Magnus Lie Hetland

Problemløsningsguide, 2021

Algoritmer og datastrukturer

Oversikt

Om du står overfor en type problem du har løst mange ganger, eller noe som ligner veldig, så kan det hende du umiddelbart ser hva løsningen skal bli, eller at du finner inspirasjon til en fremgangsmåte. Men om du jobber med problemer som er mer fremmede og utfordrende, kan det være lurt å gå frem systematisk, og prøve å finne grundige svar på de følgende spørsmålene:

- Hva skal du finne ut? (*tolkning*)
- Hva vet du? (*analyse*)
- Hva kan du slutte deg til? (*syntese*)

Med presise og relativt uttømmende svar på disse spørsmålene, og med tilhørende gode mentale modeller, bør veien frem til en løsning være noe mindre strevsom.

I det følgende skisserer jeg noen punkter du kan bruke for å få ting litt klarere. Som med de fleste slike fremgangsmåter, så vil du i praksis ende med å hoppe litt frem og tilbake og gjøre noen ting flere ganger, etc., men forhåpentligvis er rekkefølgen jeg gir relativt logisk. Jeg har delt inn prosessen i tre faser: tolkning, analyse og syntese.

Tolkning **T**

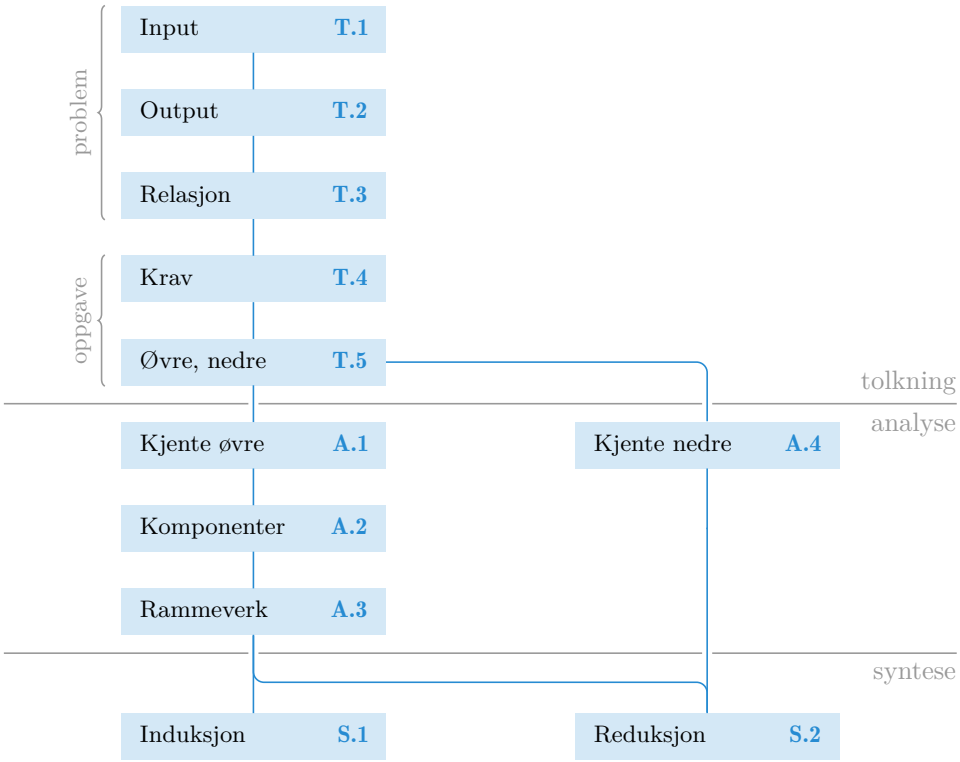
Definer problemet eller problemene du står overfor. Klargjør hva din oppgave er: Hva skal du gjøre med problemene?

Her handler det altså om å få det helt klart for seg hva man faktisk står overfor, *før* man begynner å konstruere en løsning. Det gjelder både å beskrive de problemene og objektene man har å gjøre med, og å få det klart hva det er man faktisk skal gjøre. Hva er den overordnede målsettingen?

Analyse **A**

Plukk problemet fra hverandre og plasser det i en større kontekst. List opp alt du har av relevant kunnskap og relevante verktøy.

I stedet for å prøve å løse et monolittisk problem fra bunnen av, så prøver vi å finne en eller annen struktur vi kan utnytte, og eksisterende resultater vi kan bygge på, så det vi ender opp med å konstruere selv blir relativt begrenset. Målet er å ikke finne opp kruttet på nytt – i det hele tatt finne opp så lite som mulig, og helst i små biter.



