Rekursion och induktion för algoritmkonstruktion

Att lösa ett problem *rekursivt* innebär att man uttrycker lösningen i termer av samma typ av problem som dock måste vara i någon mening enklare. Man delar alltså upp problemet i ett eller flera delproblem (av samma typ), löser dessa (på samma sätt) och kombinerar sedan lösningarna av delproblemen till en lösning av ursprungsproblemet.

Vi skall demonstrera hur det går till genom ett antal exempel. Avsnitt märkta med asterisk (*) kan överhoppas.

Exempel 1: Beräkning av fakultet

Iterativ definition:

$$n! = n(n-1)(n-2)\ldots 2 \cdot 1$$
$$0! = 1$$

Iterativ beräkning:

```
public static int fac(int n) {
   int p = 1;
   for (; n>1; n--)
      p = p*n;
   return  p;
}
```

Rekursiv definition:

$$n! = \begin{cases} 1 & \text{om } n = 0 \\ n(n-1)! & \text{om } n > 0 \end{cases}$$

Rekursiv beräkning:

```
public static int fac(int n) {
   if (n == 0)
      return 1;
   else
       return n*fac(n-1);
}
```

Förutom i det så kallade *basfallet* (dvs när $n = 0$) så anropar metoden `fac` alltså sig själv med ett argument som är 1 mindre den egna parametern.

Antag att vi anropar med `fac(3)`. Java kommer då behöva beräkna `3*fac(2)`. Då behöver man först beräkna `fac(2)`. Detta görs genom uttrycket `2*fac(1)` som, först, måste beräkna `1*fac(0)`. Vid anropet `fac(0)` hamnar vi på ett basfall och returnerar 1 vilket gör att uttrycket `1*1` kan beräknas osv. Det hela kan illustreras på vidstående sätt.

```
fac(3):
   3*fac(2)
      2*fac(1)
         1*fac(0)
         1*1
      2*1
   3*2
6
```

Det finns alltså, som mest, fyra stycken ”`fac`-uppväckningar” igång samtidigt var och en med sitt eget värde på `n`.

Observera att det allid måste finnas minst ett basfall och en beräkning alltid måste ”bottna” i ett basfall. Om man anropar metoden ovan med ett negativt argument kommer man inte att basfallet (i alla fall inte på något vettigt sätt)

Exempel 2: Beräkning av x^n - första version

Vi skall beräkna x^n, n heltal ≥ 0, med upprepade multiplikationer.

Iterativ definition:

$$x^n = x \cdot x \cdot x \cdot x \cdot \ldots \cdot x$$

Iterativ beräkning:

```
public static double power(double x, int n) {
   double p = 1.;
   for (int i = 1; i<=n; i++) {
      p = p*x;
   }
   return p;
}
```

Rekursiv definition:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ xx^{n-1} & \text{om } n > 0 \end{cases}$$

Övning 1: Implementera ovanstående rekursiva algoritm i en metod

```
public static double power(double x, int n)
```

♢

Övning 2: Skriv en rekursiv metod (dvs utan iteration)

```
public static int multiply(int m, int n)
```

som beräknar produkten `m*n` *utan* att använda multiplikation. Produkten skall alltså beräknas med *additioner*. Du kan förutsätta att `m` och `n` är icke-negativa.

♢

Övning 3: Skriv en rekursiv metod (dvs utan iteration)

```
public static int divide(int t, int n)
```

som beräknar hur många gånger `n` går i `t` (dvs en *heltalsdivision*) *utan* att använda division

eller multiplikation. Du kan förutsätta att `t` och `n` är större än 0.

♢

Övning 4: Skriv en rekursiv metod

```
public static double harmonic(int n)
```

som beräknar den harmoniska summan

$$h(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$$

♢

Övning 5: Skriv en rekursiv metod

```
public static int largest(int[] a, int i)
```

som returnerar det största värdet av elementen `a[0]`, `a[1]`, ... `a[i]`.

♢

I ovanstående exempel blir de rekursiva metoderna varken enklare eller effektivare än den iterativa men rekursiva resonemang är ett ändå kraftfullt sätt att lösa problem och hitta effektiva algoritmer.

Att uttrycka sig rekursivt är ofta naturligt i matematiken. Deriveringsreglerna ("derivatan av en summa är summan av derivatorna") är ett exempel på detta. Inte desto mindre brukar rekursion betraktas som svårt när man börjar med det i programmering. Orsaken till detta är säkert att man i de flesta fall lärt sig programmera med iterationer.

