Hvordan man angriber et programmeringsproblem

Torben Mogensen

20. oktober 2015

Resumé

Programmering er i bund og grund problemløsning, så derfor er metoder til at løse programmeringsproblemer ikke fundamentalt anderledes end metoder til at løse alle andre slags problemer.

Der findes heldigvis problemløsere, der har studeret deres egen problemløsningsproces og endda formuleret denne som en opskrift til nye problemløsere. En af dem er George Pólya, der i 1945 skrev bogen "How to Solve it", som omhandler løsning af matematiske problemer. Men metoderne er gangbare ikke bare til matematik, men til stort set alle former for teknisk problemløsning, herunder programmering.

Denne vejledning tager udgangspunkt i Pólyas metoder, men fokuserer på de dele, der passer bedst til programmering.

De fire skridt

Pólya opdeler problemløsningsprocessen i fire skridt:

1. Forstå problemet.
2. Lav en plan til løsning af problemet.
3. Udfør planen.
4. Reflekter over, hvordan det gik.

For hver af de ovennævnte skridt giver Pólya et antal teknikker, som man kan bruge. Det er normalt kun en lille del af teknikkerne, der passer til det givne problem, og det er ikke altid oplagt, hvilke der gør. Men en begyndende problemløser kan prøve de mest lovende teknikker, indtil hun finder en eller flere, der virker.

Selv om de fire skridt er beskrevet som selvstændige faser, vil der ofte være overlap: Mens man søger at forstå problemet, vil ideer til løsningsplaner opstå, og mens man laver planer, vil konkrete løsning på delproblemer vise sig så oplagte, at man laver dem med det samme. Og undervejs i løsningen, finder man ofte ud af, om det går godt eller ej. Og nogle gange vil man, når man er i gang med sin afsluttende refleksion, at man faktisk ikke havde forstået problemet ordentligt, og så starter man igen fra skridt 1.

For meget små problemer kan nogle af skridtene synes overflødige, men det kan være en gode ide at lave dem alligevel, så du får træning. Når du har erfaring nok til bare at kunne hælde korrekte og optimale løsninger på små problemer ud af ærmet, kan du springe skridtene over for den slags små problemer.

I de efterfølgende afsnit beskriver jeg et antal metoder til de fire skridt, som jeg selv synes virker godt til programmeringsopgaver af den slags, man bliver stillet på et kursus. Der vil i beskrivelserne indimellem blive brugt begreber, som I ikke vil have hørt om ved kursets begyndelse. Så hvis der er noget uforståeligt, så ignorer det og vend tilbage til beskrivelserne senere i kurset.

1 Forstå problemet

Det er selvsagt ikke nemt at løse et problem, man ikke forstår. Uden forståelse risikerer man nemt, at ens "løsning" slet ikke løser det rigtige problem – hvis man i det hele taget kommer så langt. Så forståelse af problemet er det måske allervigtigste skridt mod en løsning. Her er et par teknikker, der kan være nyttige:

Forklar problemet til andre

Programmeringsopgaver på indledende programmeringskurser er i reglen beskrevet meget præcist og koncist, så man kunne tro, at forståelse ikke er det store problem. Men et er at have en beskrivelse af problemet, et andet er at forstå beskrivelsen. Når man har læst beskrivelsen kan man afprøve sin forståelse af problemet ved at lægge beskrivelsen væk og forklare problemet til en anden ved at bruge sine egne ord og formuleringer. Hvis man ikke kan dette, har man nok ikek forstået problement. Hvis man ikke lige har en anden, man kan forklare til, så kan man prøve at skrive sin egen forståelse af problemet ned som tekst, som er udformet sådan at en uindvidet ville kunne læse teksten og forstå problemet ud fra denne tekst. Pas på ikke at genbruge fraser fra den oprindelige problembeskrivelse.

Når du har lavet din egen forklaring, så kig på den oprindelige opgave tekst: Fik du det hele med? Lavede du antagelser, der ikke var givet i opgaven? Betyder de i det hele taget det samme. Hvis der er problemer, så lav om på din forklaring, indtil den (efter din bedste overbevisning) passer med den oprindelige.

