Rekursjon

I. TRE AV REKURSIVE KALL,

rekursjonsdybde

terminering — ordning

II. INDUKTIVE DATA TYPER

og Rekursjon over slike

III. ŒSPLITT OG HERSKŒ— PROBLEMLfISNING VED REKURSJON

IV. STABEL AV REKURSIVE KALL

iterasjon til rekursjon

rekursjon implementert som iterasjon

V. KORREKTHET

terminering

invarianter

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    **/

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

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    **/

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

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

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    **/

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

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

- fact(n) vil alltid terminere – Hvorfor?

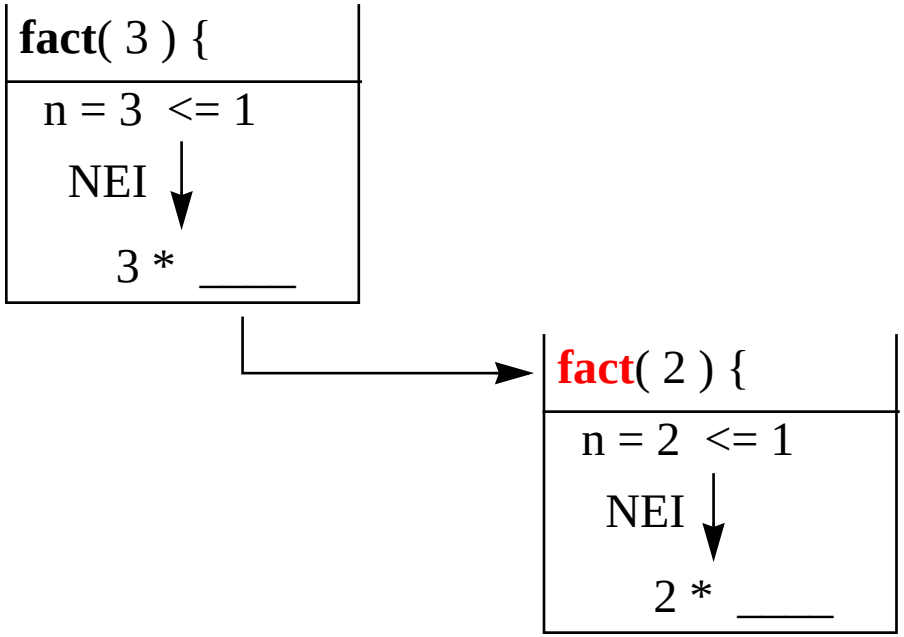
- Hva med:

```
    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!
*/
```

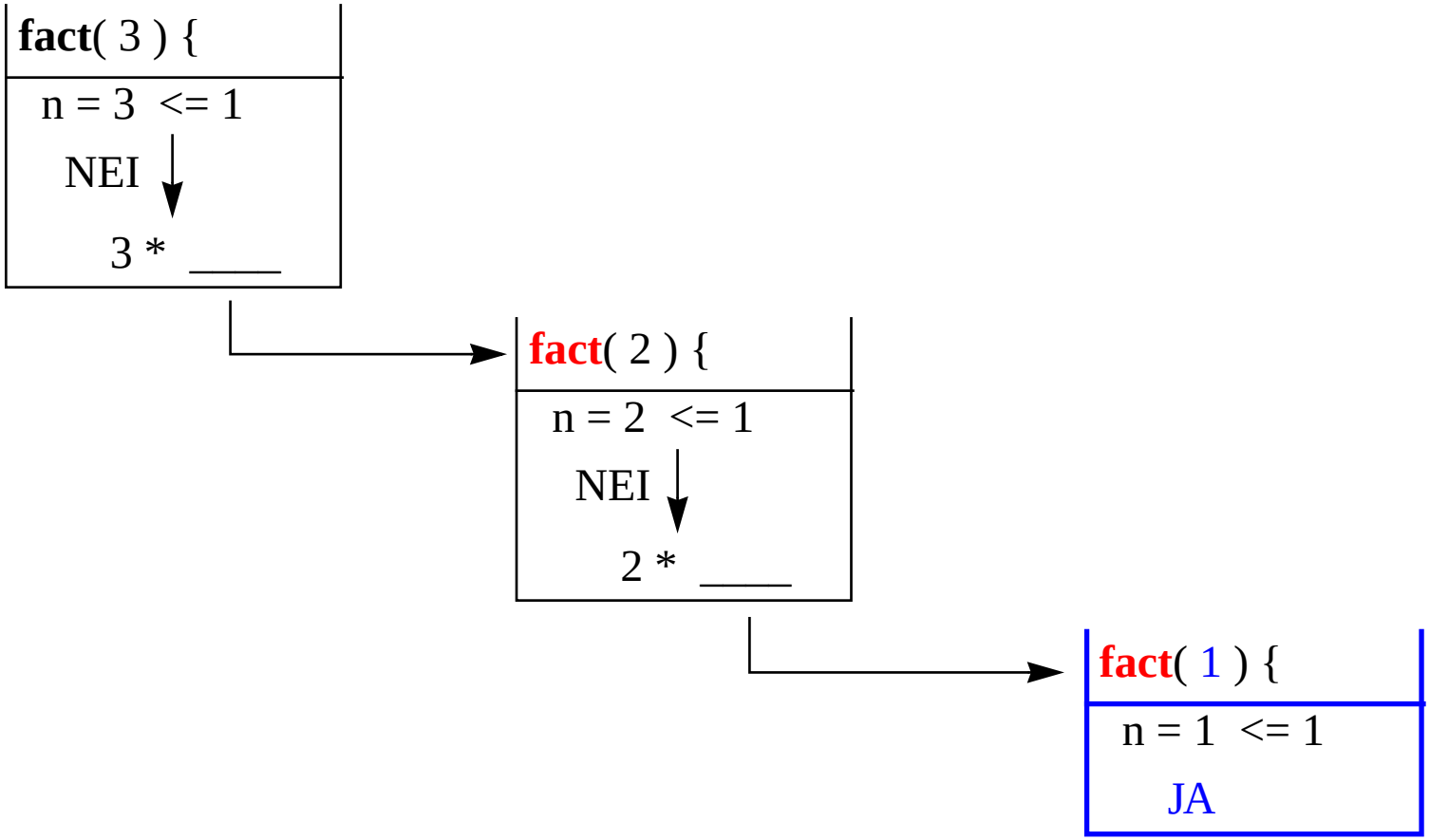
```
int fact( int n ) {
    if (n <= 1) return 1;
    else   return  n *  fact( n–1 );
}
```



1. En metode “kaller seg selv”

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

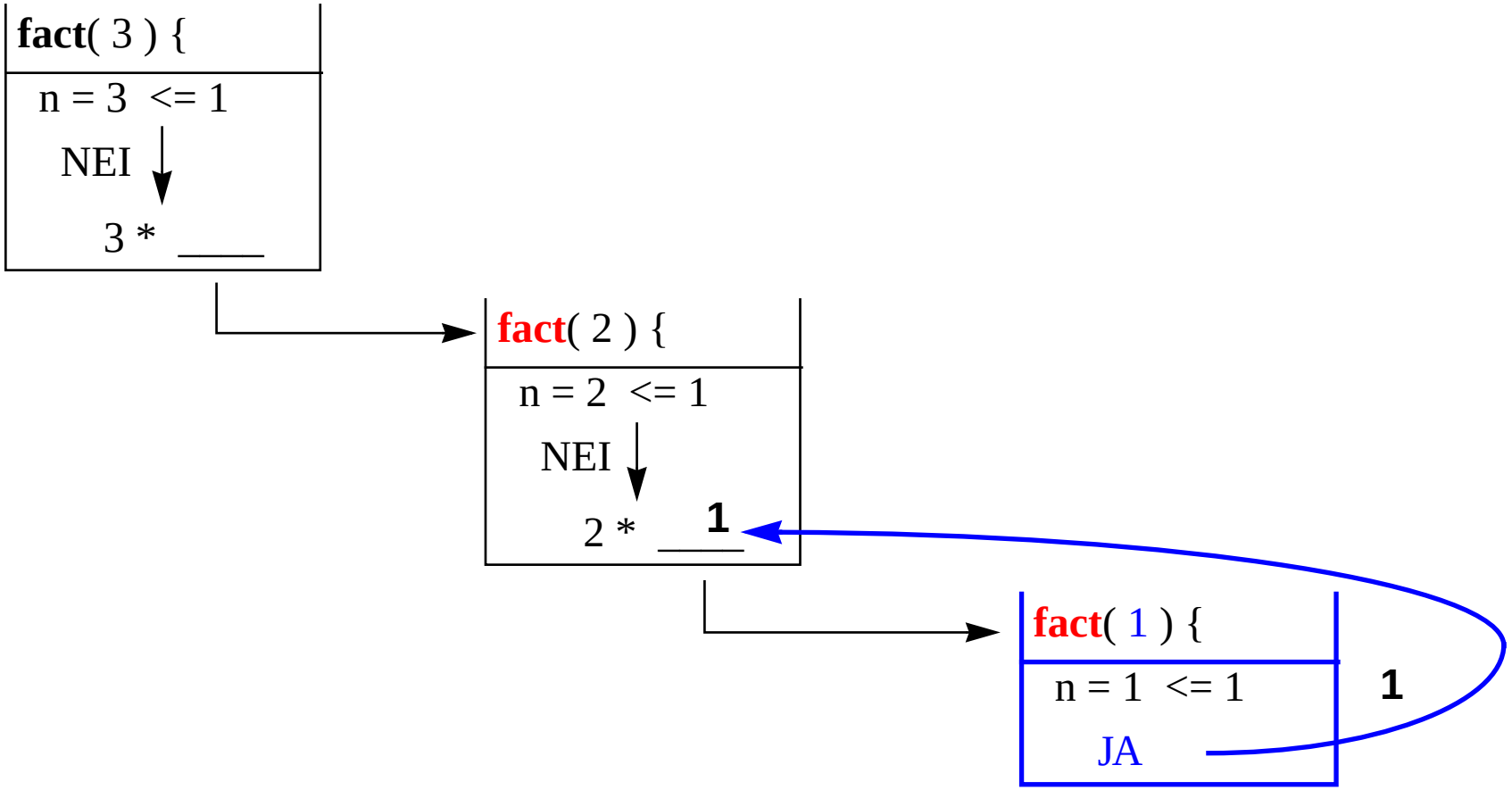
```
int fact( int n ) {
    if (n <= 1) return 1;
    else   return  n *  fact( n–1 );
}
```



1. En metode “kaller seg selv”

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

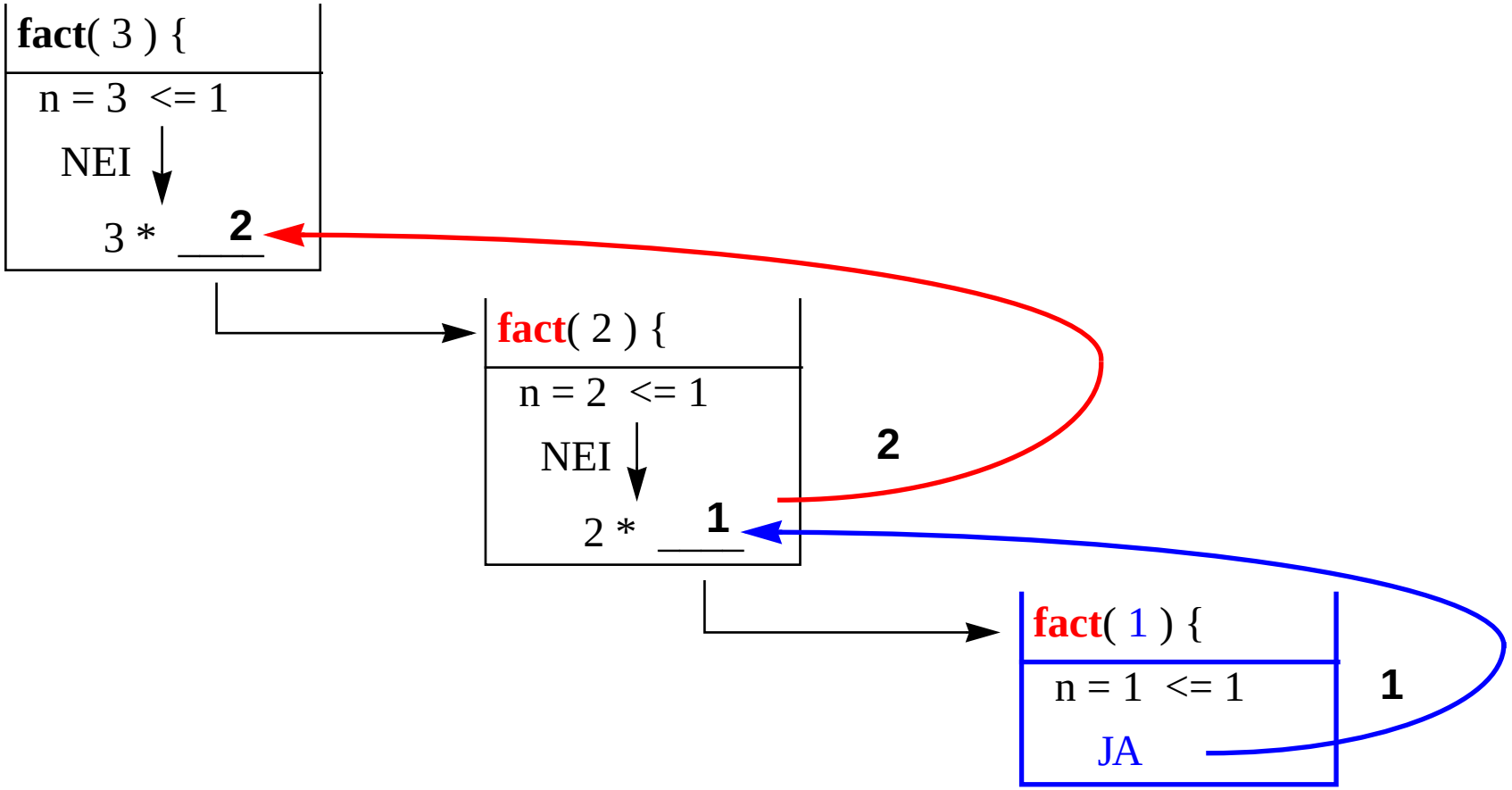
```
int fact( int n ) {
    if (n <= 1) return 1;
    else   return  n *  fact( n–1 );
}
```



1. En metode “kaller seg selv”

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

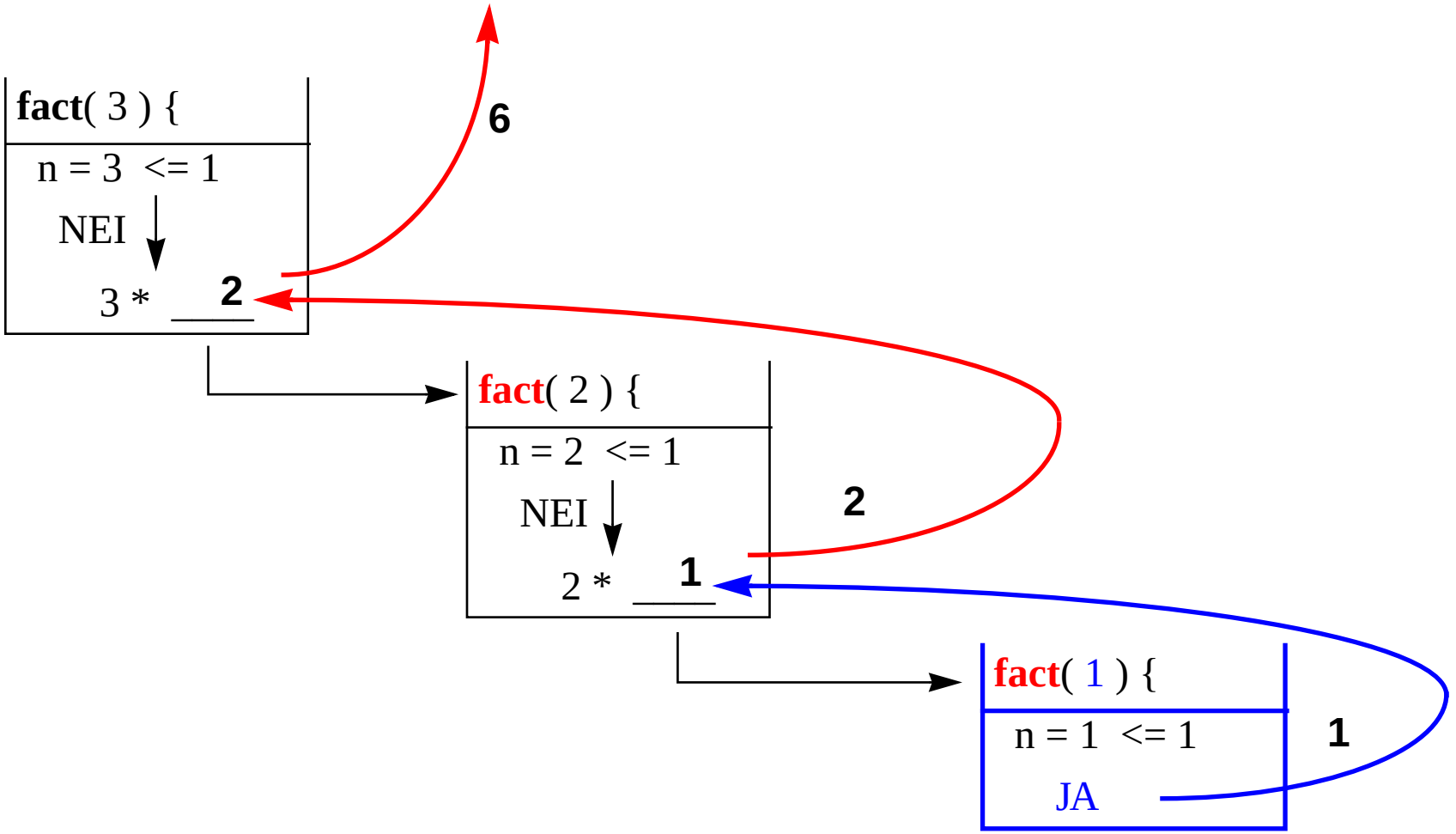
```
int fact( int n ) {
    if (n <= 1) return 1;
    else   return  n *  fact( n–1 );
}
```



1. En metode “kaller seg selv”

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

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



1. Rekursjons***tre*** og ***-dybde***

fact(0) = 0

fact(0) = 1

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

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

1. Rekursjons***tre*** og ***-dybde***

fact(0) = 0

fact(0) = 1

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

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

returner i **basistilfelle**

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

fact(0) = 1

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

```
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 sam-

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

fact(0) = 1

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