Frågor att besvara vid konstruktion av en rekursiv algoritm:

1. Hur kan jag dela upp ursprungsproblemet i mindre problem av samma slag?
2. Hur kombinerar jag lösningarna till delproblemen till en lösning på ursprungsproblemet?
3. Vilka rekursionsterminerande fall är lämpliga? Kommer man alltid till ett sådant oberoende av indata?

Detta är mycket besläktat med induktionsbevis: Vi antar att vi kan lösa problemet för ett eller flera mindre problem. Sedan visar man hur man med hjälp av dessa lösningar kan lösa ursprungsproblemet. Precis som i induktionsbevis så måste man ha minst ett basfall.

Exempel 3: Skriv ut en sträng med n tecken i omvänd ordning

Hur definiera problemet i termer av sig självt?

Induktionsantagande: Antag att vi kan lösa problemet för de $n - 1$ första tecknen i strängen.

Basfall: Att skriva ut *ett* tecken i omvänd ordning. Trivialt.

Eftersom det sista tecknet skall vara först måste vi börja med att skriva det.

Algoritm i pseudokod:

```
Skriv strängen i omvänd ordning
   om längden är 1 eller mindre så
      skriv strängen
   annars
      skriv det sista tecknet
      skriv alla utom det sista
        i omvänd ordning
```

Java-kod:

```
public static void reverse(String s) {
   if (s.length()<=1) {
      System.out.println(s);
   } else {
      System.out.print(s.charAt(s.length()-1));
      reverse(s.substring(0, s.length()-1));
   }
}
```

Övning 6: Skriv en metod

```
public static String reverse(String s)
```

som returnerar en sträng där tecknen kommer i omvänd ordning mot tecknen i `s`. Använd antagandet att vi kan lösa problemet för ut de $n-1$ *sista* tecknen i strängen.

$\diamondsuit$

Det finns ofta möjlighet att välja de mindre problemen på olika sätt. I ovanstående exempel hade vi två naturliga val men vilket vi valde spelar egentligen ingen roll. Nästa exempel visar hur man kan hitta en bättre (effektivare) algoritm genom att välja andra delproblem.

Exempel 4: Beräkning av x^n - bättre version

Den förra versionen av `power` som byggde på definitionen $x^n = x \cdot x^{n-1}$ är ineffektiv. T ex så kommer anropet `power(x,1000)` generera 1000 multiplikationer. Den definitionen är dock inte den enda möjliga. Vi skulle också kunna använda:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ (x^{n/2})^2 & \text{om } n > 0 \text{ och } n \text{ jämn} \\ x(x^{n/2})^2 & \text{om } n > 0 \text{ och } n \text{ udda} \end{cases}$$

Den är rekursiv precis som den förra men i stället för att återföra lösningen av ett problem av storlekn n på ett problem av storleken $n-1$ så använder vi oss av problem av storleken $n/2$ dvs ett betydligt mindre problem.

Javakod:

```
public static double power(double x, int n) {
   if (n == 0)
      return 1.;
   else  {
      double p = power(x, n/2);
      if (n % 2 == 0)           //n jämn
        return  p*p;
      else                      //n udda
        return  x*p*p;
   }
}
```

Anropet `power(x,1000)` kommer att generera följande sekvens av anrop:

```
power(x,1000)   ->   power(x,500)   ->   power(x,250)   ->   power(x,125)  ->
power(x,124)    ->   power(x,62)    ->   power(x,31)    ->   power(x,30)   ->
power(x,15)     ->   power(x,14)    ->   power(x,7)     .>   power(x,6)    ->
power(x,3)      ->   power(x,2)     ->   power(x,1)     ->   power(x,0)
```

Varje ”uppväckning” av `power` utom den sista innehåller en multiplikation (kvadrering eller multiplikation med `x`) vilket innebär att resultatet beräknas med sammanlagt 15 multiplikationer.

Övning 7: Modifiera programmet så att det även hanterar negativa exponenter. Tänk rekursivt!

Övning 8*: Implementera ovanstående algoritm utan rekursion!

