TDT4120

Algoritmer og datastrukturer

Forelesninger, pensum og læringsmål

MAGNUS LIE HETLAND

NTNU, Høsten 2016

Innledning

De overordnede læringsmålene for faget er som følger.*

Dere skal ha kunnskap om:

- Et bredt spekter av etablerte algoritmer og datastrukturer
- Klassiske algoritmiske problemer med kjente effektive løsninger
- Komplekse problemer uten kjente effektive løsninger

Dere skal kunne:

- Analysere algoritmers korrekthet og effektivitet
- Formulere problemer så de kan løses av algoritmer
- Konstruere nye effektive algoritmer

Dere skal være i stand til:

- Å bruke eksisterende algoritmer og programvare på nye problemer
- 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* (Cormen et al., 2009).

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

Forkunnskaper

En viss bakgrunn er nødvendig for å få fullt utbytte av faget, som beskrevet i fagets anbefalte forkunnskaper.* 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:

- Kjenne til begreper rundt *monotone funksjoner*
- Kjenne til notasjonene $\lceil x \rceil$, $\lfloor x \rfloor$, $n!$ og a mod n
- Vite hva *polynomer* er
- Kjenne grunnleggende potensregning
- Ha noe kunnskap om grenseverdier
- Være godt kjent med logaritmer med ulike grunntall
- Kjenne enkel sannsynlighetsregning, indikatorvariable og forventning
- Ha noe kjennskap til rekkesummer (se også s. 21)
- Beherske helt grunnleggende mengdelære
- Kjenne grunnleggende terminologi for relasjoner, ordninger og funksjoner
- Kjenne grunnleggende terminologi for og egenskaper ved grafer og trær
- Kjenne til enkel kombinatorikk, som permutasjoner og kombinasjoner

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

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

Det vil bli holdt en ekstraforelesning (som en del av en øvingstime) som gjennomgår en del av dette stoffet. Vi vil også 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 i øvingsopplegget.

Gjennom semesteret

Det følgende er pensum knyttet til flere forelesninger gjennom semesteret:

☐ Lysark fra ordinære forelesninger

☐ Appendiks til dette dokumentet

Læringsmål for hver algoritme:

- Kjenne den formelle definisjonen av det generelle problemet den løser
- Kjenne til eventuelle tilleggskrav den stiller for å være korrekt
- Vite hvordan den oppfører seg; kunne utføre algoritmen, trinn for trinn
- Forstå korrekthetsbeviset; hvordan og hvorfor virker algoritmen egentlig?
- Kjenne til eventuelle styrker eller svakheter, sammenlignet med andre
- Kjenne kjøretidene under ulike omstendigheter, og forstå utregningen

Læringsmål for hver datastruktur:

- Forstå algoritmene for de ulike operasjonene på strukturen
- Forstå hvordan strukturen representeres i minnet

Læringsmål for hvert problem:

- Kunne angi presist hva input er
- Kunne angi presist hva output er og hvilke egenskaper det må ha

Merk:*

(*i*) Appendiks B kan det være lurt å se på tidlig i semesteret.

(*ii*) Det vil bli holdt en ekstraforelesning om bevisteknikk (appendiks A).

(*iii*) Stoffet om induksjon (appendiks C) vil gjennomgås gradvis gjennom semesteret, og de tre seksjonene egner seg til forberedelse før eller repetisjon etter hhv. forelesning 1, 3 og 6. (Temaene er også oppgitt som læringsmål under disse forelesningene.)

(*iv*) Stoffet i appendiks A og C er ment å hjelpe på forståelsen av resten av pensum, og vil i mindre grad bli spurt om direkte på eksamen.

*Gjelder appendiks i dette dokumentet, ikke i læreboka.

