Et enkelt eksempel

har en metode som

```
/**  leser en linje fra terminalen
 *   @return innleste String
 *   @exception IOException – i tilfelle i/o problem
 */
     public String readln()
```

og vil lage en som

```
/**  leser en linje fra terminalen
 *      inntil den leser et heltall
 *   @return innleste tall
 *   @exception  ingen unntak
 *      – anta det kommer et heltall
 */
/*  public int iRead() {
 *      String s= readln();
 *      int k= hent int fra s;
 *      while (! alt ok)
 *         gjenta: k = hent int fra neste linje;
 *      return k;
*/
```

```
/* public int myRead() {
 *       String s= readln();
 *       int k= hent int fra s;
 *       if (alt ok) return k;
 *       else   // prøv neste linje
 *           return myRead();
 */      }

  public int myRead() {
    try{
         return Integer.parseInt(readln());  }
    catch(IOException e) {
         return myRead(); }
    catch(NumberFormatException e) {
         return myRead(); }
  }
```


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. 8.1.1)

IV. Rekursjons effektivitet

"memoisering"

avskjæring

V. Stabel av rekursive kall

iterasjon til rekursjon

rekursjon implementert som iterasjon

VI. Korrekthet

terminering

invarianter (notat til Krogdahl&Haveraaen)

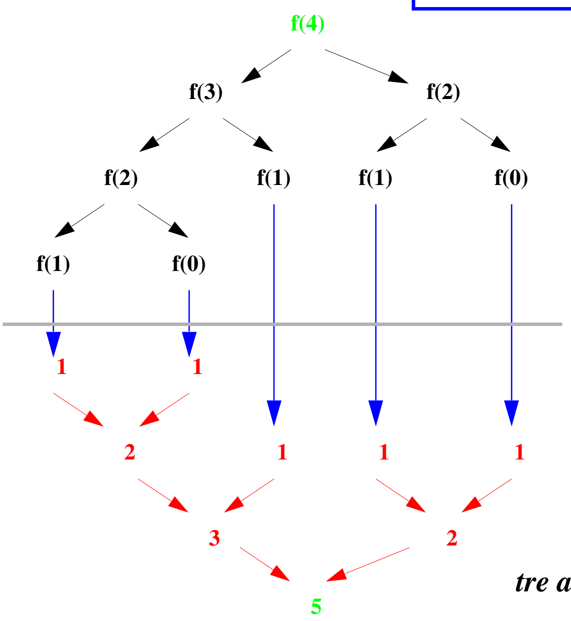
1. Rekursjons*tre* og *-dybde;* Eks: Fibonacci-tallene

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)*

Iterasjon til rekursjon

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

```
/** @param n > 0
    @return 1+2+...+n */
int sumR(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;
    else return Kroppen(n, Rekursiv(oppdater(n)));
}
```

*Enhver **iterasjon** kan skrives som **rekursjon***
... t.o.m. som hale-rekursjon

Variasjoner over tema

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

N
basis: 0
ind: n+1

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

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

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

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

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

Liste[N]
basis: null
ind: (L,n)

```
class LS {
    int hodedata;
    LS restliste; }
```

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

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

BinærtTre[N]
basis: null
ind: (t1,n,t2)