Exempel 5: Trava brickor

En mängd med n brickor av olika storlek ligger travade på en plats (A) i storleksordning med den största underst. Problemet går ut på att flytta hela traven till en annan plats (C) under iakttagande av följande regler:

1. Endast en bricka får flyttas per gång.
2. En större bricka får aldrig läggas på en mindre.

Till hjälp har man ytterligare en plats (B) som får användas för mellanlagring.

Induktionsantagande: Vi kan lösa problemet med $n-1$ brickor.

Basfall: Flytta en bricka. Trivialt.

Induktionssteg: Vi gör på följande sätt

1. Flytta de $n-1$ översta brickorna till B.
2. Flytta den kvarvarande från A till C.
3. Flytta de $n-1$ brickorna på B till C.

Problemet löses således rekursivt genom att lösa två problem av storlek $n-1$.

Totala antalet brickförflyttningar $b(n)$ ges av följande differensekvation:

$$b(n) = \begin{cases} 1 & \text{om } n = 1, \\ b(n-1) + 1 + b(n-1) & \text{om } n > 1. \end{cases}$$

som har lösningen

$$b(n) = 2^n - 1$$

(Lösningen kan erhållas antingen genom att expandera ekvationen eller genom någon standardteknik för att lösa linjära differensekvationer.)

Algoritmen är således mycket tidskrävande om n är stort men det är den bästa möjliga! Det är lätt att inse att lösningen till ett problem av storlek n faktiskt kräver lösning av två problem

av storlek $n-1$ – först för att frilägga understa brickan och sedan för att få tillbaka alla brickor på den understa på en ny pinne.

Övning 9: Implementera en metod

```
bricklek(char from, char to, char help, int n)
```

som skriver ut hur flyttningen av `n` brickor från `from` till `to` med hjälp av `help` skall göras. En förflyttning från 'A' till 'B' följd av en förflyttning från 'C' till pinne 'B' ska skrivas ut såhär:

```
A -> B
C -> B
```

Observera radbrytningarna!

$\diamondsuit$

Exempel 6: Fibonaccitalen

Fibonaccitalen F_n definieras enligt

$$F_n = \begin{cases} 0 & \text{om } n = 0, \\ 1 & \text{om } n = 1, \\ F_{n-1} + F_{n-2} & \text{om } n > 1. \end{cases}$$

Definitionen ovan kan användas för att skriva en metod som returnerar det n:te Fibonaccitalet:

```
public static long fib(int n) {
   if ( n==0 )
      return 0;
   else if ( n==1 )
      return 1;
   else
      return fib(n-1) + fib(n-2);
}
```

Detta är en korrekt metod men den är hopplöst ineffektiv för stora värden på n.

Påstående: Tiden $t(n)$ det tar att exekvera denna metod växer exponentiellt med n. Mer exakt så är

$$t(n) \approx c \cdot 1.618^n$$

där c beror på dator, programmeringsspråk, ...

Bevis* För att inse det kan vi räkna hur många additioner anropet `fib(n)` utför. För detta antal $t(n)$ gäller:

$$t(n) = \begin{cases} 0 & \text{om } n \leq 1, \\ 1 + t(n-1) + t(n-2) & \text{om } n > 1. \end{cases}$$

Observera likheten med Fibonaccitalen!

Detta är en linjär differensekvation och den homogena ekvationen (som är lika med Fibonaccitalen!) har karaktäristiska ekvationen

$$r^2 - r - 1 = 0,$$

som har lösningen

$$r_{1,2} = \frac{1 \pm \sqrt{5}}{2},$$

dvs. den homogena ekvationen har lösningen

$$F(n) = ar_1^n + br_2^n,$$

där a och b bestämmes ur begynnelsevillkoren.

Eftersom

$$t(n) = -1$$

är en partikulärlösning, kan den allmänna lösningen skrivas

$$t(n) = ar_1^n + br_2^n - 1.$$

Genom att använda begynnelsevillkoren kan a och b bestämmas och ger lösningen

$$\begin{aligned} a =& (1 - r_2)/(r_1 - r_2) \\ b =& -(1 - r_1)/(r_1 - r_2) \end{aligned}$$

Eftersom $r_1 \approx 1.618$ och $r_2 \approx 0.618 < 1$ så ser man att, för stora n, gäller

$$t(n) \approx 1.618^n$$

Antalet additioner växer således exponentiellt och därmed också tiden $t(n)$

QED

Övning 10: Tag reda på hur lång tid anropen `fib(50)` respektive `fib(100)` tar (skulle ta) på din dator.

