for det opprinnelige problemet (gitt opprinnelig input). Av og til tillater vi oss å transformere svaret tilbake, og regner denne transformasjonen også som en del av reduksjonen.



Ut fra dette kan vi trekke to logisk ekvivalente konklusjoner:

(*i*) Hvis vi kan løse B, så kan vi løse A

(*ii*) Hvis vi *ikke* kan løse A, så kan vi *ikke* løse B

Det første utsagnet ser vi av figuren over. Straks vi plugger inn en løsning for B, så kan den kombineres med reduksjonen for å lage en løsning for A. Det andre utsagnet følger ved kontraposisjon. Om vi lar LA og LB være de logiske utsagnene at *vi har en løsning* for hhv. A og B, så kan vi skrive om konklusjonene:

$$\text{LB} \;\Rightarrow\; \text{LA} \qquad \text{og} \qquad \neg\,\text{LA} \;\Rightarrow\; \neg\,\text{LB}$$

Vi kan også slenge på et par betraktningene:

(*iii*) Hvis vi *ikke* kan løse B, så *sier det ingenting* om A

(*iv*) Hvis vi *kan* løse A, så *sier det ingenting* om B

Selv om vi *ikke* kan løse B, så kan det jo hende at vi kan løse A på en *annen* måte. Og om vi *kan* løse A, så er det jo ikke sikkert at vi gjorde det ved å redusere til B. Disse tilfellene gir oss altså ingen informasjon. Med andre ord: Om vi allerede er kjent med et problem X og så støter på et nytt og ukjent problem Y, så har vi to scenarier der vi kan gjøre noe fornuftig. Hva vi gjør avhenger av om vi lar Y få rollen som A eller B i figuren over. Hvis vi vi vise at Y *ikke er vanskeligere enn* X, så kan vi la Y innta rollen som A, og prøve å finne en reduksjon fra Y til X. Det er dette vi ofte gjør når vi prøver å bruke eksisterende algoritmer for et problem X å løse et nytt problem Y. Vi reduserer Y til X, og løser så X.

Men av og til mistenker vi at et problem vi støter på er vanskelig. Kanskje vi kjenner til et problem X som vi *vet* er vanskelig, og vi vil vise at Y er *minst like vanskelig*. Da må vi i stedet la Y innta rollen som B, og redusere *fra* det vanskelige problemet. Vi skriver $A \leqslant B$ for å uttrykke at problemet A er *redusibelt* til B, det vil si, at det finnes en reduksjon fra A til B, så A kan *løses ved hjelp av* B. Det betyr altså at A ikke er vanskeligere enn B, siden vi jo kan redusere til B; og ekvivalent, at B er minst like vanskelig som A.

Vanligvis ønsker vi å være noe mer nyanserte enn at det finnes eller ikke finnes en løsning på et problem. Vi kan for eksempel lure på hvor effektive algoritmer vi kan finne for problemet. Da er det viktig at også *reduksjonen* vår er effektiv. For dersom reduksjonen bruker ubegrenset lang tid, så vil jo ikke en effektiv løsning på B fortelle oss noe om hvor effektivt vi kan løse A.

Appendiks B

Iterasjonsmetoden

I læreboka er *rekursjonstrær* en sentral metode for å løse rekurrensligninger. Dette er en mer visuell fremgangsmåte enn den som ble brukt i tidligere utgaver, som de kalte *iterasjonsmetoden*. Det er egentlig akkurat samme prinsipp, bare at man jobber mer direkte på ligningene. Ideen er at man bruker rekurrensen selv til å ekspandere de rekursive termene. For eksempel:

$$\begin{aligned} \mathrm{T}(0) &= 0 \\ \mathrm{T}(n) &= \mathrm{T}(n-1) + 1 \qquad (n \geqslant 1) \end{aligned}$$