Figur 1: En oversikt over de ulike trinnene i hver fase av fremgangsmåten.

Syntese **S**

Koble sammen bitene og fyll inn det som mangler av transformasjoner, mindre beregningstrinn og eventuelle korrekthetsbevis.

Det er altså her vi konstruerer faktiske prosedyrer og algoritmer for de delene som gjenstår som uløste, gjerne reduksjoner og transformasjoner mellom ulike instanser av samme problem, eller mellom ulike problemer.

Før vi ser nærmere på fasene: Her er noen råd som kan brukes under alle tre.

Tenk utenfor hodet

Benytt deg av såkalt *distribuert kognisjon*, eller *skriftlig refleksjon*, og skriv ting ned. Tegn figurer og diagrammer. *Lag lister* med alle alternativer du kan komme på. Står du fast, bruk penn og papir, whiteboard eller din favoritt-editor

for å prøve å komme videre.

Om du ikke helt klarer å se for deg hva som foregår, prøv å lage en figur som inneholder alle elementene – eller så mange som mulig. Lag gjerne flere utkast, og prøv å gjøre figuren tydeligere. Tenk at du skal lage den for å forklare problemet til noen andre!

Og om du for eksempel har funnet ut at du skal redusere fra et problem med visse egenskaper, lag en uttømmende liste med relevante problemer du kjenner til, og stryk ut dem som ikke er relevante. Se så hva du kan få til med dem som står igjen. Jobb systematisk!

Prøv med forenkling

Lag en forenklet modell av problemet. Se på svært enkle instanser og eksempler. Prøv med en enkel algoritme til å begynne med, som f.eks. uttømmende søk.

Man kan ofte komme langt med forenklinger av problemet eller med svært enkle løsninger. Om ikke annet, så kan det gi en økt forståelse av problemet og hvilke angrepsvinkler som er lovende.

Forbedre løsningen trinnvis

Prøv å komme på eksempler som gjør at du får galt svar. Prøv så å forbedre løsningen, så den håndterer disse eksemplene.

Her kan man bruke en fremgangsmåte som minner om *test-driven development*.

Tenk logisk

Er det noen ting på listene dine som umulig kan være relevante? Er det noe du vet må være sant, som begrenser løsningen? Gir ting mening fra et fugleperspektiv?

Du kan for eksempel lage en liste med logiske slutninger du trekker underveis; det kan være lett å glemme dem, og de kan hjelpe deg senere. Og om du vet noe av konteksten for problemløsningen – kanskje det også kan hjelpe? Om du f.eks. arbeider med en oppgave i fagets øvingsopplegg eller på eksamen, så er det sannsynlig at du skal kunne basere deg på ting du har lært.

Fase 1

Tolkning

Formodentlig står du overfor ett eller flere problemer. For eksempel skal du konstruere en algoritme eller et hardhetsbevis for et problem, eller du skal redusere fra ett problem til et annet. Første fase går ut på å tolke problemstillingen grundig og rigorøst, ikke bare for å sørge for at du svarer på riktig spørsmål, men også for å få en god mental modell som du er i stand til å manipulere effektivt, og som du lettere kan koble til eksisterende kunnskap.

Problemet

I grove trekk kan man si at et problem er en binær relasjon mellom problem-instanser og løsninger. Vi begynner med å presisere denne relasjonen.

Hva er input? **T.1**

Hva slags objekt er en instans? En mengde? En sekvens? En graf? Består den av flere deler, som for eksempel en graf og en vektfunksjon? Stilles det spesielle krav for at en instans skal være gyldig?

Om det ikke er eksplisitt oppgitt noen abstrakte, matematiske objekter i problembeskrivelsen, tenk på hva du kan bruke. For eksempel kan mange problemer bli mer oversiktlige om man kan beskrive deler av dem som grafer av noe slag.

Hva er output? **T.2**

Hva slags objekt er resultatet? Har vi et søkeproblem, beslutningsproblem eller optimeringsproblem? Dersom vi har et søkeproblem: Hva er søkerommet?

Her gjelder det samme som i forrige punkt: Vi er ute etter en entydig beskrivelse av output som abstrakte objekter av noe slag. Vi har noen store klasser av output, som gir oss ulike typer problemer. De mest generelle er kanskje såkalte *søkeproblemer* eller *funksjonsproblemer*, der resultatet kan være et vilkårlig matematisk objekt, som tilfredsstiller visse krav. Hakket mer spesifikt er *optimeringsproblemer*, der resultatet enten er bare et rasjonalt tall som representerer optimum (f.eks. lengden på den korteste veien), eller et objekt som har den optimale målverdien (f.eks. selve

den korteste veien). Til slutt har vi *beslutningsproblemer*, der svaret kun er en logisk verdi (sann eller usann, ev. ja eller nei).

