Algoritmer, simulering och bruteforce

Algoritmer och problem

En ***algoritm*** är en metod som tar emot någon mängd värden, ***indata***, för att producera nya värden, ***utdatan***. Oftast använder vi algoritmer för att lösa olika sorters ***problem*** som uppkommer inom datavetenskapen.

Figur 1: indata kommer in, utdata kommer ut

Ett problem kan exempelvis vara att beräkna den största gemensamma delaren av två heltal A och B, som betecknas $\gcd(A, B)$ (greatest common divisor på engelska). Ett tal N är ett delare till ett tal A om kvoten $\frac{A}{N}$ är lika med ett heltal, dvs $A = N \cdot x$ för något heltal x. Vi beskriver problemet formellt:

Exempel 1. Största gemensamma delaren

Indata: två positiva heltal $B \geq A > 0$.

Utdata: den största gemensamma delaren av A och B, dvs $\gcd(A, B)$.

Lösning

För att en algoritm ska lösa ett problemet behöver den uppfylla två villkor:

1. Algoritmen får inte köra oändligt länge (dvs den måste ***terminera***)
2. När algoritmen terminerar måste den ha beräknat den korrekta utdatan

Till problemet **Största gemensamma delaren** finns en tämligen enkel algoritm som redan den gamle greken Euklides kom på, den så kallade ***Euklides algoritm***.

Euklides algoritm tar två positiva heltal A och B. Sedan, ända tills ett av talen blir 0, subtraherar man det mindre av talen från det större. När ett av talen blir 0 kommer det andra talet vara $\gcd(A, B)$. Vi kan specifiera algoritmen mer exakt med hjälp av ***psuedokod***:

Algoritm 1 Euklides algoritm

Antagande: $B \geq A \geq 0$

```
1: Funktion GCD(heltal A, heltal B)
2:     så länge A > 0
3:         B ← B − A                    ▷ Subtrahera det mindre talet från det större
4:         om A är större än B
5:             byt plats på A och B     ▷ Vi vill att B ≥ A alltid ska vara sant
6:     returnera B
```

När vi använder pseudokod beskriver vi hur en algoritm fungerar på en högre nivå än när vi använder ett programmeringsspråk. Vi bryr oss inte om små detaljer, vi är inte lika explicita, vi bryr oss inte om syntax etc. Denna form är väldigt praktiskt, och låter oss konkretisera våra tankar kring ett problem innan vi spenderar tid på att implementera det på datorn.

Det är fortfarande långt ifrån uppenbart varför algoritmen faktiskt fungerar, eller om den ens terminerar. Vi kan prova att exekvera algoritmen för oss själva, t.ex. med talen 15 och 6.

$$
\begin{aligned}
B = 15, A = 6 &\quad \text{(start)} \\
B = 9, A = 6 &\quad \text{(subtrahera)} \\
B = 3, A = 6 &\quad \text{(subtrahera)} \\
B = 6, A = 3 &\quad \text{(byt plats)} \\
B = 3, A = 3 &\quad \text{(subtrahera)} \\
B = 0, A = 3 &\quad \text{(subtrahera)} \\
B = 3, A = 0 &\quad \text{(byt plats)} \\
gcd(15, 6) = 3 &\quad \text{(terminera)}
\end{aligned}
$$

Algoritmen fungerade onekligen för vårt lilla exempel - 3 är det största heltal som delar både 15 och 6. Eftersom vi är matematiker nöjer vi oss såklart inte med detta, men kanske har vi nu fått ett hum om varför algoritmen är korrekt:

1. Betrakta summan $A + B$. Efter en körning av loopen kommer summan minska med A. Men $A > 0$, eftersom loopen inte ännu avslutats. Alltså kommer summan $A + B$ bli strikt mindre. Men inget av talen är någonsin negativt, så $A + B \geq 0$. Summan kan alltså inte minska i all oändlighet, och för eller senare måste därför A vara 0 så att loopen avbryts.
2. Vi noterar först att $\gcd(B, 0) = B$. Vidare måste vi inse att ett tal D är en gemensam delare till A och B **om och endast om** D är en gemensam delare till talen A och $B - A$. Detta innebär rent konkret att efter varje steg i vår algoritm kommer de gemensamma delarna till talen vara oförändrade. I synnerhet kommer den *största* gemensamma delaren vara densamma.