Her ønsker vi å bli kvitt $\mathrm{T}(n-1)$ på høyre side i ligningen. Om vi antar at $n > 1$, så kan vi bruke definisjonen $\mathrm{T}(n) = \mathrm{T}(n-1) + 1$ til å finne ut hva $\mathrm{T}(n-1)$ er, nemlig $(\mathrm{T}((n-1)-1)+1)+1$ eller $\mathrm{T}(n-2)+2$. Det er altså én *ekspansjon* (eller iterasjon). Vi kan fortsette på samme måte, trinn for trinn, helt til vi ser et mønster dukke opp:

$$\begin{aligned} \mathrm{T}(0) &= 0 \\ \mathrm{T}(n) &= \mathrm{T}(n-1) + 1 && (1) \\ &= \mathrm{T}(n-2) + 2 && (2) \\ &= \mathrm{T}(n-3) + 3 && (3) \\ &\;\;\vdots && \vdots \\ &= \mathrm{T}(n-i) \; + i && (i) \\ &= \mathrm{T}(n-n) + n = n && (n) \end{aligned}$$

Først ekspanderer vi altså $\mathrm{T}(n)$, så $\mathrm{T}(n-1)$ og så $\mathrm{T}(n-2)$, og så videre, hele tiden ved hjelp av den opprinnelige rekurrensen. Linjenummeret (til høyre) viser hvor mange ekspansjoner vi har gjort. Etter hvert klarer vi altså å uttrykke resultatet etter et vilkårlig antall ekspansjoner, i. Her kan i være 1, 2, 3, eller et hvilket som helst annet tall, så lenge vi ikke ekspanderer oss «forbi» grunntilfellet $\mathrm{T}(0) = 0$. Og det er nettopp grunntilfellet vi ønsker å nå; vi vil gi T argumentet 0. For å få til det må vi altså få $n - i$ til å bli 0, og altså sette $i = n$. Dette gir oss svaret, som vist i siste linje. Et litt mer interessant eksempel finner du i appendiks C, der hver ekspansjon *halverer* argumentet til T, så trenger bare $\lg n$ ekspansjoner før vi kommer ned til $\mathrm{T}(1)$, som er grunntilfellet der.

Appendiks C

Binærsøk

Læreboka sin beskrivelse av binærsøk, på engelsk *binary search* eller *bisection*, begrenser seg til et par oppgaver (2.3-5 og 4.5-3, med problemet definert i 2.1-3). Algoritmen er gjennomgått i mer detalj i forelesningene, men den sentrale informasjonen er oppsummert her. Selve algoritmen er definert som følger, der tabellen $\mathrm{A}[1\,..\,n]$ antas å være sortert.

```
BISECT(A, p, r, v)
1  if p ⩽ r
2       q = ⌊(p + r)/2⌋
3       if v == A[q]
4            return q
5       else if v < A[q]
6            return BISECT(A, p, q − 1, v)
7       else return BISECT(A, q + 1, r, v)
8  else return NIL
```

Dersom A inneholder v, så vil $\textsc{Bisect}(\mathrm{A}, 1, n, v)$ returnere en indeks i som er slik at $\mathrm{A}[i] == v$. Hvis ikke, så returneres NIL.

Korrekthet kan vises ved induksjon over lengden til segmentet $\mathrm{A}[p\,..\,r]$. Vi har to grunntilfeller: Enten er $p > r$, så segmentet er tomt, og vi returnerer NIL, ellers så har vi $p = r$ og $v = \mathrm{A}[q]$, og vi returnerer q. Begge disse er korrekte. Anta så at $p \leqslant r$, og at BISECT gir rett svar for kortere intervaller. Både $\mathrm{A}[p\,..\,q-1]$ og $\mathrm{A}[q+1\,..\,r]$ er kortere intervaller, så samme hvilket rekursivt kall som velges, så vil vi returnere rett indeks.

