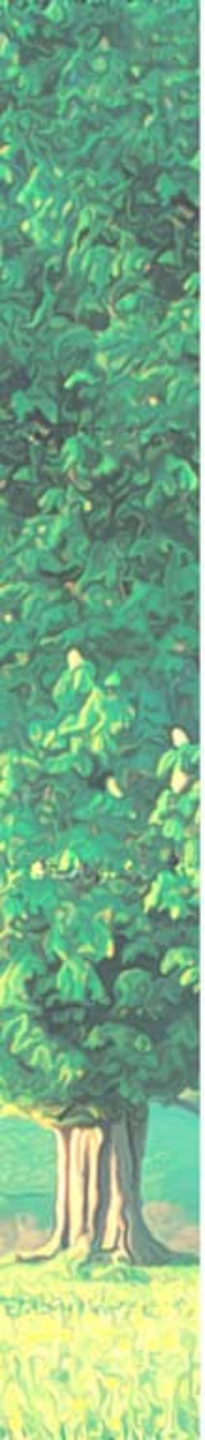
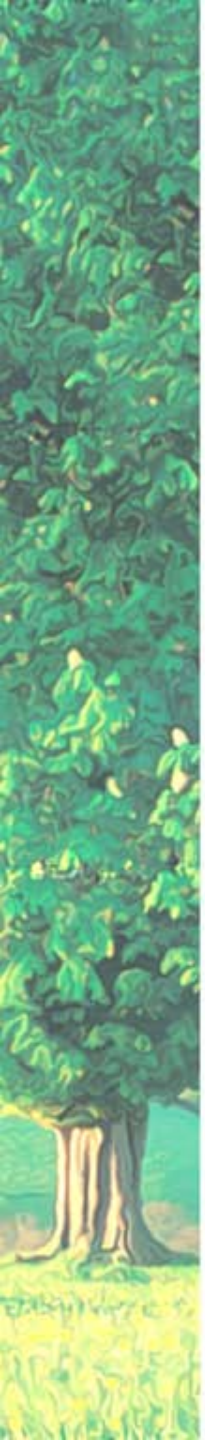


Rekursjon Kap.7

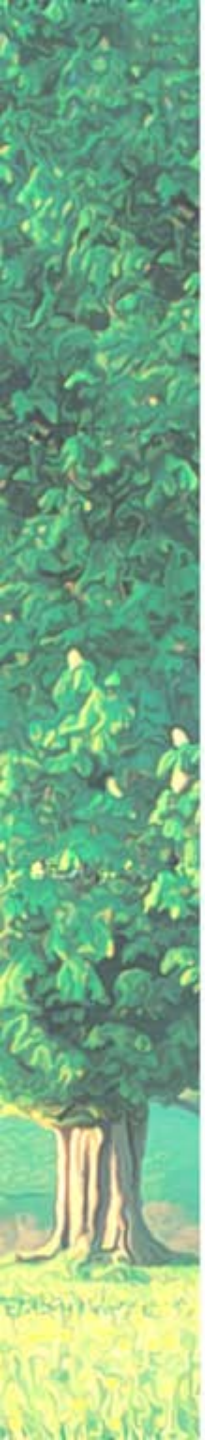
Sist oppdatert 15.02.10

- Rekursjon som programmerings-teknikk



Rekursiv tenkning

- *Rekursjon* er en programmeringsteknikk der en metode kan kalle seg selv for å løse problemet.
- En *rekursiv definisjon er en* definisjon som bruker ord eller begreper som er definert i selve definisjonen.
- I noen situasjoner kan en rekursiv definisjon være en passende måte å uttrykke et begrep på.



Rekursive definisjoner

- Gitt følgende liste av tall:

  `24, 88, 40, 37`

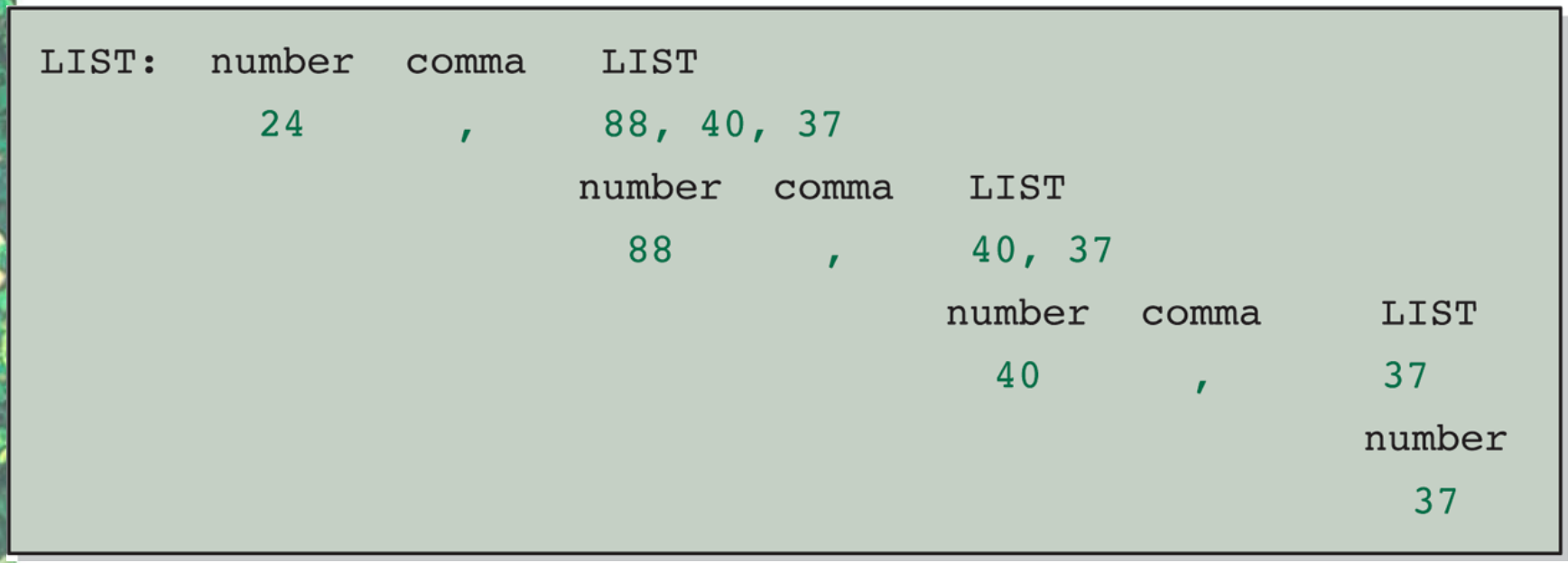
- En slik liste kan defineres rekursivt.:

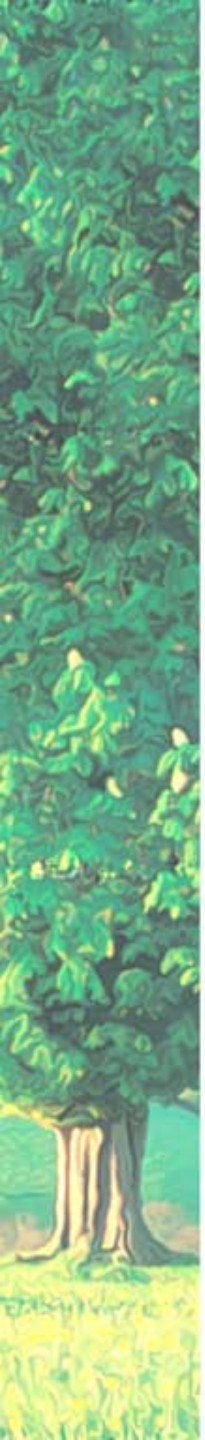
  En LISTE er et: tall

  eller et: tall komma LISTE

- Dvs., en LISTE kan være et tall, eller et tall etterfulgt av et komma etterfulgt av en LISTE. Begrepet LISTE blir brukt til å definere seg selv.

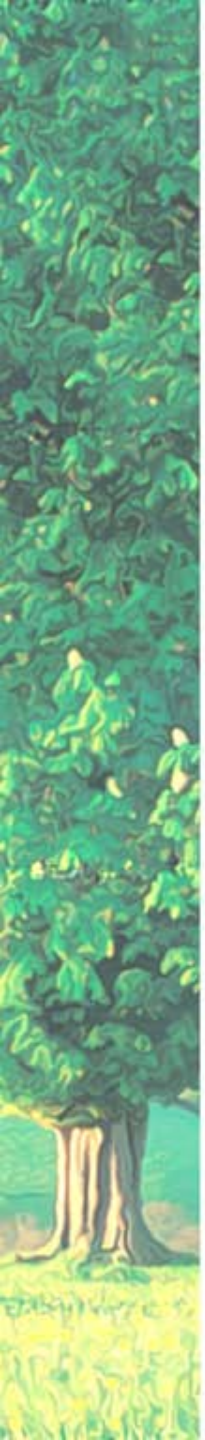
FIGURE 7.1 Gjennomgå den rekursive definisjonen av en liste.





Uendelig rekursjon

- Alle rekursive definisjoner må ha en ikke-rekursiv del.
- Hvis de ikke har, så er det ingen måte å terminere den rekursive stien på.
- En definisjon uten en ikke-rekursiv del forårsaker en *uendelig rekursjon.*
- Dette problemet ligner på en uendelig løkke- --- med definisjonen selv som forårsaker den uendelig løkkegjennomgang.
- Den ikke-rekursive delen er ofte kalt basis-tilfellet.

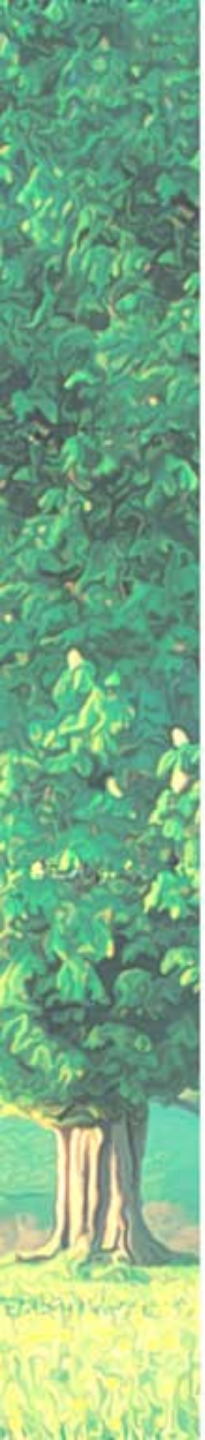


Rekursive definisjoner

- Matematiske formler er ofte uttrykt rekursivt.
- N!, for et positivt heltall N, er definert å være produktet av alle helltall mellom 1 og N inkludert.
- Definisjonen av fakultet kan uttrykkes rekursivt.:

```
1!  =  1
N!  =  N * (N-1)!
```

