Rekursjon

I. ET ENKELT EKSEMPEL

II. TRE AV REKURSIVE KALL,

rekursjonsdybde
terminering – ordning

III. INDUKTIVE DATA TYPER

og Rekursjon over Data Type

IV. ''SPLITT OG HERSK'' – PROBLEMLØSNING VED REKURSJON

Kap. 8.1.1

V. REKURSJONS EFFEKTIVITET

''dynamisk programmering''
avskjæring

VI. STABEL AV REKURSIVE KALL

iterasjon til rekursjon
rekursjon implementert som iterasjon

VII. KORREKTHET

terminering

invarianter

dermed ferdig med generell BASIS

I. 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 fra terminalen inntil faar et heltall
 *   @return innleste tallet
 *   @exception      ingen unntak – det kommer et heltall
 */

/*  public int iRead() {
 *        String s= readln();
 *        int k= hent foerste int fra s;
 *        while (! alt ok) gjenta – proev neste linje;
 *        return k; */

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

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

Iterasjon til rekursjon

```
int sumW(int k) {
    int n=0, res= 0;
    while (n <= k) {
        res= n + res;
        n++;
    }
    return res;
}
```

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

Grovt – og ikke 100% riktig – sagt:

```
int W(int k) {
    n= basis; res= init;
    while (Bet(n,k)) {
        Kroppen(n,res);
        oppd(n);
    }
    return res;
}
```

```
int R(int n) {
    if (n == basis) return init;
    else Kroppen(n, R(-oppd(n)));
}
```

Enhver iterasjon (løkke) kan skrives som en rekursiv metode

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

II. Rekursjonsdybde og -tre

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n+2) &= fib(n) + fib(n+1) \end{aligned}$$

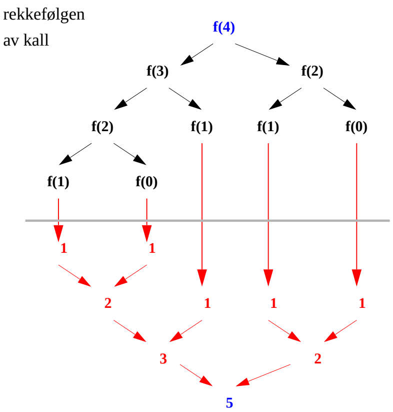
returner i **basistilfelle**

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

ellers beregn rekursivt **enklere** (**min-dre**) deler og sett de sammen

```
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 basis tilfelle er nådd*

(= høyden av treet)

III. Induktive Data Typer (vilkårlig store men endelige)

Strukturell ordering

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

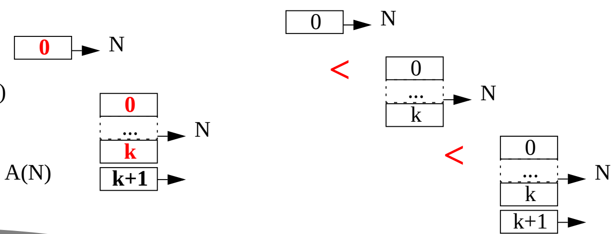
så er: **n+1** et N

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)

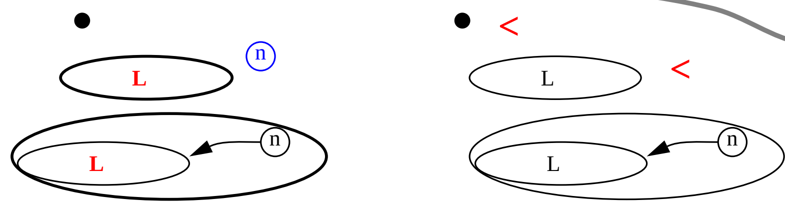


Lister av N: L(N):

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

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

så er: **(n + L)** en L(N)



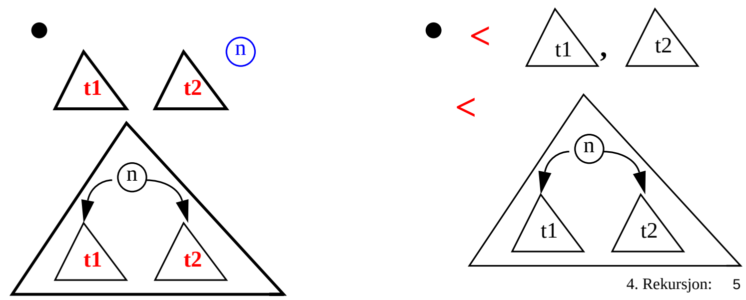
Binære Trær av N: BT(N):

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

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

så er: **(t1, n, t2)** et BT(N)

substruktur < superstruktur



Variasjoner over tema

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

N
basis: 0
ind: n+1

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

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

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

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

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

LT
basis: null
ind: (L+n)