Hva er relasjonen? **T.3**

For søkeproblemer: Hvilke krav stilles det til objektene vi leter etter? For optimering: Hva er løsningsrommet og mulighetsområdet, og hva er målfunksjonen? For beslutningsproblemer: Hva er spørsmålet?

Etter å ha etablert hva slags objekter input og output er, må vi fastslå hvilke krav som stilles til output, gitt input. For eksempel, dersom vi har et optimeringsproblem, så vil en instans definere et *løsningsrom* eller *univers* som består av alle løsninger, gyldige og ugyldige, og et *mulighetsområde* som består av de *gyldige* løsningene. I tillegg har vi en målfunksjon som vi vil maksimere eller minimere. Alle disse elementene må vi ha klart for oss.

Oppgaven

Du skal trolig se på mulige algoritmer for problemet, men algoritmene det er snakk om kan være av forskjellige typer.

Hvilke krav stilles til algoritmen? **T.4**

Må den ha en bestemt verste, gjennomsnittlig eller amortisert kjøretid? Kan den kun bruke bestemte operasjoner? Kan den være randomisert?

Det grunnleggende scenarioet er at man prøver å finne en deterministisk algoritme som gir et eksakt svar i polynomisk tid, men det er på langt nær den eneste muligheten. Sørg for at du har det klart for deg akkurat hvilke krav som stilles. Hvis beskrivelsen involverer kvantifiseringer av typen «det finnes en konstant c» eller «for enhver konstant c», vær nøye med hva som kvantifiseres hvordan.

Skal du finne en øvre eller nedre grense? **T.5**

Skal du finne en algoritme av en viss type som løser problemet, eller vise at en slik algoritme ikke eksisterer, ev. gitt antakelser som f.eks. $\mathbf{P} \neq \mathbf{NP}$?

Det viktigste veiskillet er kanskje mellom løsning og hardhetsbevis. Av og til vet du hvilken av delene du skal finne, men av og til må du jobbe med begge deler i parallell, helt til du løser problemet eller finner ut at det er uløselig.

Fase 2

Analyse

Etter å ha kartlagt hva problemet *er*, så prøver vi å plukke det fra hverandre og plassere det i en kontekst av andre problemer og metoder vi allerede kjenner til.

Øvre grenser

Hvis du i **T.5** fant ut at du skulle finne en øvre grense, og faktisk *løse* problemet, er dette stedet å begynne analysen.*

Bruk eksisterende øvre grenser **A.1**

Kan du redusere til et eksisterende problem som har de egenskapene du trenger, f.eks. som kan løses like effektivt? Husk å bevare nødvendige egenskaper (som polynomisk kjøretid) i reduksjonen.

List opp alle problemer som kan løses like bra som – eller bedre enn – det du prøver å oppnå, og som ligner på eller minner deg om problemet du definerte i første fase. Hvis du har skikkelig flaks, vil den grundige tolkningen din vise deg at du faktisk står overfor et kjent problem, og du kan anvende eksisterende resultater og algoritmer direkte. Mer generelt, så finner du kanskje et problem som har noe slektskap i hvordan input og output er satt sammen, og hvilke krav som stilles til svaret. I så fall kan du kanskje transformere problemet ditt så det ligner mer på det du alt har en løsning til, dvs., redusere det nye problemet til et kjent et? Normalt må reduksjonen bevare de egenskapene du er interessert i, men om det du reduserer til er *bedre* enn det du trenger, så kan du tåle å kaste bort litt under transformasjonen, om nødvendig.

Kartlegg parametre og komponenter **A.2**

Plasser probleminstansen i en familie med flere instanser som beskrives av et sett med parametre eller egenskaper, som for eksempel størrelse. Om mulig,

* Her er det snakk om øvre grense for kostnader som f.eks. kjøretid – altså *konservative* grenser, grenser du vet du kan nå, i motsetning til *optimistiske*, nedre grenser. Hvis du prøver å maksimere noe så vil jo en konservativ grense faktisk heller være en *nedre* grense.

prøv å dele instansen opp i komponenter, som gjerne kan være andre instanser av samme problem.