```
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 sam-

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

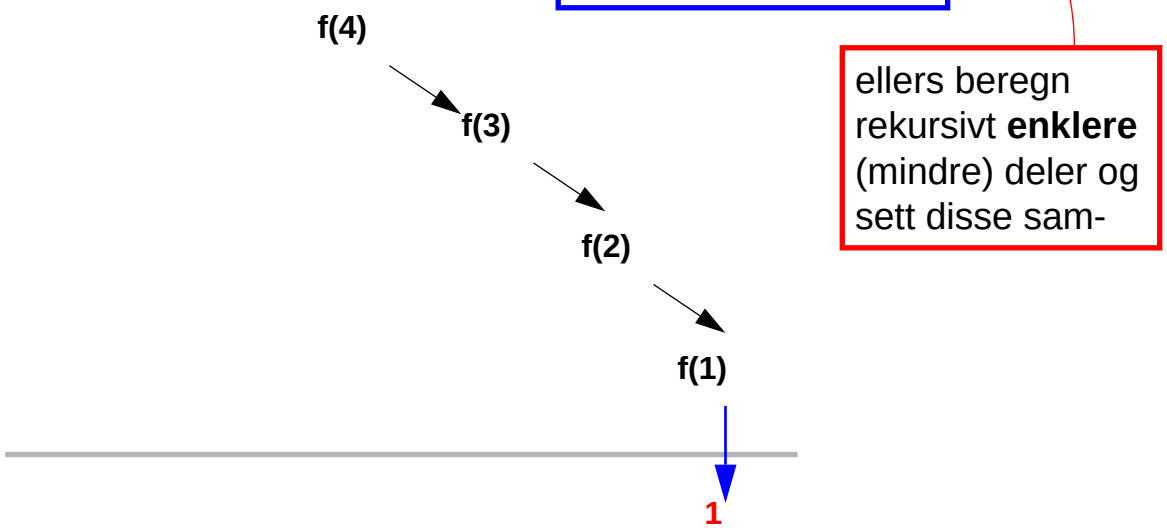
fact(0) = 1

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

```
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 sam-



1. Rekursjons*tre* og *-dybde*

fact(0) = 0

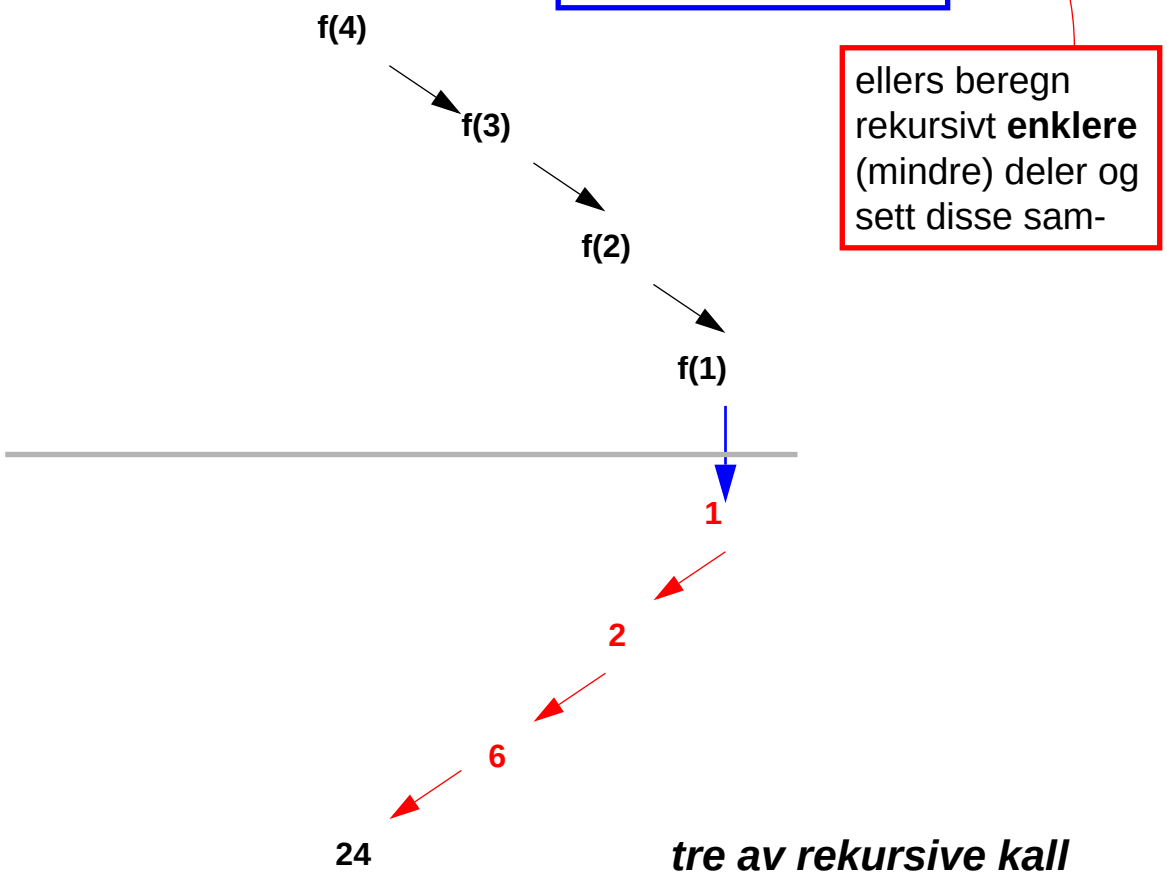
fact(0) = 1

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

```
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 sam-



tre av rekursive kall

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

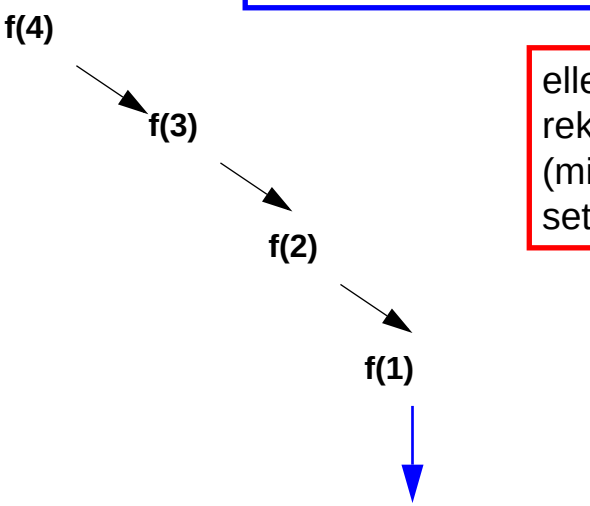
fact(0) = 1

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

```
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 sam-



tre av rekursive kall

f(4) ...?

— **f(3) ...?**

— — **f(2) ...?**

— — — **f(1) ...?**

rekkef¿lgen av kall

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

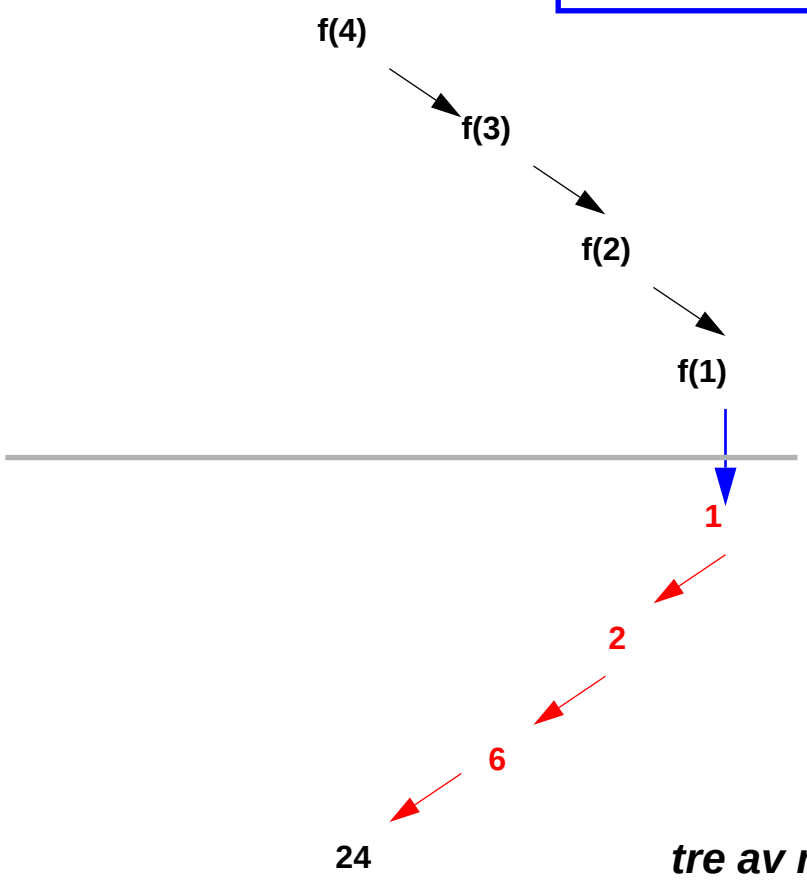
fact(0) = 1

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

```
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 sam-



tre av rekursive kall

f(4) ...?

— **f(3) ...?**

— — **f(2) ...?**

— — — **f(1) ...?**

— — — **> 1**

— — **> 2**

— **> 6**

> 24

rekkef¿lgen av kall

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

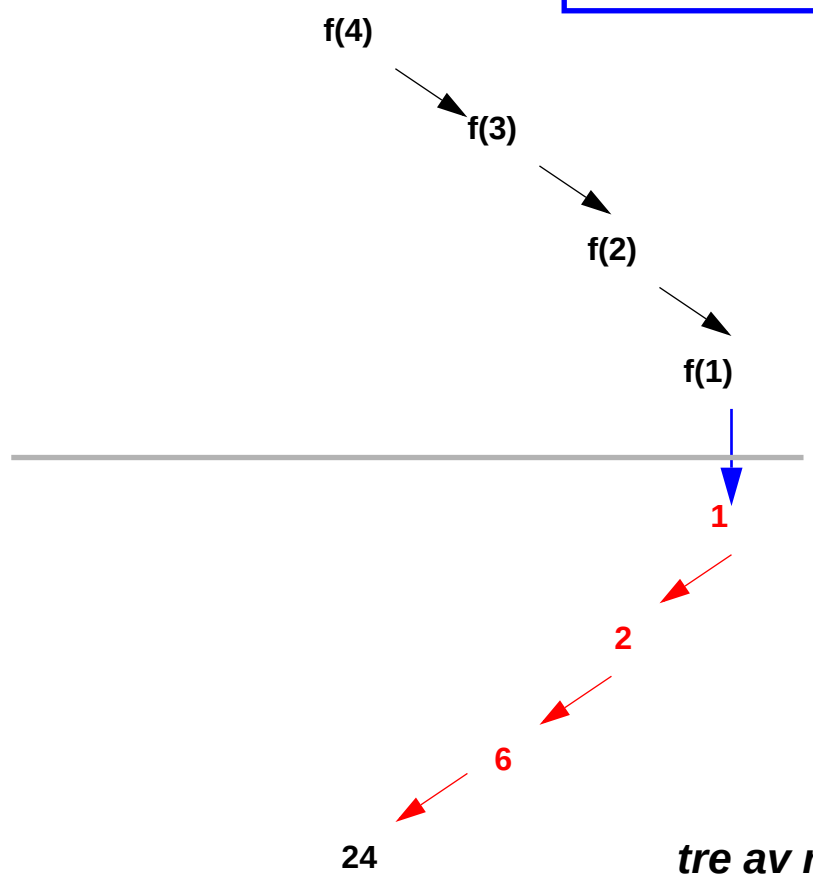
fact(0) = 1

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

```
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 sam-



tre av rekursive kall

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

rekkef¿lgen av kall

rekursjonsdybde

1. Rekursjons*tre* og *-dybde*

fact(0) = 0

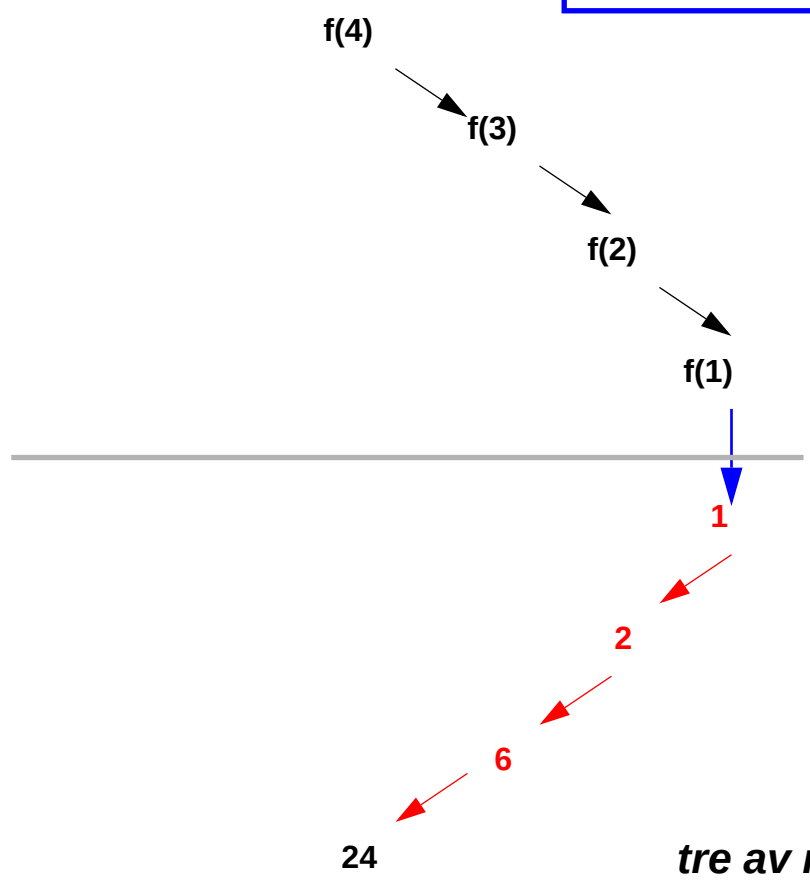
fact(0) = 1

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

```
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 sam-



tre av rekursive kall

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

rekkef¿lgen av kall

rekursjonsdybde

#svarte = #r¿de linjer = *# rekursive kall .....*

inntil basistilfelle er nådd (=høyden av treet)

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

fib(0) = 0

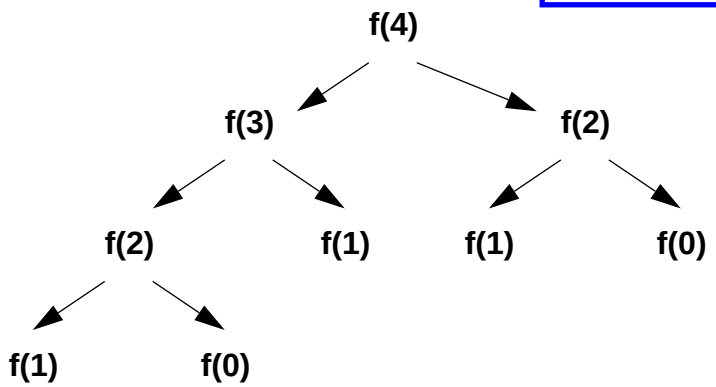
fib(1) = 1

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

```
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 sam-



tre av rekursive kall

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

fib(0) = 0

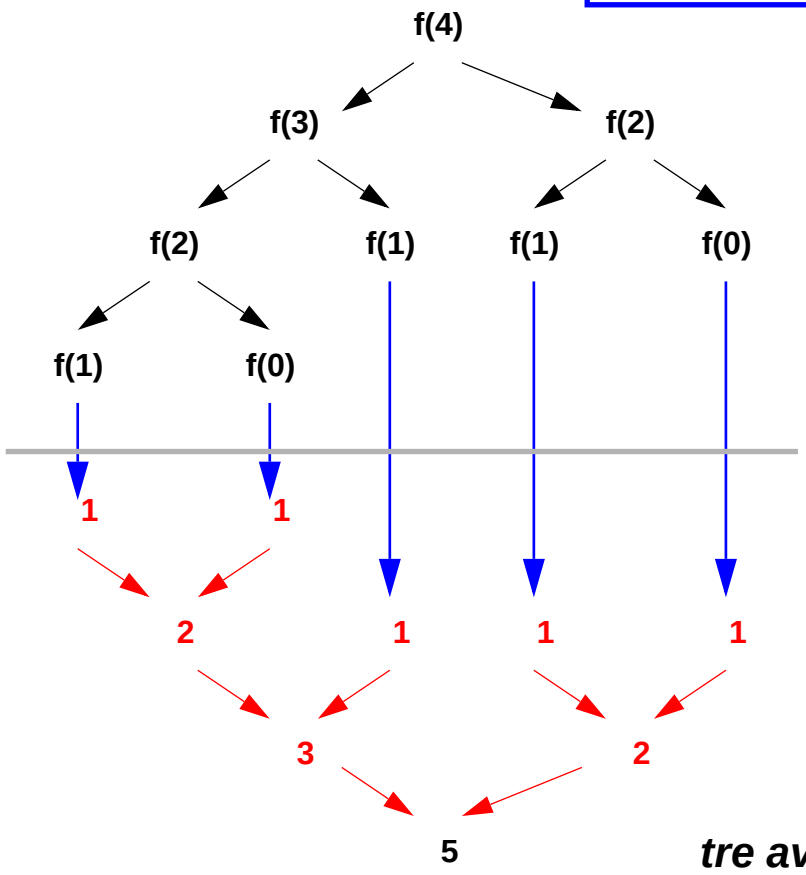
fib(1) = 1

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

```
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 sam-



tre av rekursive kall

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

fib(0) = 0

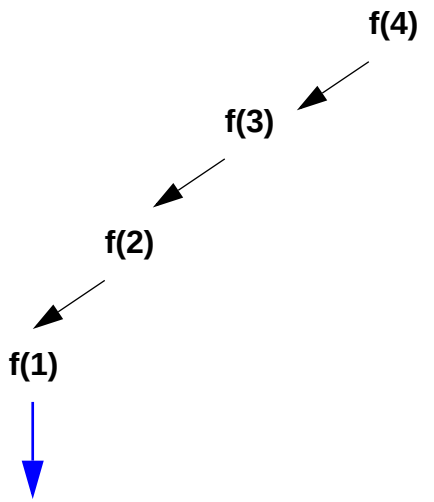
fib(1) = 1

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

```
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 sam-



f(4) ...?

— **f(3) ...?**

— — **f(2) ...?**

— — — **f(1) ...?**

rekkef¿lgen av kall

tre av rekursive kall

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

fib(0) = 0

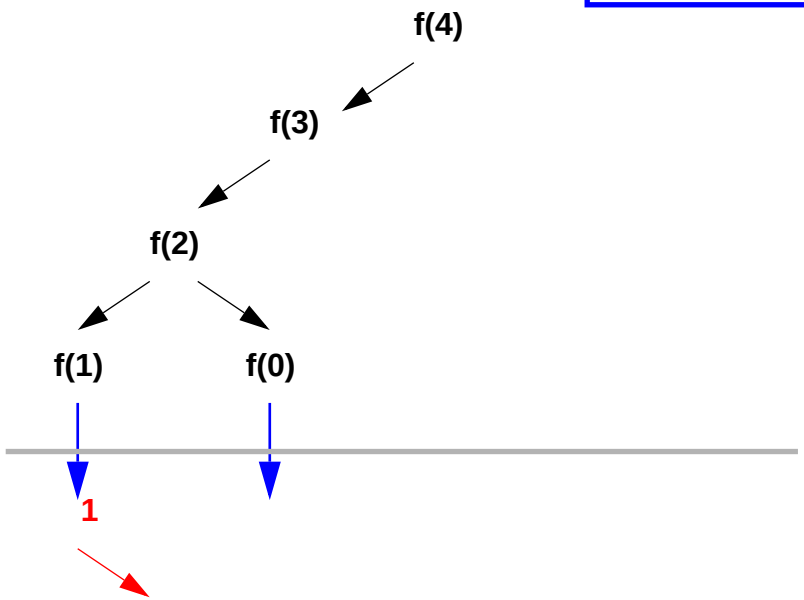
fib(1) = 1

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

```
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 sam-



f(4) ...?