For å finne kjøretiden kan vi for eksempel telle antall ganger sammenligningen $p \leqslant r$ utføres. Vi kan gjøre noen antagelser for å forenkle utregningen. Disse har ingenting å si for den asymptotiske kjøretiden. For det første antar vi at $n = 2^k$, for et eller annet heltall $k \geqslant 0$, slik at vi alltid kan dele nøyaktig på midten, helt til vi ender med $p == q$. For det andre så antar vi at v finnes i A så rekursjonen stanser her. For å sikre at vi deler på midten, kan vi til og med anta at den eneste forekomsten av v er $\mathrm{A}[n]$, så vi alltid velger det høyre segmentet, $\mathrm{A}[q+1\,..\,r]$. (Vi kan bruke induksjon, dvs., *substitusjonsmetoden*, til å verifisere at den asymptotiske kjøretiden vi finner er gyldig også om disse antagelsene *ikke* gjelder.) Vi får da

følgende rekurrensutregning.

$$
\begin{aligned}
\mathrm{T}(1) &= 1 \\
\mathrm{T}(n) &= \mathrm{T}(n/2) \ + 1 && (1) \\
&= \mathrm{T}(n/4) \ + 2 && (2) \\
&= \mathrm{T}(n/8) \ + 3 && (3) \\
&\vdots && \vdots \\
&= \mathrm{T}(n/2^i) + i && (i) \\
&= \mathrm{T}(n/n) \ + \lg n = \lg n + 1 && (\lg n)
\end{aligned}
$$

Med andre ord er kjøretiden $\Theta(\lg n)$.

Som så ofte ellers, så hjelper det å tenke rekursivt for å forstå hvordan algoritmen fungerer, men det er, som alltid, mulig å kvitte seg med rekursjonen og få en iterativ versjon:

```
BISECT′(A, p, r, v)
1  while p ⩽ r
2        q = ⌊(p + r)/2⌋
3        if v == A[q]
4              return q
5        else if v < A[q]
6              r = q − 1
7        else p = q + 1
8  return NIL
```

Denne versjonen vil generelt være mer effektiv, siden man slipper ekstra kostnader ved funksjonskall.* Både korrekthetsbevis og kjøretidsberegning kan oversettes ganske direkte fra BISECT til BISECT′.

* Siden returverdien fra det rekursive kallet returneres direkte, uten noe mer kode imellom – et såkalt *tail call* – så vil enkelte språk kunne gjøre denne transformasjonen automatisk, ved hjelp av såkalt *tail-call elimination* eller *tail-call optimization*.

Appendiks D

Ryggsekkproblemet

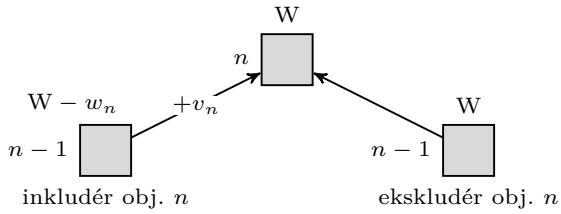
Det såkalte *ryggsekkproblemet* kommer i flere varianter, og to av dem er beskrevet på side 425–426 i læreboka. Den *fraksjonelle* varianten er lett å løse: Man tar bare med seg så mye som mulig av den dyreste gjenstanden, og fortsetter nedover på lista, sortert etter kilopris. I 0-1-varianten, derimot, blir ting litt vanskeligere—her må man ta med en hel gjenstand eller la den ligge. Løsningen er egentlig beskrevet på 426, men den er litt bortgjemt i teksten, og er beskrevet svært skissepreget.

Akkurat som i for eksempel FLOYD-WARSHALL så baserer dekomponeringen seg på et *ja-nei*-spørsmål, i dette tilfelle «Skal vi ta med gjenstand i?» For hver av de to mulighetene sitter vi igjen med et delproblem som vi løser rekursivt (i hvert fall konseptuelt). Som vanlig tenker vi oss at dette er siste trinn, og antar at vi har gjenstander $1, \ldots, i$ tilgjengelige. Vi har da følgende alternativer:

(i) Ja, vi tar med gjenstand i. Vi løser så problemet for gjenstander $1, \ldots, i-1$ men der kapasiteten er redusert med w_i. Vi legger så til v_i til slutt.

(ii) Nei, vi tar ikke med gjenstand i. Vi løser så problemet for gjenstander $1, \ldots, i-1$, men kan fortsatt bruke hele kapasiteten. Til gjengjeld får vi ikke legge til v_i til slutt.