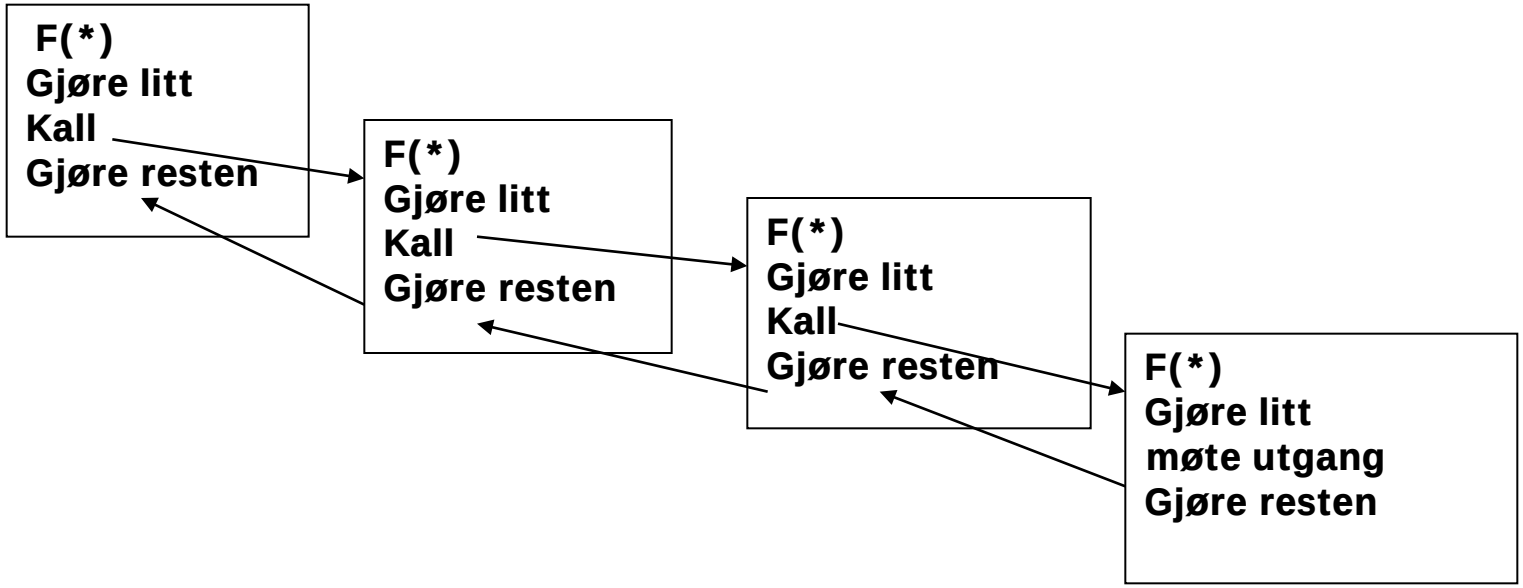
- Et tall vi tar fakultet av er definert ved ledd av et annet tall vi tar fakultet av og dette gjentas inntil basistilfellet er 1!



Rekursiv programmering

- En metode i Java kan kalle seg selv. Hvis den er satt opp på denne måten kalles den en *rekursiv metode.*
- Koden til en rekursiv metode må struktureres for å håndtere både basistilfelle(r) og det rekursive kallet.
- Hvert rekursive kall settes opp med nye parametre og nye lokale variable.
- Når metoden fullføres vil kontrollen returnere til den metoden som kalte den (en annen forekomst av seg selv).

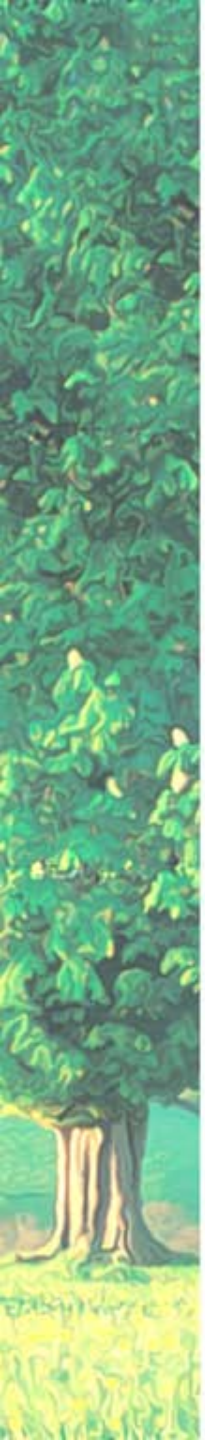
Generelt for rekursive metodekall



Realisering vha systemstabel

- Ved hvert rekursivt kall stables det på en ramme som utgjør plass til:
  - parametre, eventuell returverdi, og returadresse.

  Etter n kall vil vi ha n slike rammer på systemstabelen.
- Ved hver retur fra en rekursiv metode stables det av en slik ramme.
  Gitt at metoden har en returverdi. Når rekursjonen er ferdig, vil svaret ligge på systemstabelen.
  Vi behøver ikke vite hvordan en slik ramme ser ut for å kunne programmere rekursivt.



Rekursiv programmering

- Vi betrakter problemet med å beregne summen av alle tall mellom 1 og N inkludert.
- Hvis N er 5 vil summen være
- 1 + 2 + 3 + 4 + 5
- Problemet kan uttrykkes rekursivt ved: **Summen av 1 til N er N pluss summen av 1 til N-1**

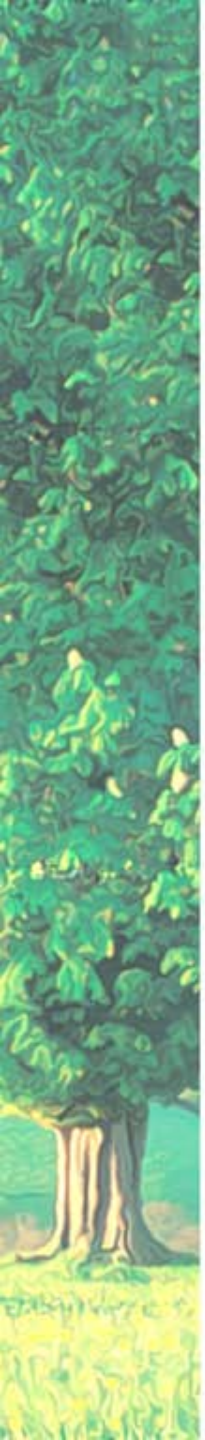


FIGURE 7.2 Summen av tallene 1 til N, definert rekursivt.