Del I

Forelesningsplan

Ekstraforelesninger

Ut over de ordinære forelesningene som beskrives i det følgende, vil det holdes ekstraforelesninger om

(*i*) programmering i Python,

(*ii*) bevisteknikk (se appendiks A i dette dokumentet), og

(*iii*) matematisk bakgrunnsstoff (se beskrivelse av forkunnskaper på side ii).

Disse holdes som en del av de første tre øvingsforelesningene.

Forelesning 1

Problemer og algoritmer

☐ Kap. 1. The role of algorithms in computing

☐ Kap. 2. Getting started

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

Her vil jeg også snakke litt om bruk av programvare for lineærprogrammering.

Læringsmål:

- Kunne definere *problem*, *instans* og *problemstørrelse*
- Forstå *løkkeinvarianter* og *naturlig induksjon*
- Forstå bokas *pseudokode*-konvensjoner
- Kjenne egenskapene til *random-access machine*-modellen
- Forstå Insertion-Sort
- Kunne definere *best-case*, *average-case* og *worst-case*
- Forstå ideen bak *divide-or-conquer*
- Forstå Merge-Sort
- Kunne definere *asymptotisk notasjon*, O, Ω, Θ, o og ω.

Forelesning 2

Datastrukturer

☐ Kap. 10. Elementary data structures

☐ Kap. 11. Hash tables: s. 253–264

☐ Kap. 17. Amortized analysis: Innledning og s. 463–465

Læringsmål:

- Forstå hvordan *stakker* og *køer* fungerer
- Forstå hvordan *lenkede lister* fungerer
- Forstå hvordan *pekere* og *objekter* kan implementeres
- Forstå hvordan *rotfaste trær* kan implementeres
- Forstå hvordan *direkte adressering* og *hashtabeller* fungerer
- Forstå *konfliktløsing ved kjeding* (*chaining*)
- Kjenne til grunnleggende *hashfunksjoner*
- Vite at man for *statiske datasett* kan ha *worst-case* O(1) for søk
- Kunne definere *amortisert analyse*
- Forstå hvordan *dynamiske tabeller* fungerer

Forelesning 3

Splitt og hersk

☐ 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)

Læringsmål:

- Forstå *strukturell induksjon**
- Forstå designmetoden *divide-and-conquer* (*splitt og hersk*)
- Kunne løse rekurrenser med *substitusjon*, *rekursjonstrær* og *masterteoremet*
- Forstå hvordan *variabelskifte* fungerer
- Forstå QUICKSORT og RANDOMIZED-QUICKSORT
- Forstå *binærsøk*

*I appendiks og på lysark, ikke i læreboka.

Forelesning 4

Rangering i lineær tid

☐ Kap. 8. Sorting in linear time

☐ Kap. 9. Medians and order statistics

Læringsmål:

- Forstå hvorfor *sammenlikningsbasert sortering* har en *worst-case* på $\Omega(n \lg n)$
- Vite hva en *stabil sorteringsalgoritme* er
- Forstå COUNTING-SORT, og hvorfor den er stabil
- Forstå RADIX-SORT, og hvorfor den trenger en stabil subrutine
- Forstå BUCKET-SORT
- Forstå RANDOMIZED-SELECT
- Forstå SELECT

Forelesning 5

Rotfaste trestrukturer

☐ Kap. 6. Heapsort

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

Læringsmål:

- Forstå hvordan *heaps* fungerer, og hvordan de kan brukes som *prioritetskøer*
- Forstå HEAPSORT
- Forstå hvordan *binære søketrær* fungerer
- Forstå flere ulike operasjoner på binære søketrær, ut over bare søk
- Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$
- Vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

Forelesning 6

Dynamisk programmering

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

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

Læringsmål:

- Forstå ideen om en *delproblemrelasjon* eller *delproblemgraf*
- Forstå *induksjon over velfunderte relasjoner**
- Forstå designmetoden *dynamisk programmering*
- Forstå løsning ved *memoisering* (*top-down*)
- Forstå løsning ved *iterasjon* (*bottom-up*)
- Forstå hvordan man *rekonstruerer* en løsning fra lagrede beslutninger
- Forstå hva *optimal delstruktur* er
- Forstå hva *overlappende delproblemer* er
- Vite forskjellen på et *segment* og en *underfølge* (*subsequence*)
- Forstå eksemplene *stavkutting*, *matrisekjede-multiplikasjon* og *LCS*
- Forstå løsningen på *0-1-ryggsekkproblemet*

*I appendiks og på lysark, ikke i læreboka.

