Algoritmer och deras beskrivning

Många tror att algoritmer endast har med matematik att göra. Även om algoritmer historiskt har introducerats av matematiker kan de användas på all problemlösning. Man kan t.o.m. tillämpa algoritmer på vardagliga problem. Samtidigt ligger de till grund för all programmering. Ett datorprogram är ingenting annat än en algoritm beskriven i datorns språk. Men även följande vägbeskrivning är ett fullgott exempel på en algoritm:

> *” ... gå ut från ditt hus till vänster, fortsätt rakt fram, sväng till höger vid trafikljuset, fortsätt sedan andra korsningen till vänster, där finns ett gult hus, på 2:a våningen bor jag ...”*

En *algoritm* är alltså ett tillvägagångssätt att lösa ett problem – vilket som helst. Och det behöver inte heller vara datorn som löser det. Vi kommer att precisera denna definition lite senare (sid 17). Problemet som ska lösas kan sakna lösning – då kan det inte heller finnas någon algoritm. Om däremot problemet är lösbart, kan det ha ingen, en eller flera algoritmer. Vi sysslar här endast med sådana problem som har minst en algoritm.

Historiens första algoritm

Det är alltid lärorikt att blicka tillbaka till historien. Själva ordet *algoritm* härstammar från ett namn på en person: namnet på den framstående persiska matematikern *Al-Kharazmi*[*]. Namnet har sedan latiniserats och blivit *algoritm.* Han levde på 800-talet. I sin berömda bok om *Algebra* ställde han upp historiens första algoritm som beskriver addition och multiplikation av heltal. Den används även idag. Men kunde man inte addera eller multiplicera heltal på 800-talet? Jo, redan långt tidigare kunde man räkna med tal i Egypten, Indien, Persien och Grekland. Vad var i så fall Al-Kharazmis historiska prestation? Ja, det var inte att komma på hur man *adderar* eller *multiplicerar* heltal – för det var ju redan känt, utan hur man i allmänna ordalag *beskriver* tillvägagångssättet, dvs formulerar en algoritm för dessa operationer.

1000 år mellan praktisk lösning och formell beskrivning

Det är anmärkningsvärt att *beskrivningen* av hur man räknar med heltal kom till mer än 1000 år efter den praktiska lösningen. Orsaken är att den korrekta, allmänna beskrivningen som ska hålla i *alla* tänkbara situationer, är mycket svårare att åstadkomma än den faktiska lösningen av ett eller en klass av problem. Att själv gå en väg som man känner till är enklare än att formulera en korrekt vägbeskrivning som alla förstår och kan följa. Anledningen är att algoritmer är *generella* till sin natur, och just det är tjusningen: Att försöka beskriva dem så att de håller i *alla* situationer, det är konsten. Detta gäller även idag: Program – det moderna sättet att beskri-

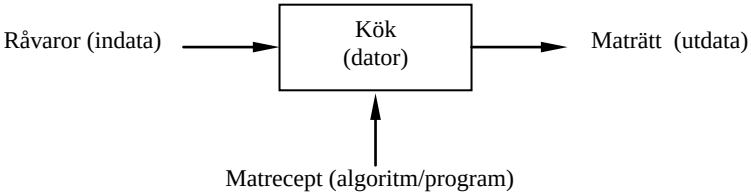
[*] Så uttalas hans namn på persiska idag (utan prefixet *Al-* som är arabiska). Han är född i *Kharazm*, en antik region som fanns i nuvarande nordöstra delen av Iran (Khorasan) mot Turkmenistan och Uzbekistan. På den tiden var Iran ockuperat av araberna.

va algoritmer – måste fungera under alla omständigheter och ska helst aldrig krascha. Dessvärre vet vi ju att så inte är fallet. En av utmaningarna inom programmering ligger just i att skriva program som fungerar i *alla* situationer. Det vi kan lära oss av det 1000-åriga glappet mellan praktisk lösning och formell beskrivning är: Satsa tid och energi på att först *analysera* det problem du vill lösa med ett program. Fokusera på att *beskriva* lösningen av problemet så generellt som möjligt.