```
class LT {
  int n;
  LT nxt; }
```

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

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

BT
basis: null
ind: (t1,n,t2)

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

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

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

FRACTALS

IV. “Splitt og Hersk”

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 **mi** < **n**

P = sorter input array A

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

O (n^2)

```
/* int[] MS(int[] A) { int lgh= A.length;
 *    if (lgh == 1) { return A; }
 *    else {      del A i midten i
 *                t1= A[0...lgh/2] og t2= A[lgh/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 */
```

O (n*log n)

P = finn et gitt element x i en array A

Hvis A er usortert : sjekk A[n]; hvis ikke der, lett i A[0...n-1] O (n)

Hvis A er sortert . . .

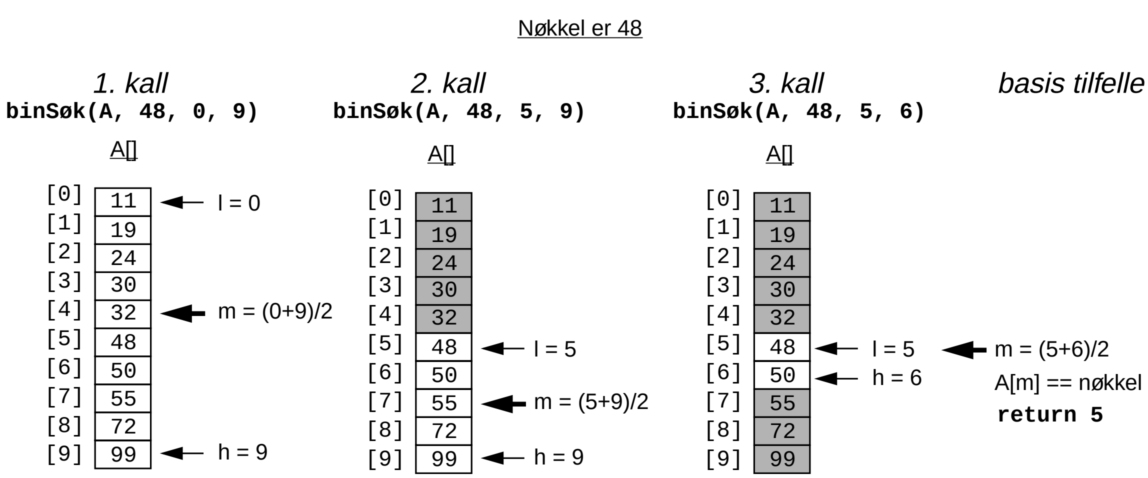
```
/* int BS(int[] A,x,l,h) { //initielt kall med BS(A,x,0,A.length-1)
 *        if (l > h) { return -1; }
 *        else {      m = midten mellom l og h = (l+h)/2 ;
 *                    if ( A[m] == x ) return m;
 *                    else if ( A[m] < x ) lett i øvre del A[m+1...h];
 *                    else lett i nedre del A[l...m-1]; } }  */
```

O (log n)

Binær Søk

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

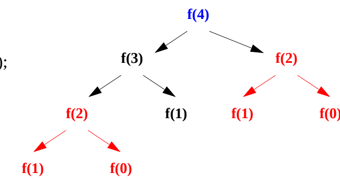
```
int BS(int[] A,x,l,h) {
      int m= (l+h) / 2 ;
      if (l > h) return -1;
      else if (A[m] == x) return m;
      else if (A[m] < x) return BS(A,x,m+1,h);
      else return BS(A,x,l,m-1); }
```



V. Rekursjon & effektivitet

– Reduser antall rekursive kall –

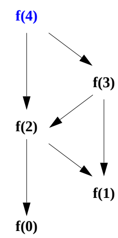
```
int fib(int n) {
    if (n==0 || n==1) return 1;
    else return fib(n-1)+fib(n-2);
}
```



1. “Dynamisk programmering” :

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