— **f(3) ...?**

— — **f(2) ...?**

— — — **f(1) ...?**

— — — **> 1**

— — — **f(0) ...?**

rekkef¿lgen av kall

tre av rekursive kall

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

fib(0) = 0

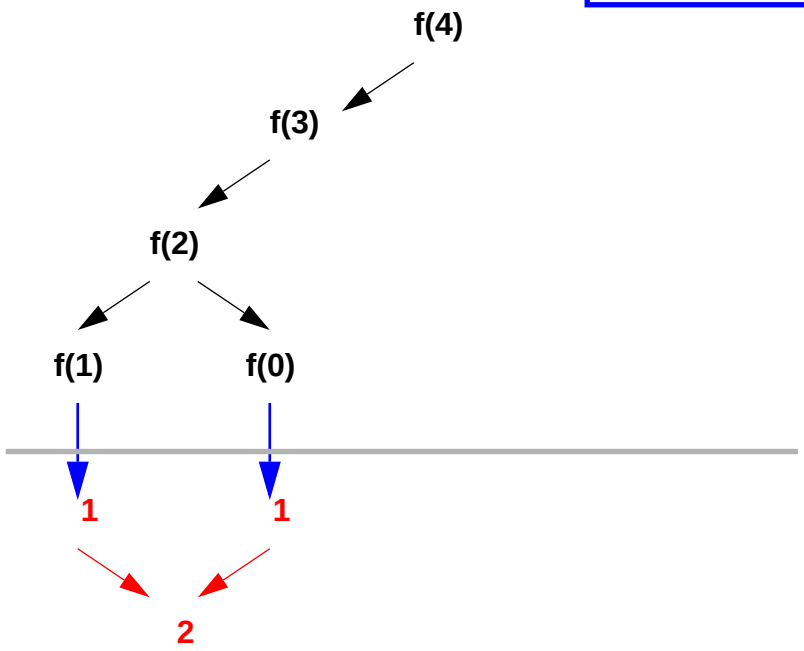
fib(1) = 1

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

```
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 sam-



tre av rekursive kall

f(4) ...?

— **f(3) ...?**

— — **f(2) ...?**

— — — **f(1) ...?**

— — — **> 1**

— — — **f(0) ...?**

— — — **> 1**

— — **> 2**

rekkef¿lgen av kall

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

fib(0) = 0

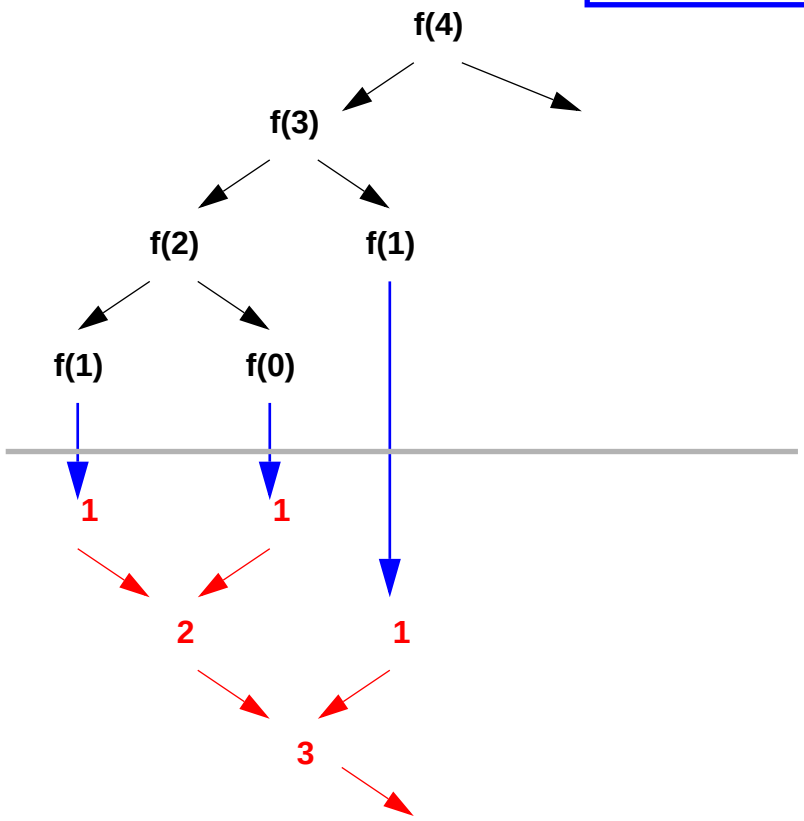
fib(1) = 1

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

```
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 sam-



tre av rekursive kall

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

rekkef¿lgen av kall

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

fib(0) = 0

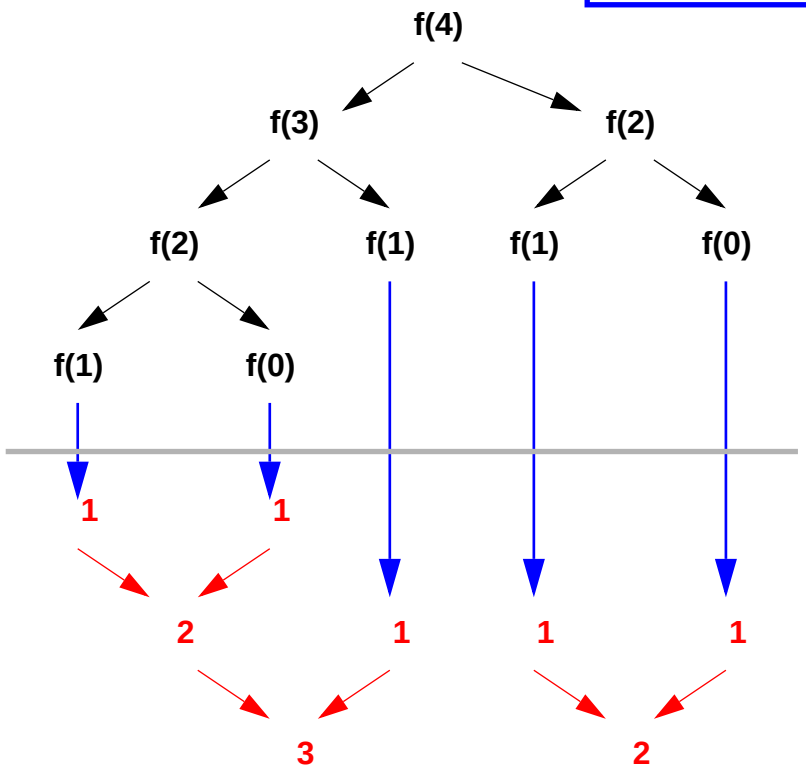
fib(1) = 1

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

```
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 sam-



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**

rekkef¿lgen av kall

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

fib(0) = 0

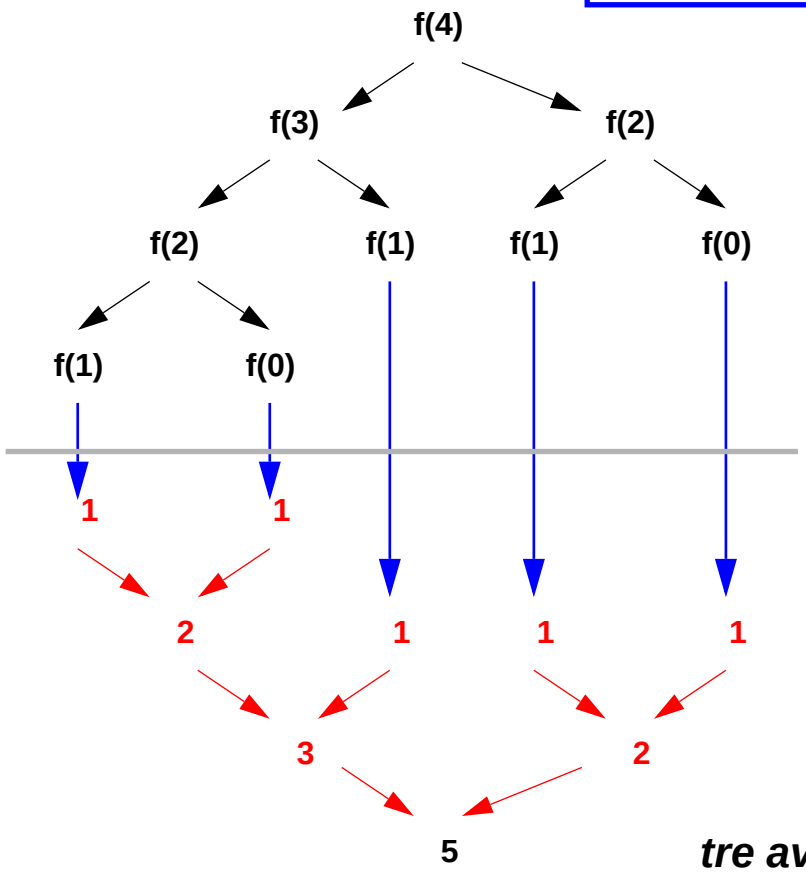
fib(1) = 1

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

```
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 sam-



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

rekkef¿lgen av kall

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

fib(0) = 0

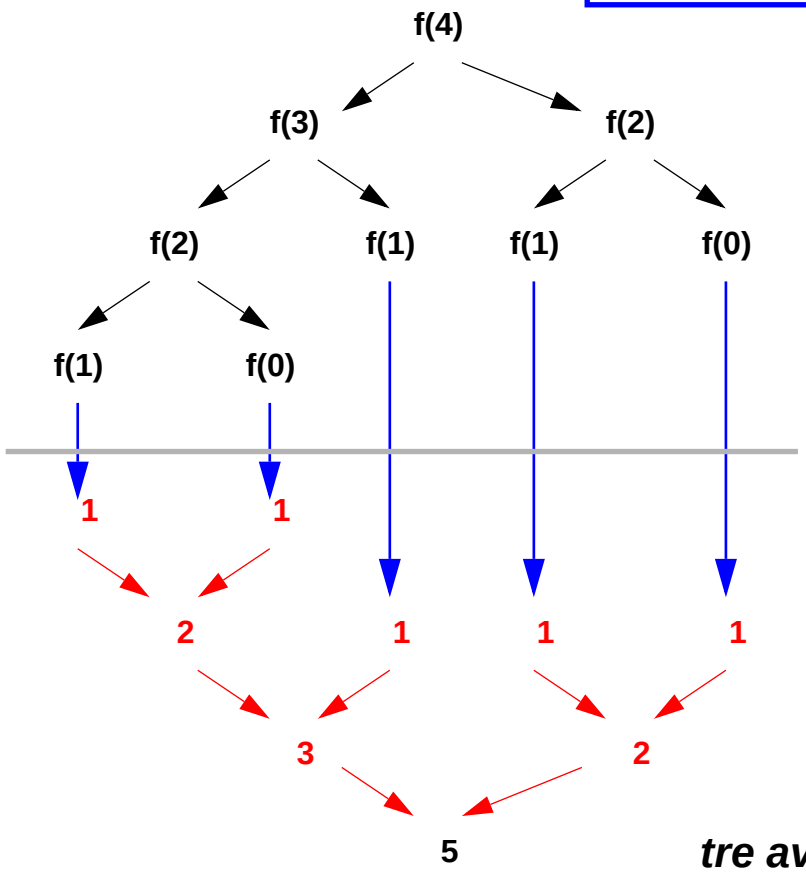
fib(1) = 1

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

```
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 sam-



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

rekkef¿lgen av kall

rekursjonsdybde

#svarte = #r¿de linjer = *# rekursive kall .....*

inntil basistilfelle er nådd (=høyden av treet)

Rekursjons***tre*** og ***-dybde;*** Quicksort

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**

Rekursjons***tre*** og ***-dybde;*** Quicksort

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); } }   */
```

Rekursjons***tre*** og ***-dybde;*** Quicksort

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**

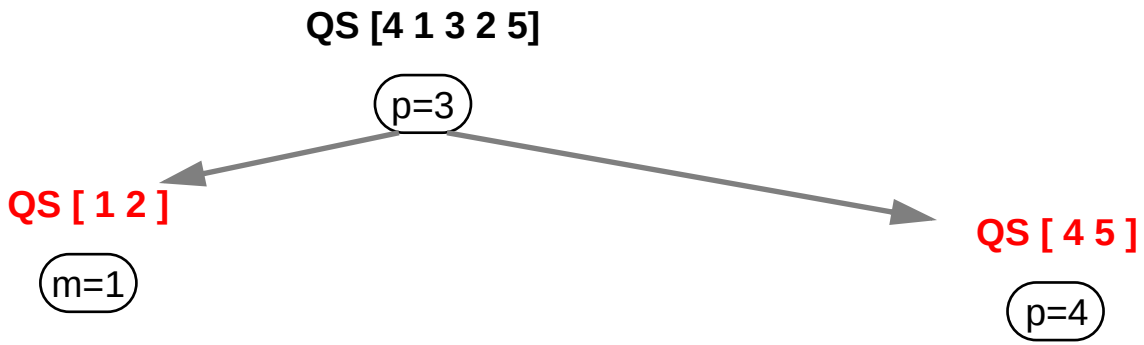
P = sorter input array A

```
/* 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); } }   */
```

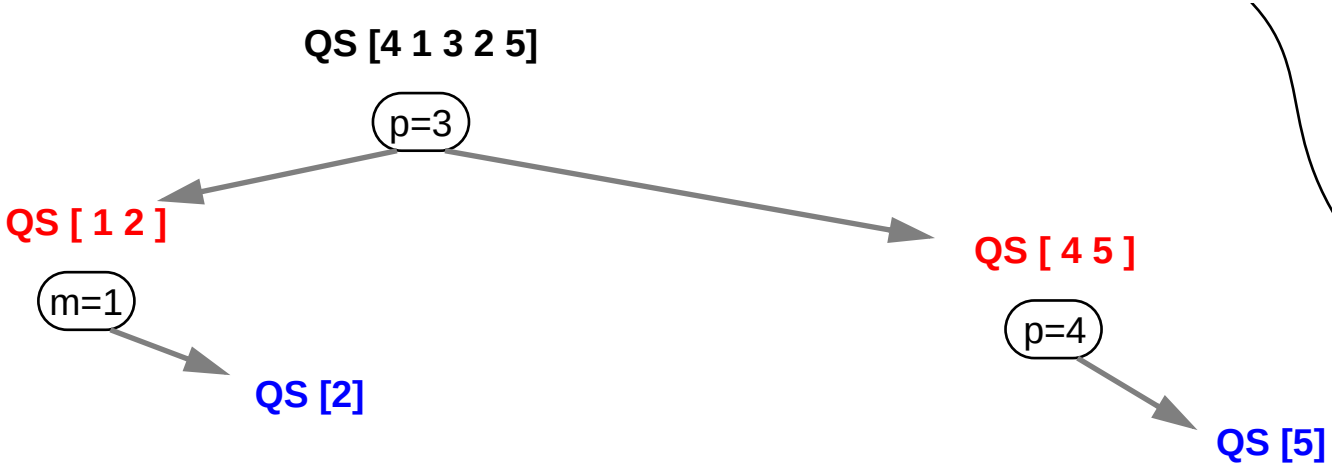
```
/* int[] QuickSort ( int[] A )  {
 *    int n = A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        p =  A[n/2]  – pivot
 *        t1 = [x : x ε A & x < p] og t3 = [x : x ε A & x > p];
 *        t2 = [x : x ε A & x = p] ;
 *        sorter rekursivt begge (mindre) array
 *              r1 = QuickSort ( t1 ) og
 *              r3 = QuickSort ( t3 )
 *        return r1 + t2 + r3
 *    }  }
 * */
```

QS [4 1 3 2 5]

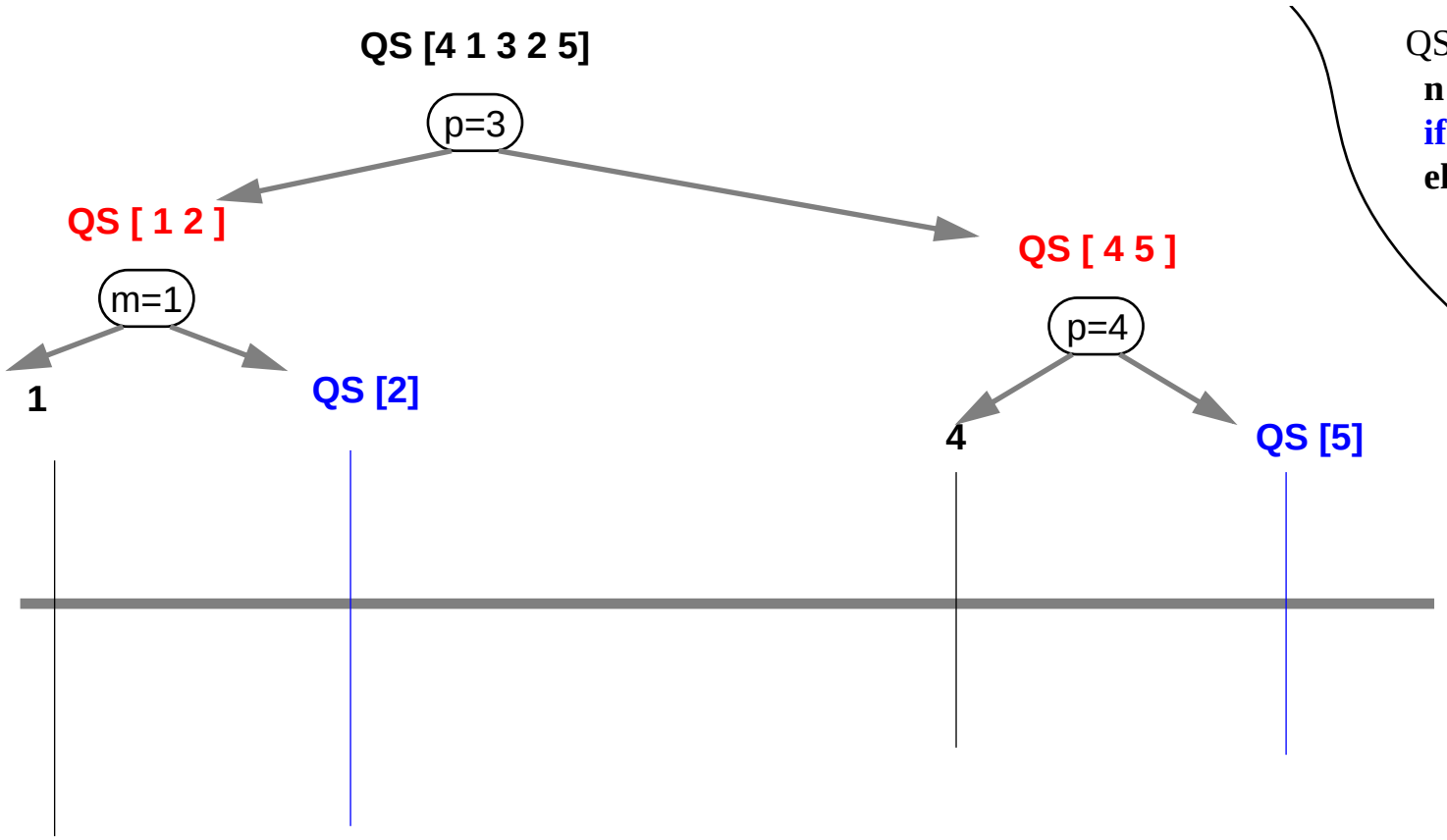
```
QS ( Ar[l...h] )
  n = Ar.lenght
  if (n == 1) return Ar;
  else
     p = A[n/2];
     return    QS ( Ar[< p] )
                + Ar[= p]
                + QS ( Ar[> p] )
```