Forelesning 7

Grådige algoritmer

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

Her vil jeg også gi eksempler på ulike metoder for å vise at grådige algoritmer er korrekte, ut over dem som diskuteres i boka.

Læringsmål:

- Forstå designmetoden *grådighet*
- Forstå *grådighetsegenskapen* (*the greedy-choice property*)
- Forstå eksemplene *aktivitet-utvelgelse* og *det fraksjonelle ryggsekkproblemet*
- Forstå Huffman og *Huffman-koder*
- Forstå bevismetoden *bevis ved fortrinn* (*exchange arguments*)*
- Forstå bevismetoden *bevis ved forsprang* (*staying ahead*)*

*Kun på lysark, ikke i læreboka.

Forelesning 8

Traversering av grafer

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

Læringsmål:

- Forstå hvordan grafer kan implementeres
- Forstå BFS, også for å finne *korteste vei uten vekter*
- Forstå DFS og *parentesteoremet*
- Forstå hvordan DFS *klassifiserer kanter*
- Forstå Topological-Sort
- Forstå hvordan DFS kan *implementeres med en stakk**
- Forstå hva *traverseringstrær* (som *bredde-først-* og *dybde-først*-trær) er
- Forstå *traversering med vilkårlig prioritetskø**

*Kun på lysark, ikke i læreboka.

Forelesning 9

Minimale spenntrær

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

☐ Kap. 23. Minimum spanning trees

Læringsmål:

- Forstå *skog*-implementasjonen av *disjunkte mengder*
- Vite hva *spenntrær* og *minimale spenntrær* er
- Forstå GENERIC-MST
- Forstå hvorfor *lette kanter* er *trygge kanter*
- Forstå MST-KRUSKAL
- Forstå MST-PRIM

Forelesning 10

Korteste vei fra én til alle

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

Læringsmål:

- Forstå ulike varianter av *korteste-vei-* eller *korteste-sti*-problemet
- Forstå strukturen til *korteste-vei*-problemet
- Forstå at negative sykler gir mening for korteste *enkle vei* (*simple path*)*
- Forstå at *korteste enkle vei* er ekvivalent med *lengste enkle vei**
- Forstå hvordan man kan representere et *korteste-vei-tre*
- Forstå *kant-slakking* (*edge relaxation*) og RELAX
- Forstå ulike egenskaper ved korteste veier og slakking
- Forstå BELLMAN-FORD
- Forstå DAG-SHORTEST-PATH
- Forstå kobling mellom DAG-SHORTEST-PATH og dynamisk programmering*
- Forstå DIJKSTRA

*Kun på lysark, ikke i læreboka.

Forelesning 11

Korteste vei fra alle til alle

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

Læringsmål:

- Forstå *forgjengerstrukturen* for *alle-til-alle*-varianten av korteste vei-problemet
- Forstå Floyd-Warshall
- Forstå Transitive-Closure

Forelesning 12

Maksimal flyt

☐ Kap. 26. Maximum flow

Læringsmål:

- Kunne definere *flytnettverk*, *flyt* og *maks-flyt-problemet*
- Kunne håndtere *antiparallelle kanter* og *flere kilder og sluk*
- Kunne definere *residualnettverket* til et nettverk med en gitt flyt
- Forstå hvordan man kan *oppheve* (*cancel*) flyt
- Forstå hva en *forøkende sti* (*augmenting path*) er
- Forstå hva *snitt*, *snitt-kapasitet* og *minimalt snitt* er
- Forstå *maks-flyt/min-snitt-teoremet*
- Forstå Ford-Fulkerson
- Vite at Ford-Fulkerson med BFS kalles *Edmonds-Karp*-algoritmen
- Forstå hvordan maks-flyt kan finne en *maksimum bipartitt matching*
- Forstå *heltallsteoremet*

Forelesning 13

NP-kompletthet

☐ Kap. 34. NP-completeness: Innledning og 34.1–34.3

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

Læringsmål:

- Forstå sammenhengen mellom *optimerings-* og *beslutnings-problemer*
- Forstå *koding* (*encoding*) av en instans
- Forstå hvorfor løsningen vår på 0-1-ryggsekkproblemet *ikke er polynomisk*
- Forstå forskjellen på *konkrete* og *abstrakte* problemer
- Forstå representasjonen av beslutningsproblemer som *formelle språk*
- Forstå definisjonen av klassen P
- Forstå definisjonen av klassene NP og co-NP
- Forstå *redusibilitets-relasjonen* $\leq_{\mathrm{P}}$
- Forstå definisjonen av NP-*hardhet* og NP-*kompletthet*
- Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC
- Forstå hvorfor CIRCUIT-SAT er NP-komplett

Forelesning 14

NP-komplette problemer

☐ Kap. 34. NP-completeness: 34.4, 34.5

Læringsmål:

- Forstå hvordan NP-kompletthet kan bevises ved én reduksjon
- Kjenne de NP-komplette problemene CIRCUIT-SAT, SAT, 3-CNF-SAT, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP og SUBSET-SUM
- Forstå NP-kompletthetsbevisene for disse problemene
- Forstå at *0-1-ryggsekkproblemet* er NP-hardt
- Forstå at *lengste enkle-vei*-problemet er NP-hardt
- Være i stand til å konstruere enkle NP-kompletthetsbevis

Del II

Appendiks

Appendiks A

Bevisteknikk

Det å forstå en algoritme er nært beslektet med å kunne bevise at den er korrekt, og det å ha en viss ferdighet i å konstruere bevis er nyttig for å lære seg å designe algoritmer. Å lære om byggesteiner og vanlige strukturer for bevis kan også være til hjelp for å forstå eksisterende bevis, som dem i læreboka. Vi beviser at et utsagn følger fra et sett med aksiomer (initielle antakelser) ved å konstruere en serie med utsagn

(*i*) som starter med aksiomene og slutter med utsagnet vårt, og

(*ii*) der hvert utsagn bortsett fra aksiomene følger av de foregående.

Det at et utsagn *følger* av noen andre viser vi ved å sette sammen eller ta fra hverandre del-utsagn som er konstruert med disse logiske operatorene:

$$\Rightarrow \quad \Leftrightarrow \quad \neg \quad \wedge \quad \vee \quad \forall \quad \exists$$

En måte å organisere bruken av logiske operatorer er å se på hvordan vi kan *bevise* utsagn som inneholder hver av dem, eller å *bruke* disse utsagnene til å bevise noe annet. La oss se på operatorene etter tur.

Implikasjon, ekvivalens og negasjon

Implikasjonen $P \Rightarrow Q$ betyr at hvis P er sann, så er Q sann. Måten vi beviser en slik implikasjon på er at vi først *antar* at P er sann, og så beviser at Q er sann *under denne antagelsen*. Straks vi har bevist dette, kan vi konkludere med at P

impliserer Q. Antagelsen P er *lokal*, og gjelder bare så lenge vi prøver å bevise implikasjonen; den brukes *ikke* i resten av beviset. En alternativ bevismetode er såkalt *kontraposisjon*: Vi antar at Q *ikke* er sann, og vise at P da *heller ikke* er sann. Det vi da har bevist er egentlig implikasjonen $\neg Q \Rightarrow \neg P$, som kalles det *kontrapositive* utsagnet. Det blir riktig fordi de to implikasjonene er ekvivalente:

$$P \Rightarrow Q \qquad \text{hvis og bare hvis} \qquad \neg Q \Rightarrow \neg P\,.$$