```
int[] ar;
void Fib(int n) {
    ar= new int[n+1];
    ar[0]=1; ar[1]=1;
    fibo(n);
}
int fibo(int n) {
    if (n==0 || n==1) return 1;
    else if (ar[n] > 0) { return ar[n]; }
    else {      int z= fibo(n-1) + fibo(n-2);
                ar[n]= z;
                return z;
}   }
```



Rekursjon & effektivitet

2. Avskjæring

Finn alle permutasjoner av [0,1,2...n-1]

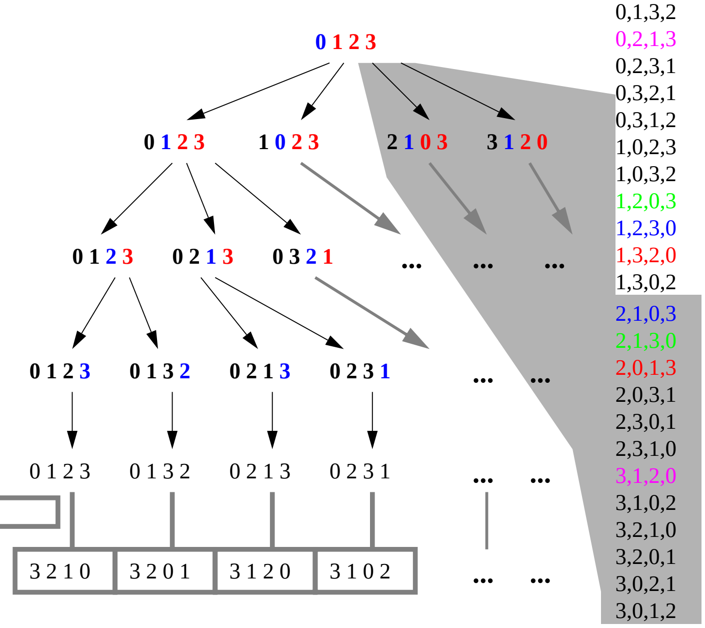
```
/*    perm(A,n) { int l= A.length-1;
 *    if (n==l) { skriv A; }
 *    else {
 *      for hver ind: n...l
 *        B= A;
 *        bytt B[n] og B[ind]
 *        perm(B,n+1) – perm B[n+1...l] }
 */
```

A[0..n-1] A[n] A[n+1..l]

perm(A,0) skriver alle perm