Exempel på algoritmer

I vardagen använder vi algoritmer hela tiden, om än omedvetet. Här några exempel:

- **Matrecept** vars användning kan jämföras med programkörning på datorn:



  Matrecept skrivs fortfarande med vanligt språk men man kan konstatera att det finns en viss stil som är typisk för alla matrecept.

- **IKEA:s monteringsanvisningar** för att sätta ihop delarna till en möbel. Här används en kombination av text och grafik som är mycket effektiv. Grafiken förenklar algoritmen avsevärt. ”En bild säger mer än tusen ord.” På köpet får man en slags internationalisering, ett oberoende av det lokala språket, vilket gör att algoritmen förstås över hela världen.

- **Bruksanvisningar** av alla slag är exempel på algoritmer, även om många av dem i praktiken är värdelösa. Men det finns dåliga algoritmer på andra områden också.

- **Manualer** för datorprogram som visar hur ett program ska användas.

- **Konstruktionsritningar** som ingenjörer gör för att en viss produkt ska kunna tillverkas i fabrik. En arkitektritning av ett hus är ett specialfall av det. Här har grafiken tagit över helt och hållet.

- **Partiturer:** Noter i musik som används för att spela ett musikstycke och som omfattar noggranna anvisningar om hur en hel orkester ska spela. Ett speciellt ”språk” används som varken består av text eller grafik, utan snarare av symboler längs en tidslinje.

- **Spelregler** är snarare ett negativt exempel: De talar mest om vad man *inte* får göra och lämnar ett stort utrymme för hur man får spela inom reglernas ram. Därför finns två skilda problemställningar. Den ena är: ”Hur *får* jag spela?”

Spelregler ger delvis (negativa) svar på det. En helt annan problemställning är: ”Hur *vinner* jag spelet?” *Spelteori* som involverar sannolikhetslära behandlar denna fråga. I spelteori brukar man tala om *strategier* snarare än algoritmer. Här befinner vi oss i ett gränsområde där problem inte alltid har en entydig lösning eller saknar algoritm. I fortsättningen kommer vi att undvika sådana frågeställningar. Vi betraktar endast problem som är lösbara och har minst en algoritm. Exemplet belyser dock en viktig aspekt: Inte bara vägen till lösning måste beskrivas. Först måste *problemställningen* vara klart och exakt formulerad så att man kan avgöra om det finns en entydig lösning och minst en algoritm.

Definition av algoritm

Låt oss titta på vad som är *gemensamt* för exemplen ovan (utom spelreglerna), för att kunna fornulera en generell definition. Vilka typiska faktorer förekommer i alla exempel?

För det första består de alla av en rad anvisningar om vad som ska göras för att lösa det givna problemet. Frågan är: Ska man tillåta alla slags anvisningar? Om de leder till problemets lösning, varför inte? Men leder alla slags anvisningar till lösningen? T.ex. anvisningen ”Bygg ett hus!” är helt värdelös. Ingen av oss kan bygga ett hus med bara denna anvisning. Problemet är ju just *hur* man bygger huset. Anvisningarna måste vara mycket enklare och mer detaljerade. Vem som helst ska kunna utföra dem. Sådana anvisningar kallas *elementära instruktioner*. Bara sådana kan tillåtas i en algoritm om de ska leda till problemets lösning.

För det andra. Undersöker man de ovannämnda exemplens innehåll kan man konstatera att anvisningarna måste utföras i en viss *ordning.* Det går inte att kasta om ordningen. Man inser redan vid receptexemplet att man *först* måste knåda degen och *sedan* ställa in den i ugnen, inte vice versa. Vid partiturexemplet är ju ordningen helt avgörande. Och så är det i alla algoritmer. Ordningsföljden för de elementära instruktionerna måste finnas med i algoritmen. Självklart måste en algoritm också ange när instruktionerna ska upphöra. Om vi sammanfattar kan vi formulera följande definition:

> En *algoritm* är en följd av precisa anvisningar, s.k. *elementära instruktioner*, som löser ett givet problem, inklusive anvisningar om i vilken *ordning* instruktionerna ska utföras och när de ska avslutas. Dvs en algoritm måste ha ett exakt *avslutningskriterium*.