Udenfor programmeringskurser er opgaverne sjældent beskrevet præcist og udtømmende, så her kan der være behov for at snakke med opgavestilleren for at forstå problemet. Men specielt her er det en god ide at formulere sin egen forståelse af problemet til opgavestilleren, så man er sikker på, at man er enige. Manglen på dette har givet fiasko i flere store offentlige IT-projekter.

Relater problemet til noget konkret

Programmeringsopgaver arbejder ofte med abstrakte begreber, ligesom matematik: Tal, lister, træer, par osv. Hvis man er matematisk inklineret, er det ikke noget problem at arbejde med abstrakte begreber, men for de mindre matematisk inklinerede kan det være en fordel at hægte det abstrakte problem op på noget konkret eller i hvert fald mere velkendt: Man kan tænke på tal som afstande mellem steder eller længder af ting, man kan tænke på talpar som scoringstal i fodboldkampe og man kan tænke på sorterede lister som opslagsværker i stil med ordbøger eller telefonbøger, hvor indholdet er sorteret for at gøre det lettere at finde information.

Men pas på ikke at bruge egenskaber ved det konkrete begreb, som ikke findes i det abstrakte. For eksempel vil afstande mellem steder aldrig være negative tal, så hvis opgaven ikke nævner, at tal aldrig er negative, skal man passe på ikke at lave den antagelse, når man senere løser problemet.

Mangler der oplysninger?

Selv om opgaver på programmeringskurser er forklaret mere præcist end større programmeringsopgaver, kan der godt mangle oplysninger, der er nødvendige for at lave en komplet løsning.

Nogle gange er opgaven ikke veldefineret for alle de angivne mulige inddataværdier og andre gange er der flere forskellige mulige svar, der alle opfylder kravene i opgaven. Hvis man kan spørge opgavestilleren, er det fint, men nogle gange bliver man nødt til selv at udfylde hullerne. Når man gør det, er det vigtigt at beskrive, hvad der mangler i opgaven, hvad de mulige måder for at udfylde mangler er, hvad man vælger og hvorfor. Denne beskrivelse bør afleveres sammen med opgaveløsningen.

Ude i "den virkelige verden" er programmeringsopgaveformuleringer ofte meget mangelfulde. Så man bør indgå en dialog med opgavestilleren (kunden eller chefen) for at afklare manglerne. I disse situationer er det ekstremt vigtigt at få formuleret, hvordan man vil fylde hullerne i formuleringen og få denne beskrivelse godkendt af opgavestilleren. Igen har mangel på dette ført til adskillige IT-skandaler.

Hvordan ser jeg, om min løsning er rigtig?

Man har ikke rigtigt forstået et problem, hvis man ikke er i stand til at genkende, om et løsningsforslag er en korrekt løsning eller ej. Så som en del af forståelsen, bør man overveje, hvordan man kan vise (eller i det mindste sandsynliggøre) sin løsnings rigtighed.

Det kan indebære, at man finder nogle relevante og nogenlunde dækkende mulige input og overvejer, hvordan resultatet bør se ud for disse input. Det er vigtigt at have afdækket hjørnetilfældende: De mindste mulige værdier, de største mulige værdier, "underlige" (men korrekte) værdier, osv.

Formuler din forståelse på skrift

Du bør nedskrive de dele af din forståelse, som ikke direkte fremgår af opgaveteksten. Dels vil en skriftlig formulering hjælpe dig til at huske denne forståelse i den senere proces, og dels vil du nogle gange opdage, at din forståelse ikke er konkret nok til at blive formuleret, og dermed ikke brugbar.

2 Lav en plan til løsning af problemet

Mange arbejdstimer er spildt af programmører, der kaster sig ud i en løsning uden en ordentlig plan, og som efter et større stykke arbejde finder ud af, at det ikke var den rigtige måde at løse problemet på.