   Vi vet att variabeln A förr eller senare kommer bli 0 i vår algoritm måste, vi vet att $\gcd(B, 0) = B$, samt att vi har samma gemensamma delare i början av algoritmen som i slutet. Vår slutledningsförmåga konstaterar då att talens största gemensamma delaren helt enkelt är variabeln B:s slutvärde.

Vi har alltså konstaterat dels att algoritmen terminerar, men också att den gör så med rätt svar - algoritmen är alltså ***korrekt***.

Den uppmärksamma läsaren kanske ifrågasatte att vi betraktade problemet som löst utan att ta någon som helst hänsyn till *hur lång tid* algoritmen tar innan den terimerar. Vår algoritm kanske är så ineffektiv att det tar flera dagar att beräkna svaret när A och B är stora. Under nästa lektion kommer vi behandla de verktyg man använder för att beskriva hur effektiv en algoritm är.

Övning 1. Formulera problemet **Polynomrötter**, som går ut på att finna alla reella nollställen till ett polynom.

Övning 2. Bevisa att D är en gemensam delare till A och B **om och endast om** D är en gemensam delare till talen A och $B - A$.

Övning 3. Bevisa att Euklides algoritm terminerar efter högst $A + B$ iterationer av loopen.

Kattisövning (ID: slowgcd)**.** Implementera Euklides algoritm.

Implementationsproblem

I den "enklaste" sortens problem är problemlydelsen så pass specifik att det svåra inte är att komma på lösningen, utan att faktiskt implementera den. Ofta handlar implementationsproblem om att simulera något tidsförlopp.

Exempel 2. Receptet (Programmeringsolympiadens skolkval 2011)

> Du har bestämt dig för att laga mat. För att laga maten behöver du N ingredienser. För varje ingrediens vet du hur mycket du redan har hemma, hur mycket du behöver totalt samt kostnaden för ingrediensen om du måste köpa den. Du skall alltså köpa den mängd av varje ingrediens som du saknar. Uppgiften är att beräkna kostnaden för att laga maten.

Problemet är inte särskilt svårt: för varje ingrediens beräknar hur mycket av varje ingrediens vi måste köpa. Sedan multiplicerar vi detta med priset för ingrediensen, Till slut summerar vi kostnaden för varje ingrediens.

Bara för att lösningen till ett implementationsproblem är uppenbar behöver den absolut inte vara enkel! Att implementera en algoritm kan vara extremt invecklat och öppnar upp för många buggar i ditt program.

Exempel 3. Matt i ett drag (Introduktion till Algoritmer 2014-2015)

> När du tittar tillbaka i dina gamla årgångar av Schacknytt hittar du massvis av schackpussel. Tyvärr inser du att det var alldeles för länge sedan du spelade schack, och även triviala pussel som att hitta matt i ett drag är numera långt över din förmåga.
>
> Men skam den som ger sig. Du inser såklart att du istället kan använda dina nyfunna algoritmiska kunskaper för att lösa problemet genom att göra ett program som hittar det vinnande draget.
>
> Brädet du får uppfyller följande villkor:
>
> Ingen spelare kan göra rockad.
>
> Ingen spelare kan göra en passant.
>
> Vit kan sätta svart i matt i ett drag.
>
> Ditt program ska skriva ut vilket drag vit ska göra för att sätta svart i matt.

Hur löser man ett sådant hemskt problem? Man skriver *bra kod*, kod som är så generell som möjligt, som är så explicit i vad den gör som möjligt. En bra strukturerad lösning till problemet skulle kunna se ut såhär:

```
Boolean Main() {
  B <- inläsning()
  // Prova alla möjliga vita drag
  för varje (r, c) i B {
    om B(r, c) är en vit pjäs {
      D <- Drag(B, r, c)
      för varje d i D {
        B' = GörDrag(B, d)
        // Är draget ogiltigt?
        om ÄrSchack(B', Vit): prova nästa vid-drag

        // Prova alla möjliga svarta mot-drag
        för varje (r, c) i B' {
          om B'(r, c) är en svart pjäs {
            D' <- Drag(B', r, c)
            för varje drag d' i D' {
              B'' <- GörDrag(B', d')
              // Lyckades svart undvika schack?
              om inte ÄrSchack(B'', Svart): prova nästa vit-drag
            }
          }
        }
        // Inget svart drag hjälpte!
        Svara med D
```

```
      }
    }
  }
}

Vector<Drag> Drag(B, r, c) {
  ....
}

Bräde GörDrag(B, D) {
  ....
}

Boolean ÄrSchack(B, Färg) {
  ...
}
```

Kan tyckas enkelt, men att generera drag och verifiera om en pjäs står i schack är förvånansvärt klurigt att få till. Underskatta inte hur tidskrävande implementationsproblemen kan vara.

