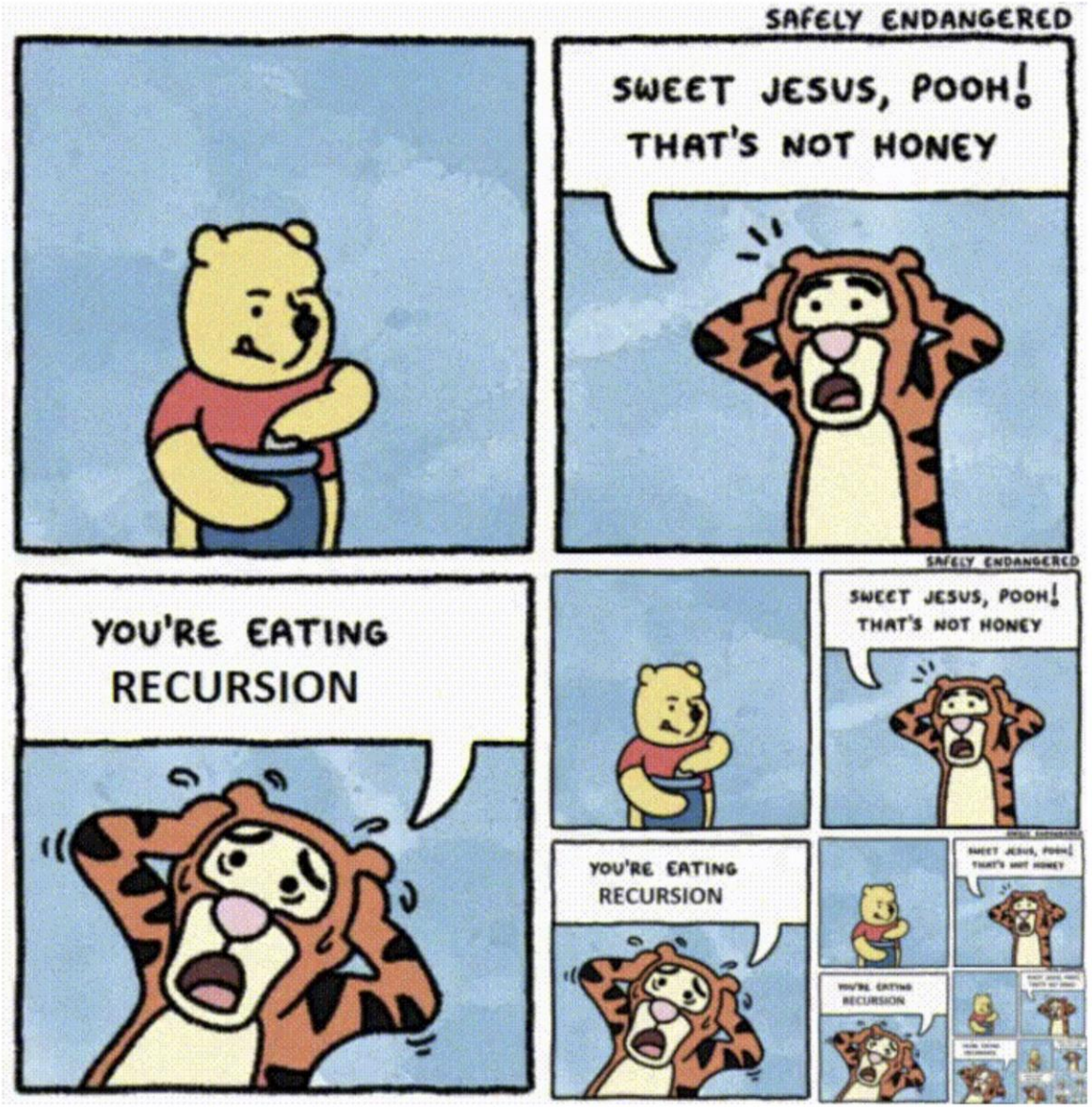
IN1010 - Rekursjon

Eyvind W. Axelsen – eyvinda@ifi.uio.no

