Rekursjon

I. TRE AV REKURSIVE KALL,

rekursjonsdybde

terminering – ordning

II. INDUKTIVE DATA TYPER

og Rekursjon over slike

III. "SPLITT OG HERSK" – PROBLEMLØSNING VED REKURSJON (Kap. 10.1.1)

IV. STABEL AV REKURSIVE KALL

iterasjon til rekursjon

rekursjon implementert som iterasjon

V. KORREKTHET

terminering

invarianter (notat til Krogdahl&Haveraaen)

1. En metode “kaller seg selv”

```
/**  en heltallsfunksjon fakultet (factorial (!)) defineres som
*       0! = 1
*       (n+1)! = (n+1) * n!
*/
```

```
/** Algoritme som beregner fact(n)
*   @param n >= 0
*   @return fact(n)
*   @exception  ingen unntak    **/

public int fact( int n ) {
    int r = 1;
    while (n > 1)
        r = r * n;
        n = n – 1;
    return r;
}
```

```
public int fact( int n ) {
        if (n <= 1) return 1;
        else   return  n *  fact( n–1 );
}
```

- fact(n) vil alltid terminere – Hvorfor?

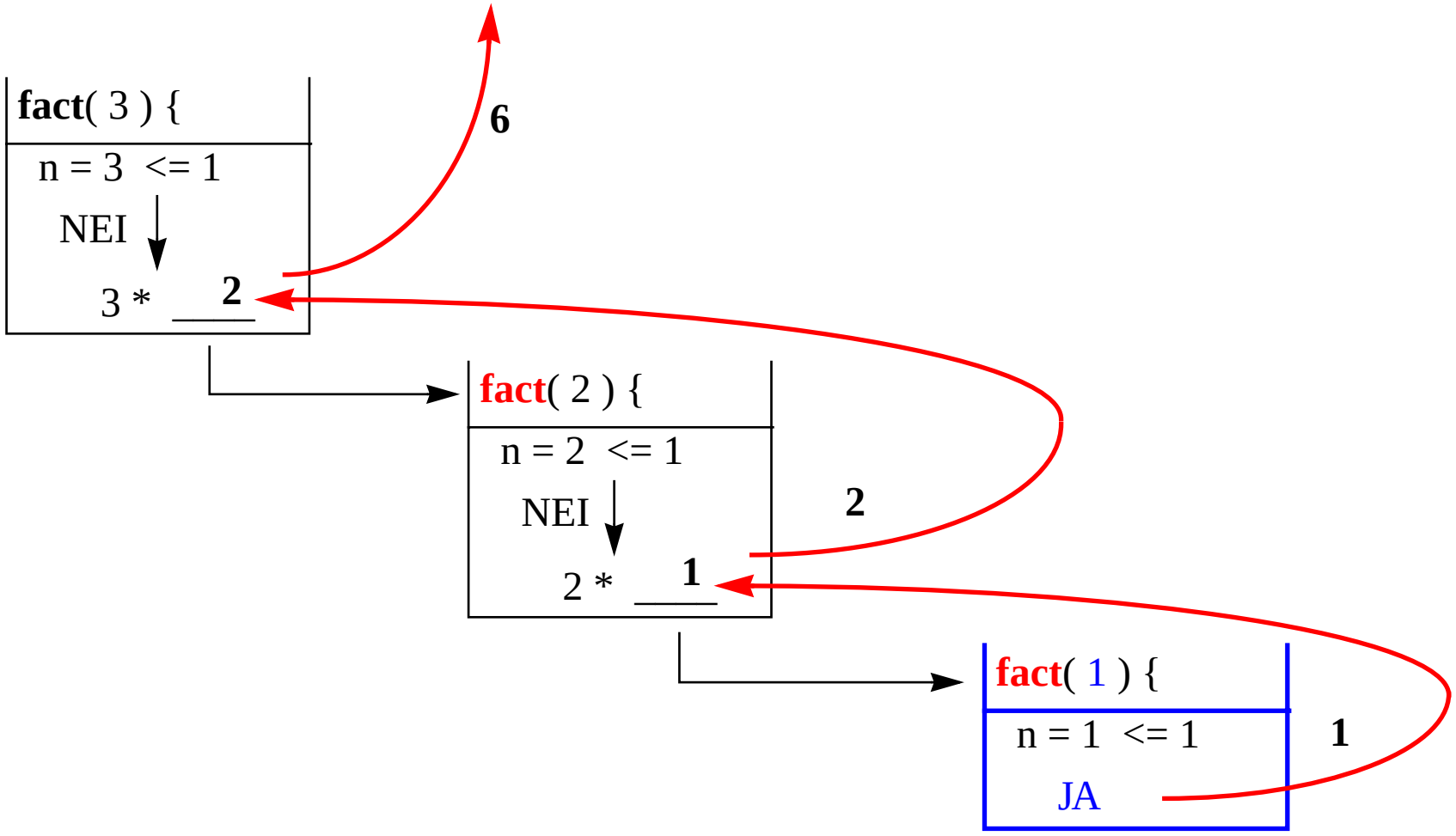
- Hva med:

```
    public int hack( int n ) {
        if (n <= 1) return 1;
        else   return  n *  hack( n+1 );
    }
```

1. En metode “kaller seg selv”

```
/**  heltallsfunksjon fact (!):
*      0! = 1
*      (n+1)! = (n+1) * n!
*/
```

```
public int fact( int n ) {
    if (n <= 1) return 1;
    else   return  n *  fact( n–1 );
}
```



1. Binær Søk

```
/*   finn indeks i A til et element x:
*    @param A  int A[...] sortertint
*    @param x  finn x i A
*    @param l, h søk i A bare fom. l tom. h
*    @return     indeks til x;
*                –1 hvis x ikke finnes
*/
```

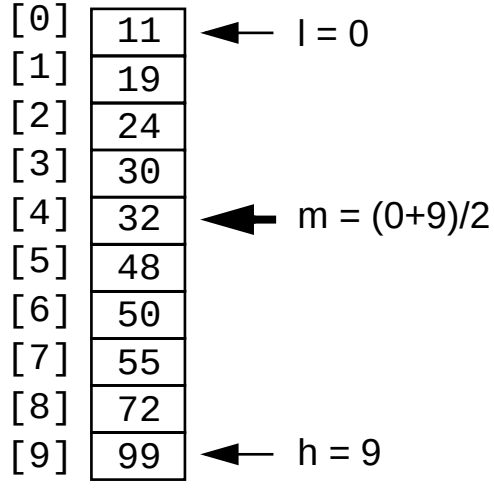
```
int BS(int[] A,x,l,h) {
    m= (l+h) / 2 ;
    if (l > h) return -1;
    else if (A[m] == x) return m;
    else if (A[m] < x) return BS(A,x,m+1,h);
    else return BS(A,x,l,m-1); }
// initielt kall med BS(A, x, 0, A.length–1)
```

Nøkkel er 48

1. kall

binSøk(A, 48, 0, 9)

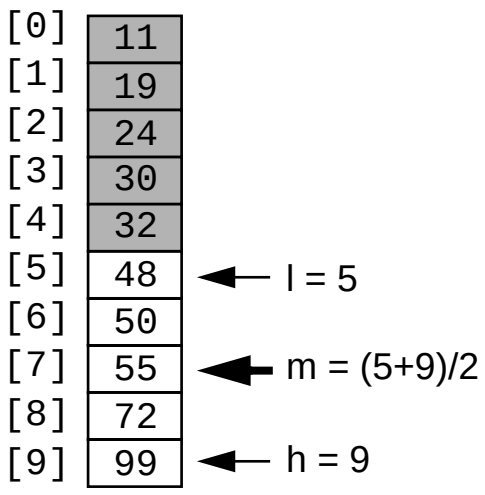
A[]



