Vad är pseudokod?

En vägledning till hur man skriver pseudokod i mästarprovet

Emma Enström

1 oktober 2013

1 Att beskriva en algoritm

Saker som kan variera mellan olika beskrivningar av en algoritm är detaljnivå, avsett syfte, målgrupp och vilka uttrycksmedel som står till förfogande. Syftet med beskrivningen kan variera, och syftet kan också påverka hur beskrivningen ser ut. Är den t.ex. skriven för att pedagogiskt förklara något, för att ge en snabb inblick i idén bakom en algoritm, eller som skiss för att senare implementera algoritmen – kanske till och med i något särskilt programspråk? Är den avsedda publiken matematiskt skolad, är de programmerare, eller är de skolbarn?

Man kan förstås beskriva algoritmer utan vare sig programspråk eller pseudokod. Det finns fördelar och nackdelar med alla framställningsformer.

Ju fler detaljer man beskriver, desto längre blir texten och desto svårare blir det att överblicka beskrivningen. Å andra sidan, om det finns för få detaljer kan det både bli svårt att förstå hur algoritmen ska kunna lösa sitt problem och svårt att analysera den.

De uttrycksmedel som vi jämför här är naturligt språk, programspråk, matematisk notation av idag och pseudokod.

1.1 Naturligt språk

Varför krångla till det med notation? Varför inte bara berätta på ren svenska vad algoritmen gör? Vardagsspråket kan uppfattas som mindre krångligt än mer tekniska notationssätt, men går budskapet fram? Det kan visa sig att många begrepp och termer blir luddiga och det behövs väldigt många ord. Så många, att det blir jobbigt att läsa texten. Läsaren får då anstränga sig mycket för att skaffa sig en överblick över algoritmen.

1.2 Ett Javaprogram

Här finns det regler och inget annat än regler! Kompilatorn har ingen som helst förmåga att tolka oklara beskrivningar, och inget kan tas för givet. Datorn kan utföra instruktionerna - om man kan programmera algoritmen kan man beskriva den för en dator. Den här notationen blir pladdrig på ett annat sätt än om man använder naturligt språk, och det görs ingen som helst skillnad på viktiga, bärande principer och triviala påståenden.

1.3 Modern matematisk notation

Kompakthet! Att införa notationssätt som har en mycket väldefinierad, begränsad betydelse gör att man slipper skriva så mycket. Det är dock ofta en lång väg att gå från notationen till idén bakom algoritmen, och även om texten tar liten plats går den inte nödvändigvis snabbt att läsa.

1.4 Pseudokod

För att kunna välja vilka av ovanstående för- och nackdelar man vill ha, kan man blanda in olika mycket av ovanstående notationsformer när man skriver pseudokod. Om algoritmen handlar om matematik, kan matematisk notation användas för att beskriva vad man håller på med, t.ex. $aktuellaNoder = \{v \in V : d(v) > 3\}$ för ”skapa en mängd av alla hörn i grafen som har minst tre grannar” Om det blir tydligare så kan istället vanliga meningar användas, som t.ex. ”för varje bok i biblioteket”.

Pseudokodens första kännetecken är att den inte har en syntax. Vilken kodliknande sekvens som helst som inte kompilerar är inte användbar som pseudokod. För att producera bra pseudokod kan det till och med vara bra om du inte börjar med ”vanlig” kod och försöker ta bort det som är onödigt, utan istället låtsas att du ska hålla en lektion om det du vill beskriva, hitta vilka poänger du absolut vill få fram, skriva ner dem och sedan komplettera med de detaljer som behövs för att någon annan lätt ska kunna implementera algoritmen utifrån din beskrivning.

Vid labbredovisningar händer det att assistenten vill kolla studentens uppfattning om sitt program genom att fråga ”Så vad gör det här programmet?”. Den frågan brukar besvaras på något av följande sätt: ”Det går igenom en text och kontrollerar vilka ord som finns med mer än en gång”, respektive: ”Först läser det in filen och sen skapar det en lista och sen sätter man i till 0 och sen...”. Personen som svarar på det senare sättet kan mycket väl förstå sitt program, men har en annan uppfattning om vilket slags svar som var förväntat än vad assistenten troligen hade. Det är som bekant en besvärlig uppgift att modellera verkligheten i matematik eller programkod, men det är en lika delikat uppgift att översätta åt andra hållet – att beskriva i vardagsspråk vad ett program gör, det vill säga vilket problem det löser och genom vilka principer. En övergripande beskrivning kan ibland gå att ha med i själva pseudokoden, men behöver ibland ingå i en kompletterande beskrivning i naturligt språk.