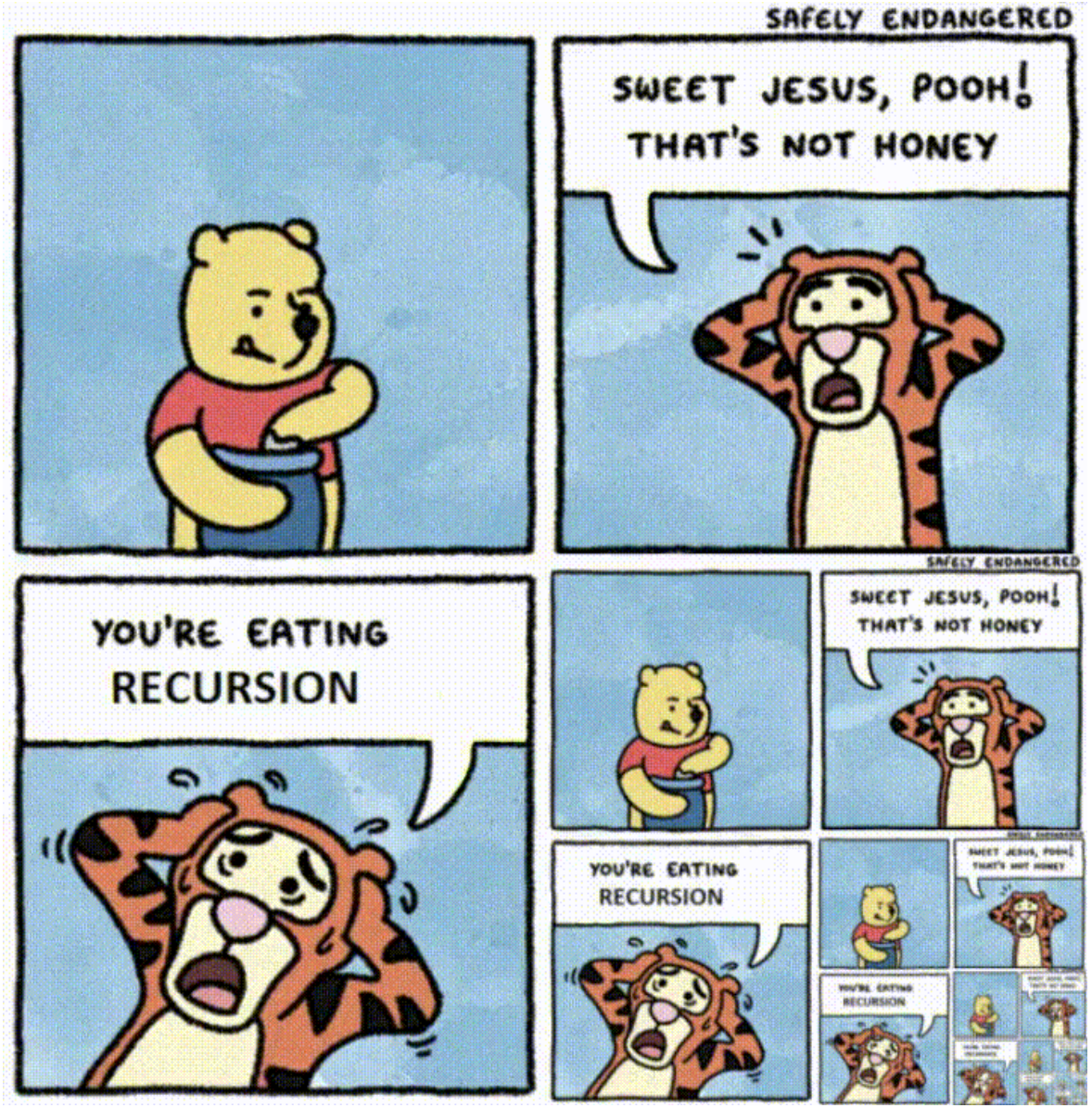
Foilene er basert på Dag Langmyhrs tidligere foiler om samme tema (dag@ifi.uio.no)