2. kall

binSøk(A, 48, 5, 9)

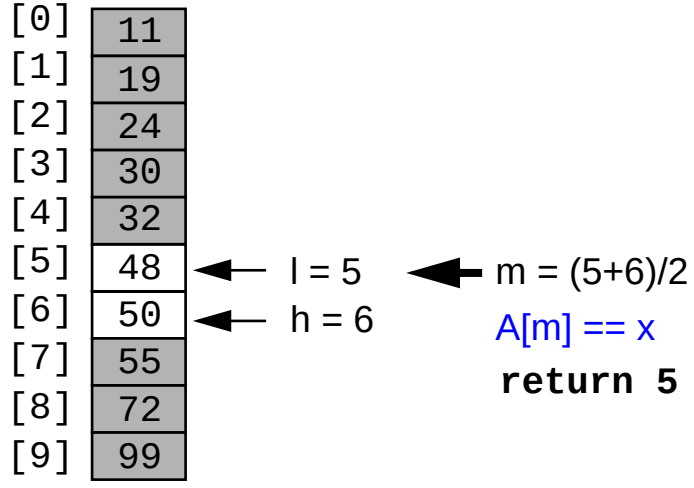
A[]



3. kall

binSøk(A, 48, 5, 6)

A[]



basis tilfelle

1. Rekursjons*tre* og -*dybde*

fact(0) = 0

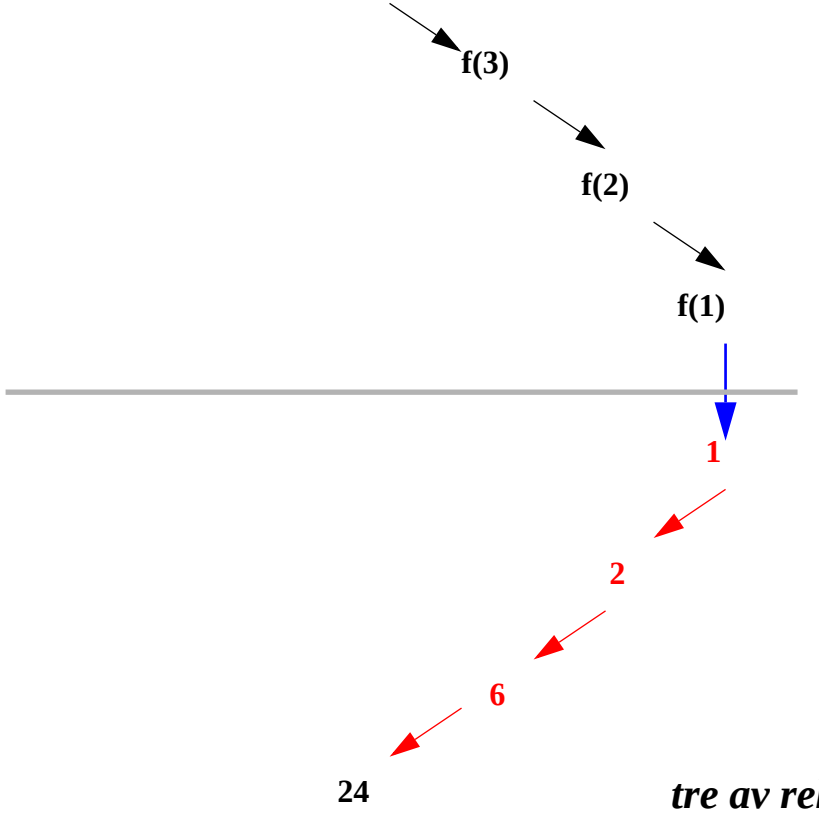
fact(0) = 1

fib(n+1) = (n+1) * fact(n)

```
public int fact(int n) {
   if (n <= 1) return 1;
   else
      return n * fact(n-1);
}
```

returner i **basistilfelle**

ellers beregn rekursivt **enklere** (mindre) deler og sett disse sammen



tre av rekursive kall

f(4) ...?
— **f(3) ...?**
— — **f(2) ...?**
— — — **f(1) ...?**
— — — **> 1**
— — **> 2**
— **> 6**
> 24

rekkefølgen av kall

rekursjonsdybde

#svarte = #røde linjer = *# rekursive kall .....*
inntil basistilfelle er nådd (=høyden av treet)

1. Rekursjons*tre* og -*dybde;* Eks: Fibonacci-tallene

fib(0) = 0

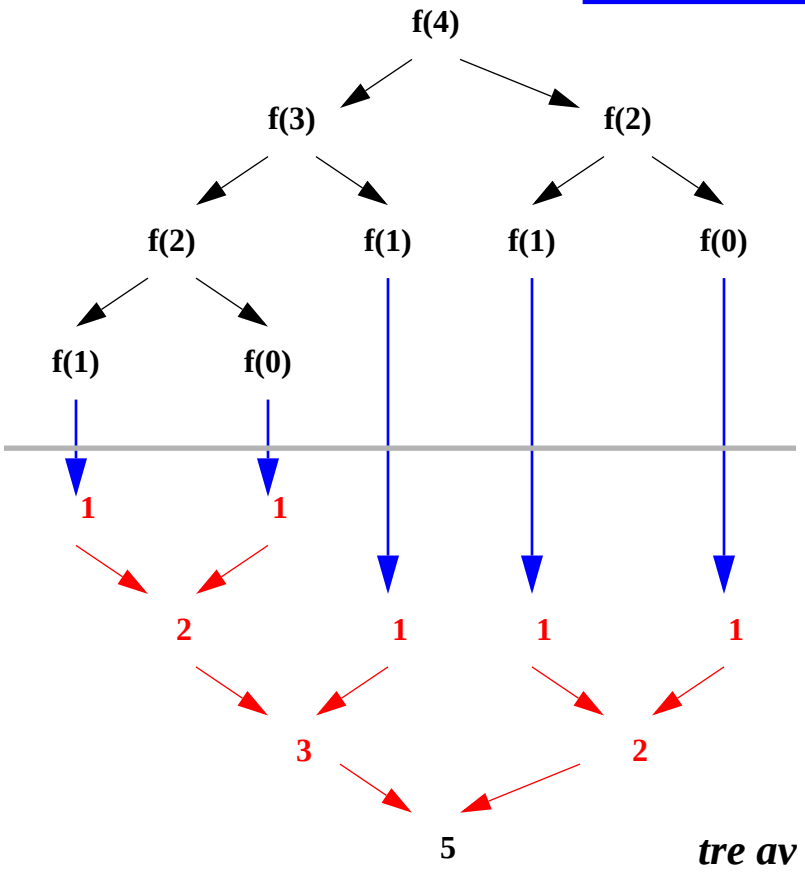
fib(1) = 1

fib(n+2) = fib(n) + fib(n+1)

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

returner i **basistilfelle**

ellers beregn rekursivt **enklere** (mindre) deler og sett disse sammen



tre av rekursive kall

f(4) ...?
— **f(3) ...?**
— — **f(2) ...?**
— — — **f(1) ...?**
— — — **> 1**
— — — **f(0) ...?**
— — — **> 1**
— — **> 2**
— — **f(1) ...?**
— — **> 1**
— **> 3**
— **f(2) ...?**
— — **f(1) ...?**
— — **> 1**
— — **f(0) ...?**
— — **> 1**
— **> 2**
> 5

rekkefølgen av kall

rekursjonsdybde

#svarte = #røde linjer = *# rekursive kall .....*

inntil basistilfelle er nådd (=høyden av treet)

2. Induktive Data Typer *(vilkårlig store men endelige)*

"Strukturell ordering"