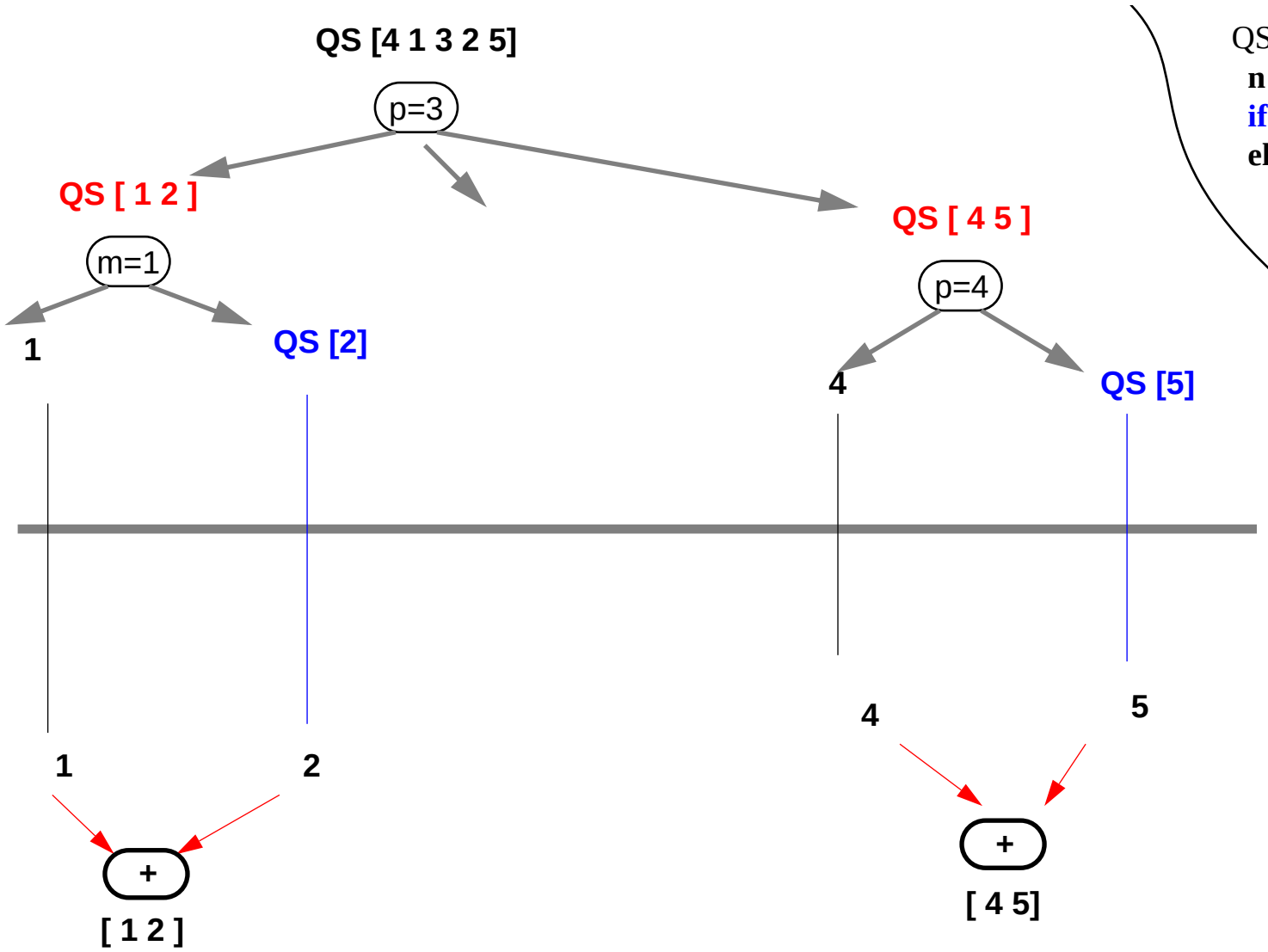

```
QS ( Ar[l...h] )
  n = Ar.lenght
  if (n == 1) return Ar;
  else
    p = A[n/2];
    return    QS ( Ar[< p] )
            + Ar[= p]
            + QS ( Ar[> p] )
```



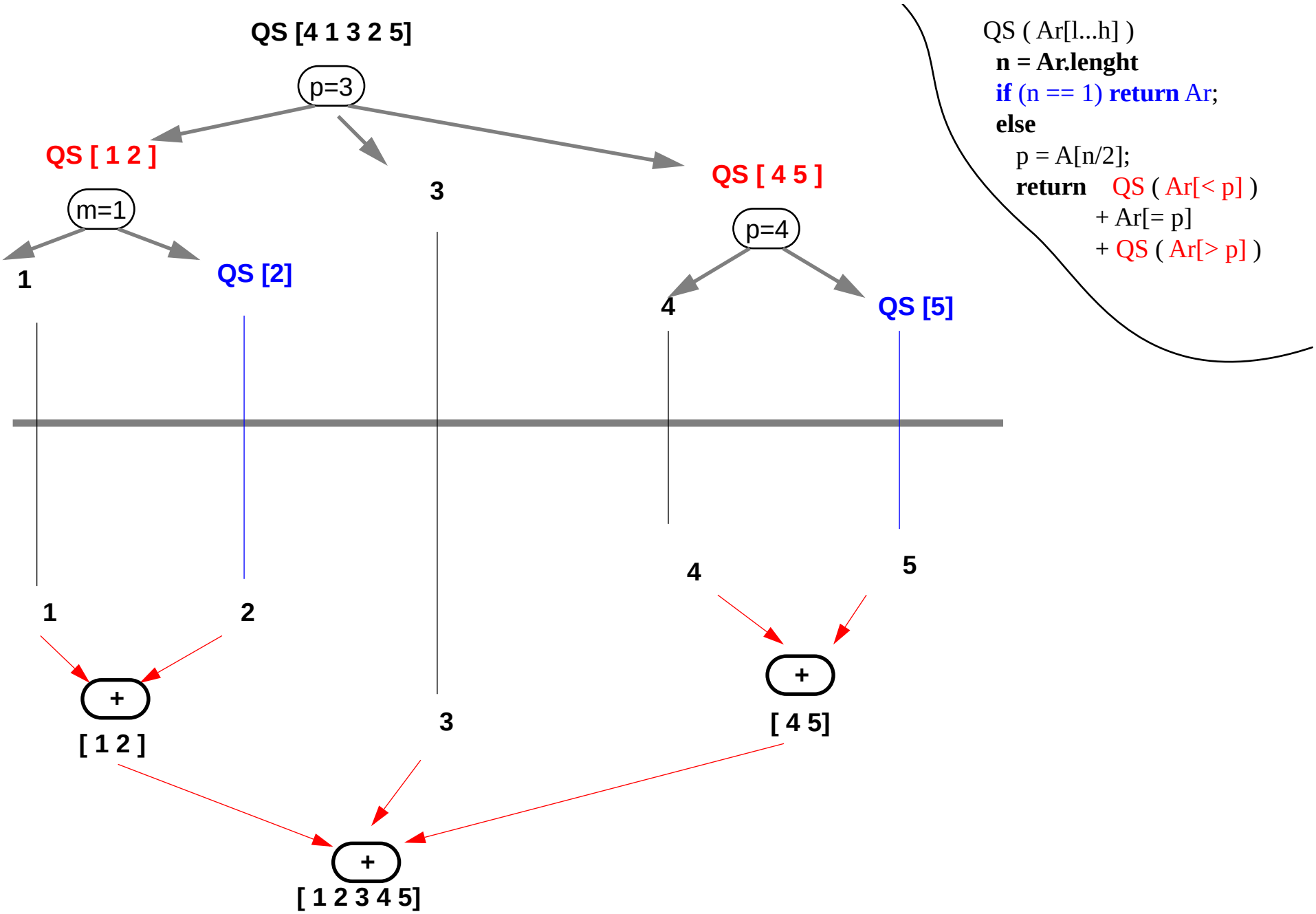

```
QS ( Ar[l...h] )
  n = Ar.lenght
  if (n == 1) return Ar;
  else
    p = A[n/2];
    return    QS ( Ar[< p] )
            + Ar[= p]
            + QS ( Ar[> p] )
```




```
QS ( Ar[l...h] )
  n = Ar.lenght
  if (n == 1) return Ar;
  else
    p = A[n/2];
    return    QS ( Ar[< p] )
            + Ar[= p]
            + QS ( Ar[> p] )
```

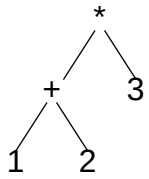



```
QS ( Ar[l...h] )
  n = Ar.lenght
  if (n == 1) return Ar;
  else
    p = A[n/2];
    return    QS ( Ar[< p] )
            + Ar[= p]
            + QS ( Ar[> p] )
```



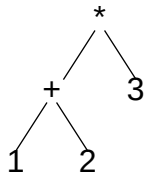

```
QS ( Ar[l...h] )
  n = Ar.lenght
  if (n == 1) return Ar;
  else
    p = A[n/2];
    return    QS ( Ar[< p] )
            + Ar[= p]
            + QS ( Ar[> p] )
```

pre-, post- og (for binære trær) innorden




```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

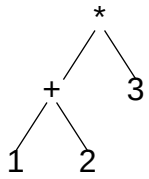
pre-, post- og (for binære trær) innorden




```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

1 + 2 * 3

pre-, post- og (for binære trær) innorden

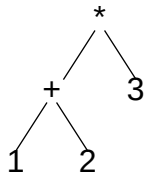



```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

1 + 2 * 3

```
pre(T) =
if (T =! null)
{   write(T.data)
    pre(T.left)
    pre(T.right)
}
```

pre-, post- og (for binære trær) innorden



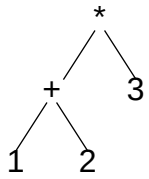

```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

1 + 2 * 3

```
pre(T) =
if (T =! null)
{   write(T.data)
    pre(T.left)
    pre(T.right)
}
```

* + 1 2 3

pre-, post- og (for binære trær) innorden




```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

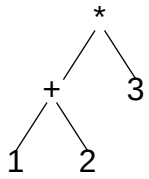
1 + 2 * 3

```
pre(T) =
if (T =! null)
{   write(T.data)
    pre(T.left)
    pre(T.right)
}
```

* + 1 2 3

```
post(T) =
if (T =! null)
{   post(T.left)
    post(T.right)
    write(T.data)
}
```

pre-, post- og (for binære trær) innorden




```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

1 + 2 * 3

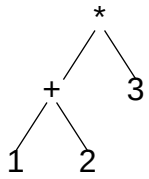
```
pre(T) =
if (T =! null)
{   write(T.data)
    pre(T.left)
    pre(T.right)
}
```

* + 1 2 3

```
post(T) =
if (T =! null)
{   post(T.left)
    post(T.right)
    write(T.data)
}
```

1 2 + 3 *

pre-, post- og (for binære trær) innorden




```
inn(T) =
if (T =! null)
{   in(T.left)
    write(T.data)
    in(T.right)
}
```

1 + 2 * 3

```
pre(T) =
if (T =! null)
{   write(T.data)
    pre(T.left)
    pre(T.right)
}
```

* + 1 2 3

```
post(T) =
if (T =! null)
{   post(T.left)
    post(T.right)
    write(T.data)
}
```

1 2 + 3 *

```
inn(T) =
if (T =! null)
{   write(()
    in(T.left)
    write(T.data)
    in(T.right)
    write())
}
```

(((1) + (2)) * (3))

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

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

s er: **n+1** et N

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

naturlige tall N:

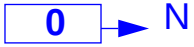
basis: **0** er et N

hvis **n** er et N

s er: **n+1** et N

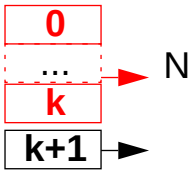
array av N: A(N)

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



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

naturlige tall N:	**array av N: A(N)**
basis: **0** er et N	basis **0 -> N** er A(N)
hvis **n** er et N	hvis **[0...k] -> N** er A(N)
s er: **n+1** et N	s er **[0...k,k+1] -> N** en A(N)



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

naturlige tall N:

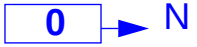
basis: **0** er et N

hvis **n** er et N

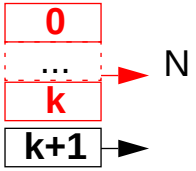
s er: **n+1** et N

array av N: A(N)

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



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



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

Lister av N: L(N):

basis: **null** er en L(N) ●

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

naturlige tall N:

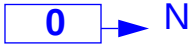
basis: **0** er et N

hvis **n** er et N

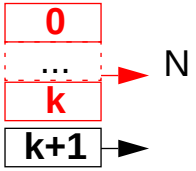
s er: **n+1** et N

array av N: A(N)

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



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

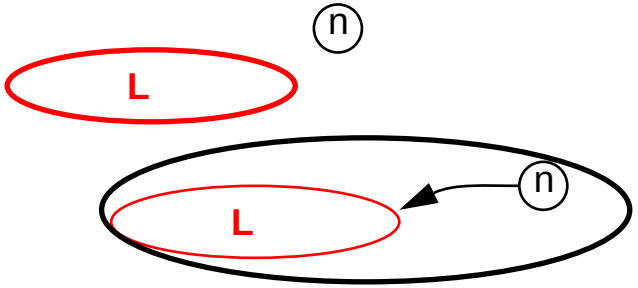


s er **[0...k,k+1] -> N** en A(N)

Lister av N: L(N):

basis: **null** er en L(N) ●

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



s er: **(n,L)** en L(N)

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

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

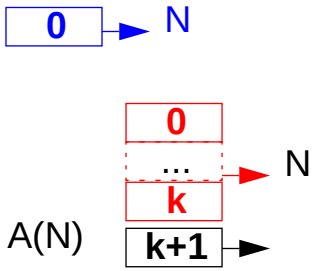
s er: **n+1** et N

array av N: A(N)

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

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

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

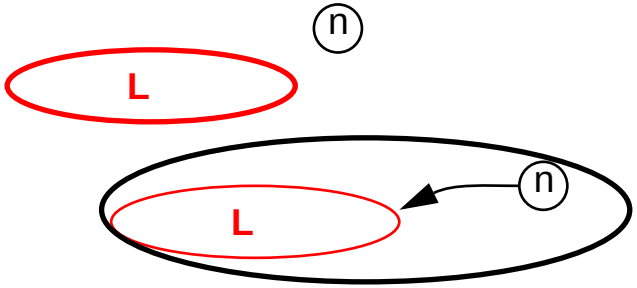


Lister av N: L(N):

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

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

s er: **(n,L)** en L(N)

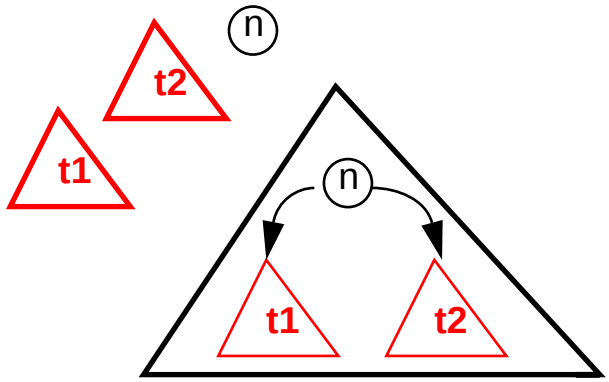


Bin¾e Tr¾av N: BT(N):

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

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

s er: **(t1, n, t2)** et BT(N)



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

"Strukturell ordering"

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

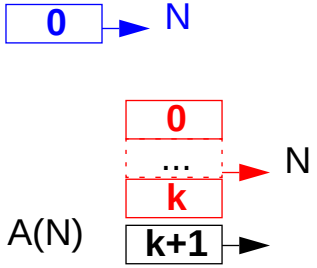
s er: **n+1** et N

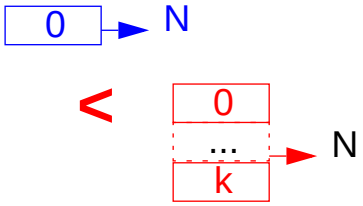
array av N: A(N)

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

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

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



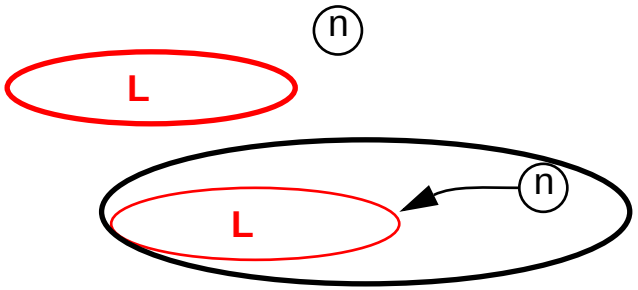


Lister av N: L(N):

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

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

s er: **(n,L)** en L(N)