IN1010 - Rekursjon

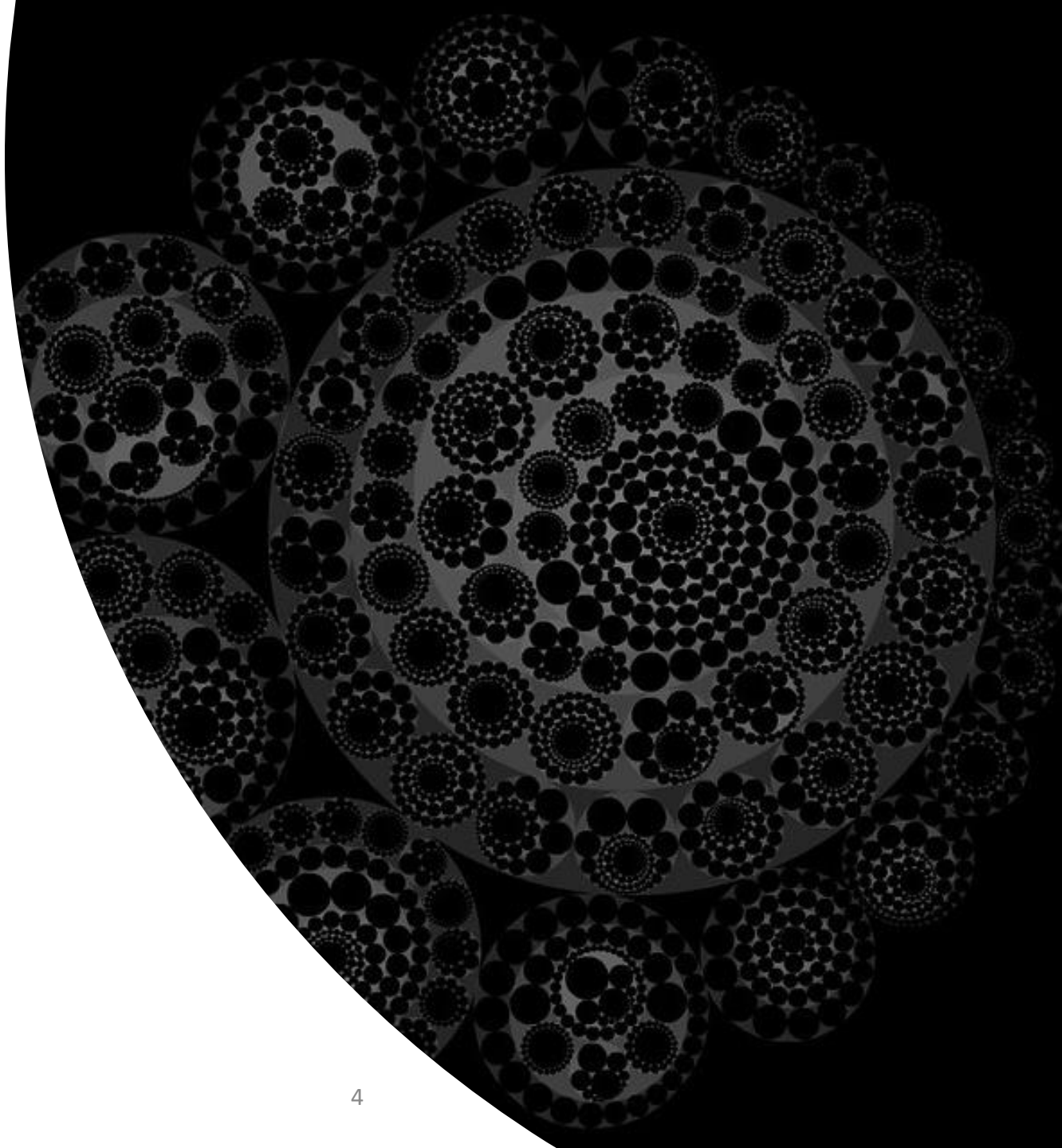
Eyvind W. Axelsen – eyvinda@ifi.uio.no

Foilene er basert på Dag Langmyhrs tidligere foiler om samme tema (dag@ifi.uio.no)



```java
public class GodEttermiddag {
  public static void main(String[] args) {
    System.out.println("Hei på deg!");
    System.out.println("Du bør kunne lese dette - sitter du for langt bak?");
  }
}
```

Plan for dagen



- Forklare rekursiv programmering ved å se på et sett av eksempler:
  - Fakultet
  - Fibonacci-tallene
  - Hanois tårn
  - Reversere lenket liste

- Etter forelesningen skal dere:
  - Forstå hovedprinsippene bak rekursiv programmering
  - Kunne skille rekursive løsninger fra iterative
  - Kunne anvende dette selv til å løse problemer

Hvor mange kjenner til begrepet rekursjon fra før?