Pseudokod ska typiskt sett innehålla tillräckligt mycket information för att vara precis och inte luddig i kanten, men inte så mycket information att den blir svåröverblickbar och pladdrig. Normalt är det onödigt att ta med variabeldeklarationer, speciellt deklarationer av styrande variabler i slingor. Indextrixande ska finnas med i den mån algoritmen bygger på ett särskilt sätt att hantera index (i t.ex. arrayer), eller om det lättaste sättet att förklara vad som görs är att numrera element. Här kan formuleringar liknande dem i satser och bevis i matematiken ibland vara tydligare än formuleringar hämtade från programspråk: jämför ”För varje bokstav b i ordet” med ”for $i \leftarrow 1$ to ordlängd(w) do $b \leftarrow w[i]$”.

För att göra något bra, kan man ibland behöva försöka göra det dåligt först. Försök att beskriva en beräkning av summan av alla element som är delbara med 3 som förekommer i mängden X...

1. ... pladdrigt med vardagsspråk,
2. ... korrekt i Java eller C++ på ett sätt som inte är gjort för att vara lättläst,
3. ... så kompakt som möjligt med hjälp av matematisk notation,
4. ... med pseudokod inspirerad av vardagsspråk, nästan utan programspråkskonstruktioner,
5. ... med pseudokod inspirerad även av programspråk,
6. ... med pseudokod inspirerad även av matematik.

Något av ovanstående var antagligen lättare att utföra, eller kändes naturligare, för dig. Någon av beskrivningarna kändes svår att överhuvudtaget klämma ur sig (speciellt som den inte var bra till något). Alla kommer inte att ha samma favoritbeskrivning.

2 Exempel: Euklides algoritm

Här ställer vi ytterligheten naturligt språk mot kompakt pseudokod där matematisk notation ingår.

2.1 Språket i Euklides Elementa

När Euklides Elementa skrevs, fanns inte många av de konventioner vi idag använder tillgängliga för den som ville uttrycka sig om matematik. Nollan och positionssystemet var inte införda fullt ut, och det hade inte etablerats notationsstandarder för så triviala saker som + och -. Bevisen av satser och algoritmer i Euklides Elementa är skrivna med naturligt språk, men man använde ”variabler” för att resonera om olika entiteter. För att titta närmare på boken, se t.ex. `http://aleph0.clarku.edu/~djoyce/java/elements/bookVII/propVII1.html`.

2.2 Euklides algoritm, engelsk översättning av Elementa

Proposition 1: Two unequal numbers (being) laid down, and the lesser being continually subtracted, in turn, from the greater, if the remainder never measures the (number) preceding it, until a unit remains, then the original numbers will be prime to one another.

Proposition 2: *To find the greatest common measure of two given numbers not relatively prime to one another.*
Let AB and CD be two given numbers (which are) not prime to one another. So it is required to find the greatest common measure of AB and CD.

In fact, if CD measures AB, CD is thus a common measure of CD and AB, (since CD) also measures itself. And (it is) manifest that (it is) also the greatest (common measure). For nothing greater than CD can measure CD.

But if CD does not measure AB then some number will remain from AB and CD, the lesser being continually subtracted, in turn, from the greater, which will measure the (number) preceding it. For a unit will not be left. But if not, aB and CD will be prime to one another (Prop. 7.1). The very opposite thing was assumed. Thus, some number will remain which will measure the (number) preceding it. And let CD measuring BE leave EA less than itself, and let EA measuring DF leave FC less than itself, and let CF measure AE. Therefore, since CF measures AE, and AE measures DF, CF will thus also measure DF. And it also measures itself. Thus, it will also measure the whole of CD. And CD measures BE. Thus, CF also measures BE. And it also measures EA. Thus, it will also measure the whole of BA. And it also measures CD. Thus, CF measures (both) AB and CD. Thus CF is a common measure of AB and CD. So I say that (it is) also the greatest (common measure). For if CF is not the greatest common measure of AB and CD then some number which is greater than CF will measure the numbers AB and CD. Let it (so) measure (AB and CD) and let it be G. And since G measures CD, and CD measures BE, G thus also measures BE. And it also measures the whole of BA. Thus, it will also measure the remainder AE. And AE measures DF. Thus, G will also measure DF. And it also measures the whole of DC. Thus, it will also measure the remainder CF, the greater (measuring) the lesser. The very thing is impossible. Thus, some number which is greater than CF cannot measure the numbers AB and CD. [(Which is) the very thing it was required to show].