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

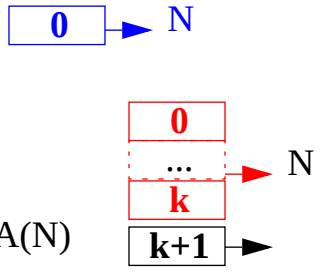
så er: **n+1** et N

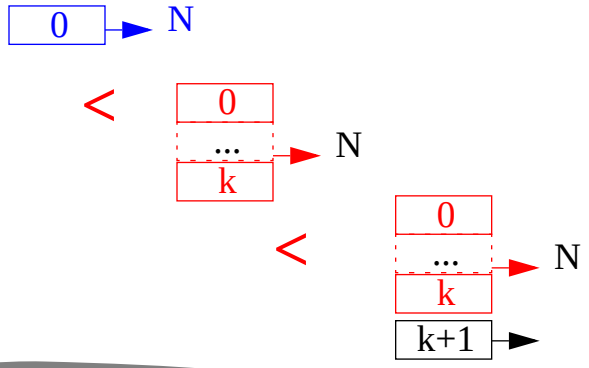
array av N: A(N)

basis **0** -> **N** er A(N)

hvis **[0...k]** -> **N** er A(N)

så er **[0...k,k+1]** -> **N** en A(N)



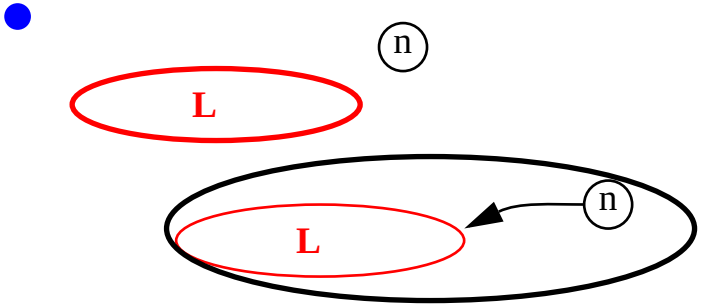


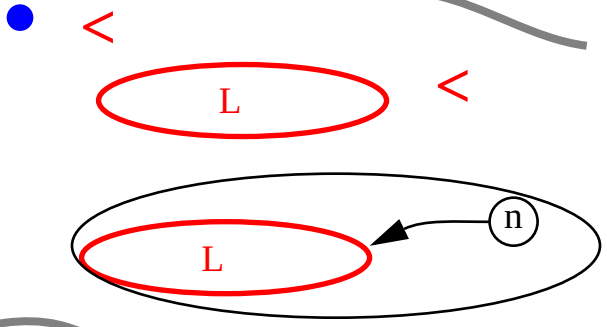
Lister av N: L(N):

basis: **null** er en L(N)

hvis **L** er L(N) og n er N

så er: **(n,L)** en L(N)



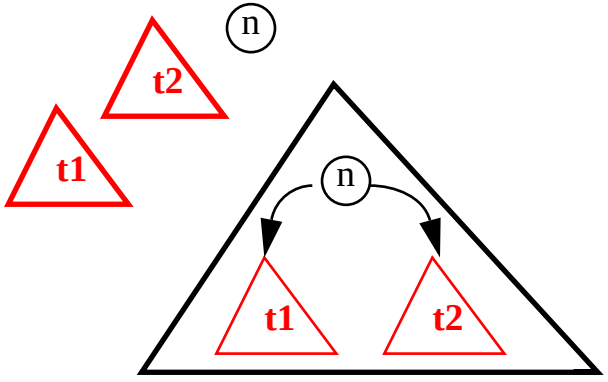


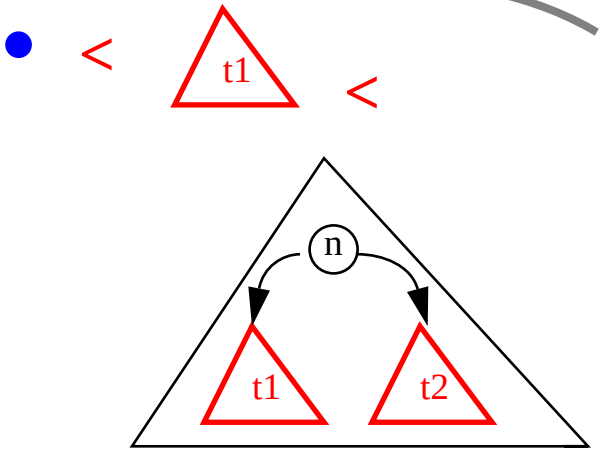
Binære Trær av N: BT(N):

basis: **null** er et BT(N)

hvis **t1**, **t2** er BT(N) og n er N

så er: **(t1, n, t2)** et BT(N)





2. Induktiv oppbygging -> rekursive beregninger

induktiv definisjon *= fra basis og oppover* ***** ***rekursjon*** *= fra toppen mot basis*

Nat
basis: 0
ind: n+1

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

```
int fact(n) {
   if (n <= 0) return 1;
   else return n * fact(n-1);
}
```

Array[Int]
basis: [0] -> Int
ind: [0.. k-1, k] -> Int

```
void inc(int[] A, int k) {
   A[k]++;
   if (k > 0) inc(A,k-1); }
```

```
int sum(int[] A, int k) {
   if (k==0) return A[0];
   else return A[k] + sum(A,k-1); } }
```

Liste[Int]
basis: null
ind: (n,hale)

```
class LS {
   int n;
   LS hale; }
```

```
void inc(LS L) {
   if (L==null) {}
   else { n++;
      inc(L.hale); } }
```

```
int sum(LS L) {
   if (L==null) return 0;
   else  return
         sum(L.hale) + n; }
```

BinærtTre[Int]
basis: null
ind: (left,n,right)

```
class BT {
   int n;
   BT left;
   BT right;}
```

```
void inc(BT B) {
   if (T==null) { }
   else {n++;
      inc(T.left);
      inc(T.right); } }
```

```
int sum(BT T)  {
   if (T==null) return 0;
   else return
         n + sum(T.left) + sum(T.right) ;
}
```

FRAKTALer

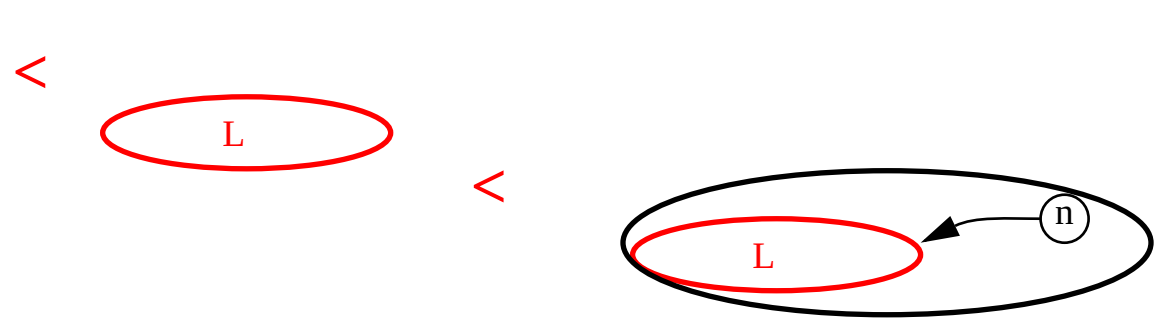
2a. En teknisk bemerkning

Lister av N: L[N]:

basis: **null** er en L[N]

hvis **L** er L[N] og n er N

så er: **(L,n)** en L[N]