$\diamondsuit$

Som framgår av övningen är programmet ”inte så bra” för stora n. Programmet är *trädrekursivt* det vill säga varje anrop resulterar i två nya anrop. Detta kan potentiellt ge orimliga exekveringstider. Som följande exempel visar så är det dock inte alltid så. I själva verket är många klassiska effektiva algoritmer trädrekursiva (t.ex. sorteringsalgoritmer, snabb Fouriertransform etc.).

Exempel 7: Sortering

Algoritm 1

Induktionsantagande: Vi kan sortera $n - 1$ element.

Basfall: Vi kan sortera 1 element.

Induktionssteg: Stoppa in det n-te elementet bland de $n - 1$ redan sorterade elementen så att sorteringen bibehålls.

Javakod:

```
public static void sort(double [] a, int n) {
  if ( n > 1 ) {
     sort( a, n-1 );            // sortera de n-1 första
     double x = a[n-1];
     int i = n-2;               // flytta undan
     while ( i>=0 && a[i]>x ) {
         a[i+1] = a[i];
         i--;
     }
     a[i+1] = x;                // lägg in sista
  }
}
```

Detta är den vanliga enkla instickssorteringen.

Tiden att sortera n tal med denna algoritm beror på hur talen är permuterade. Det är dock enkelt att visa att tiden i såväl *genomsnitt* som i *värsta* fall är $\Theta(n^2)$ medan den är $\Theta(n)$ i bästa fall. (Uttrycket "tiden är $\Theta(f(n))$" kan, löst uttryckt, tolkas om att tiden är proportionell mot $f(n)$ dvs $cf(n)$.)

Övning 11: Inse detta!

$\diamondsuit$

Algoritm 2

Induktionsantagande: Vi kan sortera $n/2$ element.

Induktionssteg: Dela mängden i två delar med vardera $n/2$ element, Sortera dessa var för sig och sammanfoga sedan de två sorterade delarna.

Sortera n element

1. dela i två lika stora delar
2. sortera delarna var för sig
3. sammanfoga delarna

Arbetet att sammanfoga de två sorterade delarna är proportionellt mot antalet element.

Analys

Låt $t(n)$ beteckna tiden att sortera n element. Då gäller

$$t(n) = \begin{cases} c & \text{om } n = 0, \\ 2t(n/2) + dn & \text{om } n > 0. \end{cases}$$

Om n är en jämn 2-potens, $n = 2^k$ så gäller

$$\begin{aligned} t(n) =& 2t(n/2) + dn = 2(2t(n/4) + dn/2) + dn = \\ =& 4t(n/4) + dn + dn = \dots \\ =& 2^k t(n/2^k) + dnk = \\ =& nt(1) + dn \log n \end{aligned}$$

dvs tiden är $\Theta(n \log n)$.

Denna sortering brukar kallas *mergesort* (*samsortering* på svenska) och är som nedanstående övning visar väsentligt bättre än instickssorteringen i föregående exempel.

Övning 12: Antag att algoritm 1 och 2 tar lika lång tid för 1000 tal — säg 1 sekund. Hur lång tid tar det för respektive algoritm att sortera 10^6 tal respektive 10^9 tal?

$\diamondsuit$

Vi ger just nu ingen kod för denna sorteringsmetod. Det beror på att den är lite ”knölig” att implementera och inte heller blir så bra för just arrayer. Vi återkommer till den senare för andra datastrukturer.

Exempel 8: Växlingsproblemet

På hur många sätt kan man växla a kronor i 100, 50, 10, 5 och 1-mynt (sedlar)? (t.ex. 90 kronor i 50+4*10, 9*10, 8*10 + 10*1 etc.)

Formulera en lösning av problemet i termer av sig självt. Viktigt att rekursionsfallet/fallen löser ett i någon mening mindre problem: Ordna myntsorterna i någon ordning.

Dela in växlingsförsöken i två grupper:

- de som *inte använder* något mynt av första sorten
- de som *använder* första sortens mynt

Problemets lösning kan nu formuleras:

Antalet sätt att växla a kronor vid användande av n olika sorters mynt är

1. antalet sätt att växla a kronor vid användande av alla utom den första sortens mynt ($n-1$ sorter) *plus*
2. antalet sätt att växla $a-d$ kronor användande alla n sorters mynt (d = 1:a myntsortens valör)

Delproblem 1 är mindre än ursprungsproblemet eftersom det använder färre myntslag och delproblem 2 är mindre eftersom det växlar en mindre summa.