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

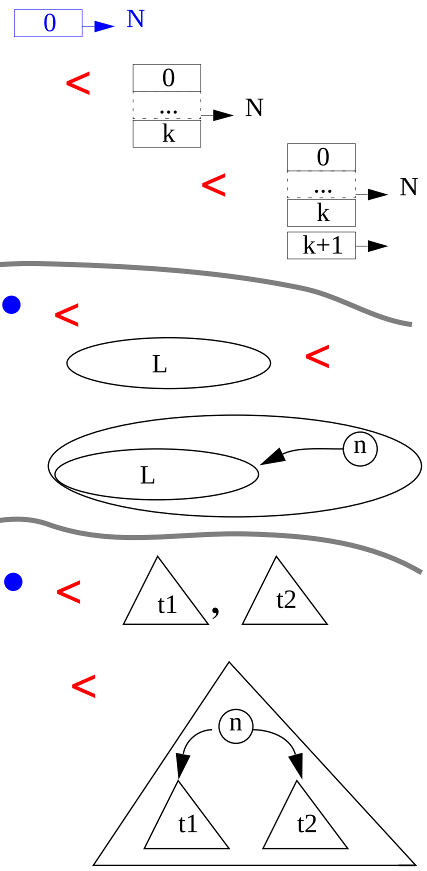
```
void inc(BT B) {
    if (T==null) { }
    else { n++;
           inc(T.left);
           inc(T.right); } }
```

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

FRAKTALer

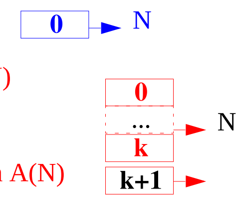
2. Induktive Data Typer (*vilkårlig store men endelige*)

Strukturell ordering

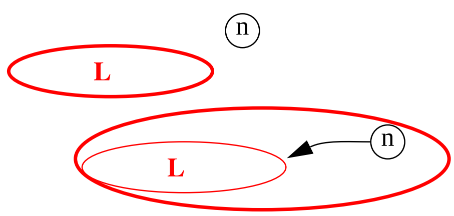


naturlige tall N:
basis: **0** er et N
hvis **n** er et N
så er: **n+1** et N

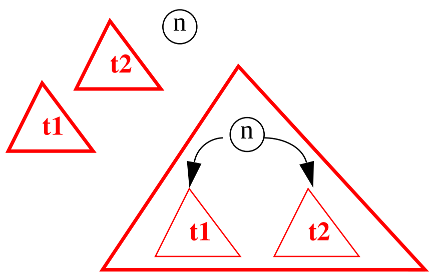
array av N: A(N)
basis **0 -> N** er A(N)
hvis **[0...k] -> N** er A(N)
så er **[0...k,k+1] -> N** en A(N)



Lister av N: L(N):
basis: **null** er en L(N)
hvis **L** er L(N) og n er N
så er: **(L,n)** 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)



Iterativt eksempel: Seleksjonsortering

```
/*   SS - sorterer input array (SeleksjonSort)
 *     @param  - int tab[0...n]
 *     @return - sortert tab
 *
 *   for (k = 0,1,2...n) {
 *     i = k
 *     for ( j = k+1...n)
 *       if (tab[j] < tab[i]) i = j;
 *     bytt elementene ved indeks k og i
 *     }
 */
```

for en vlikårlig input tabell med lengde **n**:

- *utfører* **n** *iterasjoner (for k=1,2...***n**) *og*
- *i hver iterasjon går gjennom sluttsegment* **[k...n]**, *(for j=k+1...n), dvs.*

$$\text{tidskompleksistet } SS(n) = \left(\sum_{k=1}^{n} k = 1 + 2 + \ldots + (n-1) + n \right) = (n + n^2)/2 = \mathbf{O}(n^2)$$

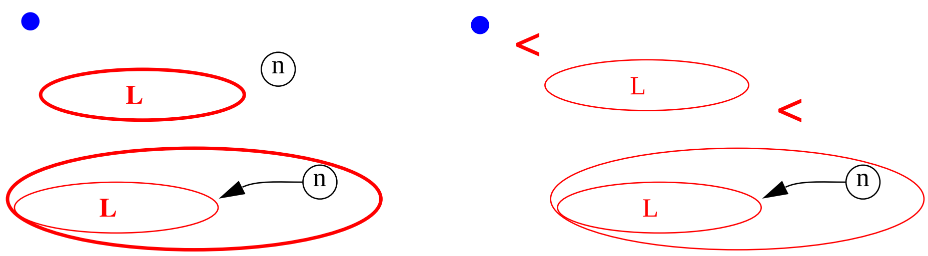
En teknisk bemerkning

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]

