Lektion 4

Denna lektion ska vi studera *rekursion*.

Principen om induktion

Principen om induktion är ett vanligt sätt att bevisa ett påstående $P(n)$ om ett heltal n. Vi kan t.ex ha påståendet "$P(n)$: summan av de n första positiva heltalen är $\frac{n(n+1)}{2}$".

Man kan bevisa detta påstående i två steg:

- Visa att $P(1)$ är sant.
- Visa att om $P(n)$ är sant, så är även $P(n+1)$ sant.

Dessa två steg bevisar tillsammans att påståendet $P(n)$ är sant för alla heltal $n \geq 1$ och kallas alltså för ett *induktionsbevis*.

Exempel 1. Visa att summan av de n första positiva heltalen är $\frac{n(n+1)}{2}$.

Bevis. Påståendet är uppenbarligen sant för $n = 1$, eftersom $\frac{n(n+1)}{2} = \frac{1 \cdot 2}{2} = 1$.

Antag nu att påståendet är sant för något heltal k, dvs att summan av de k första heltalen är $\frac{k(k+1)}{2}$. Om vi adderar $k+1$ till båda led får vi att summan av de $k+1$ första heltalen är $\frac{k(k+1}{2} + k + 1 = \frac{k(k+1)+2(k+1)}{2} = \frac{k^2+3k+2}{2} = \frac{(k+1)(k+2)}{2}$, dvs formeln är sann även för $k+1$.

Enligt principen om induktion är då påståendet samt för alla heltal $n \geq 1$. □

Att principen faktiskt är sann går att bevisa beroende på inom vilket axiomsystem man valt att konstruera heltalen. Vi kan i alla fall bevisa att den följer av den lite mer intuitiva *välordningsprincipen* – att varje icke-tom delmängd av positiva heltal har ett minsta element.

Sats 1. *Principen om induktion är korrekt.*

Bevis. Antag att för ett visst påstående $P(n)$ gäller att $P(1)$ är sant och $P(n) \Rightarrow P(n+1)$.

Låt $S = \{n \in \mathbb{Z}_+ : P(n) \text{ är falskt }\}$. Vi vill visa att $S = \emptyset$, d.v.s att $P(n)$ faktiskt är sann för alla positiva heltal. Antag att så intet är fallet. Då finns enligt välordningsprincipen ett minsta element $a \in S$. Vi vet att $a > 1$, eftersom $P(1)$ är sann. Då är $a - 1 \geq 1$. Vi får nu två fall.

Om $a - 1 \in S$ var a inte det minsta elementet, så detta kan inte vara fallet.

Alltså är $a - 1$ inte ett element i S. Detta innebär med andra ord att $P(a-1)$ är sann. Enligt antagandet har vi $P(a-1) \Rightarrow P(a)$. Ur detta följer alltså att a inte heller är i S.

Antagandet att S hade ett minsta element är alltså falskt. Den enda möjligheten som är kvar är att S faktiskt är tom, alltså att $P(n)$ är sann för alla positiva heltal. □

Rekursiva definitioner

Induktion är nära förknippat med s.k. *rekursiva definitioner*. En rekursiv definition är ett sätt att definera någon storhet i termer av sig själv. Om vi betecknar summan av de första n positiva heltalen med S_n kan vi t.ex. uttrycka S_n som $S_{n-1} + n$ – vi uttrycker summan av n heltal med hjälp av summan av $n-1$ heltal. Detta självreferrerande är alltså vad som brukar kalls för rekursion.

Den vakna läsaren noterade antagligen att $S_n = S_{n-1} + n$ dock inte duger som en definition av S_n. När vi ska definera S_0 får vi nämligen $S_0 = S_{-1} + 0$. Summan av de -1 första heltalen är dock svårt att definera rent semantiskt. Vad som fattades i vår beskrivning är det så kallade *basfallet*

– ett värde på S_n som vi bestämmer oss för att definera direkt. I vårt fall är det lämpligt att använda oss av basfallet $S_0 = 0$.

Ett antal basfall samt ett rekursivt samband utgör alltså en rekursiv definition.

En annan vanlig rekursiv definition som de flesta är bekanta med är *Fibonaccitalen*. De brukar defineras med basfallen $F_0 = 0$ och $F_1 = 1$ samt rekursionssambandet $F_{n+2} = F_{n+1} + F_n$.