Bruteforce

Många problem går ut på att prova en stor mängd möjligheter i lösningen. Om uppgiften är att man ska skriva ut antalet heltalslösningar till en ekvation kan det tänkta tillvägagångssättet helt enkelt vara att prova alla möjliga värden på ekvationens variabler och se om dessa uppfyller ekvationen. Detta kallas för ***bruteforce***. Man använder här att en dator är väldigt snabb, så att man behöver tänka mindre i själva lösningen. Vi kommer senare i kursen gå igenom ett antal standardproblem som man inte känner till en bättre lösning än bruteforce till.

Exempel 4. Maximum Clique.

> Du ska bjuda in några av dina $N \leq 15$ vänner på din födelsedagsfest. Tyvärr tycker vissa av dina vänner inte om varandra, och du vill bara bjuda in en delmängd där alla vänner tycker om varandra. Hur många vänner kan du som mest bjuda in?

Detta problem kallas för *Maximum Clique*, och vi har för tillfället inte någon särskilt effektiv algoritm som löser problemet. Vi skulle kunna hitta den maximala delmängden genom att helt enkelt prova *alla* delmängder:

Algoritm 2 Maximum Clique

```
1:  Funktion MaxClique(heltal N, vänskapsmatris A[N][N])
2:      returnera FindMax(0, {}, N, A)
3:  Funktion FindMax(heltal kollade, mängd nuvarande, heltal antal, vänskapsmatris A[N][N])
4:      om kollade = antal
5:          returnera |nuvarande|
6:      svar ← FindMax(kollade + 1, nuvarande, antal, A)
7:      om FriendWithEveryone(kollade, nuvarande)
8:          svar ← max(svar, FindMax(kollade + 1, nuvarande ∪ kollade, antal, A)
9:      returnera svar
10: Funktion FriendWithEveryone(heltal ny, mängd nuvarande, vänskapsmatris A[N][N])
11:     for varje vän v ∈ nuvarande do
12:         om inte A[ny][vn]
13:             returnera false
14:     returnera true
```

Vi provar att konstruera vår delmängd genom att börja på den tomma delmängden och sedan, för varje element, antingen lägga till det eller inte. Denna lösning, att **generera varje delmängd**, är ett vanligt sätt att lösa problem där indata är väldigt litet.

Många tillämpningar av bruteforce är inte lika uppenbara. Det kan t.ex. vara bara en liten en del i ett problem. Låt oss betrakta följande heltalsekvation:

Exempel 5. Heltalsekvation.

Finn alla heltal $0 < x < 10^9$ som uppfyller

$$x = a \cdot s(x)^b + c$$

där

$$|a| \leq 10\,000$$
$$1 \leq b \leq 5$$
$$|c| \leq 10\,000$$

är givna heltal, och $s(x)$ är siffersumman av talet x.

För att lösa ekvationen enklast använder man bruteforce. Det vi gör bruteforce över är inte x, som man skulle kunna tro. x kan nämligen anta alldeles för många värden för att programmet ska bli snabbt nog (en dator klarar av ungefär 10^8 "små" operatoner per sekund). Istället inser vi att ett tal med högst 9 siffror inte kan ha en större siffersumma än 81, nämligen då alla siffror är 9. Det vi ska bruteforca över är alltså siffersumman $s(x)$. Givet en viss siffersumma är x entydigt bestämt eftersom vi kan beräkna högerledet. Då återstår bara att kontrollera om x faktiskt har samma siffersumma som vi antog att talet hade.

Att lösa ett problem med ren bruteforce, det vill säga utan att kombinera med någon mer avancerad teknik eller ytterligare insikter (som ovan, där det svåra var att komma fram till rätt variabel) brukar oftast bara vara möjligt på enklare problem. Generellt blir programmet alldeles för långsamt.