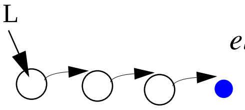


Rekursjon implementert *"utenfra" datastrukturen :*

```
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; }
```

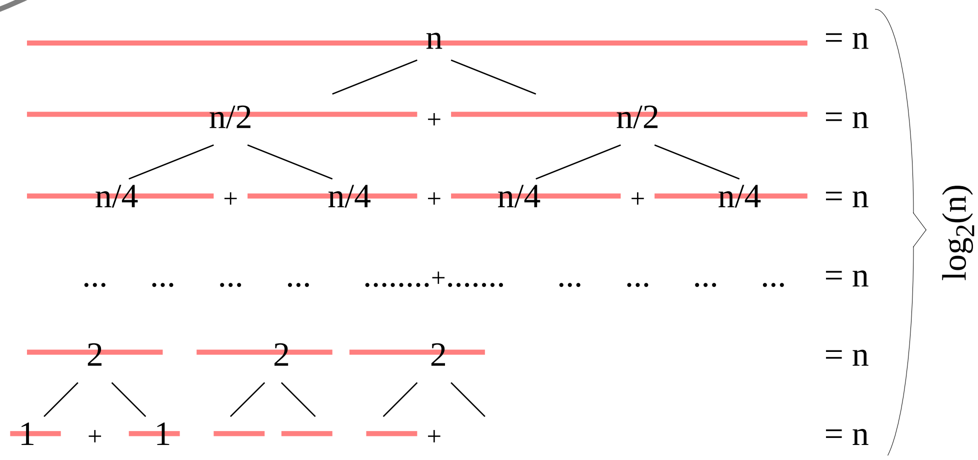
Rekursivt eksempel:MergeSort

```
/*   FL - fletter to sorterte array:
 *       @param  - int t1[0...n1], t2[0...n2] - sorterte
 *       @return - sortert t[0......n1+n2]
 *   gå (samtidig) gjennom t1 og t2 (med i1 og i2)
 *   if      t1[i1] < t2[i2] plasser t1[i1] i t og øk i1, i
 *   else                    plasser t2[i2] i t og øk i2, i
 *   hvis noe igjen i t1 eller t2, flytt det til t
 *   return t;
 */
```

$\boldsymbol{FL}(n1,n2) = \boldsymbol{O}\ (n1+n2)$

```
/*   MS - sorterer input array:
 *       @param  - int tab[0...n-1]
 *       @return - sortert tab
 *   if (n == 1) return tab
 *   else { k= n/2;
 *       return FL ( MS(tab[0...k]), MS(tab[k+1..n-1]) ); }
 */
```

$\boldsymbol{MS}(n) = \boldsymbol{O}\ (n * log_2(n)\ )$



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 (n = A.length)