Av stor betydelse, speciellt för datoriseringen, är att algoritmen måste vara tolkningsbar *på ett enda sätt*. Det får inte finnas tvetydigheter i formuleringen. Datorn kan ju bara tolka våra anvisningar på ett enda sätt. Svårigheten ligger alltså i algoritmens *beskrivning*, vilket är en god illustration till det 1000-åriga glappet mellan praktisk lösning och formell beskrivning som nämndes på sid 15. Det är i regel svårare att *beskriva* en algoritm än att lösa ett specifikt problem i en specifik situation.

Anledningen är att algoritmer måste vara *generella* till sin natur: De måste hålla i *alla* situationer. Följande dilemma uppstår:

Hur beskriver man en algoritm bäst, så att den kan tolkas *endast på ett sätt*, men samtidigt behålla sin *generella* karaktär? Vi ska nu diskutera några hjälpmedel som kan användas för att formulera sådana algoritmer:

Olika sätt att beskriva algoritmer

- **Vanligt språk** är ett sätt att beskriva algoritmer, t.ex. vägbeskrivningen till en kompis. Största fördelen med det är att alla som kan språket direkt förstår algoritmen utan att behöva lära sig något nytt. Nackdelen är att det ofta kan tolkas på olika sätt. Och tur är det! Annars skulle man ju t.ex. inte kunna skriva en dikt eller njuta av den. Men just i samband med algoritmer då man eftersträvar entydighet, är möjligheten till olika tolkningar en nackdel.

- **Pseudokod** är en hybrid (blandning) mellan vanligt språk och formaliserad kod, ett försök att minska det vanliga språkets tvetydighet genom att införa vissa strukturer och t.o.m. grafiska stilmedel i layouten. Allt som på ett entydigt sätt beskriver en algoritm, även en matematisk formel, kan användas som pseudokod. I nästa avsnitt tar vi upp ett exempel på pseudokod med vanligt språk kombinerad med generella *kontrollstrukturer* (sid 22) som förekommer i alla algoritmer. På så sätt uppnår det vanliga språket en högre grad av entydighet, noggrannhet och struktur.

- **Flödesschema** eller flödesschema är en variant av IKEA:s monteringsanvisningar som kombinerar text och grafik med en klar dominans mot det senare. Man använder sig av geometriska figurer som symboliserar algoritmens byggstenar och av pilar som visar flödet i algoritmen och definierar instruktionernas ordning. Med dessa få stilmedel uppnår man en hög noggrannhet i beskrivningen, eliminerar tvetydigheter och åskådliggör algoritmens logiska struktur. Det tänkta händelseförloppet syns tydligt. I det avseendet är flödesschema överlägset både vanligt språk och pseudokod. Flödesschemassymbolik är ett utmärkt medel som lämpar sig inte bara för beskrivning av fullständiga algoritmer, utan också för att åskådliggöra logiken hos mindre, men kritiska delar av ett program. Vi kommer att använda oss av detta medel i hela boken.

- **Programkod** är den variant av algoritmbeskrivning som används för att låta en dator utföra algoritmen. Därför måste den kunna tolkas av datorn. Programkoden översätts till ett språk, kallat *maskinkod* som datorns processor förstår. Programkoden däremot – även kallad *källkod* – är skriven i något programmeringsspråk som man måste lära sig. Medan källkod förstås av människan, men inte av datorn, förstås maskinkod av datorn, men inte av människan.

- **Andra sätt** att beskriva algoritmer finns också. Inget av dem har lyckats etablera sig som standard. Anledningen är att det är oförutsägbart vilka metoder som i allmänhet kan lösa problem. Många av de traditionella sätten kan betecknas med det samlande namnet *pattern designs*. Andra använder begrepp

som *strukturdiagram*, *Mind Maps* eller *beslutstabeller*. Mest känt är dock *UML = Unified Modeling Language* som är ett språk för objektorienterad design och modellering. Man använder UML för att att planera, utveckla och visa strukturen hos avancerade objektorienterade system. UML används för att lägga upp och modellera stora programmeringsprojekt, vilket förutsätter bekantskap med den objektorienterade programmeringens terminologi. Vi kommer att ta up UML senare i avsnitt 4.9 (sid 93). I nästa avsnitt ska vi börja utveckla de traditionella struktureringsverktygen *pseudokod* och *flödesschema*.