Hvis vi ikke skal bevise $P \Rightarrow Q$, men allerede vet at dette er sant, og vil *bruke* den kunnskapen, så kan vi for eksempel bevise Q ved først å bevise P. Dette kalles *modus ponens*. Eventuelt kan vi bevise $\neg P$ ved først å bevise $\neg Q$. Dette kalles *modus tollens*. Et viktig poeng er at vi *ikke* kommer noen vei her ved å bevise $\neg P$ eller Q.

Ekvivalensen $P \Leftrightarrow Q$ betyr at P er sann *hvis og bare hvis* Q er sann. Den vanligste strategien for å bevise et slikt utsagn er å bevise hver retning for seg. Vi beviser implikasjonen *fremover* ($P \Rightarrow Q$) ved å anta P og utlede Q, og *bakover* ($Q \Rightarrow P$) ved å anta Q og utlede P. Om vi vil vise at flere utsagn er ekvivalente, som for eksempel

$$P \;\Leftrightarrow\; Q \;\Leftrightarrow\; R \;\Leftrightarrow\; S\,,$$

så kan vi bevise hver av ekvivalensene for seg. Det er fleksibelt hvilke ekvivalenser vi beviser. Vi kan f.eks. heller bevise at alle er ekvivalente med P:

$$P \;\Leftrightarrow\; Q \qquad P \;\Leftrightarrow\; R \qquad P \;\Leftrightarrow\; S\,.$$

Ofte kan vi spare mye arbeid på å bare bevise implikasjoner *i ring*, det vil si,

$$P \;\Rightarrow\; Q \;\Rightarrow\; R \;\Rightarrow\; S \;\Rightarrow\; P\,.$$

Ved gjentatt bruk av modus ponens så kan vi nå konkludere med for eksempel at $R \Rightarrow S$, uten å måtte bevise det direkte. Om vi ikke skal bevise en implikasjon $P \Leftrightarrow Q$, men alt vet at den holder, så kan vi naturligvis bruke den til å utlede de to implikasjonene $P \Rightarrow Q$ eller $Q \Rightarrow P$, om vi har bruk for det.

Hvis vi vil bevise en negasjon $\neg P$, det vil si at utsagnet P *ikke* er sant, så kan det ofte være nyttig å omformulere $\neg P$ til et ekvivalent positivt utsagn. For eksempel, i stedet for å bevise at $x > y$ er galt, så kan vi bevise aat $x \leq y$ er riktig. En viktig alternativ strategi er det som kalles *bevis ved selvmotsigelse*, der vi beviser at P leder til en selvmotsigelse, og dermed ikke kan være sann. Vi beviser dette som alle andre implikasjoner: Vi antar midlertidig at P er sann og beviser et eller annet som ikke kan være sant. Hvis vi for eksempel allerede vet Q, kan vi prøve å bevise $\neg Q$. Vi har da bevist $P \Rightarrow \neg Q$, og siden vi alt vet Q så kan vi ved modus tollens konkludere med $\neg P$.

Konjunksjon og disjunksjon

En konjunksjon $P \wedge Q$ er sann når både P og Q er sanne. For å bevise et slikt utsagn, beviser vi P og Q hver for seg. Vi kan også *bruke* utsagnet til å bevise P og Q hver for seg. Tilsvarende er en disjunksjon $P \vee Q$ sann når minst én av P og Q er sanne, så hvis vi beviser P, så kan vi konkludere med $P \vee Q$. Det samme gjelder naturligvis Q. Men om det vi prøver å bevise er $P \vee Q$, så er det kanskje urealistisk at vi skal klare å bevise at ett av de to utsagnene alltid er sant, helt uavhengig av det andre. I stedet kan vi anta at P er usann, og vise at Q da må være sann (eller omvendt). Det vi beviser da er altså $\neg P \Rightarrow Q$, som er ekvivalent med $P \vee Q$.