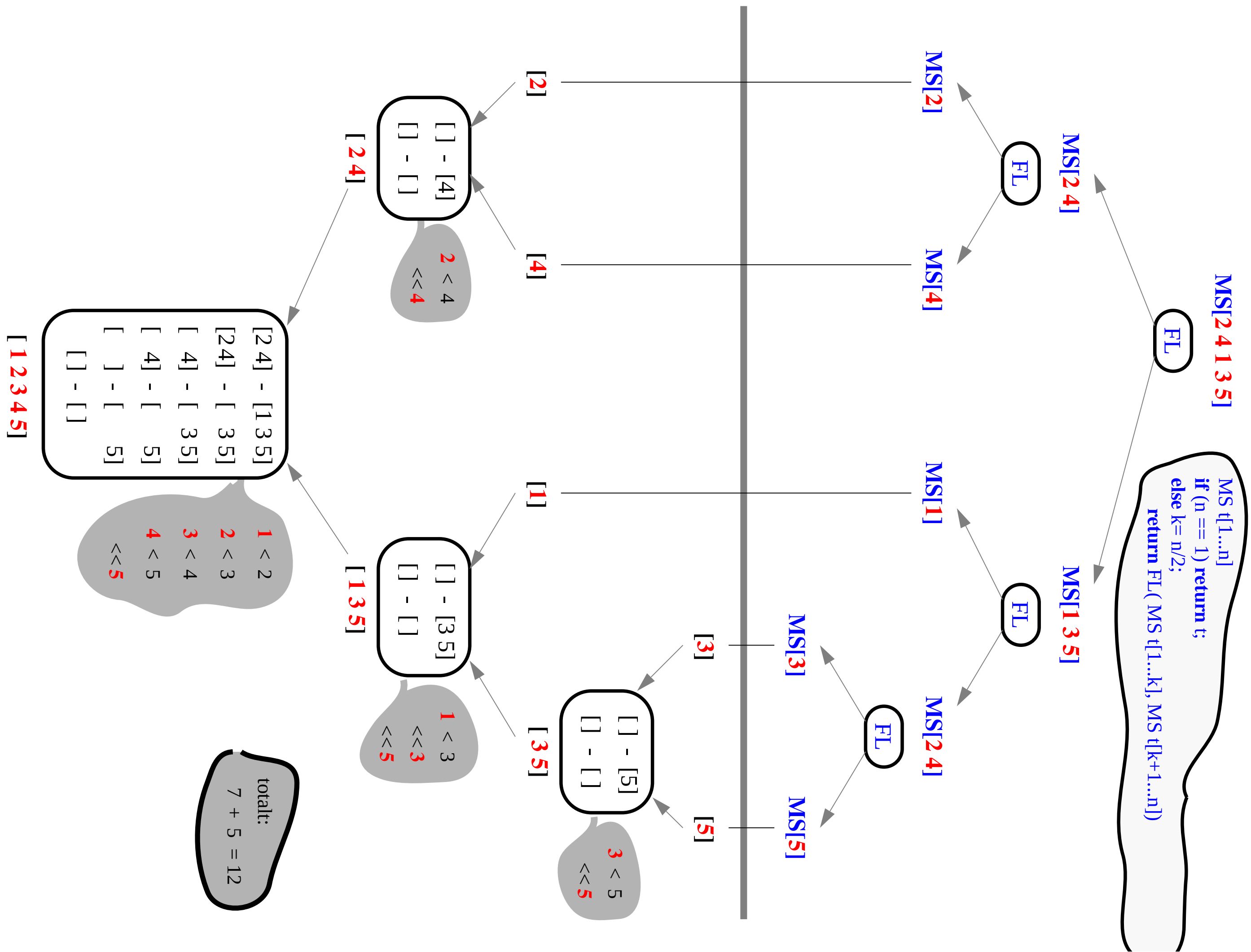
```
/* int[] SS(int[] A,k) {                                O (n^2)
 * initielt kall med SS(A,0)
 * n = A.length;
 *    if (k==n-1) { return A; }
 *    else {
 *       i= indeksen til minste elementet
 *          i A[k...n-1];
 *       bytt A[k] med A[i];
 *       return SS(A, k+1); } }   */
```

```
/* int[] MS(int[] A) { int n= A.length;                 O (n*log n)
 *    if (n == 1) { return A; }
 *    else { del A i midten i
 *       t1= A[0...n/2] og t2= A[n/2+1...lgh];
 *       sorter rekursivt begge (mindre)
 *          r1= MS(t1) og r2= MS(t2)
 *       return flettet resultat av rekursive kall FL(r1,r2)
 * FL - fletter to sorterte array i en sortert array */
```

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] O (n)

Hvis A er sortert ...