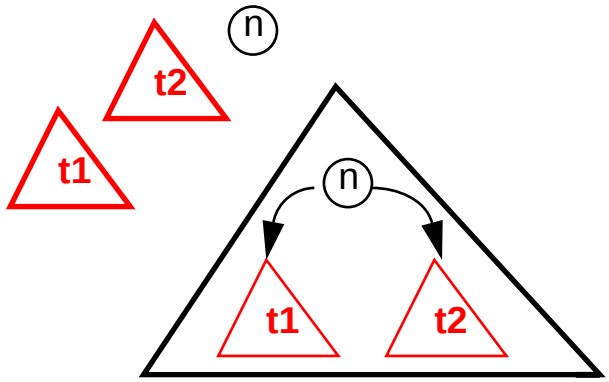


Bin³/4re Tr³/4av N: BT(N):

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

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

s er: **(t1, n, t2)** et BT(N)



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

"Strukturell ordering"

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

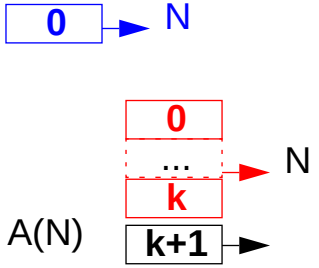
s er: **n+1** et N

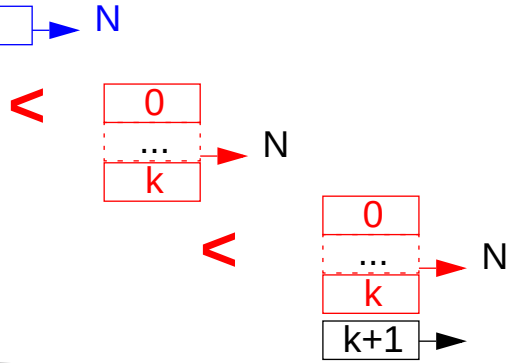
array av N: A(N)

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

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

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



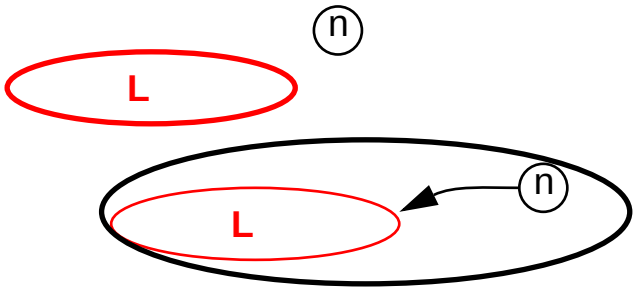


Lister av N: L(N):

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

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

s er: **(n,L)** en L(N)

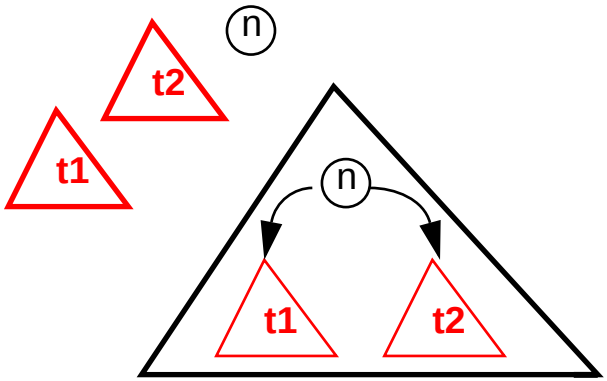


Bin*re Tr*av N: BT(N):

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

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

s er: **(t1, n, t2)** et BT(N)



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

"Strukturell ordering"

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

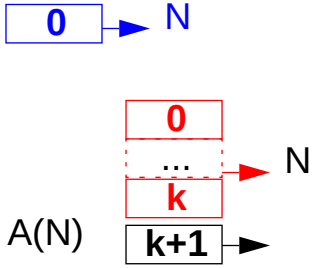
s er: **n+1** et N

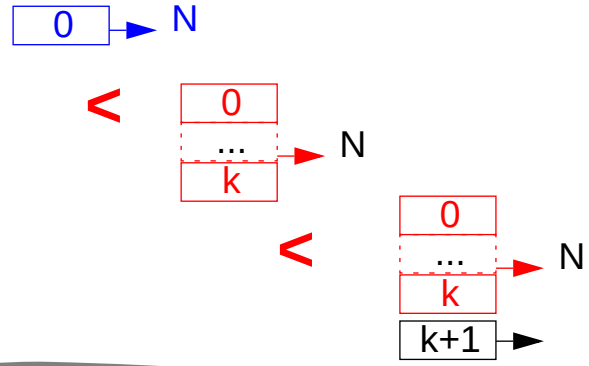
array av N: A(N)

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

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

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



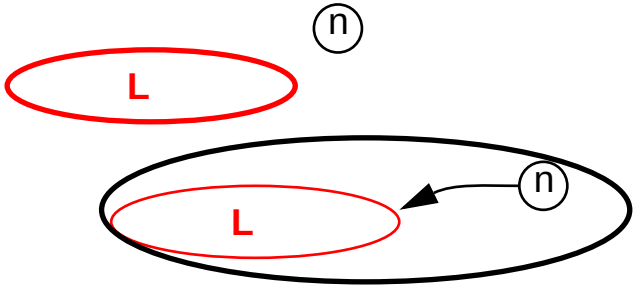


Lister av N: L(N):

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

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

s er: **(n,L)** en L(N)

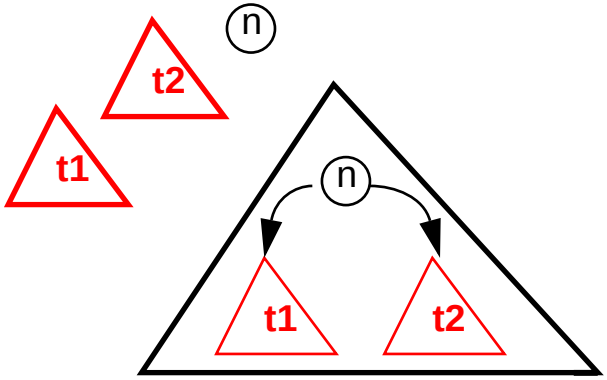


Bin¾re Tr¾r av N: BT(N):

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

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

s er: **(t1, n, t2)** et BT(N)



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

"Strukturell ordering"

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

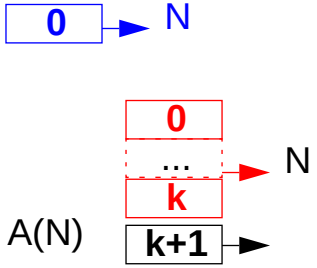
s er: **n+1** et N

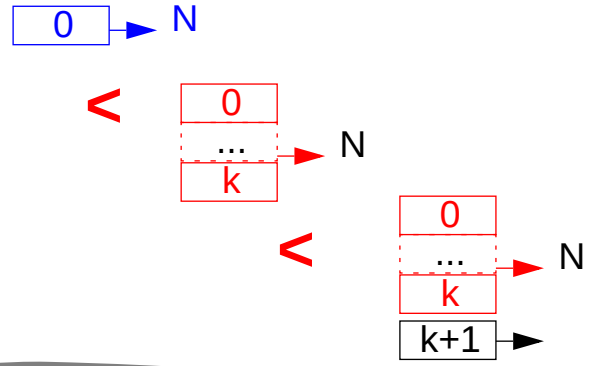
array av N: A(N)

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

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

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



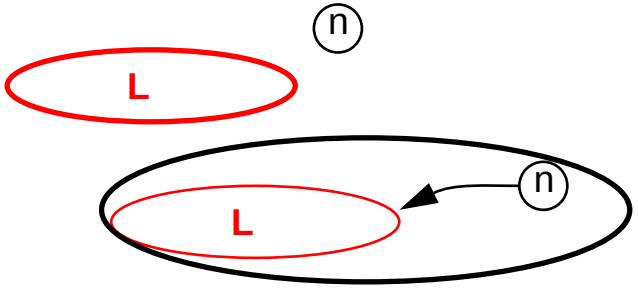


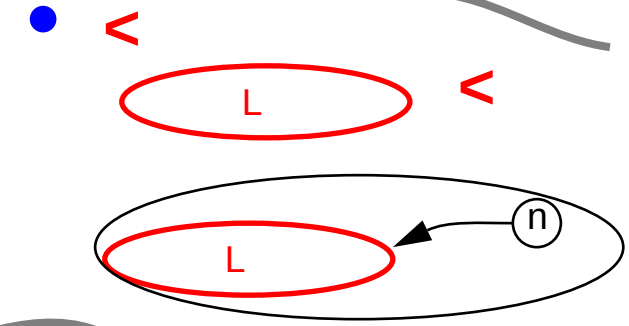
Lister av N: L(N):

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

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

s er: **(n,L)** en L(N)



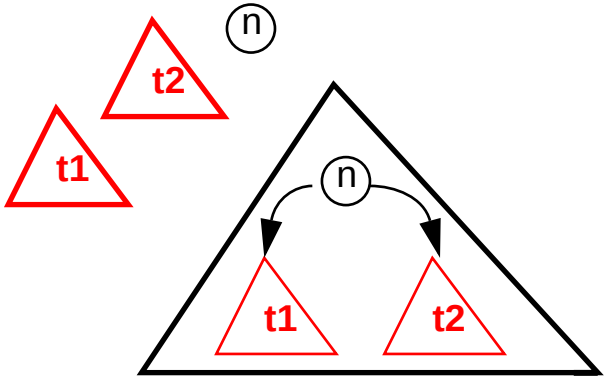


Bin³/₄e Tr³/₄av N: BT(N):

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

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

s er: **(t1, n, t2)** et BT(N)



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

"Strukturell ordering"

naturlige tall N:

basis: **0** er et N

hvis **n** er et N

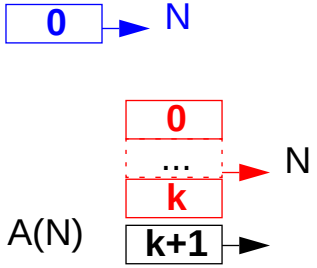
s er: **n+1** et N

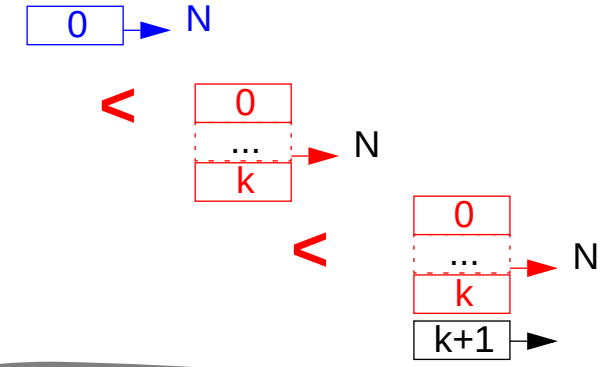
array av N: A(N)

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

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

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



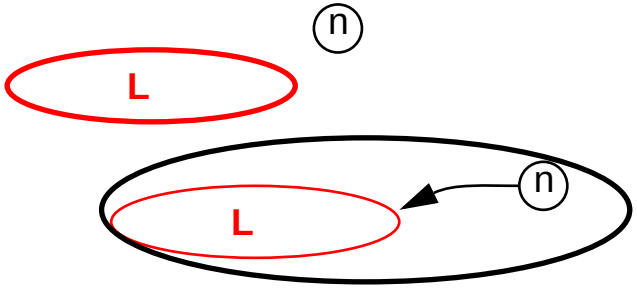


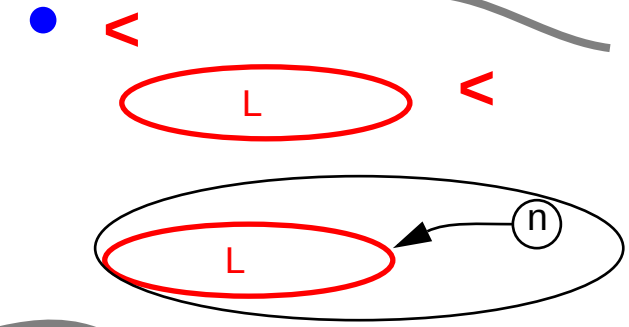
Lister av N: L(N):

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

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

s er: **(n,L)** en L(N)



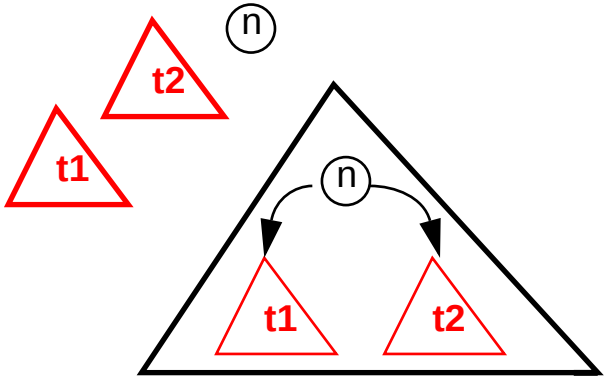


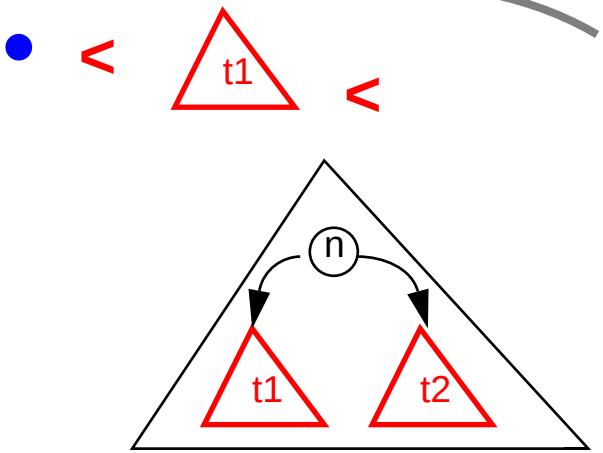
Bin³⁄₄e Tr³⁄₄ av N: BT(N):

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

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

s 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);
}
```

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); } }
```

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)
```

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; }
```

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; }
```

BinTre[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

Hva beregnes av følgende... ?

f (x: int) = if (x <= 0) then 0 else x + f(x—1) ; f(n) = ?

*f (x: int) = if (x <= 0) then 0 else x*x + f(x—1) ;* f(n) = ?

f (x: int) = if (x <= 0) then 0 else 1 + f(x—1) ; f(n) = ?

f (x: int) = if (x <= 0) then 5 else f(x—1) ; f(n) = ?

f (x: int) = if (x <= 0) then 1 else f(x—1) + f(x—1) ; f(n) = ?

*f (x: int) = if (x <= 0) then 1 else 2 * f(x—1) ;* f(n) = ?

f (x: int) = if (x <= 1) then 0 else 1+ f(x—2) ; f(n) = ?

Tegn tre av rekursive kall for hver f-funksjon ved kall *f(3);*

Hva beregnes av følgende... ?

f (x: int) = if (x <= 0) then 0 else x + f(x—1) ; f(n) = ?

*f (x: int) = if (x <= 0) then 0 else x*x + f(x—1) ;* f(n) = ?

f (x: int) = if (x <= 0) then 0 else 1 + f(x—1) ; f(n) = ?

f (x: int) = if (x <= 0) then 5 else f(x—1) ; f(n) = ?

f (x: int) = if (x <= 0) then 1 else f(x—1) + f(x—1) ; f(n) = ?

*f (x: int) = if (x <= 0) then 1 else 2 * f(x—1) ;* f(n) = ?

f (x: int) = if (x <= 1) then 0 else 1+ f(x—2) ; f(n) = ?

Tegn tre av rekursive kall for hver f-funksjon ved kall *f(3);*

Gjensidig rekursjon:

(La [] betegne en tom liste mens x::xs en liste med x som f¿rste elementet etterfulgt av listen xs;
f.eks. listen [1,2,3] kan skrives som 1::2::3::[].)

flat([]) = []
flat(l::ls) = aux(l, ls)
aux([], ls) = flat(ls)
aux(x::xs, ls) = x :: aux(xs, ls)

Hva slags argument (av hvilken type) forventes av *flat* ?
Hva blir resultatet av *flat([[1,2], [2,3]]) ;* ? av *flat([[1,2], [2,3], [], [4,5,6]]) ;* ?

Hva beregnes av følgende... ?

f (x: int) = if (x <= 0) then 0 else x + f(x—1) ; f(n) = 1 + 2 + 3 + ... + n

*f (x: int) = if (x <= 0) then 0 else x*x + f(x—1) ;* f(n) = $1^2 + 2^2 + 3^2 + ... + n^2$

f (x: int) = if (x <= 0) then 0 else 1 + f(x—1) ; f(n) = 1 + 1 + 1 + ... + 1 = n

f (x: int) = if (x <= 0) then 5 else f(x—1) ; f(n) = 5

f (x: int) = if (x <= 0) then 1 else f(x—1) + f(x—1) ; f(n) = 2^n

*f (x: int) = if (x <= 0) then 1 else 2 * f(x—1) ;* f(n) = 2^n

f (x: int) = if (x <= 1) then 0 else 1+ f(x—2) ; f(n) = n//2

Tegn tre av rekursive kall for hver f-funksjon ved kall *f(3);*

Gjensidig rekursjon:

(La [] betegne en tom liste mens x::xs en liste med x som f¿rste elementet etterfulgt av listen xs;
f.eks. listen [1,2,3] kan skrives som 1::2::3::[].)

flat([]) = []
flat(l::ls) = aux(l, ls)
aux([], ls) = flat(ls)
aux(x::xs, ls) = x :: aux(xs, ls)

Hva slags argument (av hvilken type) forventes av *flat* ?
Hva blir resultatet av *flat([[1,2], [2,3]]) ;* ? av *flat([[1,2], [2,3], [], [4,5,6]]) ;* ?

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(n) = bergen 2^n

1. *hva gjør jeg når* **n** *= 0 ?*

2. *hvordan konstruere løsning for* **n** *utfra løsning for* **n–1** *?*

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(n) = bergen 2^n

1. *hva gjør jeg når* **n** *= 0 ?*
   returner $2^0 = 1$
2. *hvordan konstruere løsning for* **n** *utfra løsning for* **n–1** *?*
   $2^n = 2 * 2^{n-1}$ *, altså*
   *returner 2 * P(***n–1***)*

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(n) = bergen 2^n

1. *hva gjør jeg når* **n** *= 0 ?*
   returner $2^0 = 1$
2. *hvordan konstruere løsning for* **n** *utfra løsning for* **n–1** *?*
   $2^n = 2 * 2^{n-1}$ *, altså*
   *returner 2 * P(***n–1***)*

P(0) = 1
P(n+1) = 2 * P(n)

P(x) = if (x=0) then 1
else 2 * P(n—1)

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**

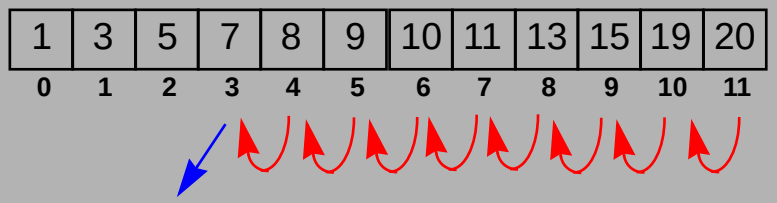
P = finn et gitt element x i en array A

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); }
```

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)

1	3	5	7	8	9	10	11	13	15	19	20
0	1	2	3	4	5	6	7	8	9	10	11

Hvis A er sortert ...