I A.1 ser vi på forholdet mellom problemet vårt og andre problemer. Her ser vi på forholdet mellom instansen vår og andre instanser. Den mest opplagte måten å organisere instanser på er etter størrelse, men det kan også være andre egenskaper som kan brukes, som for eksempel antall elementer du bruker av et eller annet slag (som hvilke noder som kan være med i korteste stier, eller hvilke gjenstander du får putte i ryggsekken. Slike parametre kan være kunstige, men kan la oss bygge opp en løsning trinnvis. Det samme gjelder dekomponering av instansen i flere komponenter (som første og andre halvdel av en sekvens, f.eks.); uansett er vi ute etter en familie med instanser av ulik kompleksitet eller størrelse. Kanskje det også går an å konstruere seg instanser av ulike typer, som kan løses på ulike vis, og så kan vi partisjonere instansen vår i f.eks. én av hver type, og så løse dem forskjellig.

Bruk et eksisterende rammeverk **A.3**

Kan problemet uttrykkes annerledes, ev. transformeres, så det passer inn i generelle designmetoder eller rammeverk for algoritmekonstruksjon?

Om du ikke kan redusere til et bestemt problem, så kan det hende du kan formulere problemet så det passer inn i et mer generelt rammeverk, hvis du ikke alt har gjort det som en naturlig del av fase 1. Analysen du har gjort så langt bør allerede ha gitt deg grunnlag for splitt-og-hersk-løsninger eller løsninger vha. dynamisk programmering, men hva med for eksempel grådighet? Kan det uttrykkes som et flytproblem?

Her finnes det også mer avanserte rammeverk (lineærprogrammering, matroider, multiplikativ vektoppdatering, ...), som i hovedsak faller utenom pensum i dette faget. Men om du skal løse et problem i praksis, så kan programvare for lineærprogrammering være svært nyttig.* Uansett: Gå gjennom rammeverkene du mener kan være relevante, og se om du kan få problemet til å passe inn.

* Se f.eks. https://pypi.org/project/PuLP/.

Nedre grenser

Om du i **T.5** fant ut at du skulle gi en *nedre* (dvs., optimistisk) grense for kjøretid eller en eller annen kostnad (ev. en øvre grense for en gevinst), så må du angripe ting litt annerledes. Det er generelt svært vanskelig å bevise slike ting fra bunnen av, så du vil nok omtrent utelukkende basere deg på eksisterende grenser i dette tilfellet.

Bruk eksisterende nedre grenser **A.4**

Kan du redusere fra et eksisterende problem som har de egenskapene du trenger, f.eks. som det er like vanskelig å løse effektivt? Husk å bevare nødvendige egenskaper (som polynomisk kjøretid) i reduksjonen.

Her, som i **A.1**, må vi sørge for at reduksjonen ikke stjeler showet. Vi må ikke la den skjule arbeid eller kvalitetstap som så forsvinner fra resonnementet vårt. Det er for eksempel ikke noe problem å redusere fra TSP til sortering, dersom reduksjonen kan ta eksponentielt lang tid! Mer om dette i fase 3.

Fase 3

Syntese

Når du kommer hit, har du forhåpentligvis en klar forståelse av problemet, hvordan det er strukturert, og hvilke andre problemer eller metoder du kan bruke til å løse oppgaven du står overfor. Det som gjenstår er å faktisk konstruere prosedyrene og algoritmene som gjør det du trenger.

Induksjon

Dersom du skal løse probemet, uten å redusere det til et annet, har du trolig i **A.2** kartlagt en familie med delproblemer. Hvis du kan løse de enkleste av disse direkte, og bygge løsninger fra delløsninger, så har du en algoritme.

Sørg for velfunderte avhengigheter S.1

Du kan la løsningen for et delproblem være avhengig av at andre delproblemer er løst allerede, men du kan ikke ha uendelige avhengighetskjeder! Bortsett fra det, så fungerer alle slags avhengighetsrelasjoner.

Så lenge alle avhengigheter kan spores tilbake til et grunntilfelle av noe slag, så er alt i orden; du trenger da bare konstruere ett beregningstrinn: Løsningen for ett delproblem, gitt at avhengighetene alt er løst. For mer om denne fremgangsmåten, se for eksempel «Using Induction to Design Algorithms» av Udi Manber [3].

For interesserte: Om velfundert induksjon

Så lenge du kan finne en total ordning av delproblemene (en topologisk sortering), og har et sted å begynne, så kan du bruke vanlig (sterk eller svak) matematisk induksjon. Det er dette f.eks. Manber gjør. Men av og til kan det være kunstig å innføre en slik total ordning, og det er heller ikke nødvendig, så lenge det i prinsippet er mulig. Anta, for eksempel, at du har en transitiv relasjon $\mathrm{C}(x, y)$ som kan bety «x er en komponent av y» eller noe tilsvarende. Du kan da bruke følgende induksjonsprinsipp:

Induksjonshypotese: For alle x der $\mathrm{C}(x, y)$, så kan vi løse x.