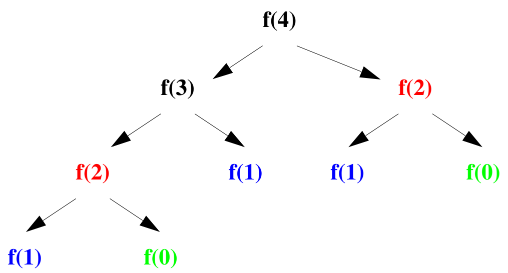


4. Rekursjon og effektivitet

– *Reduser antall rekursive kall* –

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

$O(1.6^n)$



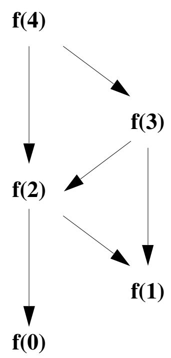
1. "Memoisering" :

*Istedenfor gjentatte rekursive kall til f(k) med samme k, kan **i dette tilfelle** resultatet av f(k) lagres for senere bruk:*

```
int Fib(int n) {
    int[] ar= new int[n+1];
    ar[0]=1; ar[1]=1;
    return fibo(n, ar);
}

int fibo(int n, int[] ar) {
    if (ar[n] > 0) {
        return ar[n]; }
    else {
        int z= fibo(n–1) + fibo(n–2);
        ar[n]= z;
        return z;
} }
```

O(n)



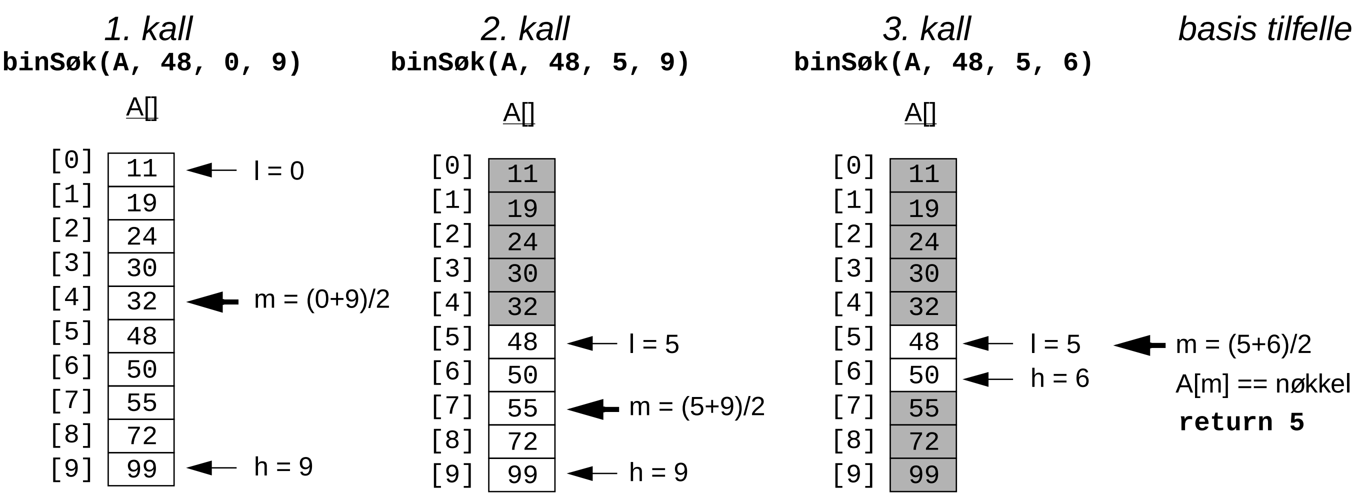
Binær Søk

```
/*  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
*/
```