Situasjonen er illustrert i Fig. D.1, der hver rute representerer en delløsning (en celle i løsningstabellen, for eksempel) og pilene er avhengigheter, som vanlig. Vi kan sette opp en rekursiv løsning slik (der vi antar at vektene og verdiene er globale variable, for enkelhets skyld):



Figur D.1: Dekomponering for *0-1-knapsack*.

```
KNAPSACK(n, W)
1  if n == 0
2       return 0
3  x = KNAPSACK(n − 1, W)
4  if w_n > W
5       return x
6  else y = KNAPSACK(n − 1, W − w_n) + v_n
7       return max(x, y)
```

Denne prosedyren vil naturligvis ha eksponentiell kjøretid. Vi kan løse det ved å bruke memoisering, eller vi kan skrive den om til en iterativ *bottom-up*-løsning, som nedenfor.

```
KNAPSACK'(n, W)
 1  let K[0..n, 0..W] be a new table
 2  for j = 0 to W
 3       K[0, j] = 0
 4  for i = 1 to n
 5       for j = 0 to W
 6            x = K[i − 1, j]
 7            if j < w_i
 8                 K[i, j] = x
 9            else y = K[i − 1, j − w_i] + v_i
10                 K[i, j] = max(x, y)
```

Forskjellen ligger i at vi her eksplisitt legger inn **for**-løkker for å gå gjennom alle delproblemene, heller enn å gjøre det direkte (implisitt) med rekursjon.

Men dette er ikke polynomisk!

Det binære ryggsekkproblemet (*0-1-knapsack*) er et såkalt NP-hardt problem, og det er ingen som har funnet noen polynomisk løsning på det; trolig vil ingen noen gang finne det heller. Det ser jo ut til å motsi diskusjonen over, der vi kom frem til det som absolutt *ser ut som* en algoritme med polynomisk kjøretid. Hva er det som foregår her?

Kjøretiden til både KNAPSACK og KNAPSACK′ er $\Theta(n\mathrm{W})$, siden det er $n\mathrm{W}$ delproblemer, og vi utfører en konstant mengde arbeid per delproblem. Og det er rett at dette er en polynomisk funksjon av n og W, men det er ikke nok til at vi kan si at algoritmene «har polynomisk kjøretid». Spørsmålet er *som funksjon av hva*? Hvis vi eksplisitt sier «som funksjon av n og W» er alt i orden. Men om vi ikke oppgir noen eksplisitte parametre, så antas kjøretiden å være en funksjon *av input-størrelsen*, som altså er hvor stor plass en instans tar. Akkurat hvordan vi måler denne størrelsen kan avhenge av problemet (se side 25 i læreboka), men når vi regner på om ting kan løses i polynomisk tid (i forbindelse med NP-kompletthet og denslags) holder vi oss til *antall bits* i input, i en rimelig encoding. Størrelsen blir da $\Theta(n + \lg \mathrm{W})$, siden vi bare trenger $\Theta(\lg \mathrm{W})$ bits for å lagre parameteren W. Poenget er altså at W vokser eksponentielt som funksjon av $\lg \mathrm{W}$, og kjøretiden er, teknisk sett, eksponentiell.

Dette er kanskje enda enklere å se hvis vi lar m være antall bits i W, så vi kan skrive kjøretiden som

$$\mathrm{T}(n, m) = \Theta(n2^m)\,.$$

Da er det forhåpentligvis tydelig at dette ikke er en polynomisk kjøretid. Kjøretider som er polynomiske hvis vi lar et *tall fra input* være med som parameter til kjøretiden (slik som $\Theta(n\mathrm{W})$, der W er et tall fra input, og ikke direkte en del av problemstørrelsen) kaller vi *pseudopolynomiske*.