Å beregne fakultet

Den matematiske funksjonen **n fakultet** (med notasjonen **n!**) er definert slik:

1! = 1	= 1
2! = 1 * 2	= 2
3! = 1 * 2 * 3	= 6
4! = 1 * 2 * 3 * 4	= 24
n! = 1 * 2 * … * n	=

Fakultet implementert med løkke

```java
class Fakultet1 {
    static long fak(int k) {
        long res = 1;
        for (int i = 1; i ⩽ k; i++)
            res = res * i;
        return res;

    }
    public static void main(String[] arg) {
        int n = Integer.parseInt(arg[0]);
        for (int i = 1; i ⩽ n; i++)
            System.out.println(i + "! = " + fak(i));
    }
}
```

```
eyvinda$ java Fakultet1 5
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
eyvinda$
```

Rekursjon

- En rekursiv metode er en metode som kaller *seg selv*
- Dette er nyttig i mange tilfeller (selv om det kanskje høres litt rart ut nå?)
  - Noen problemer er rekursive «av natur»
- Alle rekursive algoritmer kan også programmeres uten rekursjon



Rekursiv definisjon av fakultet

Om vi ser nærmere på definisjonen av fakultet, så innser vi at vi kan forenkle den ved å gjenbruke tidligere verdier:

1! = 1

n! = n * (n - 1)!

Å beregne fakultet

Den matematiske funksjonen **n fakultet** (med notasjonen **n!**) er definert slik:

1! = 1	= 1
2! = 1 * 2	= 2
3! = 1 * 2 * 3	= 6
4! = 1 * 2 * 3 * 4	= 24
n! = 1 * 2 * ... * n	=

Men hvordan koder vi dette?

Har du forslag?

```java
class Fakultet2 {
    static long fak(int k) {
        if (k == 1)
            return 1;
        else
            return k * fak(k - 1);
    }

    public static void main(String[] arg) {
        int n = Integer.parseInt(arg[0]);
        for (int i = 1; i <= n; i++)
            System.out.println(i + "! = " + fak(i));
    }
}
```

Metoden fak kaller seg selv!

```
eyvinda$ java Fakultet2 5
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
eyvinda$
```

To hoveddeler i ethvert rekursivt program:

```java
class Fakultet2 {
    static long fak(int k) {
        if (k == 1)
            return 1;
        else
            return k * fak(k - 1);
    }

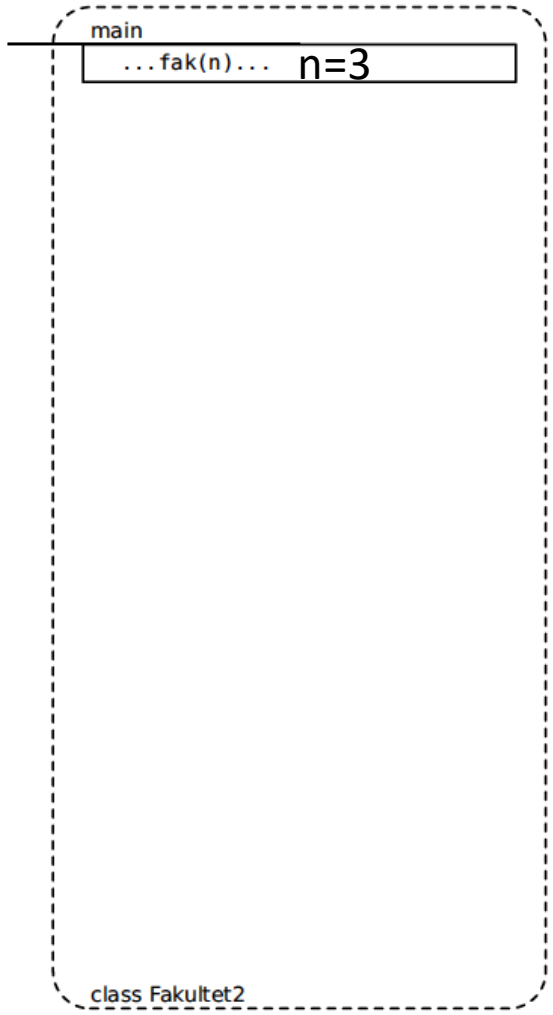
    public static void main(String[] arg) {
        int n = Integer.parseInt(arg[0]);
        for (int i = 1; i <= n; i++)
            System.out.println(i + "! = " + fak(i));
    }
}
```