Detta är från bok 7 från Euclid’s Elements, hämtat från en utgåva redigerad/editerad av Richard Fitzpatrick. (`http://farside.ph.utexas.edu/euclid/elements.pdf`)

Proposition 1 behandlar samma sak som proposition 2, men med tal som *är* relativt prima. ”The unit”, 1, betraktades inte som ett tal och därför behövde situationerna med tal som är

relativt prima och tal som inte är det behandlas separat.

Poängen med texten är att lägga fram matematiska sanningar, därför handlar den till största delen om korrekthetsresonemanget, som här är helt integrerat i framställningen men också har en egen avdelning i slutet.

2.3 Euklides algoritm i pseudokod (utan korrekthetsresonemang)

Data: Two numbers a and b, $a \geq b > 0$
Result: The greatest number that divides both a and b
function `gcd(`a,b`)`
  while $b \neq 0$ **do**
    $t := b$;
    $b := a \mod b$;
    $a := t$;
  return a

Algoritm 1: Euklides algoritm, iterativ version

Data: Two numbers a and b, $a \geq b > 0$
Result: The greatest number that divides both a and b
function `gcd(`a,b`)`
  if $b = 0$ **then**
    return a
  else
    return `gcd(`b, $a \mod b$`)`

Algoritm 2: Euklides algoritm, rekursiv version

Det är uppenbart att algoritmen är lättare att urskilja i pseudokodsversionen än i originalbeskrivningen. Korrekthetsbeviset skulle likna Euklides text lite mer, men även där går det idag att använda notation som ger bättre överblick och kortare beskrivningar.

3 Riktlinjer för pseudokodsskrivande

- Beskriv din algoritm både med text och med pseudokod. Låt texten sköta förklaringen på en övergripande nivå och pseudokoden ha med tillräckligt med detaljer för att övertyga läsaren om att algoritmen går att implementera. Om din pseudokod är skriven efter att du löst problemet i något programspråk, tänk en extra vända på vad det är för uppgifter varje slinga eller sats har och fundera på om dessa blir tydligare, kortare eller klarare med matematisk notation eller med naturligt språk istället!

- Sträva efter att ha *lättläst*, *tillräckligt detaljerad* och förhållandevis *kort/kompakt* pseudokod. Undvik i regel att deklarera variabler och att använda konstruktioner från ditt favoritspråk som egentligen bara är till för kompilatorn eller interpretatorn!

- För att åstadkomma kompakthet och inte inkräkta på läsbarhet är det ibland bra att använda matematiska symboler som $\emptyset$, $\mod$, $\exists, +, -, \cdot, \leq, \neq, \notin, \forall, \infty, \cup$ osv.

- Man kan beskriva datat i termer av mängder även om implementationen kommer att använda t.ex. en matris, hashtabell, länkad lista eller ett binärträd. Om du gör det bör du ha i åtanke att uppslagning i de olika datastrukturerna tar olika tid, och kommentera det i texten och när du gör komplexitetsanalysen. (Om uppgiften handlar om att välja lämpliga

datastrukturer är det förstås nödvändigt att basera pseudokoden och analysen på den valda datastrukturen.)

- Vanliga verb, eller hela meningar, ger ibland bättre beskrivningar än lånade uttryck från ett programspråk.
- Indentering är lika användbart för att göra pseudokod lättläst som för att producera lättläst programkod.

Konstruktioner som är lämpliga i pseudokod är t.ex. if–then, do–while, while–do, for i ←x to y do, for each, break, return, comment, function/procedure, input, output, invariant, assert.

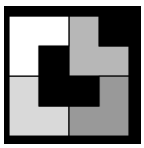
Skriva algoritmer i LaTeX

Om du renskriver dina uppgifter i Latex, kan du faktiskt få ytterligare vägledning i vad som brukar ingå i pseudokod av de olika paket som finns för att typsätta algoritmer. Där finns sådana kommandon som är vanligt förekommande i pseudokod definierade, och indenteringen sköts automatiskt. De kräver alla en viss ansträngning för att komma igång med, men de producerar snygga resultat. En hel del av algoritmerna i det här kompendiet är typsatta med hjälp av paketet *algorithm2e*. De algoritmer som beskrivs i övningarna är ofta typsatta med Viggos egna kommandon. Andra paket som finns tillgängliga är *algorithmic*, *algorithmicx*, *algpseudocode* och *program*. Paketen beskrivs bl. a. på `http://en.wikibooks.org/wiki/LaTeX/Algorithms`.