BS(A, 7, 0,11)

```
/*   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); }
```

1	3	5	7	8	**9**	10	11	13	15	19	20
0	1	2	3	4	**5**	6	7	8	9	10	11

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**

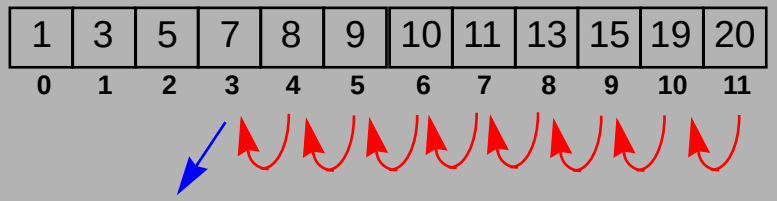
P = finn et gitt element x i en array A

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);   }
```

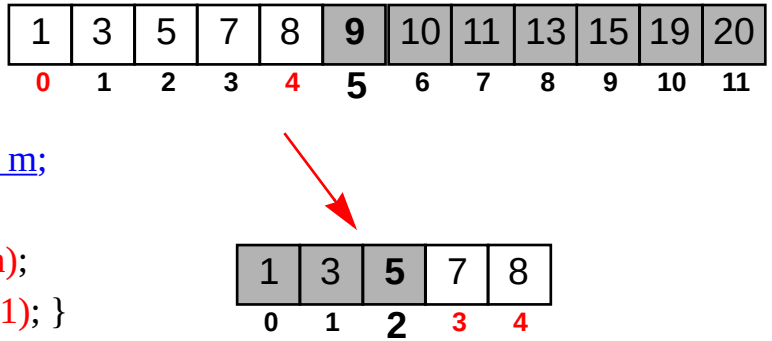
FE(A, 7, 11)



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" (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)

1	3	5	7	8	9	10	11	13	15	19	20
0	1	2	3	4	5	6	7	8	9	10	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)

1	3	5	7	8	9	10	11	13	15	19	20
0	1	2	3	4	**5**	6	7	8	9	10	11

1	3	5	7	8
0	1	**2**	3	4

7	8
3	4

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(s) = les input strengen tegn-for-tegn s lenge det er et siffer,
og returner heltallet (samt resten av strengen):

P(“101”) = (101,[]), P(“2sd”) = (2,“sd”), P(“abcd”) = (0,“abcd”), osv.

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(s) = les input strengen tegn-for-tegn s lenge det er et siffer,
og returner heltallet (samt resten av strengen):

P("101") = (101,[]), P("2sd") = (2,"sd"), P("abcd") = (0,"abcd"), osv.

1. *hva er basistilfelle? –*

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(s) = les input strengen tegn-for-tegn s lenge det er et siffer,
og returner heltallet (samt resten av strengen):

P("101") = (101,[]), P("2sd") = (2,"sd"), P("abcd") = (0,"abcd"), osv.

1. *hva er basistilfelle? –* **s** *er tom, eller* **"d::rest"** *der tegnet* **d** *ikke er et siffer,*
   hva gjør jeg da?

2. *hvordan konstruere løsning for* **"d::rest"** *utfra løsning for* **"rest"** *(når* **d** *er et siffer) ?*

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(s) = les input strengen tegn-for-tegn s lenge det er et siffer,
og returner heltallet (samt resten av strengen):

P("101") = (101,[]), P("2sd") = (2,"sd"), P("abcd") = (0,"abcd"), osv.

1. *hva er basistilfelle? –* **s** *er tom, eller* **"d::rest"** *der tegnet* **d** *ikke er et siffer,*
   hva gjør jeg da? – returnerer det akkumulerte resultatet og resten av strengen
   – **P(i, []) = (i, [])**
   – **P(i, d::rest) = (i, d::rest)**

2. *hvordan konstruere løsning for* **"d::rest"** *utfra løsning for* **"rest"** *(når* **d** *er et siffer) ?*

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(s) = les input strengen tegn-for-tegn s lenge det er et siffer,
og returner heltallet (samt resten av strengen):

P("101") = (101,[]), P("2sd") = (2,"sd"), P("abcd") = (0,"abcd"), osv.

1. *hva er basistilfelle? –* **s** *er tom, eller* **"d::rest"** *der tegnet* **d** *ikke er et siffer,*
   hva gjør jeg da? – returnerer det akkumulerte resultatet og resten av strengen
   – **P(i, []) = (i, [])**
   – **P(i, d::rest) = (i, d::rest)**

2. *hvordan konstruere løsning for* **"d::rest"** *utfra løsning for* **"rest"** *(når* **d** *er et siffer) ?*
   – **i*10 + d** *er den nye akkumulerte tallverdien*
   – *fortsett med* **P(i*10 + d, rest)**

```
parseInt(s:string) = first( P(0,s) );

P(i,[]) = ( i, [] )
P(i,d::rest) = if (not digit(d)) then ( i, d::rest )
                    else  P( i*10 + int(d), rest )
```

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;
}
```

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

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 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 / init;
     else return  Kroppen(n, Rekursiv(oppdater(n)));
}
```

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 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;
}
```

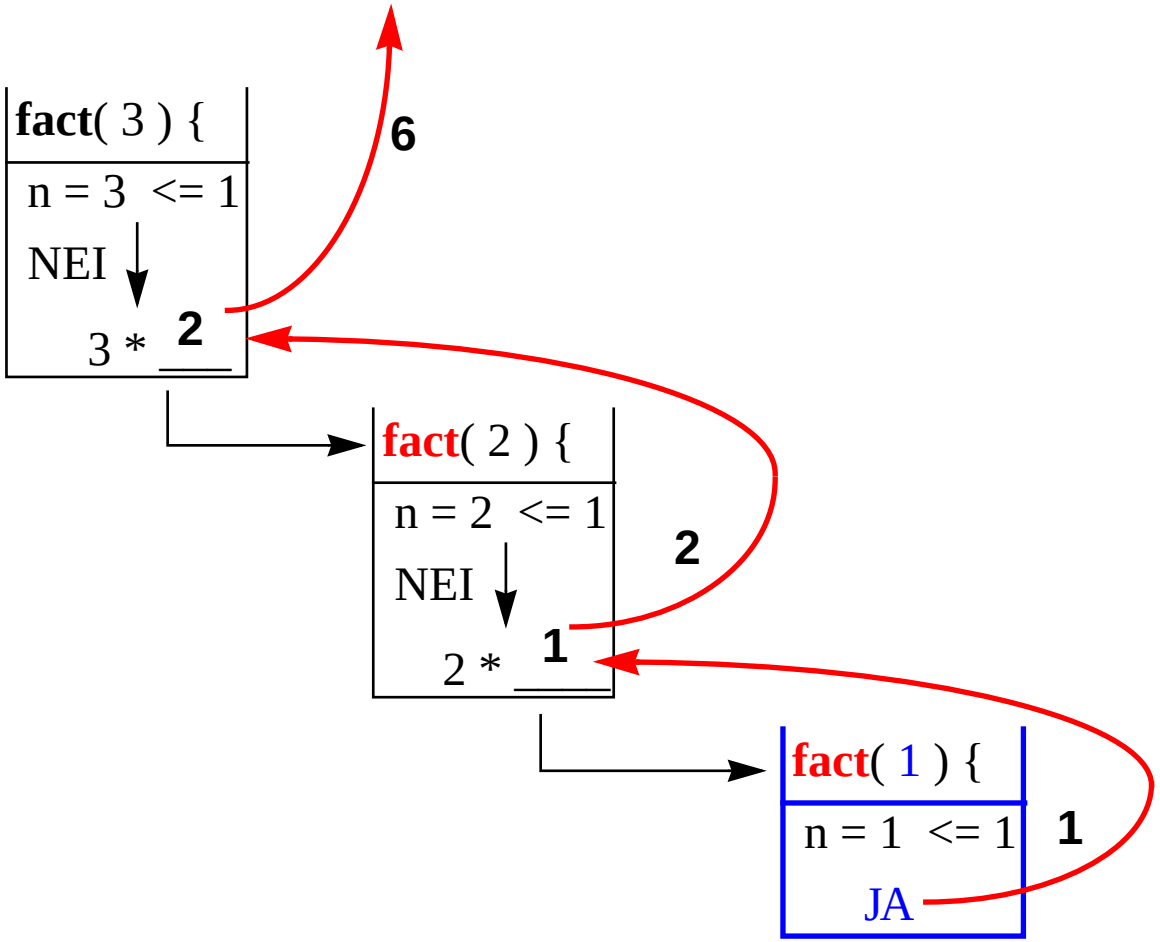
```
int Rekursiv(int n) {
     if ( !fortsett(n) ) return basetilfelle / init;
     else return  Kroppen(n, Rekursiv(oppdater(n)));
}
```

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

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

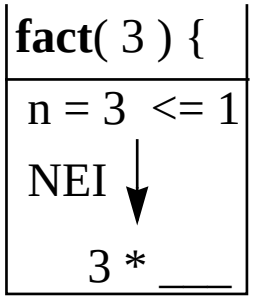
4a. Rekursjon implementeres med en stabel

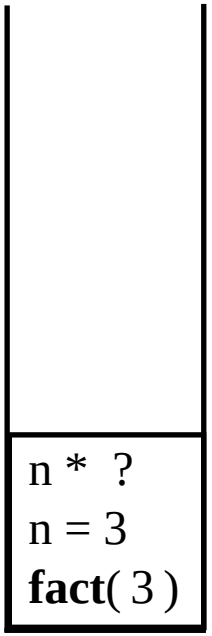
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```



4a. Rekursjon implementeres med en stabel

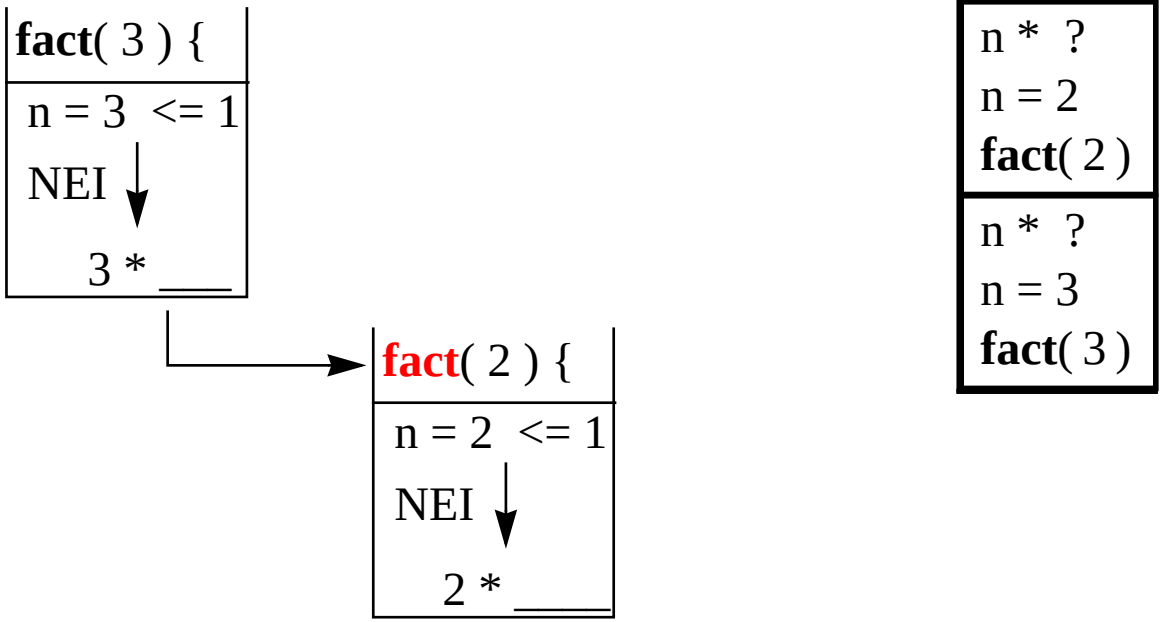
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```





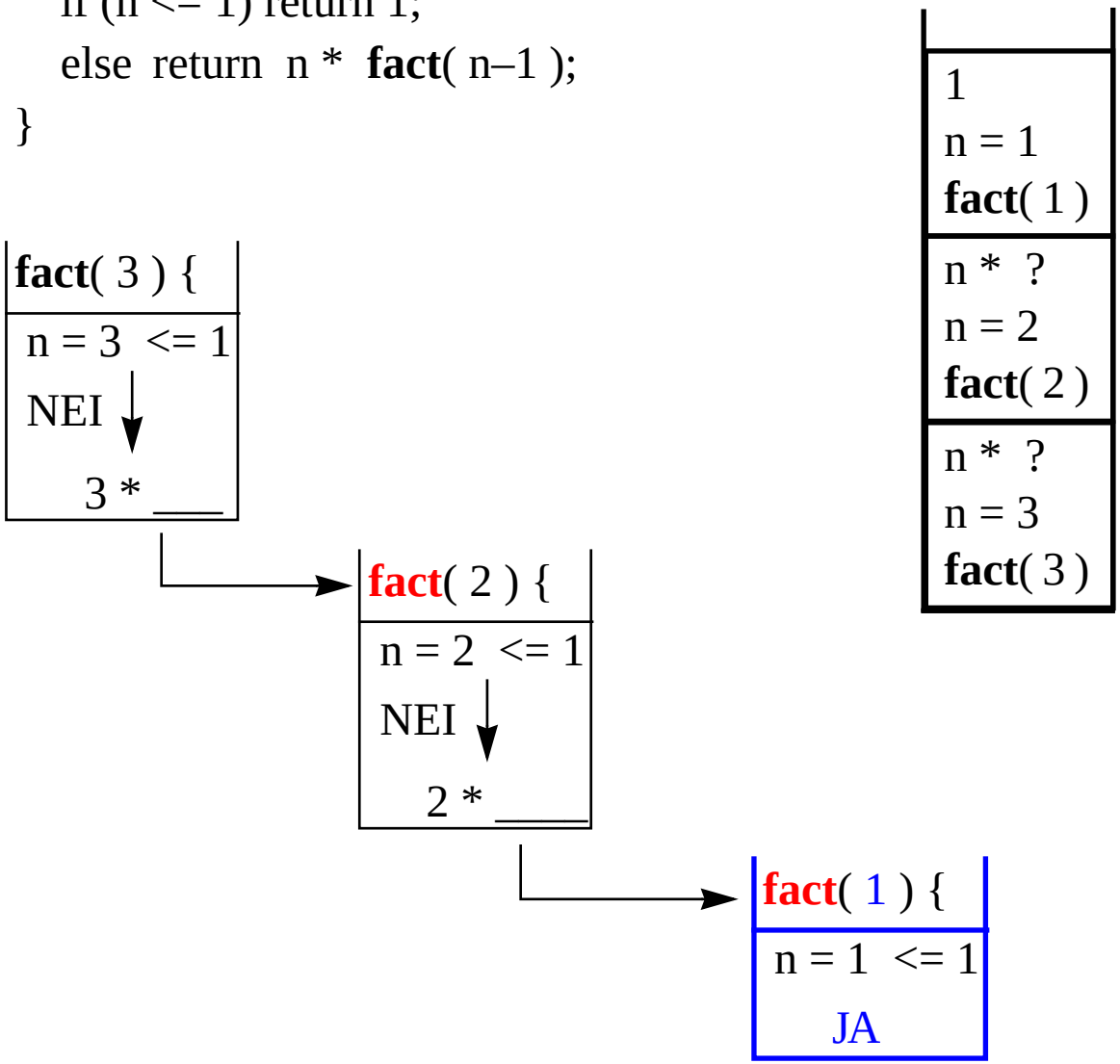
4a. Rekursjon implementeres med en stabel

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



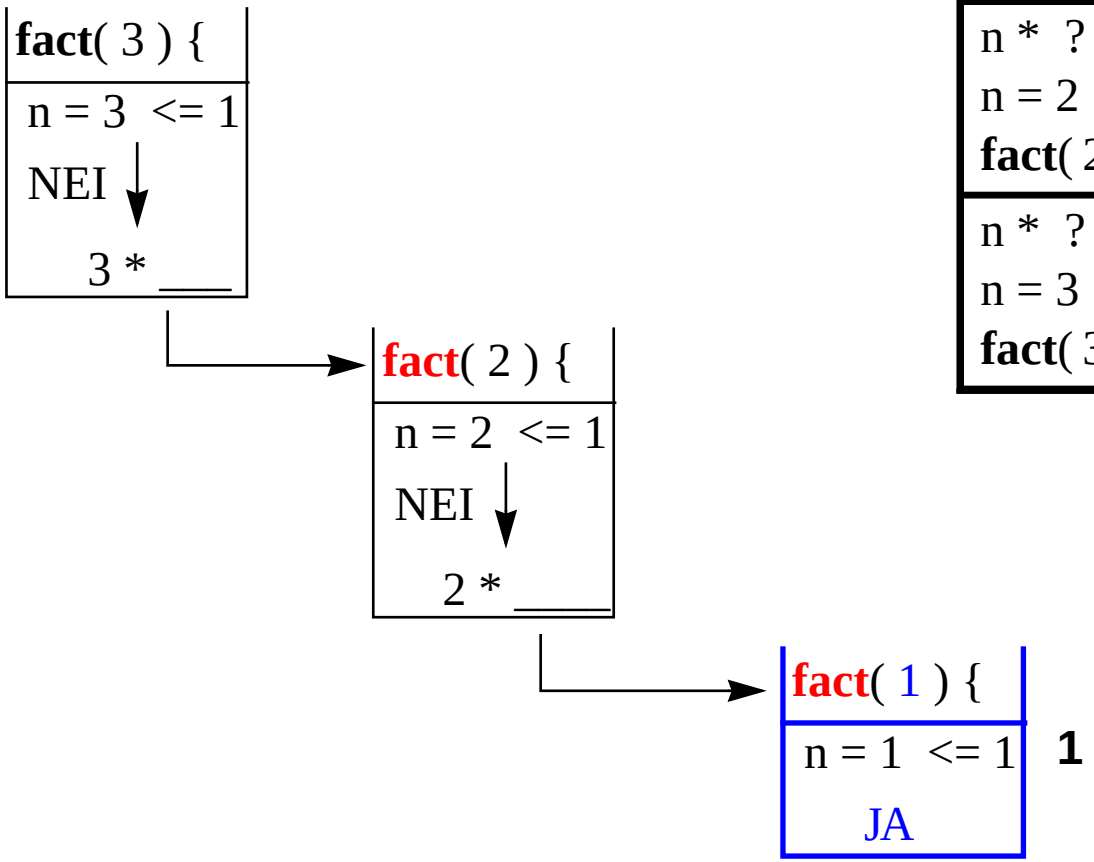
4a. Rekursjon implementeres med en stabel