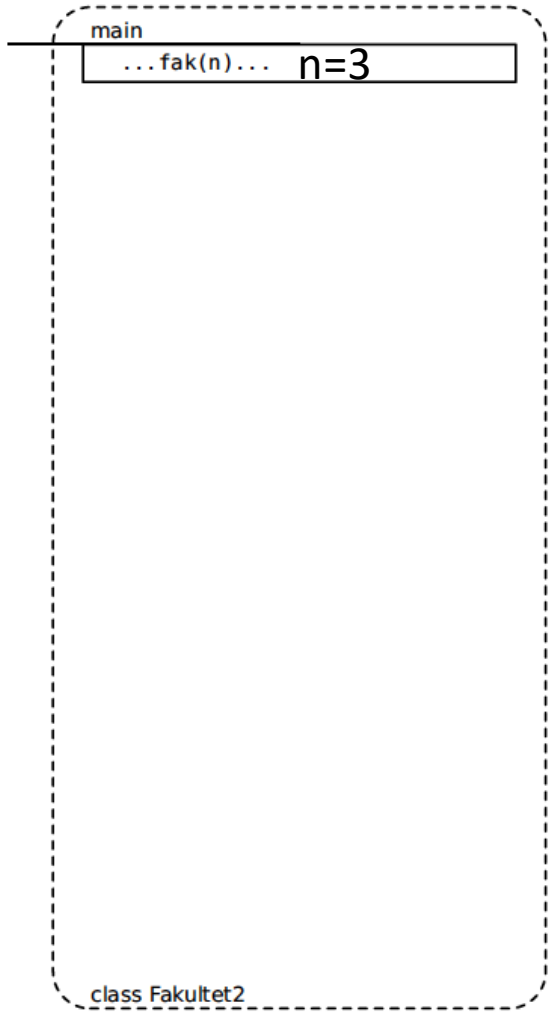
Et trivielt (enkelt) tilfelle

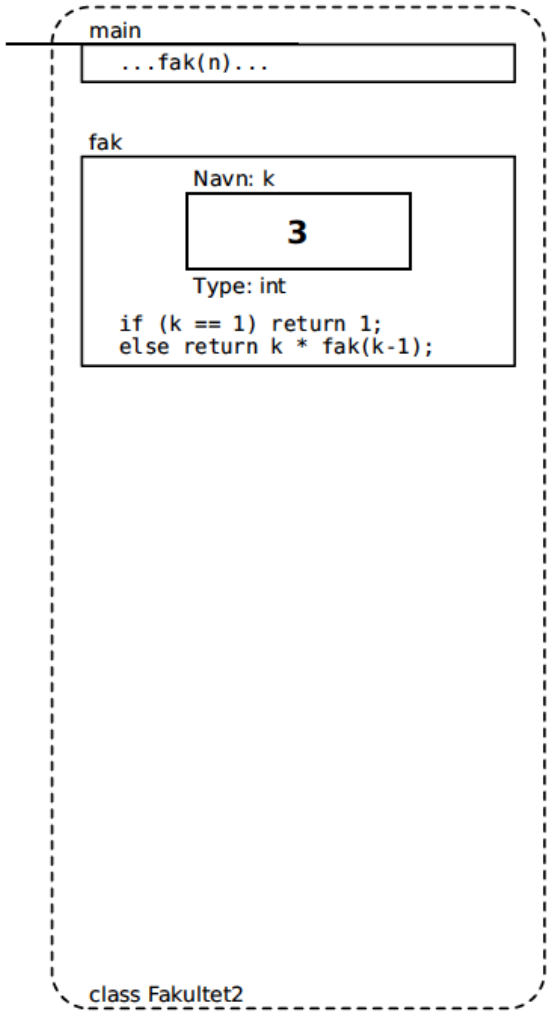
Ingen rekursjon

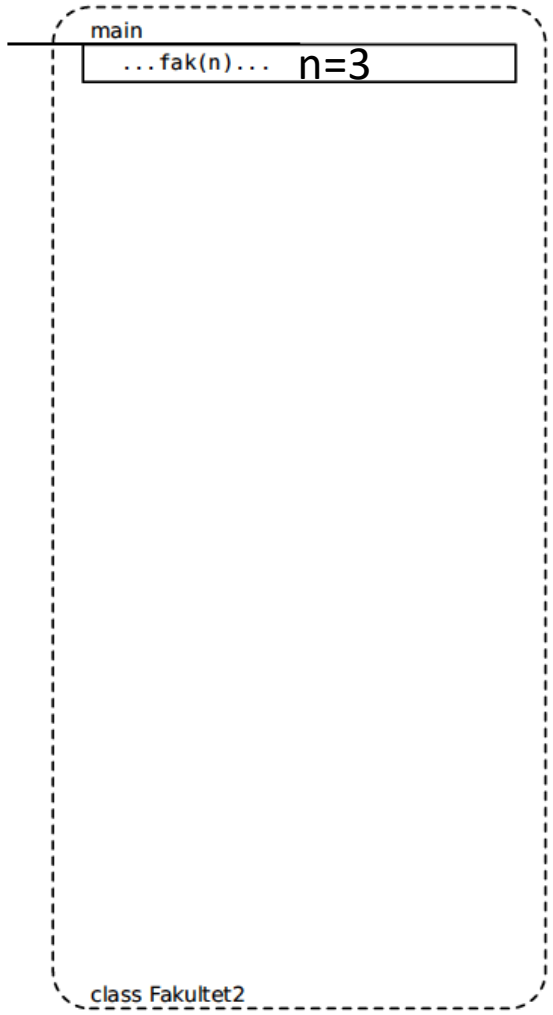