Appendiks E

Generell graftraversering

Læreboka beskriver to traverseringsalgoritmer, *bredde-først-søk* (22.2) og *dybde-først-søk* (22.3). Disse fremstilles ganske ulikt, men er nært beslektede. Prosedyren BFS (på side 595) kan tilpasses til å oppføre seg omtrent likt med DFS (side 605), ved å bytte ut FIFO-køen Q med en LIFO-kø, eller stakk (*stack*). Vi mister da tidsmerkingen ($v.d$ og $v.f$), men rekkefølgen noder farges grå og svarte på vil bli den samme. (En annen forskjell er at DFS, slik den er beskrevet i boka, ikke har noen startnode, men bare starter fra hver node etter tur, til den har nådd hele grafen; sånn sett er BFS mer beslektet med DFS-Visit.)

Køen Q i BFS er rett og slett en liste med noder vi har oppdaget via kanter fra tidligere besøkte noder, men som vi ennå ikke har besøkt. Noder er hvite før de legges inn, grå når de er i Q og svarte etterpå. Det at vi bruker en FIFO-kø er det som lar BFS finne de korteste stiene til alle noder, siden vi utforsker grafen «lagvis» utover, men vi kan egentlig velge vilkårlige noder fra Q i hver iterasjon, og vi vil likevel traversere hele den delen av grafen vi kan nå fra startnoden.

Grunnen til at en LIFO-kø gir oss samme atferd som en rekursiv traversering (altså DFS) er at vi egentlig bare simulerer hvordan rekursjon er implementert. Internt bruker maskina en *kallstakk*, der informasjon om hvert kall legges øverst, og hentes frem igjen når rekursive kall er ferdige.

Som Cormen mfl. påpeker, så bruker Prims og Dijkstras algoritmer svært lignende ideer. Prosedyrene MST-Prim (side 634) og Dijkstra (side 658) forenkler ting litt ved å sette Q = G.V før løkken som bruker Q, så vi mister den delen av traverseringen som handler om å *oppdage* noder. I eldre fremstillinger av disse algoritmene så ligner de mer på BFS, og noder legges inn i Q når de oppdages. Den sentrale forskjellen er hvilke noder som tas *ut* av Q, altså hvilke noder som *prioriteres*. I BFS prioriteres de eldste og i DFS prioriteres de nyeste; i Dijkstra prioriteres noder med lavt avstandsestimat ($v.d$), mens i Prim prioriteres noder som har en lett kant til én av de besøkte (svarte) nodene. I tillegg oppdateres disse prioritetene underveis.

Andre prioriteringer vil gi andre traverseringsalgoritmer, som for eksempel den såkalte A*-algoritmen (ikke pensum).

Appendiks F

Rekursjon og iterasjon

Iterative funksjoner kan oversettes til ekvivalente rekursive funksjoner, og omvendt. Dette er noe vi ofte gjør under konstruksjon av algoritmer basert på rekursiv dekomponering, kanskje spesielt innen dynamisk programmering. De to måtene å tenke på kan være anvendelige i ulike situasjoner; for eksempel kan det være praktisk å tenke rekursivt når man bygger løsninger fra del-løsninger, men iterative løsninger unngår ekstrakostnader til funksjonskall, som kan være viktig når en algoritme faktisk skal implementeres.

Fra iterasjon til rekursjon

Dette er en transisjon man ofte må gjøre om man skal lære seg såkalt funksjonell programmering. Å leke seg med funksjonelle språk kan være nyttig for å bli grundig vant til rekursjon.*

Det enkleste å oversette direkte er kanskje løkker av typen *repeat-while* (i mange språk kjent som *do-while*), der betingelsen kommer til slutt. For eksempel:

```
1 repeat
2       body
3 while condition
```

Dette kan man oversette til:

```
FUNC(vars)
1 body
2 if condition
3       FUNC(vars)
```

Ett kall tilsvarer her én iterasjon av løkken, og *vars* tilsvarer lokale variable som sendes med fra iterasjon til iterasjon. La oss si vi f.eks. vil skrive ut tallene $1 \ldots n$

```
1 i = 1
2 repeat
3       print i
4       i = i + 1
5 while i ⩽ n
```

* Ta for eksempel en titt på Haskell (`https://haskell.org`).