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i$$

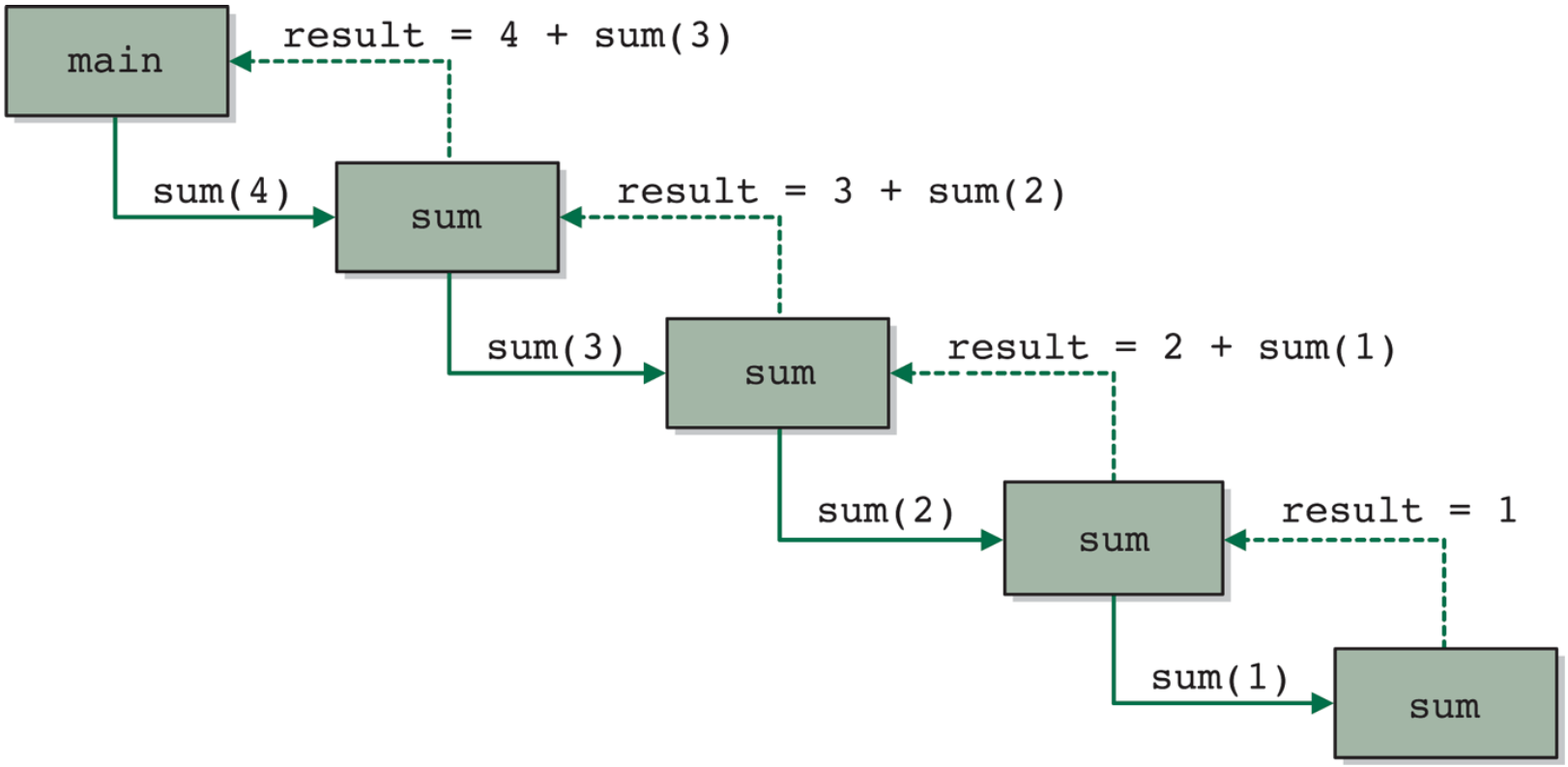
$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

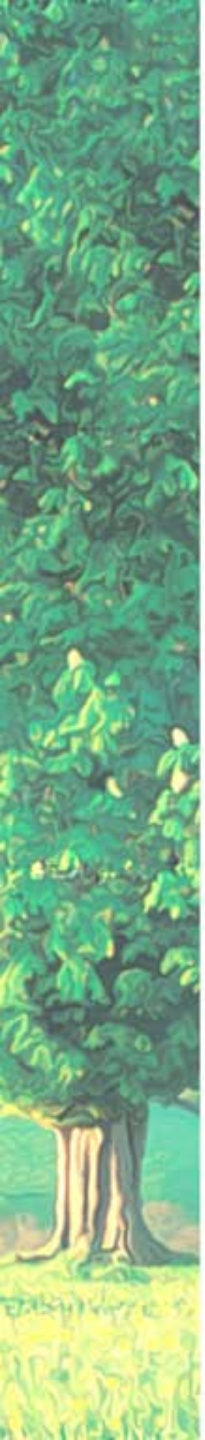
$$= N + N-1 + N-2 + \ldots + 2 + 1$$

Rekursiv programmering

```
public int sum(int n){//Summere
//de n første positive heltall
    int result;
    if (n == 1)
        result = 1;
    else
        result = n + sum(n-1);
    return result;

}//metode
```

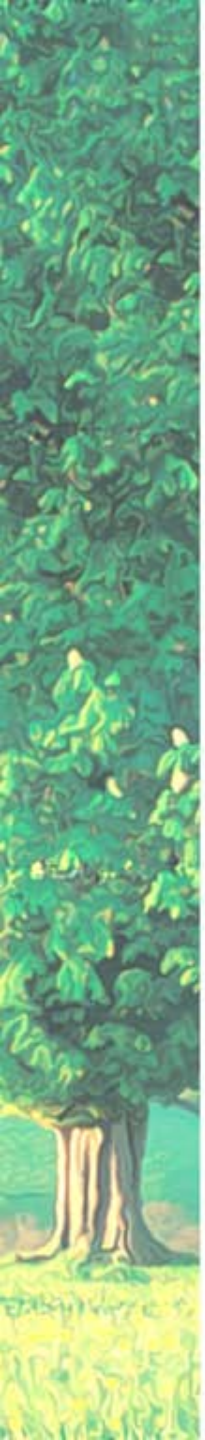
FIGURE 7.3
Rekursive kall til metoden sum.