Exempel 6. Köpa Böcker, Programmeringsolympiadens final 2010.

Du ska köpa $1 \leq N \leq 100$ böcker av olika slag och kollar därför runt hos internetbokhandlarna (som är $1 \leq M \leq 15$ till antalet). Varje bok finns hos minst en bokhandel och kan variera i pris mellan de olika bokhandlarna. Dessutom kostar det ett visst belopp i porto att beställa från varje bokhandel, oavsett hur mycket man beställer. Skriv ett program som beräknar det minimala beloppet böckerna kostar dig, inräknat porto. Du kan beställa från hur många bokhandlar som helst.

Vi kan här inte prova att köpa varje bok från varje bokhandel, ty antalet möjligheter för detta blir alldeles för stort 15^{100}. Vi noterar istället att antalet olika bokhandlar är litet (15), och attackerar problemet genom att tillämpa bruteforce över bokhandlarna. För varje bokhandel antingen köper vi eller inte köper böcker från den, vilket vi kan prova alla möjligheter över (2^{15}). När vi sedan ska köpa böcker väljer vi helt enkelt varje bok från den bokhandel som erbjuder den billigast.

Det matematiska elementet

Under kursens gång kommer vi gå igenom mängder av algoritmer, datastrukturer och tekniker vi kan använda oss av för att lösa problem. I slutändan, för de allra svåraste problemen, är det dock den matematiska problemlösningen vi begränsas av. Problem där det är ens intuition och intellekt som är nyckeln. Hur tränar man upp denna? Man löser massvis med svåra problem som är precis ovanför ens gräns, och sitter med dom tillräckligt länge.

Exempel 7. Spam Filter, KTH Challenge 2015.

Givet är en binär sträng S med högst 100 000 tecken, samt ett tal $1 \leq k \leq 100$. Hitta en delsträng S av längd minst k, som har så hög *andel* ettor som möjligt.

Vad är nyckelinsikten här? Jo, antag att den bästa delsträngen har längd L. Då kommer en av Ls halvor att ha minst en lika hög andel ettor som L självt. Med detta resonemang inser vi enkelt att vi aldrig behöver leta efter en delsträng större än $2k$ – annars finns ju en sträng med hälften så stor längd och samma andel ettor. Vi kan därmed för varje längd mellan k och $2k$ gå igenom strängen och hitta den bästa strängen med denna längd.

Exempel 8. FreeCell, KTH Challenge 2013.

> I ett solitärspel har vi en korthög med $K \leq 100$ kort i valörerna $1, 2, ..., K$ som vi vill flytta över till en annan hög *i samma ordning*. Till vår hjälp har vi $N \leq 5$ stycken fria celler och $M \leq 5$ tomma högar.
>
> Vi får alltid flytta det översta kortet i en hög eller en cell till en fri cell, men vi får bara lägga det i en hög om högen antingen är tom eller har valör $v + 1$ och vårt kort har valör v.
>
> Givet N, M och K, avgör om du kan flytta den ursprungliga högen till din nya hög.

Problemet är hyfsat matematiskt, och en explicit lösning går att hitta genom att resonera induktivt. Som exempeldata fick man dock att det maximala antalet kort man kunde flytta med 3 fria celler och 1 tom hög är 8. Den vane programmeraren ser snabbt att $(N+1)2^M = (3+1)2^1 = 8$, och gör helt enkelt den vilda gissningen att $(N+1)2^M$ är det maximala antalet kort man kan flytta för *alla* N och M. Det visade sig vara korrekt (sann historia från KTH Challenge 2013).

Att tävla i programmering

Vi avslutar lektionen genom att diskutera lite strategi och hur man går tillväga för att komma på en lösning till ett problem och implementera det i ett tävlingssammanhang.

Steg 1: Identifiera uppgiften. Börja tävlingen med att kort skumma igenom varje problem. Om du hittar ett problem du kan lösa på ≤ 10 minuter, gör det. Gå sedan igenom samtliga uppgifter och läs dem lite nogrannare. Gör små anteckningar kring vad problemet handlar om, lösningsidéer, uppskattningar på hur lång tid problemen tar att koda. Helt enkelt, bilda dig en uppfattning om varje problem.