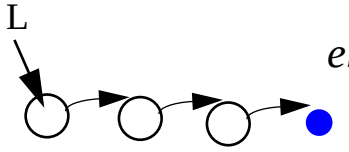


Rekursjon implementert *"utenfra" datastrukturen :*

```
class LN {
    public int hodedata;
    public LN restliste;
    ...  }
```

```
inc(LN L) {
    if (L==null) {}
    else { L.hodedata++;
        inc(L.restliste); }
```

```
int sum(LN L) {
if (L==null) return 0;
else  return sum(L.restliste)+L.hodedata;
}
```



eller "innenfor" datastrukturen :

```
class LN {
   private int hodedata;
   private LN restliste;
   inc() {...}
   int sum() {...}     }
```

```
inc() {
    hodedata++;
    if (restliste != null)
        restliste.inc(); }

inc(LN L) {
    if (L != null)
        L.inc();
}
```

```
int sum() {
    if (restliste != null)
        return restliste.sum()+hodedata;
    else return hodedata; }

int sum(LN L) {
    if (L != null)
        return L.sum();
    else return 0; }
```

3. "Splitt og hersk" (engelsk: Divide and Conquer)

***Rekursjon* som en generell strategi for problemløsning og algoritmedesign**

Gitt en instans **n** *av et problem* **P** *:*

1. *hva gjør jeg når* **n** *er basis tilfelle*
2. *hvordan konstruere løsning for* **n** *utfra løsninger for noen instanser* **mindre enn n**

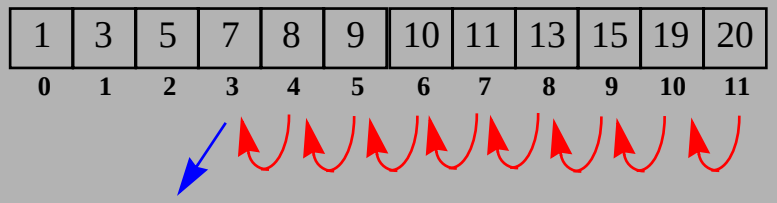
P = finn et gitt element x i en array A

Hvis A *er usortert :*

sjekk A[n]*;*
hvis x ikke er der, lett i A[0...n–1]

```
/*   initielt kall med FE(A, x?, A.length–1)   */
int FE(int[] A, x. k) {
    if (k < 0 )  return –1;
    else if (A[k]==x)  return k;
    else return FE(A, x, k–1);   }
```

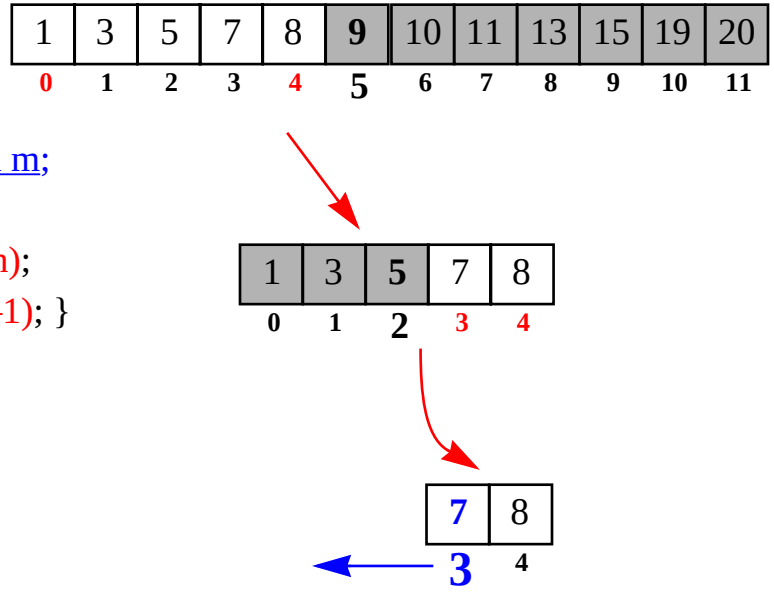
FE(A, 7, 11)



Hvis A er sortert ...

```
/*   initielt kall med
* BS(A, x?, 0, A.length) */
int BS(int[] A, x, l, h) {
   m = (l+h) / 2;
   if (l > h) return -1;
   else if (A[m] == x) return m;
   else if (A[m] < x)
       return BS(A, x, m+1, h);
   else return BS(A, x, l, m–1); }
```

BS(A, 7, 0,11)



3. "Splitt og hersk" (eng: Divide and Conquer)

***Rekursjon* som en generell strategi for problemløsning og algoritmedesign**

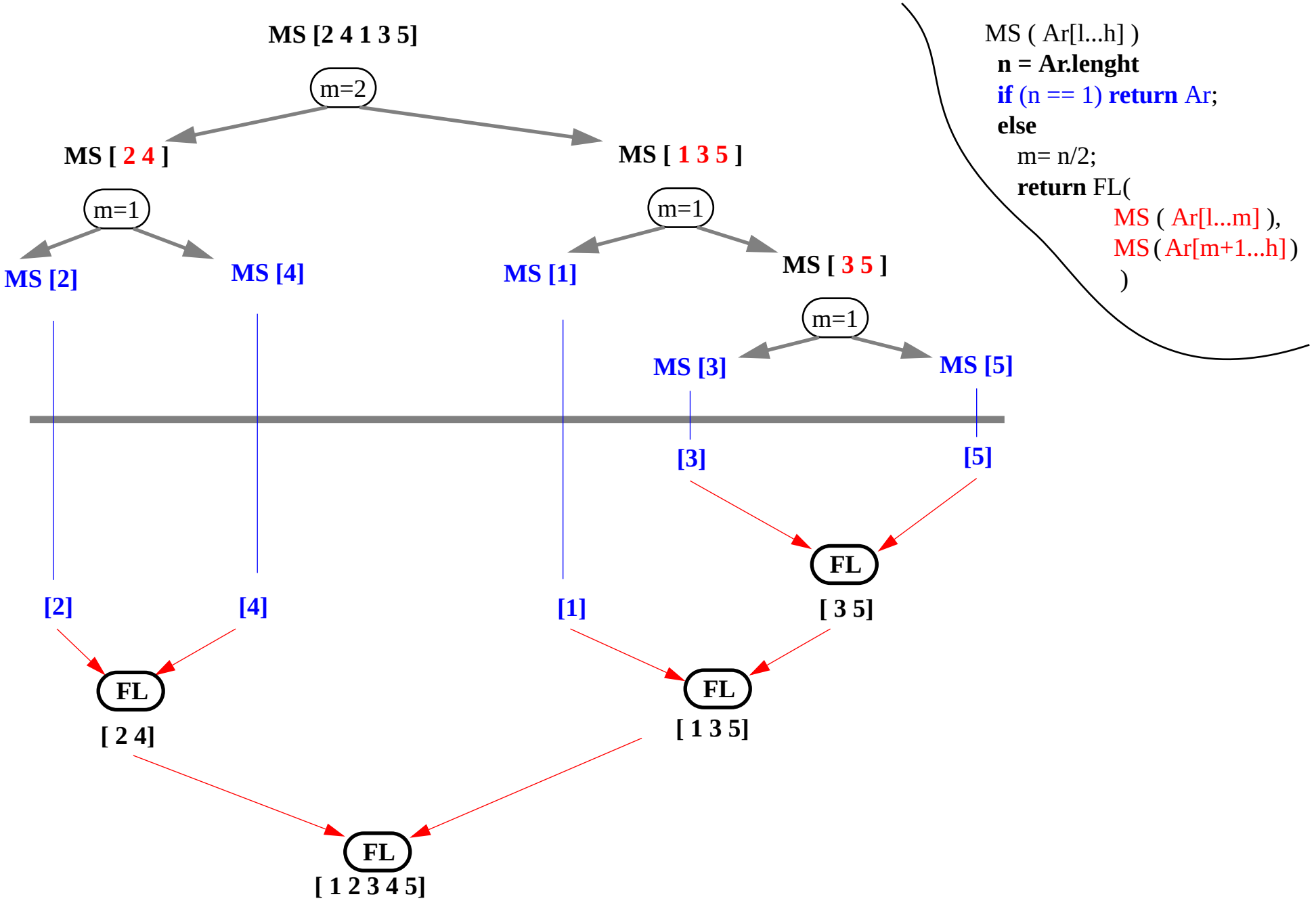
Gitt en instans **n** *av et problem* **P** *:*