Det er i sig selv en lærerig oplevelse, så prøv det bare et par gange. Men hvis du ikke vil spilde din tid, så prøv at planlæg løsningen på forhånd. Planen vil i reglen hænge stærkt sammen med den forståelse af problemet, man opnåede i skridt 1, så det er vigtigt at bruge denne forståelse.

Overvej grænsetilfældene

Ofte kan man finde et simpelt grænsetilfælde, hvor svaret direkte kan gives. Dette grænsetilfælde kan blive en del af ens løsning som et basistilfælde i en løsning, der bruger

matematisk induktion: Inddata opdeles i basistilfældet og alt andet. Svaret for basistilfældet gives direkte, og resten forsøges løst ved at bruge en induktionsantagelse.

For eksempel bruger fakultetsfunktionen denne ide: 0! (fakultet af 0) er 1, og resten af tilfældende bruger induktionsantagelsen $(n + 1)! = (n + 1) \cdot n!$. Det samme gælder funktionen, der finder den største fælles divisor af til tal: Basistilfældet er $gcd(0, n) = n$ og induktionsantagelsen er $gcd(m, n) = gcd(n \bmod m, m)$.

Der kan også være inddata, hvor der ikke findes en veldefineret løsning. Disse er også en slags grænsetilfælde, men skal sorteres fra, inden man løser problemet. Så lav en test, der kan genkende disse inddata og giv for disse en passende fejlmeddelelse i stedet for et svar. I begge de to ovenstående eksempler bør negative tal sorteres fra.

Kan man opdele i mindre delproblemer?

Ofte kan man løse opgaven for store inddata ved at dele det store data op i mindre deldata, løse problemet for disse og kombinere løsningerne til en løsning for det samlede problem.

Det kan i det simpleste tilfælde blot være at løse problemet for mindre tal og bruge denne løsning til at finde løsningen for de(t) oprindelige tal, ligesom man gør ved induktion. Både fakultetsfunktionen og *gcd* bruger denne metode.

Andre gange kan man dele problemet op i to eller flere mindre delproblemer. Hvis man vil sortere en liste af tocifrede tal kan man først dele listen over i to lige store dele, sortere hver af dem og "flette" de to sorterede lister sammen til en sorteret liste. Når delproblemerne er små nok (for eksempel en liste med et tal), er løsningen i regel nem nok, så man får også en slags induktiv løsning, bare med en slags dobbeltinduktion.

Kan man dele data op i kategorier?

Nogle gange kan man dele inddata op i flere kategorier, der har hver sin løsningsmetode. For eksempel kan man behandle lige og ulige tal forskelligt eller positive og negative tal forskelligt.

Til potensopløftning kan man for eksempel dele op i lige og ulige potenser: $x^n = (x^{\frac{n}{2}})^2$ for lige tal og $x^n = x \cdot x^{n-1}$ for ulige tal.

Det er bedst, hvos klasseopdelingen kan laves med en simpel betingelse eller med mønstergenkendelse. For eksempel er opdeling i primtal og ikke-primtal sjældent nyttig, da det kan være svært at afgøre om et tal er et primtal.

Hvis man har en datatype, der eksplicit er opdelt i underkategorier, er det oplagt at bruge denne underopdeling i sin løsning.

Er der en symmetri i problemet, som du kan udnytte?

Hvis det om en funktion f af to argumenter gælder, at $f(x, y) = f(y, x)$, kan man udnytte dette. For eksempel kan man bytte om på elementerne, så den første parameter altid er mindre end den anden og derefter løse problemet under den antagelse, hvilket kan simplificere løsningen. Det kan generaliseres til mere end to argumenter: Hvis rækkefølgen af elementer i en liste af tal ikke betyder noget for resultatet, kan man sortere listen først og udnytte denne egenskab senere.

Man kan også udnytte andre former for symmetri: Binomialkoefficienter beregnes ud fra formlen

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n(n-1)\cdots(n-k+1)}{1 \cdot 2 \cdots k}$$

Her kan man udnytte, at

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \left(\begin{array}{c} n \\ n-k \end{array} \right)$$

sådan at man bruger den venstre form, hvis $k \leq n - k$ og den højre hvis $k > n - k$, da man dermed minimerer antallet af multiplikationer.