4 Exempel från ett mästarprov

Det här är första uppgiften på mästarprov 1 hösten 2012. Sist, i Algoritm 5, är lösningsförslaget som Viggo skrev.

Kvadrattäckning med dekomposition
Betygskriterium: utveckla algoritmer med datastrukturer för enkla problem givet en konstruktionsmetod.
Ett trevligt (?) tidsfördriv är att försöka täcka en given yta med många likadana bitar. I denna uppgift ska du konstruera en algoritm som (nästan) täcker en kvadrat med sidan 2^n med bitar som ser ut som små L. Varje L-bit har ytan 3 och kan vridas och läggas på 4 sätt. Det går inte att täcka precis hela kvadraten, så innan vi börjar klipper vi bort en 1×1-ruta i övre högra hörnet på kvadraten. I vänstra figuren nedan visas hur en täckning av en 4×4-kvadrat med utklippt hörn kan göras.
Din algoritm ska ta n som indata och returnera en $2^n \times 2^n$-matris som beskriver hur täckningen kan göras. Ge varje L ett eget nummer och märk dom tre platserna i matrisen som den täcker med detta nummer, se exemplet nedan till höger.
Din algoritm ska använda *dekomposition* för att konstruera täckningen. Lägg första L-biten precis mitt i kvadraten och dela sedan upp problemet i fyra delar.
Beskriv algoritmen med pseudokod. Analysera algoritmens tidskomplexitet (med enhetskostnad).



$$\begin{pmatrix} 5 & 5 & 2 & 0 \\ 5 & 1 & 2 & 2 \\ 4 & 1 & 1 & 3 \\ 4 & 4 & 3 & 3 \end{pmatrix}$$

Nedan i Algoritm 3 följer en kort och möjligtvis pedagogiskt upplagd förklaring av samma metod. Notera att den här lösningen har ett annat n än den förra. Det finns flera saker som gör denna framställning mindre lämpad för en skriftlig inlämningsuppgift som ett mästarprov.

- Det som ska skrivas i matrisen, $aktuelltNr$, nämns bara när det ska fyllas i. Ska samma nummer alltid användas? Hur vet vi att vi kan hålla reda på vårt nummer?
- ”Fyll i basfall” är inte en så bra instruktion, även om den text man skrivit tillsammans med pseudokoden förklarar vad denna manöver går ut på.
- ”Kolla vilken bit som redan är täckt” och ”Dela upp M i fyra delar” är också oklara instruktioner – de kan dölja komplexitet och ger inte programmeraren någon idé om hur uppgiften ska utföras. De är inte heller särskilt lämpliga för att övertyga en lärare om att man har tänkt igenom hur algoritmen ska operera.

Det som beskrivningen i Algoritm 3 däremot gör bra är att svara, på ett övergripande plan, på assistentens fråga ”Så, hur fungerar din algoritm?”.

```
n ← antal rader i utdatamatrisen
M ← tom matris
M[1, n] ← täckt
if n < 4 then
 | Fyll i basfall
else
 | fyllIDel(M)
fyllIDel()
 | Kolla vilken bit som är täckt
 | Fyll mitten med aktuelltNr
 | if M.rows == 4 then
 |  | Fyll delmatrisen som basfall
 | else
 |  | Dela upp M i fyra delar
 |  | och anropa fyllIDel med dessa
 |  |
```

Algoritm 3: Lösning med pseudokod i naturligt språk utan detaljer.