1. *hva gjør jeg når* **n** *er basis tilfelle*
2. *hvordan konstruere løsning for* **n** *utfra løsninger for noen instanser* **mindre enn n**

P = sorter input array A

```
/* int[] SS(int[] A,k) {
 * initielt kall med SS(A,0)
 * n = A.length;
 *    if (k==n-1) { return A; }
 *    else {
 *        i= indeksen til minste elementet
 *            i A[k...n-1];
 *        bytt A[k] med A[i];
 *        return SS(A, k+1); } }   */
```

```
/* int[] MergeSort ( int[] A )  {
 *    int n = A.length;
 *    if (n == 1) { return A; }
 *    else {
 *       del A i midten i
 *       t1 = A[0...n/2] og t2 = A[n/2+1...n];
 *       sorter rekursivt begge (mindre) array
 *             r1 = MergeSort ( t1 ) og
 *             r2 = MergeSort ( t2 )
 *       return flettet resultat av disse FL(r1,r2)
 *    }  }

 * FL - fletter to sorterte array i en sortert array */
```


MS [2 4 1 3 5]
m=2
MS [2 4]
MS [1 3 5]
m=1
m=1
MS [2]
MS [4]
MS [1]
MS [3 5]
m=1
MS [3]
MS [5]
[3]
[5]
FL
[3 5]
[2]
[4]
[1]
FL
FL
[2 4]
[1 3 5]
FL
[1 2 3 4 5]
MS (Ar[l...h])
n = Ar.lenght
if (n == 1) return Ar;
else
m= n/2;
return FL(
MS (Ar[l...m]),
MS (Ar[m+1...h])
)

4. Iterasjon til rekursjon

```
/**  @param n > 0
     @return 1+2+...+n */
int sumIter(int n) {
     int res =0;
     while (n > 0) {
          res = res + n;
          n = n–1;
     }
     return res;
}
```

```
/**  @param n > 0
     @return 1+2+...+n */
int sumRek(int n) {
     if (n == 0) return 0;
     else return n + sumR(n-1);
}
```

Generellt, dog ikke 100% riktig:

```
int Iter(int n) {
     res= init;
     while ( fortsett(n) ) {
          res= Kroppen(n,res);
          oppdater(n);
     }
     return res;
}
```

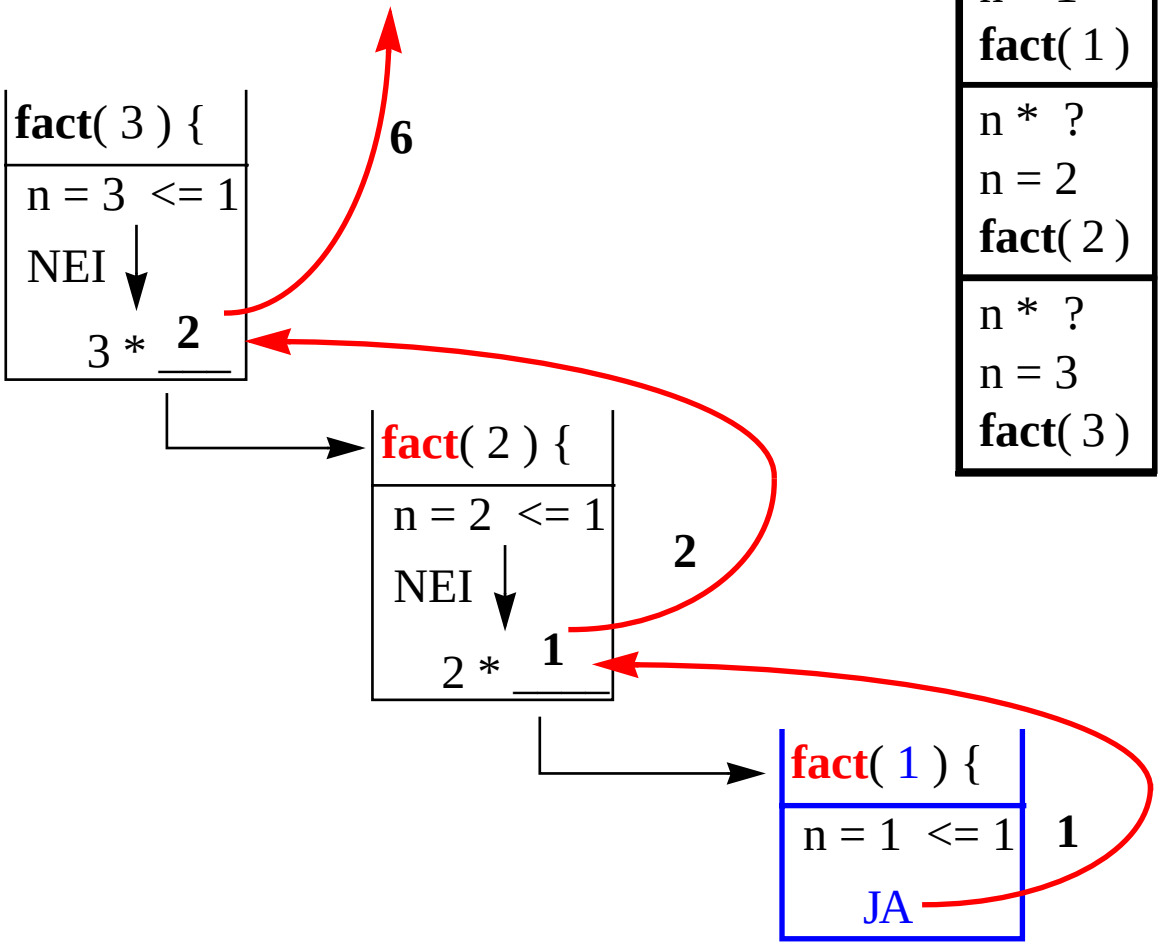
```
int Rekursiv(int n) {
     if ( !fortsett(n) ) return basetilfelle / init;
     else return  Kroppen(n, Rekursiv(oppdater(n)));
}
```

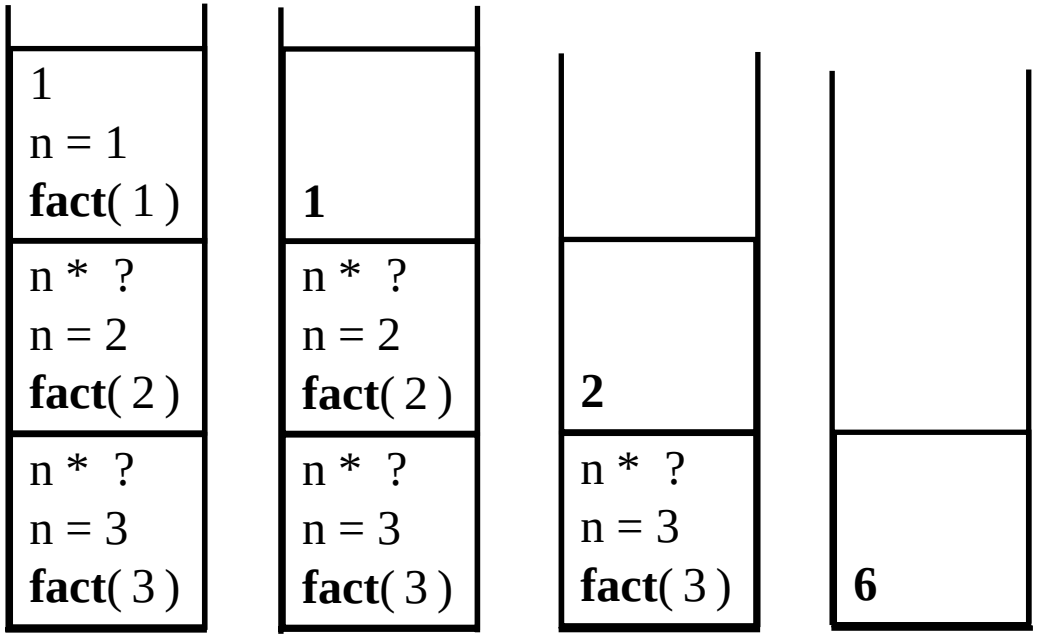
*Enhver **iterasjon** kan skrives som **rekursjon***