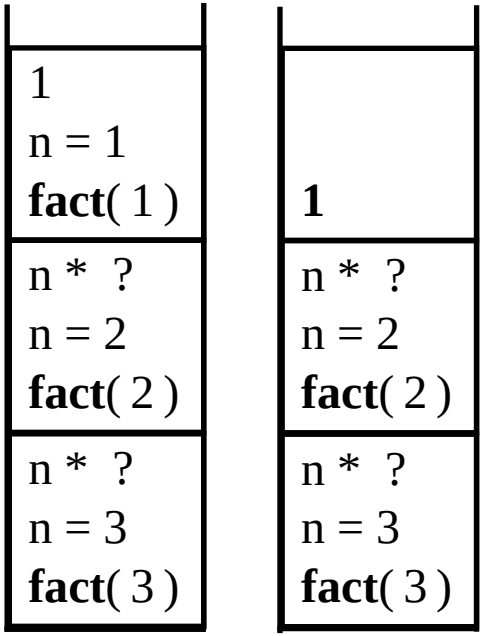
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```



4a. Rekursjon implementeres med en stabel

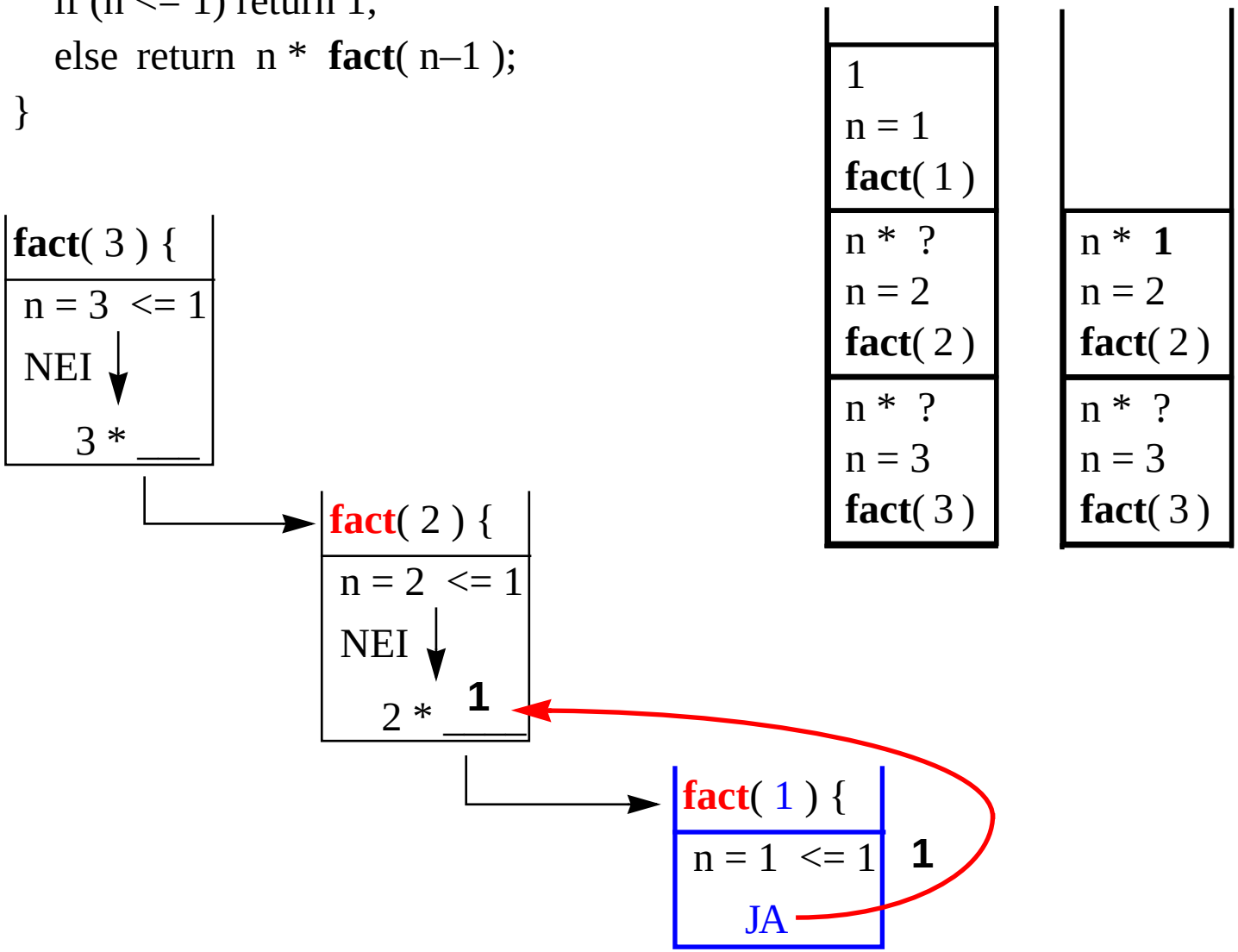
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```





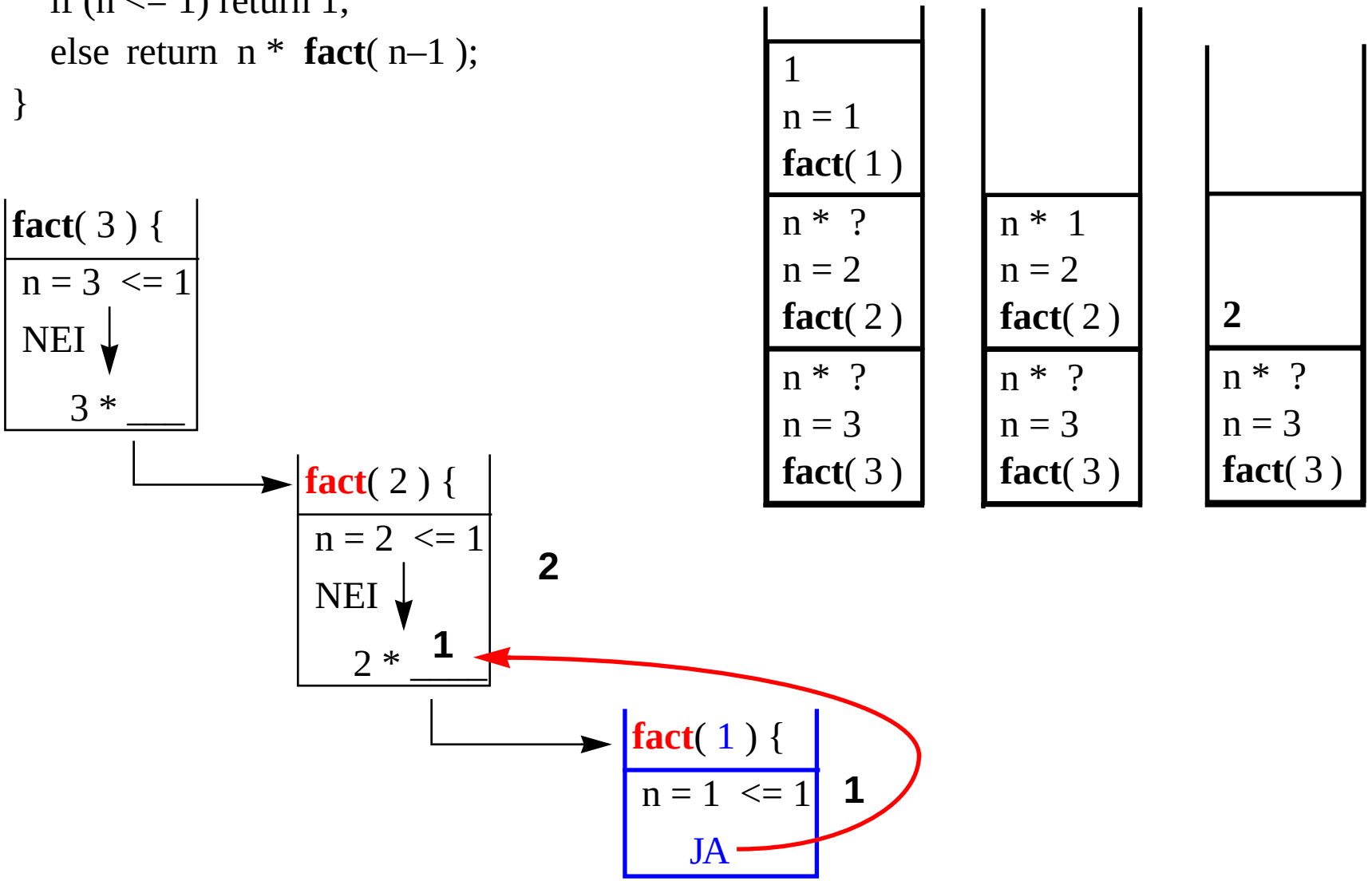
4a. Rekursjon implementeres med en stabel

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



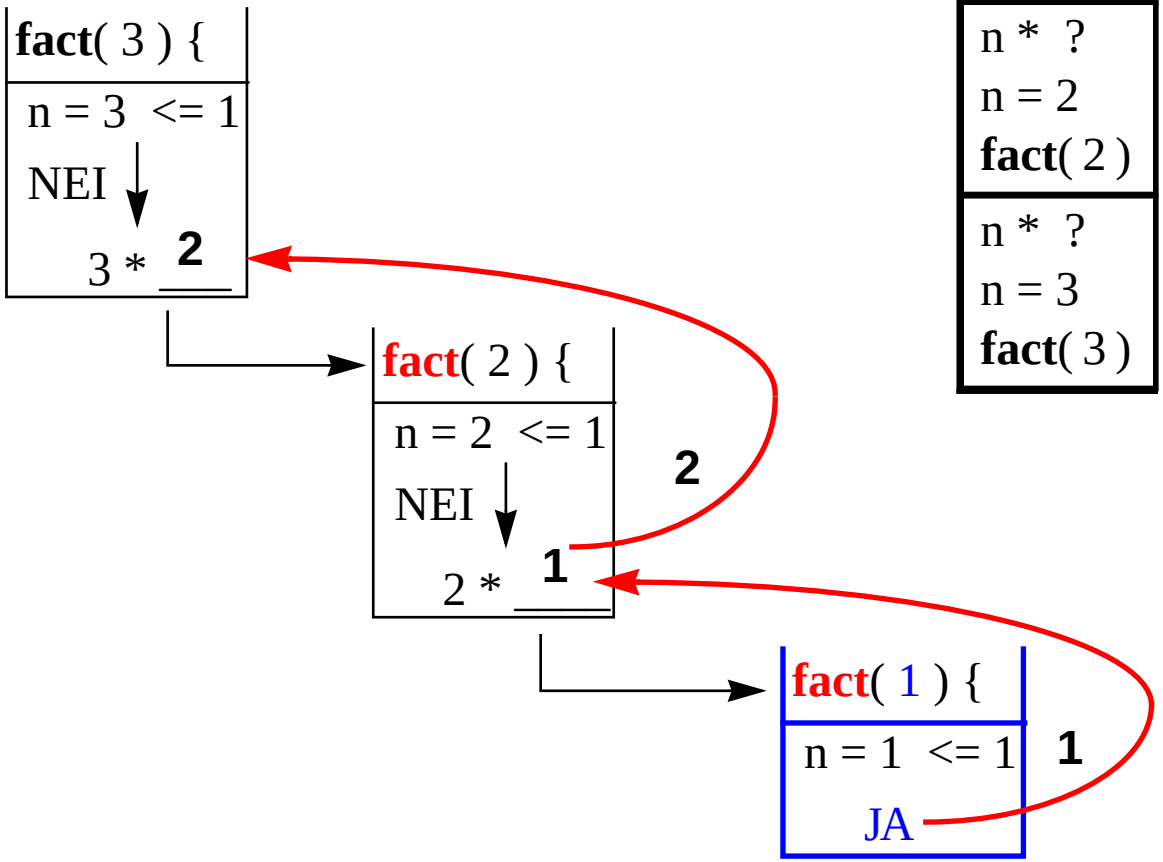
4a. Rekursjon implementeres med en stabel

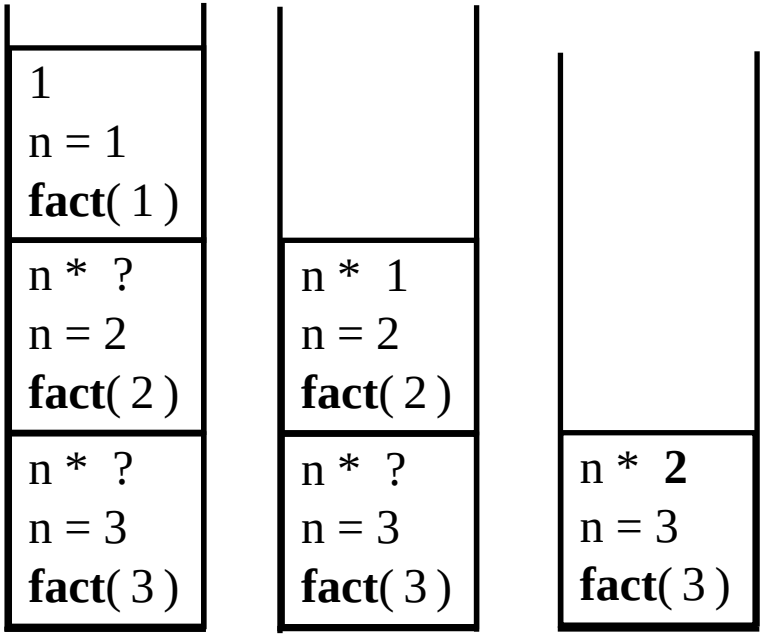
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```



4a. Rekursjon implementeres med en stabel

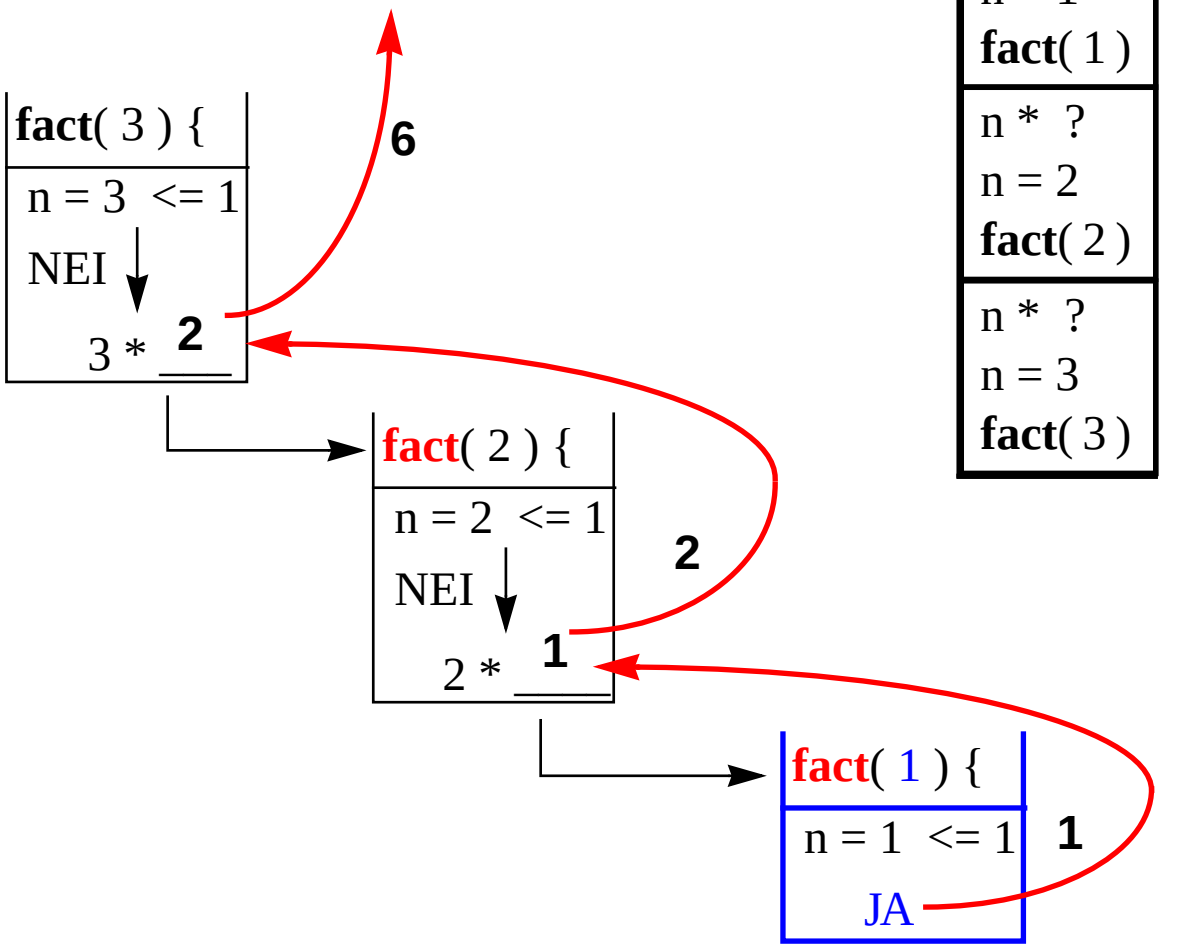
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```