Krav til rekursive løsninger:

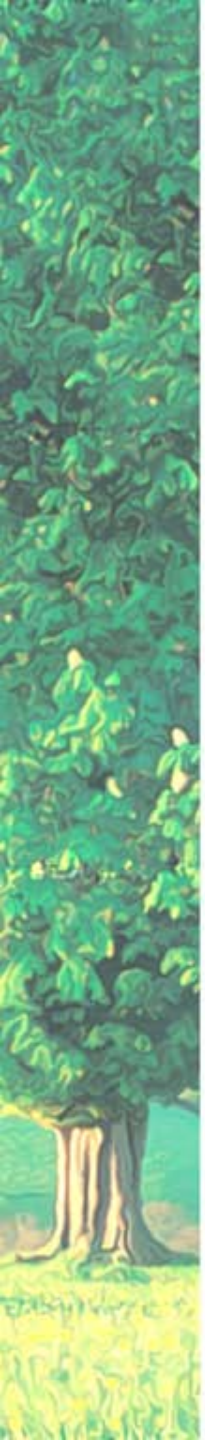
- Delproblemet er mindre enn hele problemet.
- Delproblemet kan enten løses direkte (basistilfelle) eller rekursivt ved rekursive kall.
- Løsningen av delproblemet kan kombineres med løsninger av andre delproblemer for å oppnå løsningen til hele problemet.



Rekursjon som velegnet problem-løsningsteknikk

- Når problemet naturlig kan deles i mindre enklere deloppgaver som ligner på hverandre og videre at vi får enklere og mer elegant kode.

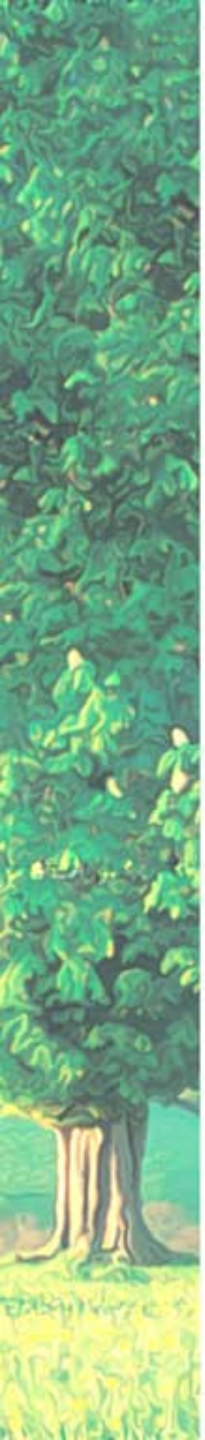
- Pga alle metodekall og bruk av stabel vil det alltid eksistere en ikke-rekursiv løsning som er mer effektiv.



Rekursjon vs. Iterasjon

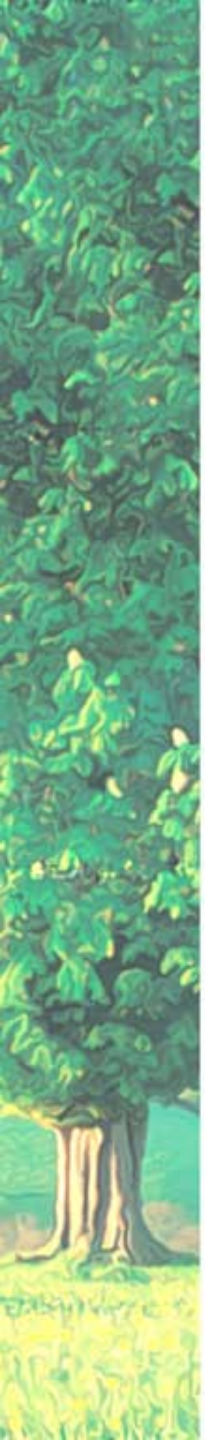
- Rekursjon:
  - Bruk av kontrollstruktur: (if, if else,…)
  - Gjentatt metodekall

- Iterasjon:
  - Bruk av løkkestruktur(for, while,…)



Labyrint(eks. på fornuftig bruk av rekursjon)

- La oss bruke rekursjon for å finne en sti gjennom en labyrint.
- Det fins en sti gjennom labyrinten fra lokasjon x hvis det fins en sti fra en av nabolokasjonene til x.
- Vi merker hver lokasjon vi besøker som “besøkt” og forsøker å finne en sti fra en av de ubesøkte naboene.



Labyrint

- Rekursjon vil bli brukt til å holde rede på stien gjennom labyrinten. Realiseringen bakom rekursjon er en “run-time”-stabel.
- Basis-tilfellene er:
  - En blokkert lokasjon, eller
  - at mål nås.

Listing 7.1

```
class Labyrintspill{//Oppretter en
      nylabyrint, skriver ut opprinnelige
      form, prøver å løse og skriver ut den
      endelige formen.
   public static void main (String[] args){
      Labyrint labyrint = new Labyrint();
      System.out.println(labyrint);
      if (labyrint.gjennomgå(0, 0))
         System.out.println ("Labyrinten
                        ble gjennomgaatt!");
      else
         System.out.println ("Det var ingen
                                mulig sti.");
      System.out.println (labyrint);
   }
  }//metode
```

Listing 7.2

```
   Representerer en labyrint av tegn.
   Målet er å gå fra øvre venstre hjørne
   til nedre høyre hjørne ved å følge en
   sti av 1'enere

class Labyrint{
    private final int FORSØKT = 3;
    private final int STI = 7;

   private int[][] gridd = {
   {1,1,1,0,1,1,0,0,0,1,1,1,1},
   {1,0,1,1,1,0,1,1,1,1,0,0,1},
   {0,0,0,0,1,0,1,0,1,0,1,0,0},
   {1,1,1,0,1,1,1,0,1,0,1,1,1},
   {1,0,1,0,0,0,0,1,1,1,0,0,1},
   {1,0,1,1,1,1,1,1,0,1,1,1,1},
   {1,0,0,0,0,0,0,0,0,0,0,0,0},
   {1,1,1,1,1,1,1,1,1,1,1,1,1}};
```