Pseudokoden i Algoritm 4 nedan lyckas beskriva hur många anrop som görs och i stort sett hur matrisen delas upp för de olika anropen. Den har istället fastnat på basfallen, som inte beskrivs i pseudokod alls. Ett tecken på att den som gjorde så här tyckte att ”matematiseringen” av basfallen är knepig är användandet av bokstaven L i olika orienteringar. Grafiska framställningar kan fungera i pseudokod, men de kan också missa målet. Det finns detaljer som i det här fallet är onödiga att ha med, som initieringen av alla värden i matrisen till -1 och att ange returdatatyp på funktionerna. Algoritmen bortser dock helt ifrån något väldigt viktigt i uppgiftslydelsen - att beskriva vad som fylls i. Inget utdata finns heller beskrivet, så det är faktiskt oklart vad vi gör med delarna. Själva dekompositionssteget beskrivs som sagt detaljerat – men blir det rätt hela vägen? Kommer den här algoritmen att fylla i matrisen på rätt sätt, om vi antar att basfallsfunktionen $skrivL$ i varje anrop skriver nästa siffra på de ställen som svarar mot de beskrivna fallen?

```
Data: matris med sidan 2^n
for i ← 1 to 2^n do
    for j ← 1 to 2^n do
        matris[i, j] ← −1
void rekursivSkrivL(sida, posX, posY, vinkel)
    if vinkel = 0 then
        matris[posX][posY] ← L
    else if vinkel = 1 then
        matris[posX][posY] ← ⅂
    else
        matris[posX][posY] ← ⅃
    if sida = 2 then
        skrivL(posX, posY, "Kvadrant 1/3")
    else if sida < 8 then
        skrivL(posX, posY, "Kvadrant 1/3") skrivL(posX, posY, "Kvadrant 1/3")
        skrivL(posX, posY, "Kvadrant 4")
        skrivL(posX, posY, "Kvadrant 2")
    else if sida ≥ 8 then
        rekursivSkrivL(sida/2, posX + sida/2, posY − sida/2, 0) // Här går vi
            uppåt till höger.
        rekursivSkrivL(sida/2, posX − sida/2, posY + sida/2, 0) // Här går vi
            uppåt till vänster.
        rekursivSkrivL(sida/2, posX − sida/2, posY − sida/2, 0) // Här går vi
            uppåt till vänster.
        rekursivSkrivL(sida/2, posX + sida/2, posY + sida/2, 0) // Här går vi
            nedåt till höger.
void skrivL(posX, posY, Matrissort)
    /* Här lägger vi till en 4x4-matris utifrån positionerna som
       skickats med. Vi lämnar 1 tom ruta i ett hörn. Parametern
       Matrissort avgör vilket hörn som ska lämnas tomt.
       T.ex. är matrisen med L i mitten en ''Kvadrant 1/3''-matris och
       den med ⅂ i mitten är en ''Kvadrant 2''-matris.                 */
rekursivSkrivL(2^n, 2^n/2, 2^n/2, 0)
```

Algoritm 4: Lösning som har mycket detaljer trots att det finns viktiga steg som utelämnats.

Algoritmen blir:

$Q[1, 2^n] \leftarrow 0$; Cover$(1, 1, n, 1, 0)$; **return** Q

Där den rekursiva funktionen definieras som:

Cover$(x, y, n, quadrant, lastL) =$
// L-täcker en $2^n \times 2^n$-kvadrat med övre vänstra hörnet i (x, y),
// där den urklippta biten finns i kvadrant *quadrant* (tal mellan 1 och 4)
// och där senast utlagda L-biten har nummer $lastL$.
// Numret för den sista L-biten som användes i täckningen returneras.
$lastL \leftarrow lastL + 1$
$h \leftarrow 2^{n-1}$
if $quadrant \neq 1$ **then** $Q[x + h - 1, y + h] \leftarrow lastL$
if $quadrant \neq 2$ **then** $Q[x + h - 1, y + h - 1] \leftarrow lastL$
if $quadrant \neq 3$ **then** $Q[x + h, y + h - 1] \leftarrow lastL$
if $quadrant \neq 4$ **then** $Q[x + h, y + h] \leftarrow lastL$
if $n = 1$ **then return** $lastL$
$lastL \leftarrow$ Cover$(x, y + h, n - 1,$**if** $quadrant = 1$ **then** 1 **else** $3, lastL)$
$lastL \leftarrow$ Cover$(x + h, y + h, n - 1,$**if** $quadrant = 4$ **then** 4 **else** $2, lastL)$
$lastL \leftarrow$ Cover$(x + h, y, n - 1,$**if** $quadrant = 3$ **then** 3 **else** $1, lastL)$
$lastL \leftarrow$ Cover$(x, y, n - 1,$**if** $quadrant = 2$ **then** 2 **else** $4, lastL)$
return $lastL$

Algoritm 5: Viggos lösningsförslag till kvadrattäckning med dekomposition.

I Algoritm 5 finns Viggos pseudokodsbeskrivning av hur algoritmen för kvadrattäckning med dekomposition ska gå till. Hans fokus har varit på att visa exakt vad som ska göras, ner på detaljnivå, dvs vilken siffra som skrivs på vilket ställe. Pseudokoden är kompakt och precis, men eftersom uppgiften handlar om indextrixande behöver den åtföljas av en text som sköter den övergripande förklaringen av vilka principer algoritmen bygger på.