... t.o.m. som hale-rekursjon

4a. Rekursjon implementeres med en stabel

```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```

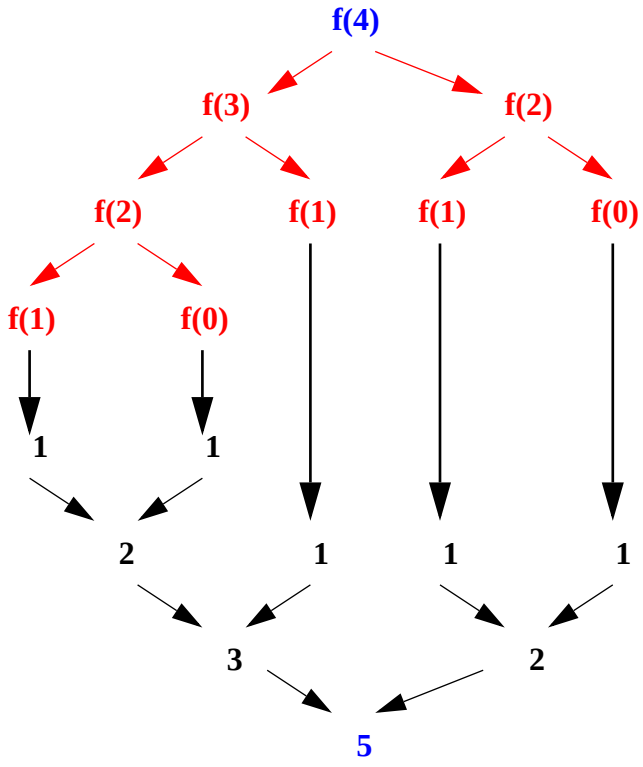


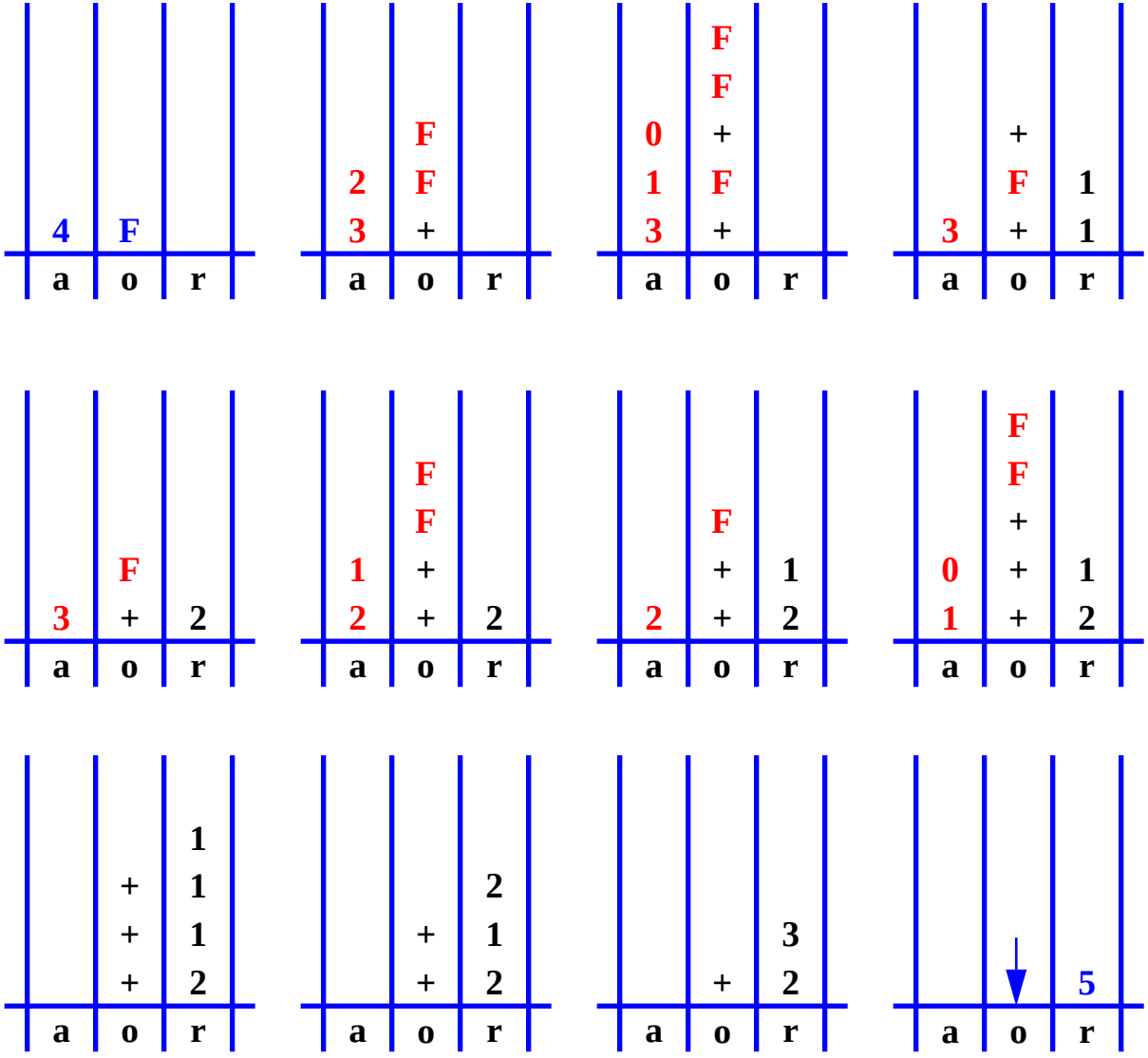


4b. Rekursjon implementeres med en stabel

For Fib kan vi bruke f.eks. 3 stabler ar(argument), op(operator), re(resultat)

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





Rekursjon til iterasjon

(kan alltid omgjøres v.hj.a. Stabel)

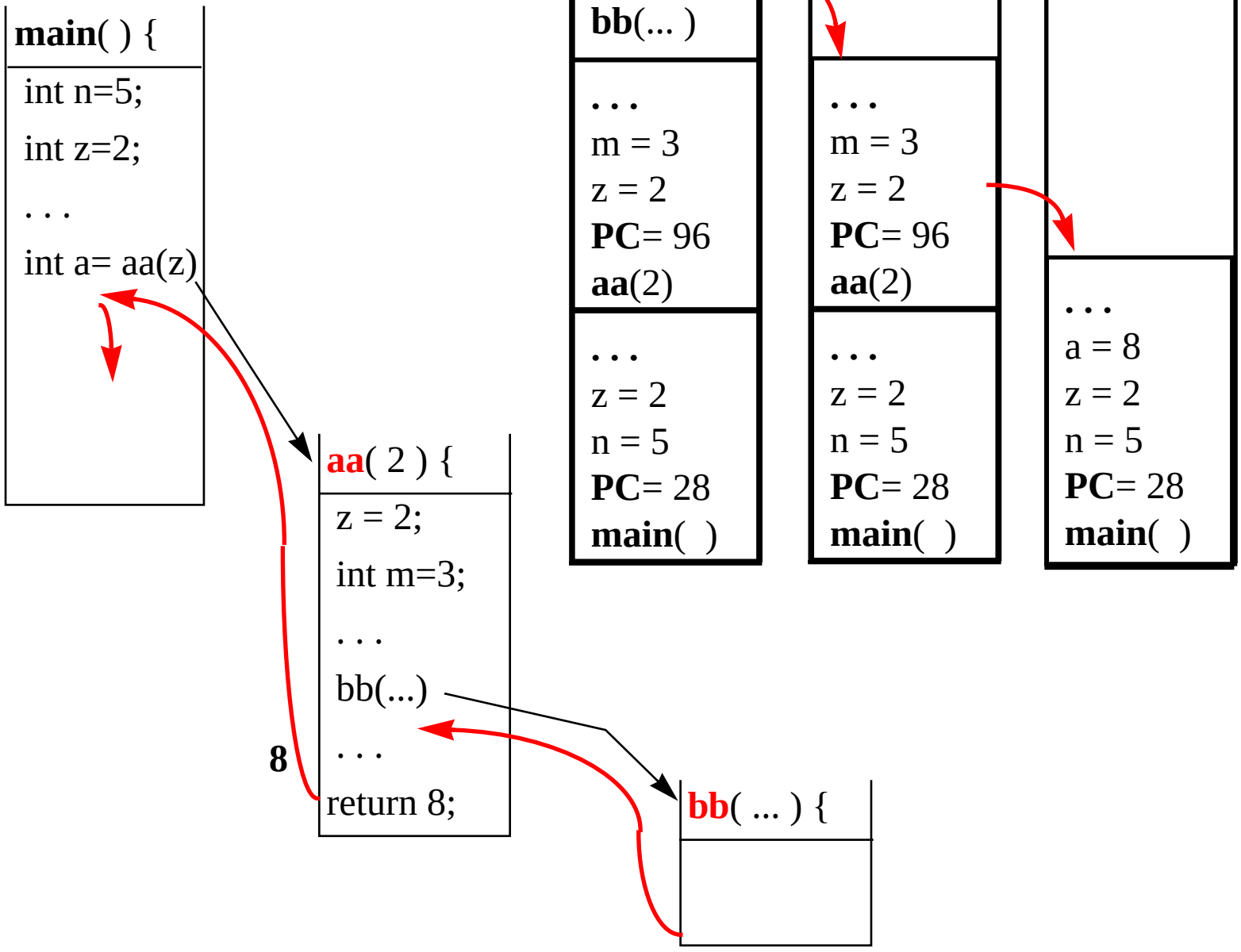
```
int Fib(int n) {
  if (n==0 || n==1) return 1;
  else
      return Fib(n-1) + Fib(n-2);
}
```

Noen rekursjoner (f.eks. hale-rekursjon) kan omgjøres til iterasjon på en enklere måte.