Antag att myntsorterna representeras i en array `change` med värden `{1, 5, 10, 50, 100}` på index $1, 2, 3, 4, 5$.

Ger grundprogram:

```
public static int count( int a, int n ) {
    return count( a, n-1 ) + count( a-change[n], n );
}
```

Vilka specialfall behövs för att undvika oändlig rekursion?

- a kan bli $= 0$ vilket innebär ett lyckat försök (räkna det)
- a kan bli < 0 vilket innebär ett misslyckat försök (räkna ej)
- n kan bli $= 0$ vilket innebär ett misslyckat försök (räkna ej)

Slutlig version:

```
public static int count( int a, int n) {
   if ( a == 0 )
      return 1;
   else if (a < 0) || (n == 0)
      return 0;
   else
      return count(a,n-1) + count(a-change[n],n);
}
```

Detta program är användbart om inte a och n är alltför stora men precis som i Fibonacciexemplet så är tillväxten exponentiell. Programmet kan dock förbättras i detta avseende med hjälp av s.k. *dynamisk programmering* som dock inte beskrives här.

Övning 13: Testkör programmet ovan. Låt programmet läsa in belopp som skall växlas.

♢

Övning 14*: Modifiera programmet så att det även skriver ut de växlingar som görs.

♢

Övning 15: Skriv en metod `static String reverseInt(int x)` som returnerar talet `x` i decimalform men med siffrorna i omvänd ordning dvs så att `reverseInt(12345)` returnerar `"54321"`

♢

Övning 16: Skriv en rekursiv metod `static String longToString(long x, int b)` som returnerar en sträng med de siffror som representerar talet `x` i basen `b`. Lös först problemet under förutsättning att `b`≤ 10. Skriv sedan en metod som fungerar för `b`≤ 16.

♢

Övning 17: Skriv en metod

```
public static String reverseNumbers(Scanner scan)
```

som läser en följd av hela tal från `Scanner`-objektet `scan` och returnerar en sträng med talen talen i omvänd ordning det vill säga med det sist inlästa talet först. Metoden skall inte använda någon array, lista eller sträng och skall bara deklarera en enda variabel. Inläsningen av tal skall avbrytas när det inte längre går att läsa ett heltal.

I returströngen skall varje tal föregås av exakt ett blanktecken.

Exempel: Anropet `reverseNumbers(new Scanner("11 23 31 49 56 611"))` skall returnera strängen `" 611 56 49 31 23 11"`

Skriv sedan en metod `public static void reverseNumbers()` som läser tal från standard input och skriver ut talen i omvänd ordning. Talen skrivs på en enda rad med radbyte efter sista talet.

♢

Övning 18: En *stack* är en mekanism som lagrar ett antal element. Till stacken hör operationerna *lagra nytt element* respektive *ta bort ett lagrat element*. Operationerna kallas traditionellt *push* och *pop*. Stackens väsentliga egenskap är att uttagsordningen är omvänd mot inläggningsordningen dvs *pop* tar ut det som senast lagrades med *push*. Av detta skäl kallas stacken även *LIFO* för *Last In First Out*. (Av motsvarande skäl kallas en *kö* för *FIFO* för *First In First Out*.) Stackmekanismen används bl a för att implementera rekursion.

Denna övning går ut på att implementera en stack som lagrar heltal. Stacken ska dock ha en speciell begränsning: det är förbjudet att lägga ett större tal på ett mindre.

Se specifikation av stacken!

Observera att stacken inte får ha några andra metoder än de här angivna!

Tips: En arraylista är en lämplig struktur för att implementera en stack.

Se till att tiden för metoderna `push` och `pop` är oberoende av stackens storlek!

Ladda ner filen StackPlay.java och implementera metoderna `move` och `getNumberOfMoves` enligt dokumentationen i koden.

Exempel på utskrift från `main`-metoden:

```
Start state
   From: [3, 2, 1]
   To:   []
   Help: []
   Number of moves: 0
End state
   From: []
   To:   [3, 2, 1]
   Help: []
   Number of moves: 7
```

Tag genom mätning och teoretiska beräkningar reda på hur lång tid programmet tar med 10, 20, 30, 50 och 100 tal. Svar i sekunder, minuter, timmar, ...

♢