Er der en matematisk ækvivalens, du kan udnytte?

Vi har allerede set eksempler på dette, både for binomialkoefficienter, ved potensopløftning, hvor vi brugte $x^{2n} = (x^n)^2$, og ved største fælles divisor, hvor vi brugte $gcd(m, n) = gcd(n \bmod m, m)$. Men der er mange andre matematiske ækvivalenser, der kan bruges i andre problemer.

Er der et mere generelt problem, der er lettere at løse?

Det kan virke kontraintuitivt, at dette kan være tilfældet, men det sker alligevel: For at beregne en funktion $f(x)$ kan det være nemmere at beregne en funktion $g(x, y)$, hvor $f(x) = g(x, 0)$ eller lignende. Det kan skyldes, at det ved beregningen af g kan være nemmere at opdele problemet i mindre delproblemer af samme form og samle løsningerne på disse til en samlet løsning.

Hvis man for eksempel i en ikke-sorteret liste af n tal skal finde medianen, altså det mindste tal x i listen, hvor mindst $n/2$ af elementerne i listen er mindre end eller lig med x, så kan det være nemmest ved at lave en funktion, der for et vilkårligt i og en vilkårlig liste med mindst i elementer finder det mindste tal x, hvor mindst i af elementerne i listen er mindre end eller lig med x. Jeg vil lade det være en opgave for læseren, hvordan dette kan gøres.

Et andet eksempel er funktionen til omvending af en liste, som løses ved at bruge en hjælpefunktion med en ekstra parameter.

Planlæg din afprøvning

Mens du planlægger din programmering, bør du også planlægge, hvordan du vil afprøve din løsning. Lav evt. nogle funktioner, der kan hjælpe med afprøvningen.

Hvis du bruger hjælpefunktioner i din løsning, så overvej, hvordan disse kan testes.

Tænk over din navngivning

Find passende navne til dine funktioner, hjælpefunktioner og parametre. Hvis en hjælpefunktion skal finde det mindste tal i en liste, så er det bedre at kalde den `findMindste` end at kalde den `g` eller `hjaelpefunktion`. Hvis en liste indeholder personer, så kald den `personer` i stedet for `liste`. Men overdriv ikke: `listeAfPersoner` kan være for langt, specielt fordi man som regel let kan se af programmet, at der er tale om en liste. Navnene bør reflektere de ting, der ikke tydeligt vil fremgå af programteksten.

Formuler dine overvejelser og din plan på skrift

Ligesom i skridt 1 er det vigtigt at konkretisere sine tanker og planer på skrift, dels for at sikre sig, at de ikke er for vage og uklare, og dels for bedre at kunne huske dem senere.

3 Udfør planen

Her bør du som udgangspunkt følge den plan, man lavede i skridt 2. I nogle tilfælde er det blot at omformulere de grænsetilfælde, problemopdelinger, klassifikationer, matematiske regler osv, man lavede i skridt 2, til et konkret program. I andre tilfælde bliver der brug for at lave ekstra hjælpefunktioner og lignende.

Hvis du finder ud af, at planen er uhensigtsmæssig, så lav om på planen og start forfra. Vær ikke bange for at kassere selv store mængder kode, hvis den er uhænsigtsmæssig. Det kræver ofte mere arbejde at ændre koden til noget, der kan bruges, end at starte forfra, og resultatet ved at starte forfra er i regel bedre.

Find konkrete datarepræsentationer

I skridt 1 og 2 bruger man ofte abstrakte databegreber såsom mængder, grafer og træer. Når disse abstrakte begreber skal implementeres i et program, er der ofte flere forskellige måder at gøre det på. En mængde kan for eksempel implementeres som en usorteret liste, som en sorteret liste, som en indikatorfunktion, som en bitvektor eller som noget helt femte. Hvad, der er bedst, afhænger meget af de operationer, man skal lave på sine data. Lav derfor funktioner til disse operationer med det samme.