```
int fibS(int a) {
  String  o; int n, a1, a2;
  Stack op = new StackImp();
  Stack re = new StackImp();
  Stack ar = new StackImp();

  op.push("F"); ar.push( new Integer(a) );
  while (!op.empty()) {
    o= (String) op.pop();

    if ( o.equals("F") ) {
        n= ( (Integer)ar.pop() ).intValue();
        if (n==0 || n==1) re.push( new Integer(1) );
        else {
            op.push("+"); op.push("F"); op.push("F");
            ar.push( new Integer(n-1) );
            ar.push( new Integer(n-2) ); }

    } else if ( o.equals("+") ) {
        a1= ( (Integer)re.pop() ).intValue();
        a2= ( (Integer)re.pop() ).intValue();
        re.push( new Integer(a1+a2) ); }
  }
  return ( (Integer)re.pop() ).intValue();
}
```

‘Method Stack’

```
     main( ) {
         int n=5;
         int z=2;
         . . .
28       int a= aa(z);
         . . .
     }

     int aa(int z) {
         int m=3;
         . . .
96       bb(...);
         . . .
         return 8;
     }

201  void bb(...) {
         . . .
     }
```



5. Korrekthet

Gitt en instans **n** *av et problem* **P** *:*

1. hva gjør jeg når **n** *er basis tilfelle*

2. hvordan konstruere løsning for **n** *utfra løsninger for noen instanser* **mindre enn n**

P(n)
  if Basis(**n**)
    return ???

  else
    return
    Kombiner(**P(m1)** ... **P(mk)**)

Terminering:

P(n)
  if Basis(**n**)
    – stopper rekursjon

  else
    – garanter at hver **mi** < **n**, er nærmere Basis

Korrekthet:

P(n)
  if Basis(**n**)
    – kontroller korrekt utførelse

  else HER MÅ VI VISE **HVIS -> SÅ**
    – **HVIS** hvert rekursivt kall **P(mi)** returnerer riktig resultat

    !!! DET OVENSTÅENDE ANTAR VI !!!

    – **SÅ** gir Kombiner(**P(m1)** ... **P(mk)**) riktig resultat

kombinasjon opprettholder ***rekursjons-invariant***

5. Korrekthet

Gitt en instans **n** *av et problem* **P** *:*

1. hva gjør jeg når **n** *er basis tilfelle*

2. hvordan konstruere løsning for **n** *utfra løsninger for noen instanser* **mindre enn n**

```
sum(n)
  if n == 0
    return 0

  else
    return
     n + sum(n–1)
```

Terminering:

sum(n)

if **n <=1**

– stopper rekursjon

else

– hver **n–1 < n**,
er nærmere Basis

kombinasjon opprettholder

$$sum(n) = \sum_{i=0}^{n} i$$

Korrekthet:

sum(n)

if **n == 0**

– resultat er 0 : $\sum_{i=0}^{0} i = 0$

else HER MÅ VI VISE **HVIS -> SÅ**

– **HVIS** rekursivt kall **sum(n–1)** returnerer riktig resultat

!!! DET OVENSTÅENDE ANTAR VI !!!

– **SÅ** gir n + **sum(n–1)**

riktig resultat : $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$

Korrekthet: rekursjons-invariant

```
/* int[] MS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        del A i midten i :
 *        t1= A[0...n/2]  og  t2 = A[n/2+1...n];
 *        sorter rekursivt (mindre) delene
 *            r1= MS(t1) og
 *            r2= MS(t2)
 *        return flettet resultat av
 *            rekursive kall FL(r1,r2) }
 */
```

Invariant:

MS(A) *returnerer sortert argument* A:

if **lgh==1** – ***da er A sortert***

else – deler A i to disjunkte deler

t1= **A[0...n/2]** og t2= **A[n/2+1...n]**

r1= MS(**t1**) ***returnerer sortert t1***

r2= MS(**t2**) ***returnerer sortert t2***

hvis FL *fletter korrekt to sorterte array,*

så returnerer hele else-grenen sortert A