Vi kan da skrive om det til:

```
NUMBERS(i, n)
1  print i
2  i = i + 1
3  if i ⩽ n
4       NUMBERS(i, n)
```

Funksjonen kalles da med argumenter 1 og n til å begynne med.

Intuitivt kan dette tolkes på flere måter. Én veldig enkel tolkning er at kallet til NUMBERS forteller maskina at vi skal gå tilbake til start, og utføre definisjonen av NUMBERS, akkurat som i en løkke. Det blir som om vi hadde skrevet om løkken til det følgende (om vi antar at i alt er satt til 1):

```
1  print i
2  i = i + 1
3  if i ⩽ n
4       go to 1
```

Internt er det nettopp denne typen *go to*-utsagn kompilatoren oversetter løkker til. I mange språk er det også nettopp slik kode man får om man bruker rekursjon slik som vi har gjort over! Med andre ord: Både løkker og rekursjon kan ende med samme primitive maskinkode, som inneholder verken løkker eller rekursjon.

En annen måte å forstå koden på, intuitivt, er at vi skiller mellom første iterasjon og *resten*. Første iterasjon består av linje 1 og 2, og om vi ikke har kommet til slutten, så utfører vi resten av iterasjonene rekursivt.

Om man trenger verdier som beregnes i løkken, kan disse returneres fra funksjonen. For eksempel kan returverdiene være de samme variablene som før:

```
FUNC(vars)
1  body
2  if condition
3       vars = FUNC(vars)
4  return vars
```

Merk at nå kan ikke lenger kompilatoren uten videre oversette dette til et enkelt *go to*-utsagn (såkalt *tail-call optimization*) siden vi har kode etter det rekursive kallet.

La oss si at vi f.eks. vil *summere* tallene fra 1 til n, i stedet, med en løkke som:

```
1  s, i = 0, 1
2  repeat
3       s = s + i
4       i = i + 1
5  while i ⩽ n
```

Vi kan skrive om dette til:

```
SUM(i, n, s)
1  s = s + i
2  i = i + 1
3  if i ⩽ n
4        i, n, s = SUM(i, n, s)
5  return i, n, s
```

Kallet som starter det hele blir $\textsc{Sum}(1, n, 0)$. (Her er det naturligvis unødvendig å sende med *alle* variablene tilbake, siden vi kun trenger s, men det illustrerer at oversettelsen her kan gjøres ganske mekanisk.)

Å håndtere andre typer løkker er relativt rett frem. For eksempel kan du begynne med å skrive dem om til å være på *repeat-while*-formen (selv om du nok ofte vil kunne finne mer naturlige rekursive måter å skrive koden på).

Fra rekursjon til iterasjon

Som nevnt over: Hvis det rekursive kallet kommer helt til slutt i funksjonen, kan vi ofte oversette det direkte til en ekvivalent løkke. Dersom koden fortsetter etter et rekursivt kall, derimot, så må vi simulere rekursjonen på noe vis; vi må huske hvilke verdier de lokale variablene hadde *før* det rekursive kallet, og gi dem disse verdiene igjen!

Måten vi gjør dette på er med en stakk, akkurat slik som rekursjon er implementert internt i programmeringsspråkene (vha. en *kallstakk*). La oss si vi har en rekursiv funksjon som dette:

```
FUNC(...)
1  before
2  if condition
3        FUNC(...)
4  after
```

La oss si at koden både før og etter bruker de samme variablene, *vars*. Da kan vi fjerne rekursjonen slik, der S er en LIFO-stakk:

```
1  repeat
2        before
3        PUSH(S, vars)
4  while condition
5  while S is not empty
6        vars = POP(S)
7        after
```

Det rekursive kallet til FUNC mottar formodentlig andre argumenter enn dem vi får inn; det kan her bare håndteres med vanlige tilordninger.

Bibliografi

[1] T. H. Cormen mfl. *Introduction to Algorithms*. 3. utg. The MIT Press, 2009.

[2] B. A. Oakley. *A mind for numbers: How to excel at math and science (even if you flunked algebra)*. TarcherPerigee, 2014.