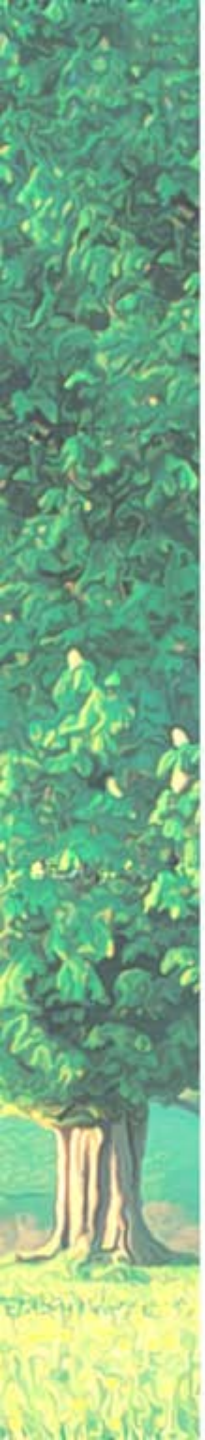
Listing 7.2 (forts.)

```
/**********************************************
  Forsøk på rekursiv
  gjennomgang av labyrinten.
  Setter inn et spesielt tegn
  ('3') for å indikere at
  lokasjon er blitt forsøkt og
  et annet spesielt tegn '7'
  for  at lokasjonen eventuelt
  er en del av løsningen.
  **********************************************
```

```
public boolean gjennomgå(int rekke, int kolonne) {
   boolean ferdig = false;
     if (gyldig (rekke, kolonne)) {
       gridd[rekke][kolonne] = FORSØKT;//cellen er forsøkt
         if (rekke == gridd.length-1 && kolonne ==
                        gridd[0].length-1)
            ferdig = true;  // spillet er løst
         else{
              ferdig = gjennomgå(rekke+1,kolonne);// ned
            if (!ferdig)
              ferdig = gjennomgå(rekke,kolonne+1);// høyre
            if (!ferdig)
              ferdig = gjennomgå(rekke-1,kolonne);// opp
            if (!ferdig)
              ferdig = gjennomgå(rekke,
                                 kolonne-1);//venstre
         }
        if (ferdig) //denne lokasjonene er en del av den
                    endelige stien
            gridd[rekke][kolonne] = STI;
         }
      return ferdig;}//metode
```

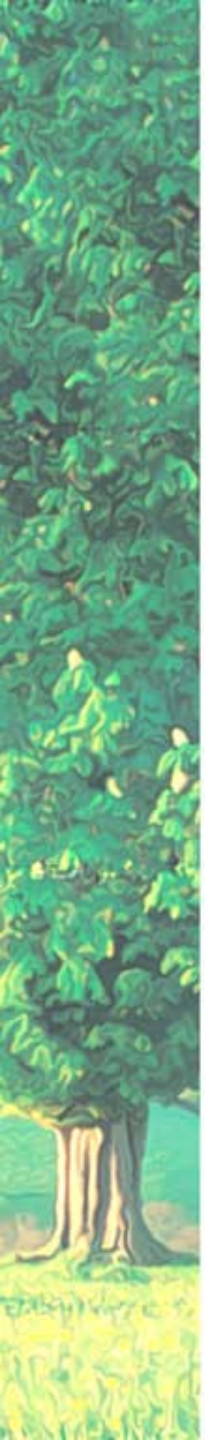
```
  //Avgjør om en spesifisert lokasjon er
    gyldig.

private boolean gyldig(int rekke,int kolonne){
  boolean resultat = false;
  /*sjekker om cellen er innenfor gridd */
   if(rekke >= 0 && rekke < gridd.length
               && kolonne >= 0 && kolonne <
               gridd[rekke].length)

/* sjekker om cellen ikke er blokkert og ikke
  forsøkt tidligere */

  if (gridd[rekke][kolonne] == 1)
      resultat = true;
  return resultat;

 }//metode
```

```
// Returnerer labyrinten som en streng.
public String toString (){
      String resultat = "\n";
      for (int rekke = 0; rekke <
            gridd.length; rekke++){
         for (int kolonne = 0; kolonne <
               gridd[rekke].length;
                     kolonne++)
           resultat += gridd[rekke][kolonne]
                                 + "";
         resultat += "\n";
      }
      return resultat;
   }

  }//class
```



Kjøring

```
111011000111
101110111001
000010101000
111011101011
101000011001
101111110111
100000000000
111111111111

Labyrinten ble gjennomgaatt!

777011000111
307770771001
000070707300
777077707333
707000077003
707777703333
700000000000
777777777777
```