Steg 2: Lös problemet. Så, du har löst alla de enkla problemen, uppskattat vilket kvarvarande problem som är lättast och ska försöka lösa det. Börja med att ta fram en teoretisk lösning på problemet. Bra frågor att fokusera kring är:

- Hur stort är indata?
- Hur snabb måste din algoritm vara?

Oftast vid problemlösning kommer man väldigt enkelt på *någon* algoritm som löser problemet, men kommer fram till att den är för långsam. Oftast är det lättare att jobba utifrån en för långsam algoritm, och sedan snabba upp den. Frågor att fokusera kring då kan vara:

- Kan jag en algoritm som löser något delproblem snabbt?
- Kan jag använda bättre datastrukturer i min lösning?
- Kan jag använda någon algoritmisk teknik (t.ex. bruteforce) för att snabba upp lösningen?
- Kan jag förenkla ett delproblem genom något antagande?

Målet med den här kursen är att bygga upp en reportoar av algoritmer och datastrukturer vi kan tillämpa för att skriva effektiva lösningar på problem, samt att få en intuition för när en viss algoritmisk teknik går att använda på ett problem. Ibland räcker dock inte det till, och man måste ta till den stora släggan: sin matematiska intuition. I slutändan har de flesta problemen element

av ren matematisk problemlösning, och då finns det inte mycket mer man kan göra än att försöka hitta de kritiska observationerna som möjliggör en lösning. Frågor som kan guida en då är:

- Vad gäller i problemet? Finns det några egenskaper vi kan visa?
- Kan vi lösa några små instanser för hand? Ser vi några mönster?
- Vad säger vår magkänsla?

Till skillnad från matematiken behöver vi inte skriva formella bevis när vi ska skicka in en lösning. Vi behöver inte motivera "uppenbaraöbservationer för någon annan än oss själva. Ibland kan vi inte ens det, men vår magkänsla säger ändå att lösningen är korrekt.

Steg 3: Implementera algoritmen. Kvar återstår att koda vår lösning och skicka in den. Att koda bra lösningar snabbt är en konst som man blir bättre på endast genom att koda många problem, och varje gång fundera på hur man skulle kunnat skriva enklare och mer elegant kod. När man skickar in en lösning finns några möjligheter:

- Godkänd – du är färdig! Gå tillbaka till steg 1.
- Ditt program är för långsamt. Gå tillbaka till steg 2.
- Ditt program kraschar. Dessa problem är oftast väldigt enkla att hitta.
- Ditt program får fel svar – den värsta möjligheten.

Fel svar är varje programmerares bane. Inte bara har man sannolikt ingen aning om vad som är fel i ens lösning, man blir helt plötsligt tveksam till om ens algoritm ens är korrekt. Gjorde man ett antagande för mycket? Var formeln man skrev fel? I det här fallet har man inte så mycket annat val än att läsa igenom sin kod för att hitta uppenbara fel (buggar), och att testa sin lösning och algoritm mot många små testfall. Kanske kan man om man funderar ett tag finna motexempel till sin algoritm.

Sammanfattning

- En *algoritm* är en procedur som löser ett *problem* genom att transformera *indata* till *utdata*.
- En algoritmen löser ett problem om den *terminerar* och *ger korrekt svar*.
- Vi kan fylla ut våra tankar med *pseudokod* innan vi börjar programmera en algoritm på riktigt.
- För *implementationsproblem* behövs ingen intelligent problemlösning, utan istället krävs kodningserfarenhet och försiktighet.
- Små problem som involverar många val kan vi *bruteforca* prova alla möjliga val på.
- Att använda bruteforce på en liten del av ett problem kan förenkla resten av problemet.

Hemtal

Kattisinlämning 1 (id: okvir)**.** Lätt

Implementationsuppgift.

Kattisinlämning 2 (id: minimumscalar)**.** Lätt

Problemlösningsuppgift.

Kattisinlämning 3 (id: sjecista)**.** Medel

Problemlösningsuppgift.

Kattisinlämning 4 (id: equalsumseasy)**.** Medel

Bruteforceuppgift.

Kattisinlämning 5 (id: 2048)**.** Svår

Implementationsuppgift.

Kattisinlämning 6 (id: friends)**.** Svår

Bruteforceuppgift.