```
/*    PE(A) { int l= A.length;
 *    for hver n: 0...l/2 {
 *      B= A;
 *      bytt B[0] og B[n];
 *      perm2(B,1); }
 *    if (l odd) {
 *      bytt A[0] og A[l/2+1];
 *      perm(A,1); } }
 */
```

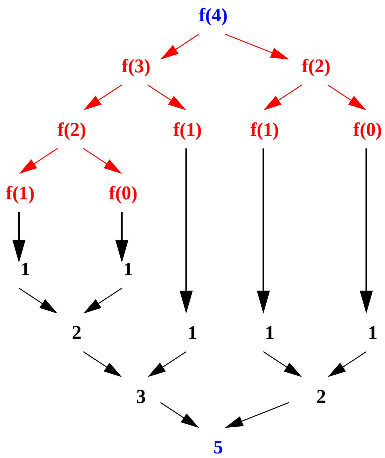
skriv inv(A);

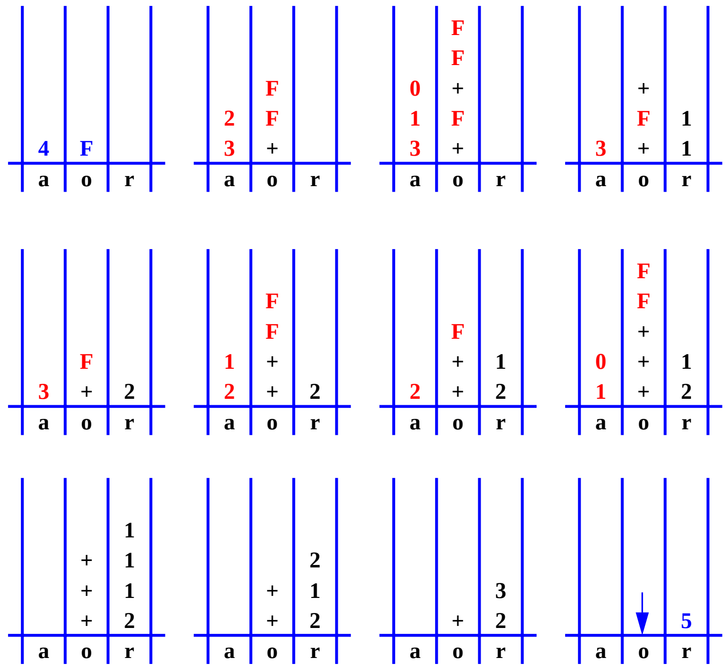


VI. Rekursjon . . .

implementeres med Stabel

```
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);}
```

```
/*int fibS(int a)
   op = new Stack();
   re = new Stack();
   ar = new Stack();
   op.push("F"); ar.push(a);
   while (!op.empty()) {
     o= op.pop();
     if (o == "F") {
            int n= ar.pop();
            if (n==0 || n==1) re.push(1);
            else { op.push("+"); op.push("F"); op.push("F");
                 ar.push(n-1);
                 ar.push(n-2); }
     } else if (o == "+") {
            a1= re.pop();
            a2= re.pop();
            re.push(a1+a2); }
   }
      return re.pop();
*/
```

+ to Object !!!

+ Kast !!!

(Noen rekursjon, som f.eks. hale-rekursjon, kan gjøres om til iterasjon på mye enklere måte.)

VII. 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 **mi < n**

P(**n**)
    if Basis(**n**) return ???
    else Kombiner(P(**m1**) ... P(**mk**))

Terminering:

P(**n**)
    if Basis(**n**) – stopper rekursjon
    else –   vær sikker på at hver **mi < n** er nærmere Basis

Effektivitet:

avhenger av rekursjonsdybde –
    hvor mye mindre hver mi < n er, eller
    hvor stort steg mot Basis utgjør hvert rekursivt kall

Korrekthet:

P(**n**)
    if Basis(**n**) – kontroller at det utføres riktig handling

    else –   **HVIS alle rekursive kall P(mi) utfører riktige handlinger – DETTE ANTAR VI !!!**

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

Korrekthet: **Invariant**

```
/* int[] MS(int[] A) { int lgh= A.length;
 *     if (lgh == 1) { return A; }
 *     else {      del A i midten i
 *                 t1= A[0...lgh/2] og t2= A[lgh/2+1...lgh];
 *                 sorter rekursivt (mindre) 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** t1= A[0...lgh/2] og t2= A[lgh/2+1...lgh]

MS(t1) og MS(t2) returnerer sorterte r1 og r2 – **!!! FORUTSETNING**

hvis FL fletter korrekt, så returnerer hele else-grenen riktig resultat

---

```
int BS(int[] A,x,l,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

else if **A[m] = x** – da har vi funnet den

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

!!! FORUTSETNING: rekursivt kall vil finne den der

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

!!! FORUTSETNING: rekursivt kall vil finne den der

Løkkeinvariant

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

}
```

basis: n==0 – sum(0) = 0
hvis sum(n–1) returnerer 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. **Løkkeinvariant, 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;
}
```