Övning 1. Visa att den rekursiva definitionen av Fibonaccitalen är *entydiga*, d.v.s basfallen tillsammans med rekursionssambandet ger tillräcklig med information för att unikt bestämma värdet på F_n för alla $n \geq 0$. Gör detta med hjälp av induktionsprincipen.

Rekursiva definitioner uppkommer ofta både i matematiken och datavetenskapen, ofta som svar till en uppgift. I matematikproblem försöker man ofta finna ett så kallat *slutet uttryck* från den rekursiva definitionen. Detta är alltså ett uttryck för storheten som *inte* är rekursiv. Vi bevisade i föregående avsnitt att S_n har det slutna uttrycket $\frac{n(n+1)}{2}$.

Exempel 2. Definera en sekvens a_i utifrån den rekursiva definitionen

$$\begin{aligned} a_0 &= 0 \\ a_{n+1} &= 2a_n + 1 \end{aligned}$$

Finn ett slutet uttryck för rekursionen.

Bevis. Vi börjar med att beräkna sekvensens värde för några termer i början:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 2 \cdot 0 + 1 = 1 \\ a_2 &= 2 \cdot 2 + 1 = 3 \\ a_3 &= 2 \cdot 3 + 1 = 7 \\ a_4 &= 2 \cdot 7 + 1 = 15 \end{aligned}$$

Vi kan här börja misstänka att a_n har det slutna uttrycket $2^n - 1$. Vi kan bevisa detta med hjälp av induktion:

$a_0 = 2^0 - 1 = 1 - 1 = 0$, så påståendet är sant för 0. Vidare, om $a_n = 2^n - 1$ är $a_{n+1} = 2(a_n) - 1) = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$.

Enligt principen om induktion gäller alltså $a_n = 2^n - 1$ för alla $n \geq 0$. □

Observera att principen om induktion kan användas även om vi inte börjar med just $P(1)$. Det är inte särskilt svårt att visa, med föregående bevis i hand:

Övning 2. Visa att om $P(n)$ är sant där n är vilket heltal som helst, samt $P(k) \Rightarrow P(k+1)$ för alla $k \geq n$ så är $P(k)$ sant för alla $k \geq n$.

Tips: påståenden om heltalen $n, n+1, n+2, ...$ osv. kan också uttryckas som påståenden om heltalen $1, 2, 3, ...$

Rekursion och programmering

Rekursiva definitioner är en av de viktigaste principer inom programmering. En väldigt stor klass av programmeringsproblem löser man nämligen genom att reducera problemet till samma problem, fast av mindre storlek. Om du ska beräkna summan av de 10 första heltalen kan du först beräkna summan av de 9 första heltalen, och sedan addera 10. Då har du reducerat problemet till att

beräkna summan av de 9 första heltalen istället. Detta kan du i sin tur reducera till att beräkna summan av de 8 första heltalen, osv, ända tills du ska summera de 0 första heltalen. Men detta basfall är enkelt – svaret är helt enkelt 0.

När vi programmerar ligger rekursionen rent konkret i en funktion som anropar sig själv. Den rekursiva definitionen av de n första positiva heltalen, $n!$, kan defineras rekursivt som:

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

I C++ kan vi implementera denna rekursiva definition så här:

```
int fakultet(int n){
    if(n == 0){
        return 1;
    }
    return n * fakultet(n - 1);
}
```

Fibonaccitalen har en lite mer komplicerad definition:

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

Men att vi har två basfall gör saken inte mycket svårare:

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

I programmeringsproblem är rekursionerna oftast inte lika enkla att komma på som i våra små exempel. Basfallen brukar däremot vara tämligen enkla att komma på själv.

Exempel 3. Från Programmeringsolympiadens onlinekval, 2010 kommer uppgiften *Gourmeten* (Kattis-id: *gourmeten*).