Det kan være en god ide altid at arbejde på datastrukturen gennem funktioner, der laver operationer på strukturen, i stedet for at arbejde direkte på datastrukturen. Det gør det nemmere at lave om på repræsentationen senere, hvis den viser sig ikke at være hensigtsmæssig.

Start med det lette

Implementer først grænsetilfælde og de andre nemme dele af problemet, og vent med de svære tilfælde til sidst. Du opdager måske noget ved løsningen af de simple problemer, der kan hjælpe dig med de svære delproblemer. Og hvis du ikke kan løse de simple delproblemer, så bør du overveje, om din forståelse eller plan for løsningen er rigtig, eller om du skal gå tilbage til skridt 1 eller 2.

Skriv kommentarer før kode

Inden du skriver et stykke kode, så skriv en kommentar i programmet om, hvad intentionen med koden er. Ikke så meget om, hvad der sker i koden, men hvorfor det sker, og hvad formål det tjener.

Når du så har skrevet koden, så sammenlign med kommentaren og se, om din kode rent faktisk opfylder det, som kommentaren siger.

Skriv eventuelt kommentarer, der kan hjælpe en læser til at forstå, hvordan din kode virker. Det skal ikke være en omformulering af koden til dansk, men kan for eksempel være angivelse af, hvad det er for nogle symmetrier og matematiske egenskaber, man udnytter, eller en relation, der altid vil gælde mellem de variable, der bruges i koden – en såkaldt

invariant. En formulering af disse ting i kommentarer kan også hjælpe med til at opdage fejl i koden.

Afprøv din kode hele tiden

Du burde i planlægningen af din løsning have overvejet, hvordan din kode skal afprøves. Lav test efter denne plan hele tiden under programmeringen, så du opdager fejl. Afprøv de enkelte hjælpefunktioner og gentag afprøvningen, hver gang du laver ændringer – selv om ændringerne kan synes trivielle.

Hvis du har formuleret invarianter eller lignende relationer, der forventes opfyldt mellem variable i din funktion, så tilføj evt. test i funktionen, der checker, om det rent faktisk er tilfældet og laver en fejlmeddelelse, hvis det ikke er tilfældet. Når du er færdig med programmet, kan disse test laves om til kommentarer.

Hvis en test fejler, så overvej først, om det er testen, der er forkert.

4 Reflekter over, hvordan det gik

Når du synes, at du er færdig med din løsning, så reflekter over denne følelse: Hvor sikker er du på, at din løsning er rigtig? Er der nogle ting, du kan gøre, for at få dig til at føle dig mere sikker? Kan du for eksempel komme i tanke om ting, der ikke er blevet afprøvet?

Reflekter dernæst over forløbet: Var der nogle steder, hvor du ikke havde tænkt nok over din forståelse eller plan, inden du gik i gang med at kode? Hvis du var inde på et vildspor, hvor du måtte kassere en del kode, kunne du have opdaget det tidligere? Hvis fra starten vidste det, du ved nu, var der så noget, du, ville have gjort anderledes?

Og, til sidst (men ikke mindst), reflekter over, hvad du har lært af processen, både fagligt og i forhold til dine arbejdsmetoder.

Afsluttende bemærkninger

Pólyas "How To Solve It" kan findes (i en ikke alt for godt scannet udgave) på `https://notendur.hi.is/hei2/teaching/Polya_HowToSolveIt.pdf`. Men da den ikke er voldsomt dyr, kan jeg anbefale, at man anskaffer den i bogform.

Den er med sine over 200 sider langt mere detaljeret, end denne korte vejledning. Jeg vil dog anbefale, at man giver sig tid til at læse Part I og Part II (de første 36 sider).

Derudover kan det være lærerigt og underholdende en gang imellem at læse et afsnit i den alfabetiske liste over heuristikker. Start med afnittet "Heuristic" for at forstå, hvad ordet betyder.

Bogen afslutter med 18 små, morsomme opgaver med vink og løsninger. Hvis man keder sig en eftermiddag, kan man prøve at se på opgaverne.