Hvordan kan vi så bruke en disjunksjon til å bevise noe annet? Vi kan bruke det som gjerne kalles et *uttømmende bevis* (*proof by exhaustion* eller *proof by cases*), der vi går gjennom alternativene, og beviser at noe er sant i hvert eneste tilfelle. Med andre ord: Hvis vi vet $P \vee Q$ og klarer å bevise både $P \Rightarrow R$ og $Q \Rightarrow R$, så har vi bevist R. Det gjelder selvfølgelig også om vi har flere enn to tilfeller å ta hensyn til; vi må bare vise at R følger av dem alle.

Denne strategien kan være nyttig i praksis. Vi kan gjerne selv definere et sett med alternative scenarier som tilsammen dekker alle muligheter. I hvert scenario har vi da et sett med antagelser som kan gjøre det enklere å bevise R. Og så lenge vi klarer å bevise R i hvert av de mulige scenariene, så har vi bevist at R må være sann. Et uttømmende bevis kan også brukes i motatt retning, for å *bevise* en disjunksjon. Hvis vi vil bevise $P \vee Q$, så kan vi konstruere et uttømmende sett med scenarioer, og for hvert scenario bevise enten P eller Q.

Universelle og eksistentielle utsagn

Uttrykket $\forall x P(x)$ betyr at utsagnet $P(x)$ er sant for alle mulige verdier av x. Vi beviser et slikt universelt utsagn ved å la x stå for en vilkårlig verdi som vi ikke antar noe om, og så beviser vi $P(x)$, uten universell kvantifisering. Vi kan *bruke* et universelt utsagn ved å plugge inn en hvilken som helst verdi for x. Hvis vi vet at $\forall x P(x)$ og a er en bestemt verdi, så kan vi konkludere at $P(a)$ er sant.

Et eksistensielt utsagn $\exists x P(x)$ betyr at det *finnes en* x slik at $P(x)$ er sant. For å bevise det må vi gjerne konstruere et eksempel a og vise at $P(a)$ er sant.* En spesiell variant av eksistensielle utsagn har formen $\forall! x P(x)$, som betyr at det finnes *én unik verdi* x som er slik at $P(x)$ er sant. Vi beviser et slikt utsagn

*Alternativt kan vi prøve å vise $\neg\forall x \neg P(x)$, siden det er ekvivalent med $\exists x P(x)$.

ved først å bevise $\exists x P(x)$, og deretter viser vi unikhet ved å bevise

$$\forall x \forall y((P(x) \wedge P(y)) \Rightarrow x = y)\,.$$

Det vil si, hvis P er sant for to verdier verdier, så må verdiene være like.

Å bruke et eksistensielt utsagn $\exists x P(x)$ er ofte litt vanskelig. En naturlig fremgangsmåte er å innføre en ny variabel y som representerer en verdi som er slik at $P(y)$ er sann. Vi kan da bruke $P(y)$ videre, selv om vi ikke kjenner y.