```
/* int BS(int[] A, int x, int l, int h) {
 *    int m= (l+h) / 2 ;
 *    if (l > h) return –1;
 *    else if (A[m] == x) return m;
 *    else if (A[m] < x) return BS(A, x, m+1, h);
 *    else return BS(A, x, l, m–1); }
 */
```

Invariant:

argumentet A *er sortert &*

er x *i* A, *så er den mellom* [l ... h]

(*initielt kall med* (A, x, 0, A.length-1)

if **l > h** – **x *kan ikke være der (–1 er riktig)***

else if **A[m] = x** – ***da har vi funnet den (m er riktig)***

else if **A[m] < x** –

er x *i* A, *så må den være mellom* **[m+1... h]**

BS(A, x, **m+1**, **h**) ***vil returnere riktig resultat***

else **A[m] > x** –

er x *i* A, *så må den være mellom* **[l ... m–1]**

BS(A, x, **l**, **m–1**) ***vil returnere riktig resultat***

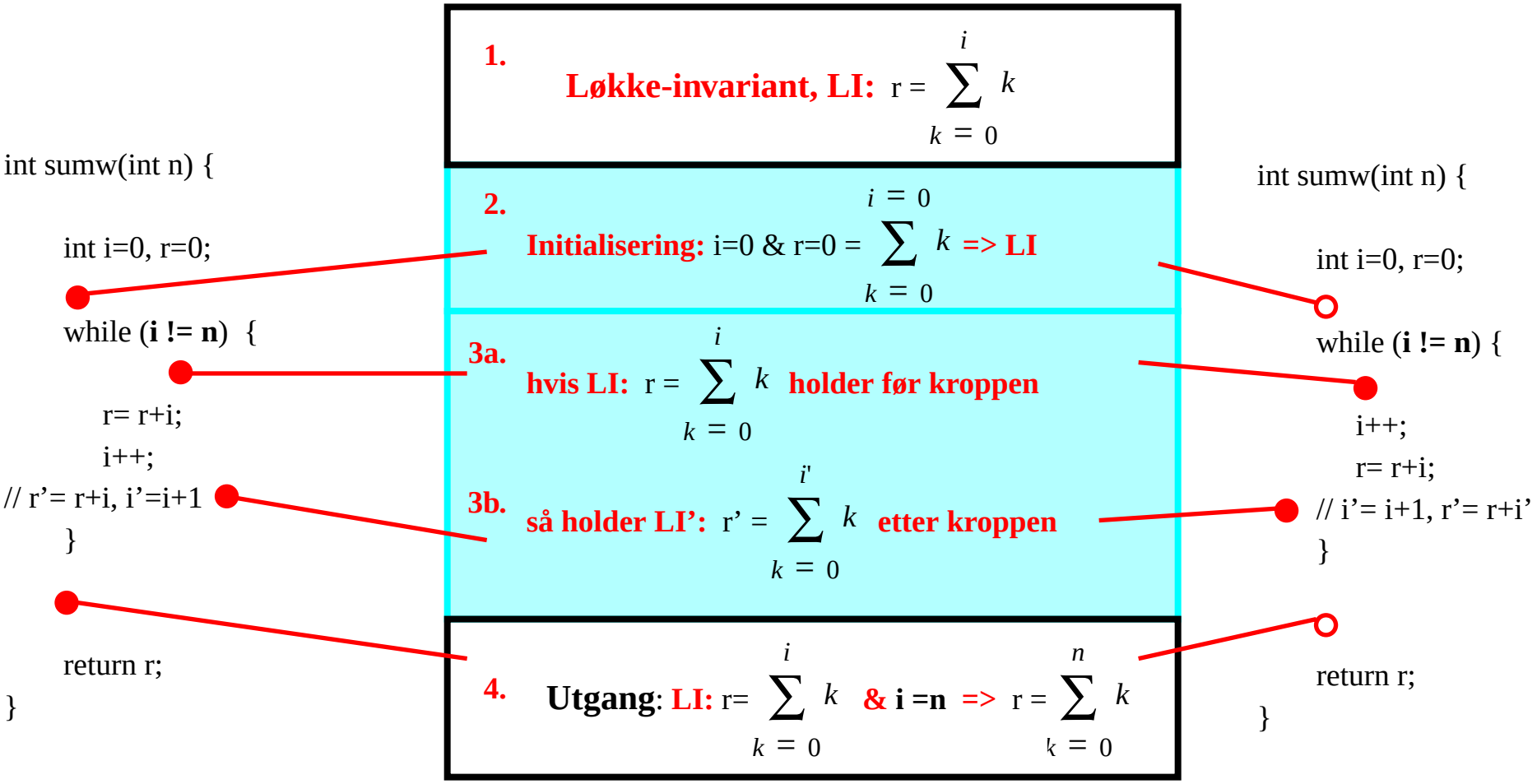
Løkke-invariant

```
int sum(int n) {
    if (n == 0) return 0;
    else return n + sum(n–1);
```

basis – gir riktig sum(0) =0

hvis sum(n–1) gir riktig = $\sum_{i=0}^{n-1} i$

så er sum(n) = n + sum(n–1) = $\sum_{i=0}^{n} i$



Løkke-invariant: eksempel 1.

```
/**  beregner heltalls kvosient samt resten
     @param x >= 0
     @param y > 0
     @return (q, r) sa. x= q*y + r  &  0 <= r < y  &  0 <= q
*/
divr(int x, int y)  {
  int q = 0 ; int r = x ;
```

initialisering: q = 0 & r = x >= 0 → **x= q*y + r & 0 <= r & 0 <= q**

```
  while (y <= r)  {
```

LI: **0 <= r & x = q*y + r & 0 <= q**

– anta at den gjelder ved inngang, samt **y <= r**

```
    q = q+1 ;
    r = r – y;
```

– da gjelder, etter løkkekroppen:

q' = q+1 & **0 <= q** → **0 <= q'** &

r' = r–y & **0 <= r** & **y <= r** → **0 <= r'**

q'*y + r' = (q+1)*y + (r–y) = q*y + y + r – y = q*y + r = **x**

– dvs. LI opprettholdes gjennom kroppen

```
  }
```

utgang fra løkken: **LI** & **r<y** → **x = r + q*y** & **0 <= r < y** & **0 <= q**

```
  return (q, r) ;  }
```

Løkke-invariant: eksempel 2.

```
/**     beregner største felles divisor
        @param x1 > 0
        @param x2 > 0
        @return y2 = gcd(x1,x2) */
gcd(x1,x2) {
    y1= x1; y2= x2;
    while (y1 != 0) {
        if (y2 < y1)
            (y1,y2) = (y2,y1);
        else // (y2 >= y1)
            y2= y2–y1;
    }
    return y2;
}
```

initialisering: x1 = y1 & x2 = y2 → **gcd(x1,x2) == gcd(x1,x2)**

LI: **gcd(y1,y2) = gcd(x1,x2)** – anta at den gjelder her

– **gcd(x1,x2)** = gcd(y1,y2) = gcd(y2,y1) = **gcd(y1',y2')**

– **gcd(x1,x2)** = gcd(y1,y2) = gcd(y1,y2-y1) = **gcd(y1,y2')**

LI': cd(y1',y2') = gcd(x1,x2)

utgang: **LI & y1 = 0** →
gcd(x1,x2) = gcd(y1,y2)
= gcd(0,y2) = y2

Hvis gcd(y1,y2) = z >=1 & y2 >= y1, så
*) y1 = z*k1 <= z*k2 = y2 & gcd(k1,k2) = 1
Men da:
y2'= y2–y1 = z*(k2–k1) & gcd(k1, k2–k1) = 1
hvis ikke, dvs. gcd(k1,k2–k1) = v > 1, da
k1= v*a & k2–k1= v*b, så
k2= v*b+v*a = v*(b+a)
dvs. da også gcd(k1,k2) = v >1 – motsier *)

Oppsummering

1. ***Rekursjon*** *– "**Splitt og hersk**"*

   – bestem hva som må gjøres i basis tilfelle(r)

   – konstruer ("hersk") en løsning fra (rekursive) løsninger for ("splitt") noen mindre instanser

2. *Enhver induktiv datatype (nat, int, lister, trær, ...) gir opphav til rekursive algoritmer*

3. *Rekursjon vs. iterasjon (rekursjon implementeres iterativt med bruk av stabel)*

4. ***Korrekthet***

   – bestem rekursjons-invarianten

   - ***verifiser*** *at basistilfelle(r) etablerer invarianten*
   - *under* ***antakelse*** *at rekursive kall etablerer invarianten,* ***vis*** *at konstruksjonen vil opprettholde den*

   – bestem løkke-invariant

   - ***vis*** *at den gjelder etter initialisering (like før inngangen i løkken)*
   - *under* ***antakelse*** *at den gjelder før løkkekroppen,* ***vis*** *at den gjelder også etter denne*