O (log n)

```
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)
```

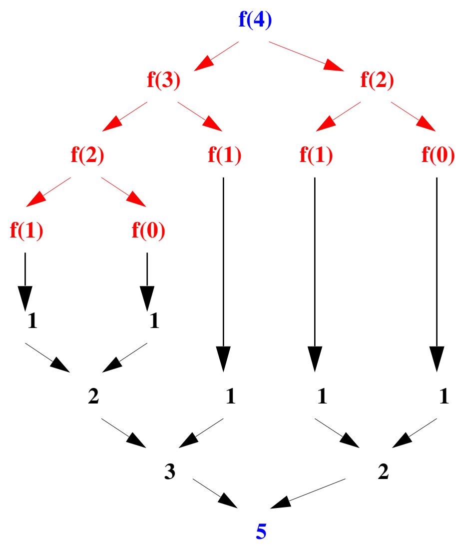
Nøkkel er 48

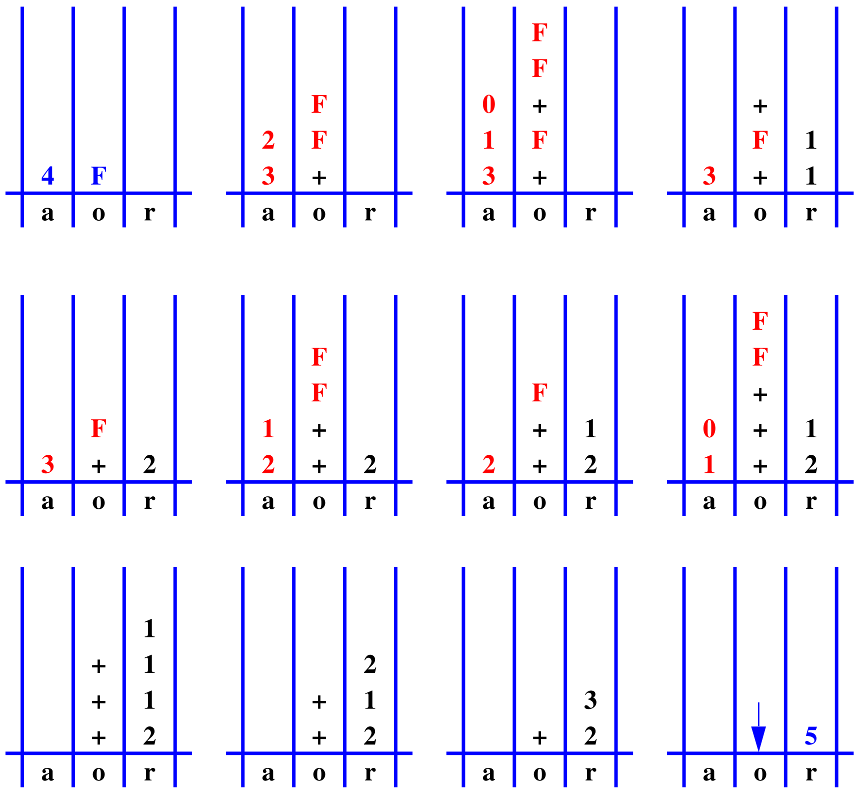


5. Rekursjon implementert med stabel. . .

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);}
```





Kompleksitet av en rekursiv funksjon

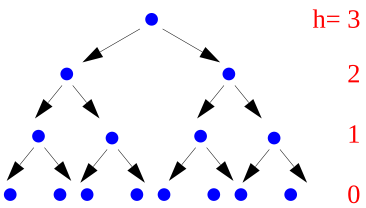
Analyse vha REKURSJONSTRE

avhenger av

- *“**størrelsen på steget**” i hvert rekursivt kall (høyden av treet)*
- ***antall rekursive kall** i hvert steg (“bredden” av forgreninger)*
- *arbeidsmengden ved “sammensetting” av resultater fra rekursive kall. **Anta dette O(1) i eksemplene under.***

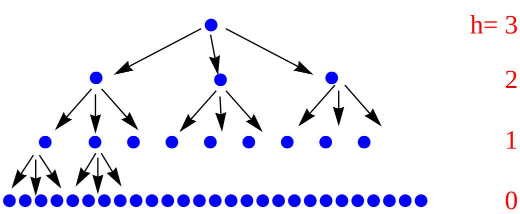
R(n) har 2 rekursive kall til R(n-1)

R(**n+1**) = R(**n**) + R(**n**) $O(2^{n+1} - 1)$



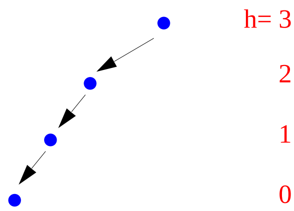
R(n) har 3 rekursive kall til R(n-1)

R(**n+1**) = R(**n**) + R(**n**) + R(**n**) $O(3^{n+1} - 1)$



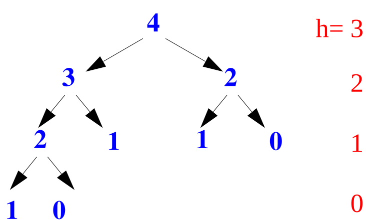
R(n) har 1 rek. kall til R(n-1)

R(**n+1**) = R(**n**) + c $O(n)$

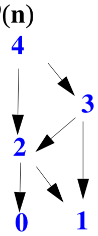


R(n) har rek kall til R(n-1) og R(n-2)

R(**n+2**) = R(**n+1**) + R(**n**) $O(1.6^n)$

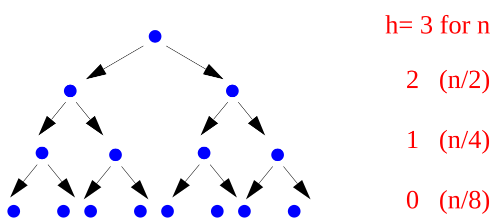


Fibonacci kan dog forenkles til: $O(n)$



R(n) har 2 rek. kall til R(n/2) $2^{\log(n)+1} - 1 = 2n - 1 = O(n)$

R(**n**) = R(**n/2**) + R(**n/2**)



6. 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

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();
}
```

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

```
int sumw(int n) {
    int i=0, r=0;
    while (i != n)  {
        r= r+i;
        i++;
// r'= r+i, i'=i+1
    }
    return r;
}
```

1. **Løkke-invariant, LI:** r = $\sum_{k=0}^{i} k$

2. **Initialisering:** i=0 & r=0 = $\sum_{k=0}^{i=0} k$ **=> LI**