Induksjonstrinn: Gitt x-løsninger, kan vi konstruere en løsning for y.

Mer generelt, og formelt, for et predikat P, og et univers U av instanser:

$$\forall y \in \mathrm{U}\big[\forall x \in \mathrm{U}(\mathrm{C}(x,y) \to \mathrm{P}(y)) \to \mathrm{P}(y)\big] \to \forall y \in \mathrm{U}\ \mathrm{P}(y) \tag{3.1}$$

I praksis vil vi ofte behandle grunntilfellet for seg – det vil si, tilfeller der y ikke har noen komponenter, og altså er minimal under C. Om vi ser på alle objekter x der $\mathrm{C}(x,y)$ eller kun de umiddelbare «barna» til y (dvs., der ingen $z \neq x,y$ eksisterer slik at $\mathrm{C}(x,z)$ og $\mathrm{C}(z,y)$) er ekvivalent, og tilsvarer forskjellen på sterk og svak induksjon over $\mathbb{N}$.

Dette induksjonsprinsippet fungerer når C er en såkalt *velfundert* relasjon, som betyr at det ikke finnes noen uendelige synkende kjeder. Det vil si, om vi skriver $x < y$ for $\mathrm{C}(x,y)$, så skal det ikke finnes noen kjeder $x_1 > x_2 > x_3 > \cdots$. (Merk at vi godt kan ha uendelige *stigende* kjeder.) Dersom vi lar C representere rekursive kall i en algoritme, for eksempel, så tilsvarer dette bare kravet om å unngå uendelig rekursjon. Om mengden vi ser på er endelig, begrenser dette seg til å forby sykler, som gir oss en rettet, asyklisk graf (DAG) eller, ekvivalent, en streng delvis ordning. Dette blir for eksempel rekursjonstreet for splitt-og-hersk eller delproblemgrafen for dynamisk programmering [1, s. 367].

Reduksjon

Enten du skal vise øvre eller nedre grenser, så må du unngå å jukse med reduksjonene.

Bevar nødvendige egenskaper S.2

Hvis du skal løse et problem i polynomisk tid, sørg for at reduksjonen har polynomisk kjøretid. Mer generelt må reduksjonen, sammen med en reell eller hypotetisk løsning, ha de egenskapene du resonnerer rundt.

Akkurat hvilke krav som stilles kommer helt an på hva du skal oppnå. Poenget er at om du har en reduksjon R fra A til B, så har du fastslått at R sammen med en løsning for B er en mulig løsning for A. Alt annet følger av dette.

En reduksjon er en algoritme, og kan i teorien konstrueres på samme vis som om man skulle ha løst et problem. I praksis er de to typene algoritmer gjerne forskjellige, og reduksjoner kan være en del «rarere» enn faktiske løsninger. I en reduksjon behandler du *til*-problemet som en maskin som du så skal bruke til å løse *fra*-problemet – og det kan være en maskin som ikke passer spesielt godt til denne oppgaven! I så fall må reduksjonen din bygge snodige komponenter (såkalte *gadgets*) fra råvarene i *til*-problemet. En fremgangsmåte for å redusere fra A til B kan være:

(*i*) Kartlegg A og B grundig, som beskrevet i fase 1 (tolkning).

(*ii*) For hvert del av problemet A, finn ut hvilke deler av B som kan brukes til å

simulere delen fra A.

(*iii*) Beskriv hvordan en instans fra A kan transformeres til en instans for B, basert på forrige trinn. Eventuelt også beskriv en transformasjon av svaret tilbake, om nødvendig.

(*iv*) Sjekk at transformasjonen resulterer i korrekt svar, og at den bevarer egenskapene du trenger å bevare (som polynomisk kjøretid).

En overordnet, skissepreget versjon av denne prosessen vil du trolig også benytte deg av i fase 2 (analyse), når du leter etter problemer du kanskje kan redusere fra, dersom ingen umiddelbart peker seg ut (f.eks. pga. grensen du skal påvise).

For mange eksempler på reduksjoner, se for eksempel Cormen mfl. sitt delkapittel 34.4 [1], eller mitt noe uformelle notat om temaet [2].

Bibliografi

[1] Thomas H. Cormen mfl. *Introduction to Algorithms*. MIT press, 2009.

[2] Magnus Lie Hetland. *Some Hardness Proofs*. 2011.

[3] Udi Manber. «Using Induction to Design Algorithms». *Communications of the ACM* 31.11 (1988).