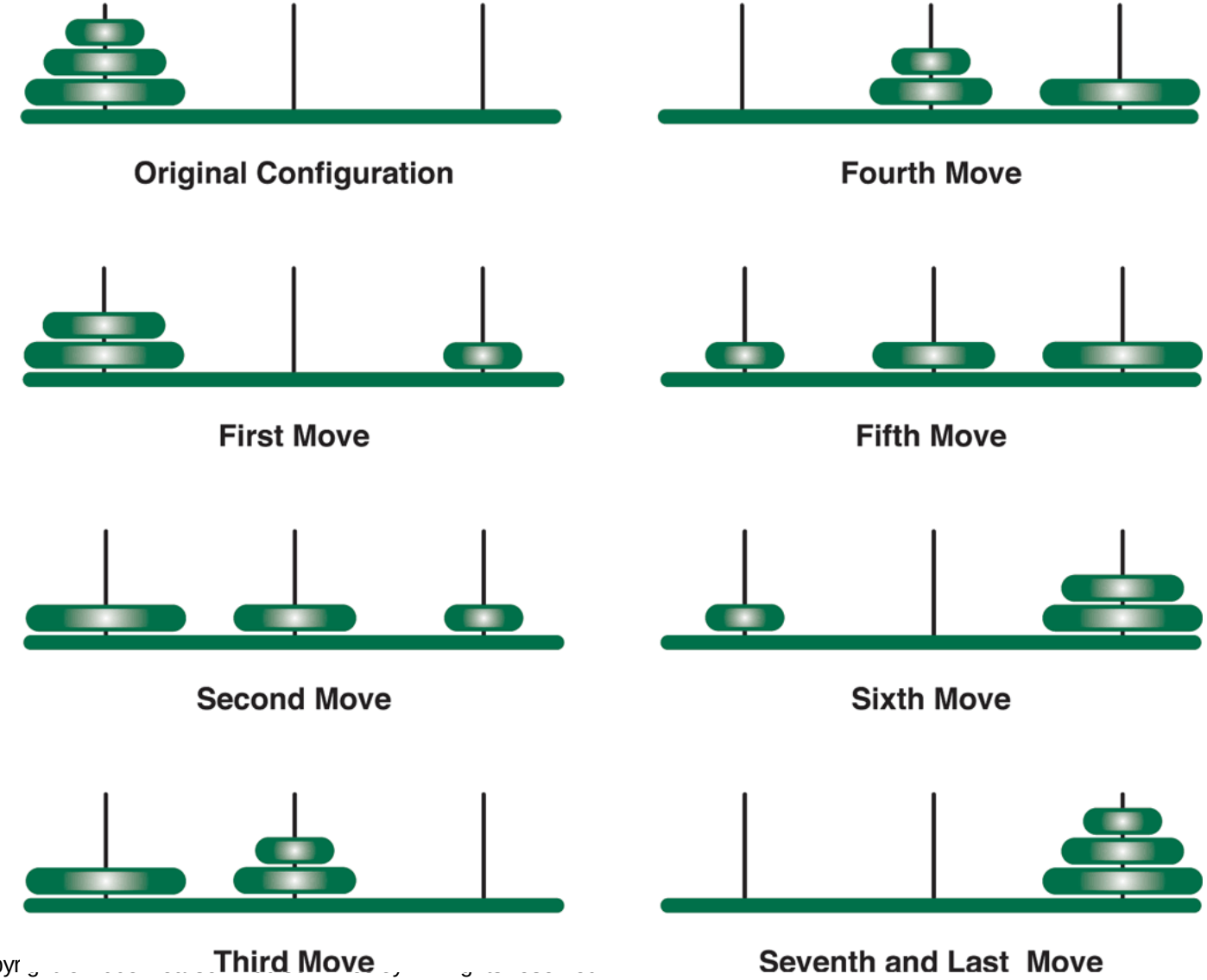


Tårnene i Hanoi (Eks på fornuftig bruk av rekursjon)

- Tre pinner og et gitt antall ringer.
- Ringene har ulike størrelser og er initielt plassert på en pinne med den største ringen på bunnen og mindre ringer plassert oppå etter avtagende størrelse med den minste på toppen.
- Målet er å flytte alle ringene fra en pinne til en annen etter følgende regler:
  - **Kun en ring kan flyttes om gangen.**
  - **En ring kan ikke plasseres over en mindre**.

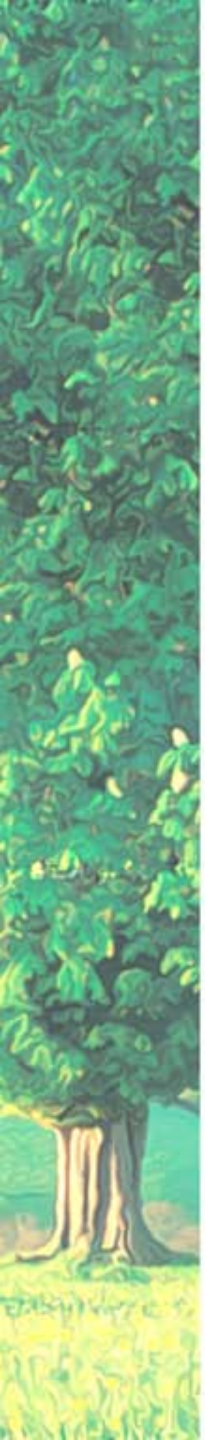
FIGURE 7.6
Tårnene i Hanoi

FIGURE 7.7 Løsning med tre ringer.



Tårnene i Hanoi (NB! Rekursiv tenkning)

- For å flytte en stabel av n ringer fra den originale startpinnen til målpinnen:
  - Flytt de n-1 øverste ringene fra den opprinnelige pinnen til den ekstra pinnen (mellom)
  - Flytt den største ringen fra den opprinnelige pinnen til målpinnen.
  - Flytt de n-1 ringene fra den ekstra pinnen til målpinnen.
- Baistilfellet er når stabelen inneholder kun en ring.



Tårnene i Hanoi

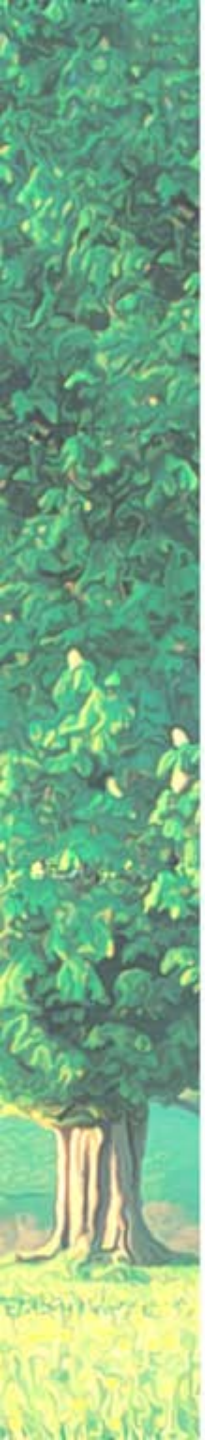
- Antall flyttinger øker eksponensielt når antall ringer øker.
- Enkel rekursiv løsning.
- En iterativ løsning er mye mer kompleks. (Dere finner kanskje en på nettet)