Appendiks B

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 løkka øker eller synker med 1 for hver gang, slik:

```
1  for i = 1..n-1
2      for j = i+1..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

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

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

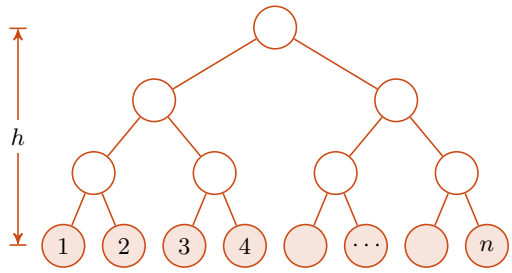
Dette er den *andre* av de to sentrale formlene:

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

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.

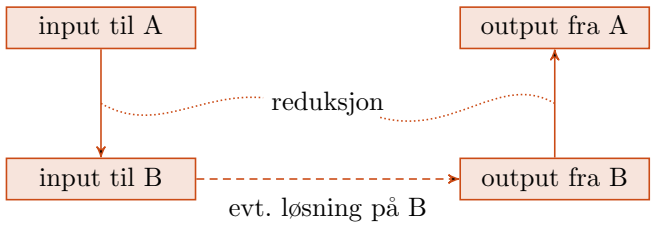
Telling 2: Antall deltakere er $n = 2^h$. I hver match blir én deltaker slått ut, helt til vi sitter igjen med én vinner etter finalen. Antall matcher trenger vi for å slå ut alle bortsett fra én er $n - 1$, eller $2^h - 1$, som er høyresiden i ligningen.

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 bytte 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 opprinelig

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 $A \leq B$ 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 C

Induksjon

Gjennom semesteret vil jeg bruke induksjon som et sentralt prinsipp for å forstå og konstruere algoritmer. Det er en bevismetode som brukes når man vil vise at alle elementene i en mengde V har en bestemt egenskap P, det vil si at $P(v)$ er sant for alle $v \in V$. Kjernen i metoden er at man viser hvordan beviset *smitter* eller *sprer seg* via en eller annen relasjon mellom elementene: Hvis noen elementer har egenskapen, så vil også noen andre *relaterte* eller *etterfølgende* elementer måtte ha den. Dersom man gjør dette på rett måte, så vil beviset spre seg, trinn for trinn, til hele mengden.

Naturlig induksjon

Den vanligste varianten av matematisk induksjon er induksjon over de naturlige tallene $\mathbb{N} = \{0, 1, 2, \dots\}$. Det som må bevises er da at

(*i*) $P(0)$ er sant; og

(*ii*) hvis $P(n-1)$ er sant, så er $P(n)$ sant.

Her kalles (*ii*) det *induktive trinnet*, mens (*i*) kalles *grunntilfellet*. For å bevise det induktive trinnet, *antar* vi at $P(n-1)$ er sant for en vilkårlig $n > 0$, for deretter å utlede at $P(n)$ må være sant. Vi kaller $P(n-1)$ *induksjonshypotesen*.

Relasjonen vi bruker holder altså for par av typen $(n-1, n)$, og beviset vårt sprer seg fra 0 til 1, fra 1 til 2, og så videre, helt til at vi har bevist $P(n)$ for alle $n \in \mathbb{N}$. Denne strategien hadde *ikke* fungert om vi hadde inkludert de negative heltallene, siden vi da ikke hadde hatt sted å starte. Grunnen til at vi beviser

grunntilfellet for seg er at 0 er det eneste naturlige tallet *uten noen forgjenger*. Om vi ser de to tilfellene under ett, så kan vi endre induksjonshypotesen litt:

> Hvis $P(k)$ er sant for alle k der $k = n - 1$, så er $P(n)$ sant.

Induksjonshypotesen er da trivielt sann for $n = 0$, siden det da ikke finnes noen slik k. Denne første varianten av naturlig induksjon kalles gjerne *svak induksjon*, for å skille den fra *sterk induksjon*, der vi bruker en sterkere induksjonshypotese:

> Hvis $P(k)$ er sant for alle k der $k < n$, så er $P(n)$ sant.

Bevismetoden er ekvivalent, men dette induksjonstrinnet kan være enklere å bevise, siden vi har flere antagelser vi kan bruke i beviset vårt. Denne varianten baserer induksjonen på relasjonen *er mindre enn*.

Strukturell induksjon