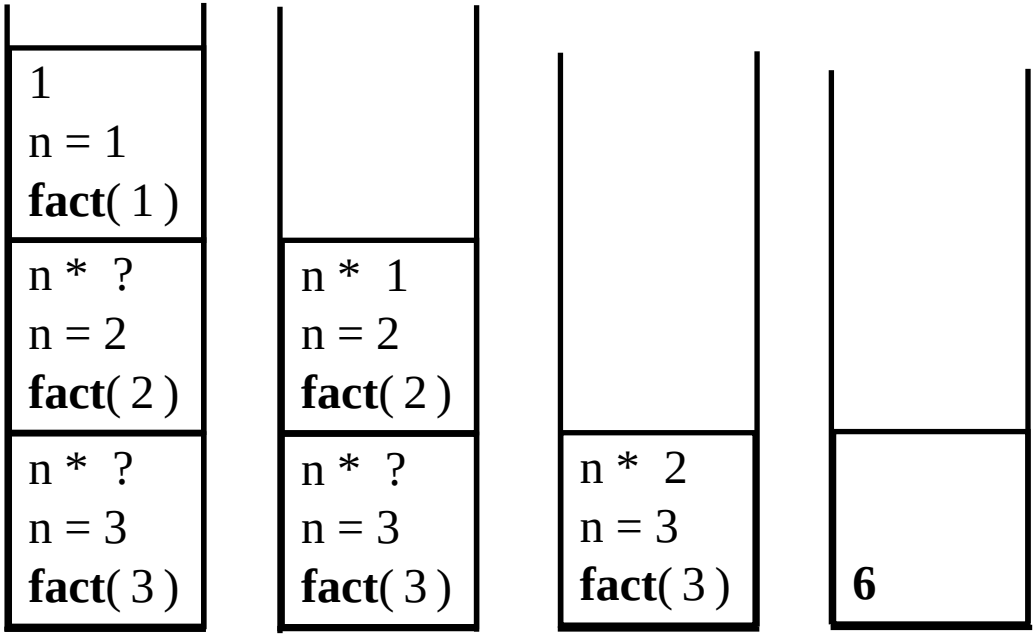




4a. Rekursjon implementeres med en stabel

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



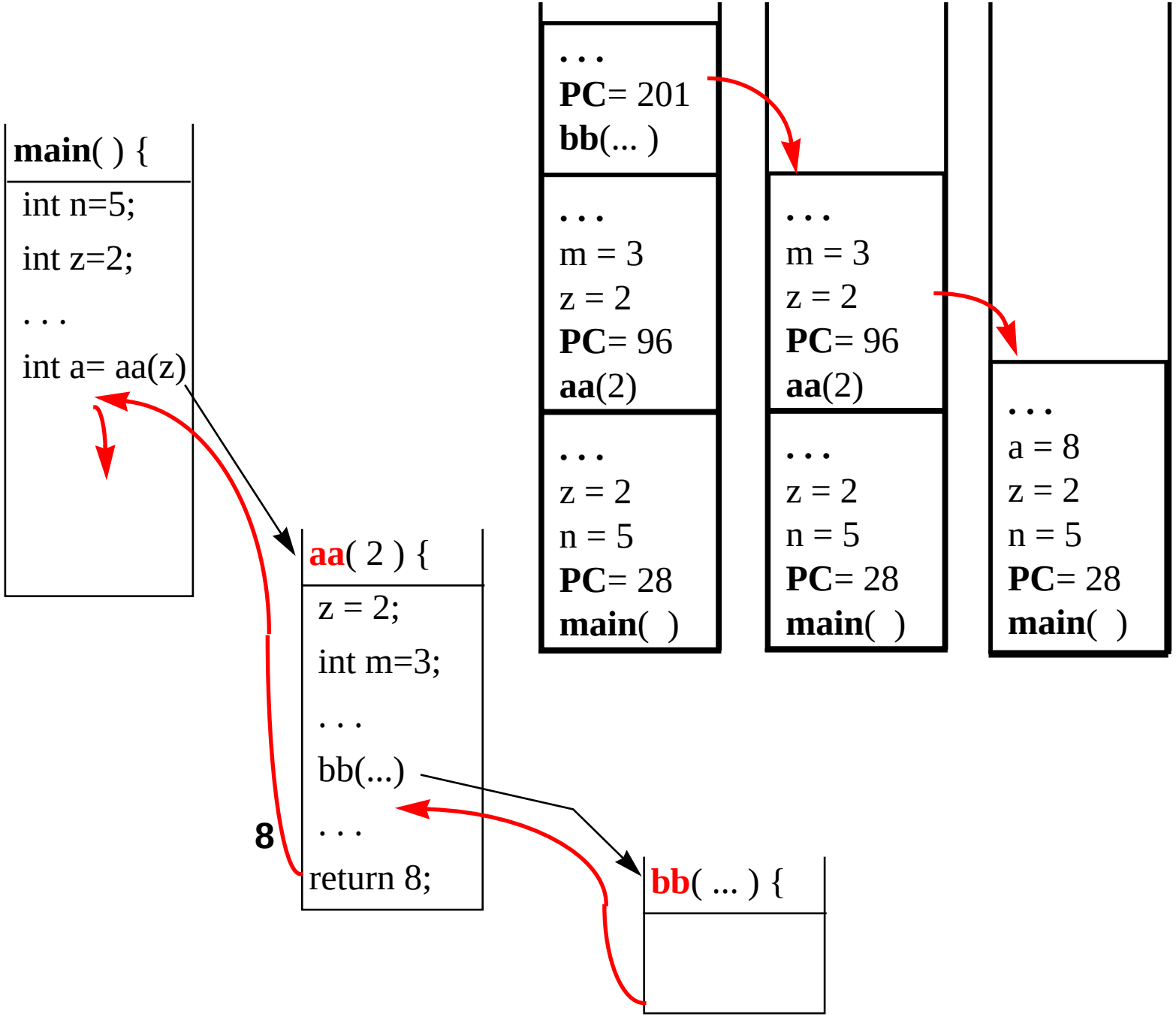


‘Method Stack’

```
      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(...) {
          . . .
      }
```



Kompleksitet av en rekursiv funksjon

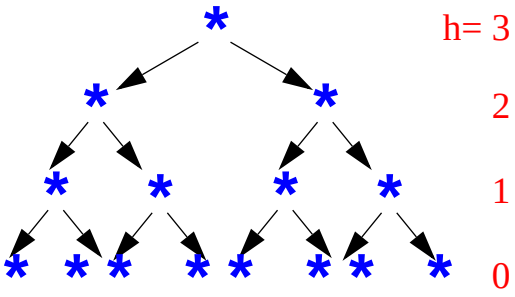
avhenger av

Analyse vha REKURSJONSTRE

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



Kompleksitet av en rekursiv funksjon

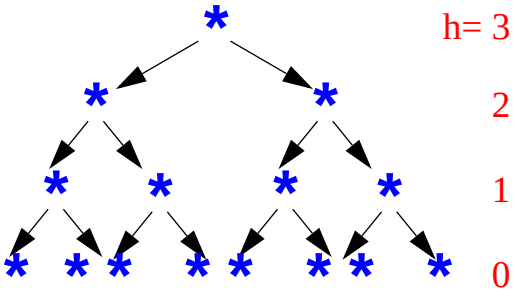
avhenger av

Analyse vha REKURSJONSTRE

- *"**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.***

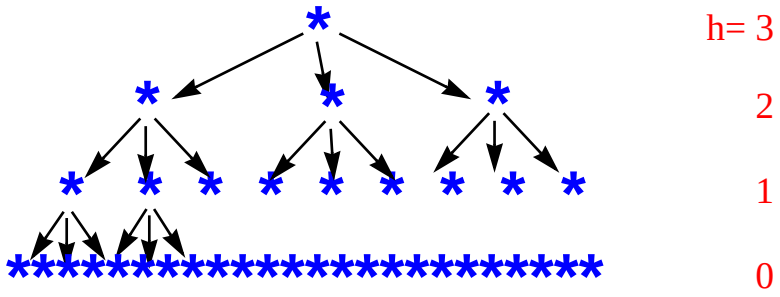
R(n) har 2 rekursive kall til R(n-1)

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



Kompleksitet av en rekursiv funksjon

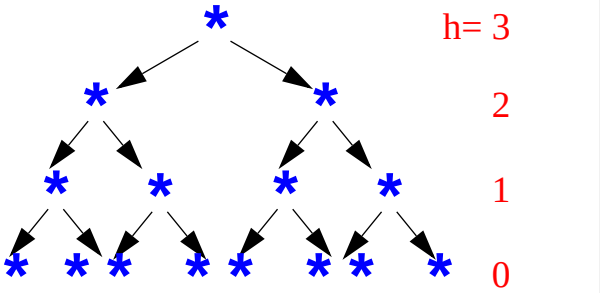
avhenger av

Analyse vha REKURSJONSTRE

- *"**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.***

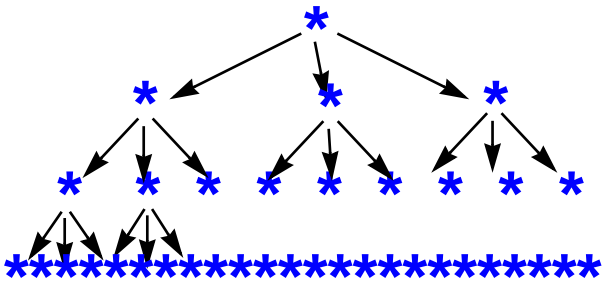
R(n) har 2 rekursive kall til R(n-1)

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



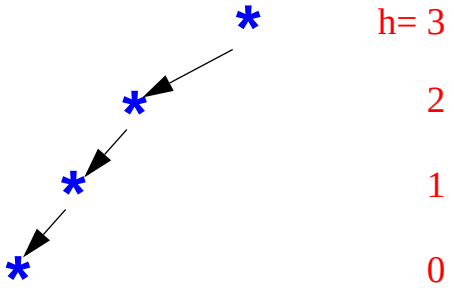
R(n) har 3 rekursive kall til R(n-1)

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**)



Kompleksitet av en rekursiv funksjon

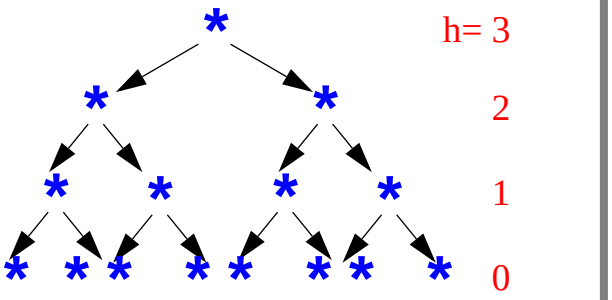
avhenger av

Analyse vha REKURSJONSTRE

- *"**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.***

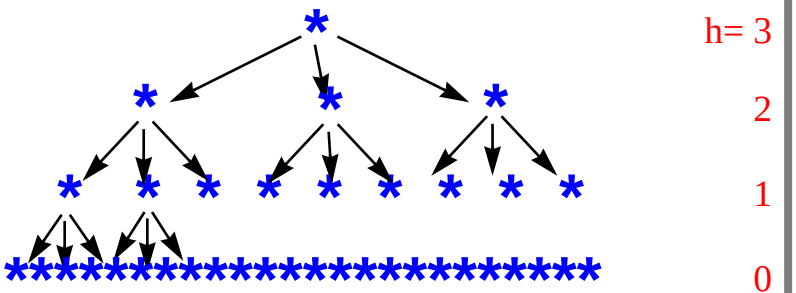
R(n) har 2 rekursive kall til R(n-1)

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



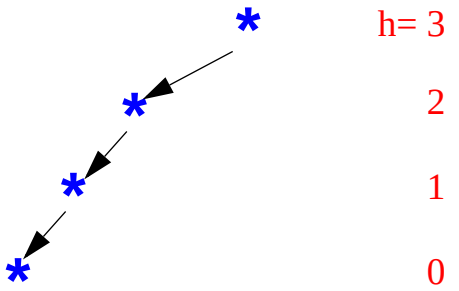
R(n) har 3 rekursive kall til R(n-1)

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



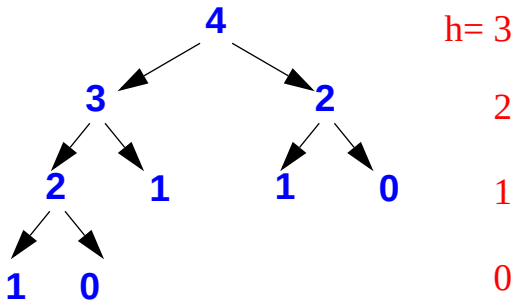
R(n) har 1 rek. kall til R(n-1)

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



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

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



Kompleksitet av en rekursiv funksjon

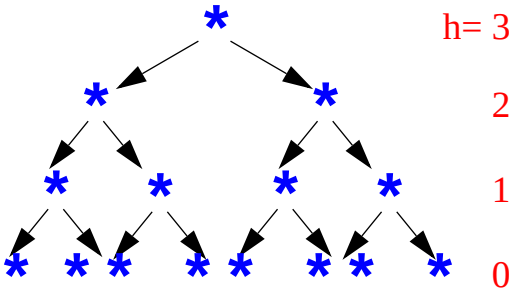
avhenger av

Analyse vha REKURSJONSTRE

- *"**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.***

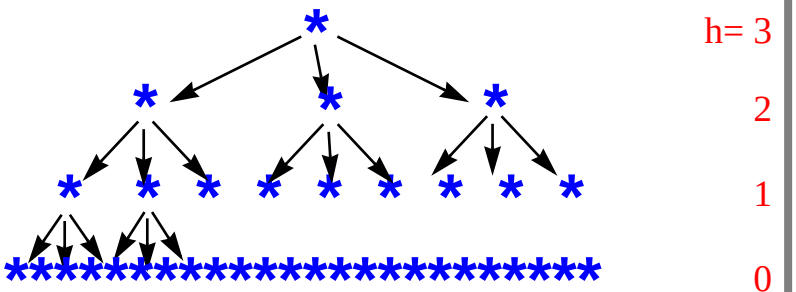
R(n) har 2 rekursive kall til R(n-1)

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



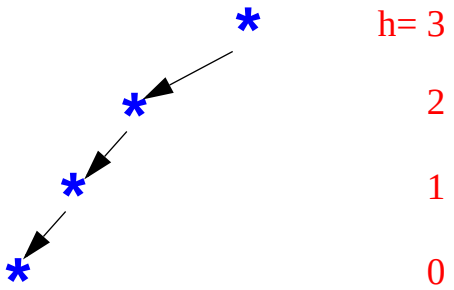
R(n) har 3 rekursive kall til R(n-1)

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



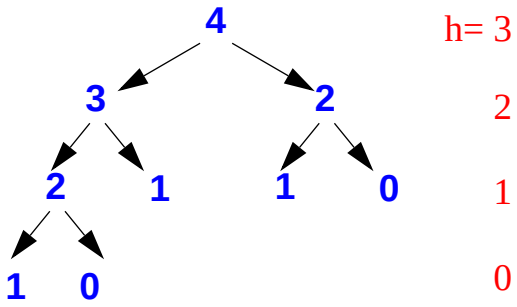
R(n) har 1 rek. kall til R(n-1)

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

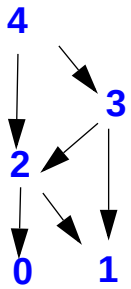


R(n) har rek kall til R(n-1) og R(n-2)

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

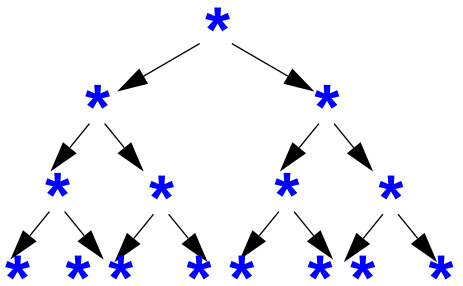


Fibonacci kan dog forenkles til: $O(n)$



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**)



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**

Terminering: Korrekthet:

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

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

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

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

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** hvert rekursivt kall **P**(**mi**) returnerer riktig resultat

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

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

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
```

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

Korrekthet:

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

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

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
```

Korrekthet:

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

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

```
  else HER MÅ VI  VISE
```

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

!!! DET OVENSTÅENDE ANTAR VI !!!

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
```

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

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[] QS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        velg pivot p ε A og
 *         del A i :
 *        A1= { xεA | x ≤ p}  og  A2 = { xεA | x > p};
 *        sorter rekursivt (mindre) delene
 *            r1= QS(A1) og
 *            r2= QS(A2)
 *        return  r1 + r2  }
 */
```

Invariant:

Korrekthet: rekursjons-invariant

```
/* int[] QS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        velg pivot p ε A og
 *         del A i :
 *        A1= { xεA | x ≤ p}  og  A2 = { xεA | x > p};
 *        sorter rekursivt (mindre) delene
 *            r1= QS(A1) og
 *            r2= QS(A2)
 *        return  r1 + r2  }
 */
```

Invariant:

QS(A) *returnerer sortert argument* A:

Korrekthet: rekursjons-invariant

```
/* int[] QS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        velg pivot p ε A og
 *         del A i :
 *        A1= { xεA | x ≤ p}  og  A2 = { xεA | x > p};
 *        sorter rekursivt (mindre) delene
 *            r1= QS(A1) og
 *            r2= QS(A2)
 *        return  r1 + r2  }
 */
```

Invariant:

QS(A) *returnerer sortert argument* A:

if **n==1** – ***da er A sortert***

else –

Korrekthet: rekursjons-invariant

```
/* int[] QS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        velg pivot p ε A og
 *         del A i :
 *        A1= { xεA | x ≤ p}  og  A2 = { xεA | x > p};
 *        sorter rekursivt (mindre) delene
 *            r1= QS(A1) og
 *            r2= QS(A2)
 *        return  r1 + r2  }
 */
```

Invariant:

QS(A) *returnerer sortert argument* A:

if **n==1** – ***da er A sortert***

else – deler A i to disjunkte deler

A1= { xε**A | x ≤ p}** og A2 = { xε**A | x > p}**

r1= QS(**A1**) ***returnerer sortert*** **A1**

r2= QS(**A2**) ***returnerer sortert*** **A2**

Korrekthet: rekursjons-invariant

```
/* int[] QS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        velg pivot p ε A og
 *         del A i :
 *        A1= { xεA | x ≤ p}  og  A2 = { xεA | x > p};
 *        sorter rekursivt (mindre) delene
 *            r1= QS(A1) og
 *            r2= QS(A2)
 *        return  r1 + r2  }
 */
```

Invariant:

QS(A) *returnerer sortert argument* A:

if **n==1** – ***da er A sortert***

else – deler A i to disjunkte deler

A1= { xε**A | x ≤ p}** og A2 = { xε**A | x > p}**

r1= QS(**A1**) ***returnerer sortert*** **A1**

r2= QS(**A2**) ***returnerer sortert*** **A2**

men da er r1+r2 sortert siden A1 < A2

Korrekthet: rekursjons-invariant

```
/* int[] QS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        velg pivot p ε A og
 *         del A i :
 *        A1= { xεA | x ≤ p}  og  A2 = { xεA | x > p};
 *        sorter rekursivt (mindre) delene
 *            r1= QS(A1) og
 *            r2= QS(A2)
 *        return  r1 + r2  }
 */
```

Invariant:

QS(A) *returnerer sortert argument* A:

if **n==1** – ***da er A sortert***

else – deler A i to disjunkte deler

A1= { xε**A | x ≤ p}** og A2 = { xε**A | x > p}**

r1= QS(**A1**) ***returnerer sortert*** **A1**

r2= QS(**A2**) ***returnerer sortert*** **A2**

men da er r1+r2 sortert siden A1 < A2

```
/* int[] MS(int[] A) { int n= A.length;
 *    if (n == 1) { return A; }
 *    else {
 *        del A i midten i :
 *        A1= A[0...n/2]  og  A2 = A[n/2+1...n];
 *        sorter rekursivt (mindre) delene
 *            r1= MS(A1) og
 *            r2= MS(A2)
 *        return flettet resultat av
 *            rekursive kall FL(r1,r2) }
 */
```

Invariant:

.....

if **lgh==1** – ***da er A*** .....

else – deler A i to disjunkte deler

A1= **A[0...n/2]** og A2= **A[n/2+1...n]**

r1= MS(**A1**) .....

r2= MS(**A2**) .....

hvis .....

så returnerer hele else-grenen sortert A

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***

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*