Listing 7.3, 7.4

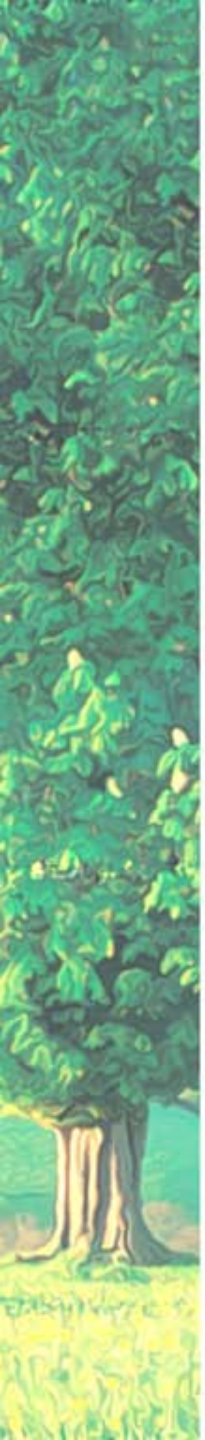
```
//  Tårn i Hanoi
class SpillTårnIHanoi{

   public static void main
                  (String[] args){
        TårnIHanoi tårn = new
        TårnIHanoi (4);
       tårn.spill();
   }//main
}//class
```

```
// Tårn i Hanoi
class TårnIHanoi{
   private int antallRinger;
//Setter opp spillet med spesifisert
  antall ringer.
   public TårnIHanoi (int startRinger)
    antallRinger = startRinger;
   }//metode
//Utfører det første kallet til
  flyttRinger.
    Flytter ringene fra tårn 1 til tårn 3
    ved å bruke tårn 2.
   public void spill(){
   flyttRinger (antallRinger, 1, 3, 2);
}//metode
```

```
//Flytter det spesifiserte antallet ringer
  fra et tårn til et annet ved å flytte
  n-1 ringer, deretter flytte en ring og
  til slutt flytte den de n-1 ringene.
private void flyttRinger (int antRinger,
      int start, int slutt, int temp){
      if (antRinger == 1)// Basistilfellet
         flyttEnRing (start, slutt);
      else{//Rolleskifte fra, til, mellom,
         flyttRinger (antRinger-1, start,
                      temp,slutt);
         flyttEnRing (start, slutt);
         flyttRinger (antRinger-1, temp,
                      slutt,start);
      }
   }//metode
```

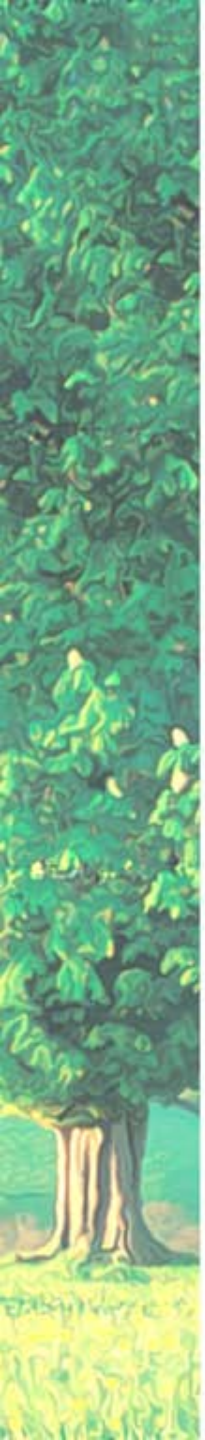


```
/*Skriver ut mellomresultatene
  ved flytting.
private void flyttEnRing(int
                  start, int slutt){
    System.out.println ("Flytt
     en ring fra " + start + ”
     til ” + slutt);
 }//metode
```



Kjøring med 4 ringer, ant flyttinger: $2^4 - 1 = 15$

```
flytt en ring fra 1 til 2
flytt en ring fra 1 til 3
flytt en ring fra 2 til 3
flytt en ring fra 1 til 2
flytt en ring fra 3 til 1
flytt en ring fra 3 til 2
flytt en ring fra 1 til 2
flytt en ring fra 1 til 3
flytt en ring fra 2 til 3
flytt en ring fra 2 til 1
flytt en ring fra 3 til 1
flytt en ring fra 2 til 3
flytt en ring fra 1 til 2
flytt en ring fra 1 til 3
flytt en ring fra 2 til 3
```



Analyse av rekursive algoritmer.

- Når vi analyserer en løkke, bestemmer vi orden av løkkekroppen og multipliserer med antall ganger løkken utføres.
- Rekursiv analyse foregår tilsvarende.
- Vi avgjør orden av metodekroppen og multipliserer den med *orden av rekursjonen* (antall ganger den rekursive definisjonen/kall er brukt).

Analyse av sum (summere de n første pos. heltall).

```
public int sum (int n){
     int result;
     if (n == 1)
        result = 1;
     else
        result = n + sum(n-1);
   return result;
}//metode
```

Analyse av sum

- Den relevante operasjonen i metoden sum er å legge sammen to verdier, O(1).
- n antall ganger rekursive kall. Rekursjonen er O(n).
- Det betyr at hele algoritmen er O(n).

Analyse av Tårnene i Hanoi

I Tårnene i Hanoi er størrelsen på problemet antall ringer. Operasjonen som er interessant er **flytting av en ring**.

- La a_n være antall flyttinger når antallet ringer er n. Med (n-1) ringer vil vi ha a_{n-1} flyttinger og $a_n = 2\ a_{n-1}+1$.
  Se tidligere lysbilde.
- Har vi kun en ring klarer vi oss med en flytting, slik at $a_1 = 1$.

Analyse av Tårnene i Hanoi

- Unntatt for basistilfellet vil hvert rekursivt kall resultere i to rekursive kall ($2\ a_{n-1}$).
  $a_n = 2\ a_{n-1} + 1$.
- Med n ringer får vi: $a_n = 2^n - 1$.
- Med 64 ringer får vi $(2^{64} - 1)$ flyttinger
- For å løse et problem med n ringer gjør vi altså 2^n-1 flyttinger.
- Derfor er algoritmen $O(2^n)$, altså eksponensiell kompleksitet.