> Den franska gourmeten Frank är en väl respekterad gourmet; hans yrke är att gå runt till olika restauranger, äta av deras mat och ge sitt omdöme om restaurangen. Men han bär på en hemlighet: han är egentligen bara intresserad av att äta så mycket som möjligt och i vilken ordning som helst!
>
> Frank förstår dock att en äkta gourmet inte kan äta hur som helst, t.ex. börja med sin dessert och avsluta med en sallad. Därför ber han om din hjälp att ta fram en lista med alla olika sätt att ordna maträtterna under en bjudning, så han kan välja ut den ordning som är finast.
>
> På bjudningen har Frank M minuter på sig att äta. Restaurangen bjuder på N olika rätter som han kan äta hur många portioner som helst av. Varje rätt tar ett visst givet antal minuter att äta. Frank vill äta kontinuerligt under alla de M minuter han har på sig, och han vill hinna äta klart alla rätter han påbörjat. Han vill aldrig påbörja en ny rätt innan han ätit färdigt den förra. Din uppgift är att skriva ett program som räknar ut på hur många olika sätt han kan lägga upp middagen, givet dessa restriktioner.

Indata: Ett heltal M, antalet minuter. Ett heltal N, antalet rätter. N heltal T_i, tiden det tar att äta rätt nummer i.

Utdata: Antalet möjliga sätt Frank kan äta under exakt M minuter.

Bevis. För att göra det enkelt för oss låter vi $a(k)$ vara antalet sätt Frank kan äta om vi har k minuter kvar.

Vi noterar först att om Frank har exakt 0 minuter kan han äta på ett sätt, nämligen genom att inte äta något alls. Man kan tvista om att det egentligen borde vara 0 sätt (är att äta inget verkligen att äta något?), men man brukar traditionellt betrakta det "tomma" sättet som ett giltigt sätt (på samma sätt som den tomma mängden är en delmängd i alla mängder). Om han har mindre än 0 minuter kvar har han redan överskridit sin tid, så då är antalet sätt han kan äta istället 0. Detta innebär att $a(0) = 1$ och $a(k) = 0$, när $k < 0$.

Detta visar sig faktiskt tillräckligt som basfall. För alla tider större än 0 minuter kan vi istället formulera svaret rekursivt.

Antag att vi har $k > 0$ minuter kvar att äta på. Om den sista rätten vi äter var rätt i som tar t_i minuter, så kan vi äta på totalt $a(k - t_i)$ sätt. Vi kan alltså gå igenom varje rätt i och summera $a(k-t_i)$, eftersom *någon* rätt måste vara den sista rätten vi åt. Detta ger oss rekurionssambandet $a(k) = \sum_{i=1}^{N} a(k - t_i)$. Tillsammans med våra basfall kan vi nu beräkna $a(M)$. □

Vi implementerar detta i kod såhär:

```
// indatan
int N, M;
vector<int> t;

int gourmeten(int minuter){
    if(minuter < 0) return 0;
    if(minuter == 0) return 1;
    int svar = 0;
    for(int i = 0; i < N; ++i){
        svar += gourmeten(minuter - t[i]);
    }
    return svar;
}
```

Dekomposition

En speciell sorts rekursion är så kallad *dekomposition* (eng. *divide and conquer*). Här bryter vi ned problemet vi ska lösa i mindre, separata delar. Sedan löser vi delarna för sig och kombinerar svaren på dessa till svaret för det stora problemet.

Exempel 4. Vi har en vektor med n stycken heltal, som har elementen $a[0], ..., a[n-1]$. Vi undrar nu vilket element som är det största av dessa.

Bevis. Vi löser ett mer generellt problem: vad är det största av elementen $a[l], ..., a[r-1]$, för två heltal $0 \leq l < r \leq n$. Om vi betecknar detta med $f(l, r)$ så är svaret till det ursprungliga problemet $f(0, n)$.

Om $r = l+1$ har vi endast ett element, nämligen $a[l]$. Detta är vårt basfall i rekursionen: $f(l, l+1) = a[l]$.

Annars kan vi låta $m = \lfloor (l+r)/2 \rfloor$ och dela upp vår array i $a[l], ...a[m-1]$ samt $a[m], ..., a[r]$. Det största värdet av dessa två delar är ju $f(l,m)$ respektive $f(m,r)$. Det största värdet av hela intervallet måste ju ligga i någon av dessa två delar. Därmed är $f(l,r) = \max\{f(l,m), f(m,r)\}$.