3a. **hvis LI:** r = $\sum_{k=0}^{i} k$ **holder før kroppen**

3b. **så holder LI':** r' = $\sum_{k=0}^{i'} k$ **etter kroppen**

4. **Utgang: LI:** r= $\sum_{k=0}^{i} k$ **& i =n** **=>** r = $\sum_{k=0}^{n} k$

```
int sumw(int n) {
    int i=0, r=0;
    while (i != n) {
        i++;
        r= r+i;
    // i'= i+1, r'= r+i'
    }
    return r;
}
```


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:

<u>if **lgh==1** – ***da er A sortert***</u>

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)

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

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

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: eksempel 2.

```
/**     beregner største felles divisor
        @param x1 > 0
        @param x2 > 0
        @return y2 = gcd(x1,x2) */
gcd(x1,x2) {
    y1= x1; y2= x2;      <----  initialisering: x1 = y1 & x2 = y2  →  gcd(x1,x2) == gcd(x1,x2)
    while (y1 != 0) {
                         <----  LI:  gcd(y1,y2) = gcd(x1,x2) – anta at den gjelder her
      if (y2 < y1)
         (y1,y2) = (y2,y1);      – gcd(x1,x2) = gcd(y1,y2) = gcd(y2,y1) = gcd(y1',y2')
      else // (y2 >= y1)
         y2= y2–y1;              – 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
    return 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 *)

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) ;  }
```

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. *Kompleksitet av rekursiv funksjon avhenger av*

   – antall noder i rekursjonstre ("splitt")

   - *dybden (høyden) av treet – hvor stort steg mot basis utgjør hver "splitting"*
   - *antall rekursive kall (bredden av treet) på hvert nivå*

   – arbeidsmengden for å konstruere en løsning utfra løsninger for mindre instanser ("hersk")

5. ***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*