Hvis vi vil bevise en egenskap for alle strukturer av en eller annen sort, er det ofte nyttig å bruke sterk eller svak induksjon over *størrelsen* på strukturene. Men vi kan også bruke induksjon direkte på strukturen, over *delstrukturer*. Relasjonen vår blir da en eller annen form for *delstruktur-relasjon*. Hvis strukturen vi ser på er en probleminstans, kaller vi ofte delstrukturene *delinstanser*, eller noe mindre presist, *delproblemer*. Induksjonstrinnet blir:

> Hvis $P(D)$ er sant for enhver delstruktur D av T, så er $P(T)$ sant.

Hvis T er et rotfast tre, for eksempel, så vil D representere *deltrær*. Hvis T er en *sekvens*, så kan D være vilkårlige *subsekvenser*; mest vanlig er det kanskje å bruke *prefiks*, altså et initielle segmenter. Hvis T er sekvensen $(0, 1, 2, \dots)$ så er dette svært likt sterk induksjon over de naturlige tallene. Forskjellen er at vi ser på *hele sekvensen så langt* heller enn bare *nåværende tall*. Vi kan følge opp denne analogien, og tenke oss en *svak* variant av strukturell induksjon også:

> Hvis $P(D)$ er sant for umiddelbare delstrukturer D av T, så er $P(T)$ sant.

For et tre T med rot r kan for eksempel de umiddelbare delstrukturene være trær som har barna til r som røtter. For en sekvens $(a, \dots, y, z)$ kan en umiddelbar delstruktur være $(a, \dots, y)$. Denne strategien, både i sterk og svak utgave, vil fungere så lenge vi ikke kan fortsette å dele opp strukturene i det uendelige.

Velfundert induksjon

Vi kan generalisere ideen enda lenger. Det som trengs for induksjon er en binær relasjon E mellom elementene. Vi kan da bruke følgende induksjonstrinn:

> Hvis $P(u)$ er sant for alle u der $(u, v) \in E$, så er $P(v)$ sant.

Hvis vi ser på $G = (V, E)$ som en rettet graf, kan vi uttrykke induksjonstrinnet slik: Hvis alle nodene med inn-kanter til v har egenskapen, så vil v også ha den. Hvis vi klarer å bevise dette éne trinnet, så kan vi umiddelbart konkludere at noder *uten inn-kanter* har egenskapen, fordi induksjonshypotesen automatisk er sann for dem. La oss kalle slike noder *startnoder*. Om vi ser på relasjonen E som en ordning, vil startnodene være *minimale elementer*. For vanlig induksjon over $\mathbb{N}$ er 0 den eneste startnoden; for strukturell induksjon vil alle minimale delstrukturer være det. Vi kan tenke oss at vi *sletter* startnodene, fordi vi er ferdige med å bevise P for dem. Hvis ting går som de skal, vil vi enten være ferdige eller ha fått noen nye startnoder. De nye startnodene hadde bare innkanter fra noder vi alt er ferdige med, så induksjonstrinnet kan brukes på dem. Vi kan dermed også slette disse startnodene, og fortsette slik helt til vi er ferdige med alle nodene i V.

Hvis V er uendelig stor, kan vi bare tenke oss at prosessen over fortsetter uendelig lenge; det er ikke noe problem. Det som *er* et problem er om vi ikke har noe sted å *begynne*, altså om vi på noe tidspunkt står helt uten startnoder. Vi ønsker ikke å anta for mye om hvilken rekkefølge induksjonstrinnene gjøres i, så for å sikre oss krever vi rett og slett at *alle* delmengder av V må inneholde minst én startnode. Mer presist:

> Alle $U \subseteq V$ har minst én $v \in U$ slik at $(u, v) \notin E$ for alle $u \in U$.

Vi sier da at relasjonen E er *velfundert*, og kaller gjerne den resulterende induksjonsprosessen *velfundert induksjon*. Dersom V er endelig, er dette ekvivalent med at grafen G er en *rettet, asyklisk graf*.

Bibliografi

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. *Introduction to Algorithms*, The MIT Press, 3. utg., 2009.