Detta ger oss den rekursiva definitionen:

$$f(l, l+1) = a[l]$$
$$f(l,r) = \max\{f(l, \lfloor (l+r)/2 \rfloor), f(\lfloor (l+r)/2 \rfloor, r)\}$$

som vi implementerar i C++ som

```
int findMax(vector<int>& a, int l, int r){
    if(r == l + 1){
        return a[l];
    }
    return max(findMax(a, l, (l + r)/2), findMax(a, (l + r)/2, r));
}
```

□

Övning 3. Bevisa att den rekursiva definitionen är korrekt med hjälp av induktion.

Övning 4. Visa att tidskomplexiteten för $findMax$-funktionen är $O(r-l)$ med induktion.

Merge sort

En känd algoritm som använder sig utav dekomposition är sorteringsalgoritmen *merge sort*. Algoritmen använder sig av samma princip som vår $findMax$-funktion, i att den delar upp indatan i delar som den rekursivt sorterar, men att kombinera dessa delsvar är lite mer komplicerat.

Antag att vi har två sorterade vektorer a och b. Vi vill kombinera dessa till en sorterad vektor c. Vi inser att det lägsta elementet i c, dvs $c[0]$ måste vara det lägsta av de det lägsta elementet i a, dvs $a[0]$, samt det lägsta elementet i b, alltså $b[0]$. Vi har alltså $c[0] = \min(a[0], b[0])$. Om a däremot är tom sätter vi $c[0] = b[0]$ och vice versa.

Självfallet kan vi nu upprepa denna procedur med de kvarvarande elementen, dvs vi tar bort det lägsta elementet från antingen a eller b och fortsätter med detta tills båda vektorer är tomma. Algoritmen är alltså:

```
vector<int> sortVector(vector<int>& a, int l, int r){
    int m = (l + r)/2;
    sortVector(a, l, m); // sort left half
    sortVector(a, m, r); // sort right half

    //combine the answers
    vector<int> answer;
    int x = l;
    int y = m;
    while(x < m || y < r){
        if(x == m || (y < r && a[y] < a[x])){
            answer.push_back(a[y]);
            y++;
        } else {
            answer.push_back(a[x]);
            x++;
        }
```

```
18      }
19      return answer;
20
21  }
```

Övning 5. Skriv ner funktionens beteende för vektorn $(3, 6, 1, 5, 4, 0, 2, 7)$.

Övning 6. Visa att *sortVector* har tidskomplexitetn $O(n \log n)$, där n är antalet element i vektorn. Det finns två sätt att göra detta på: dels via induktion (ganska svårt), och dels genom att rita upp ett diagram över de olika anropen och hur lång tid kombinationssteget tar i varje anrop.

Det som utmärker dekomposition från vanlig rekursion är just att man *delar upp* sitt problem i två eller flera *oberoende* delar, som man sedan kan kombinera för att lösa det ursprungliga problemet. Vi kommer se fler tillämpningar av dekomposition senare i kursen.

Uppgifter

Övning 7. Bevisa med hjälp av induktion att

$$\sum_{i=1}^{n} i \cdot i! = (n+1)! - 1$$

Övning 8. Bevisa med hjälp av induktion att ett schackbräde av storlek $2^n \times 2^n$ med en ruta borttagen kan täckas med hjälp av triominos (L-formade brickor formade av tre 1×1-rutor).

Övning 9. Bevisa med hjälp av induktion att varje komplett, riktad graf har en hamiltonsk väg (en väg som besöker alla hörn exakt en gång).

Övning 10. Bakåtinduktion Ett komplement till principen om induktion är *bakåtinduktion*. Givet ett påstående $P(n)$ bevisar vi att det är sant för alla $n \geq 1$ genom följande steg:

- Visa att $P(1)$ är sann
- Visa att $P(n) \Rightarrow P(n-1)$ för $n > 1$ (observera minustecknet!)
- Visa att $P(n) \Rightarrow P(n')$ där $n' > n$

Med hjälp av villkor 1 och 3 bevisar man att $P(n)$ är sant för godtyckligt stora n, och använder sedan villkor 2 för att "backa" beviset till mindre n.

Bevisa att om de tre villkoren är sanna så är $P(n)$ sant för alla $n \geq 1$. Tips: välordningsprincipen.