Et rekursivt kall

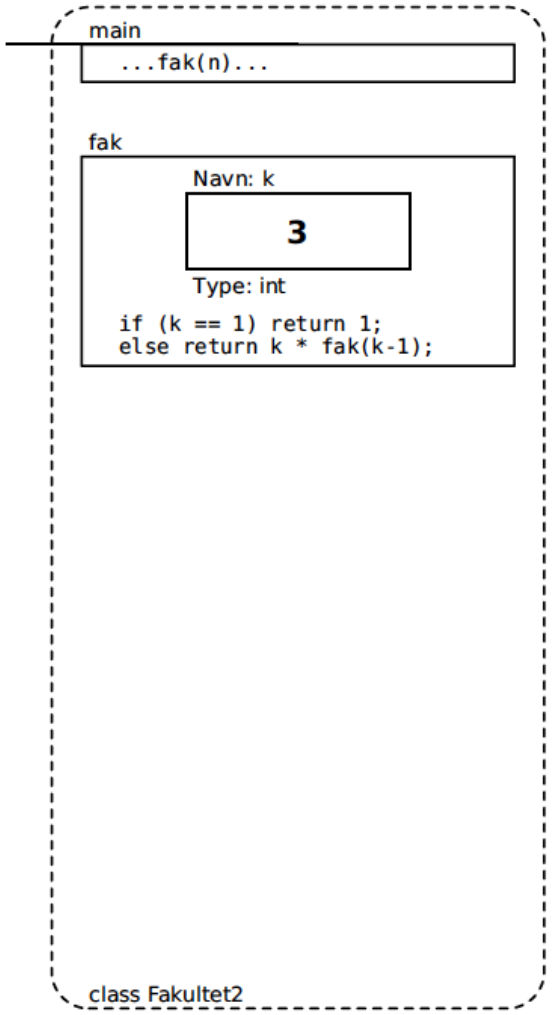
Må være enklere enn forrige kall

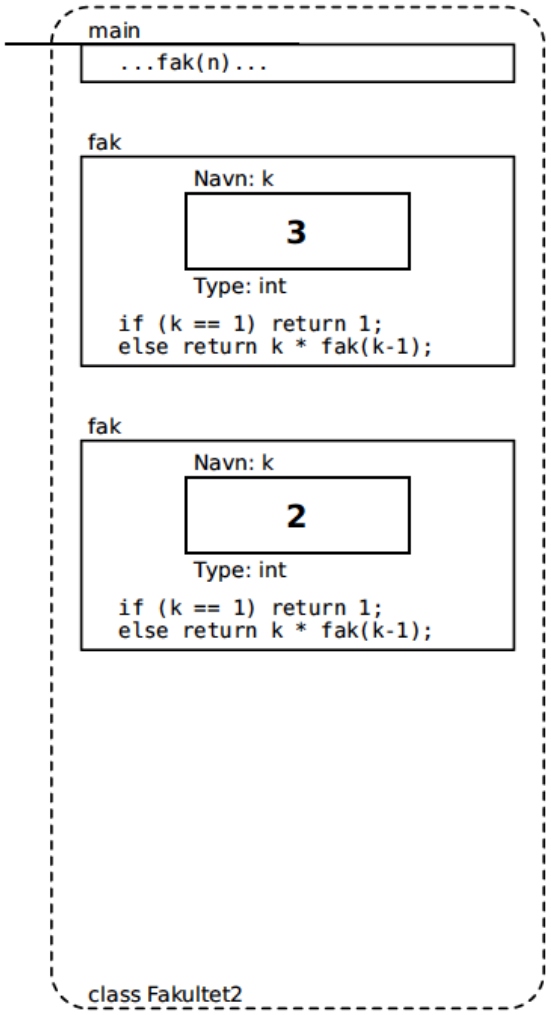

main
...fak(n)...
n=3
class Fakultet2

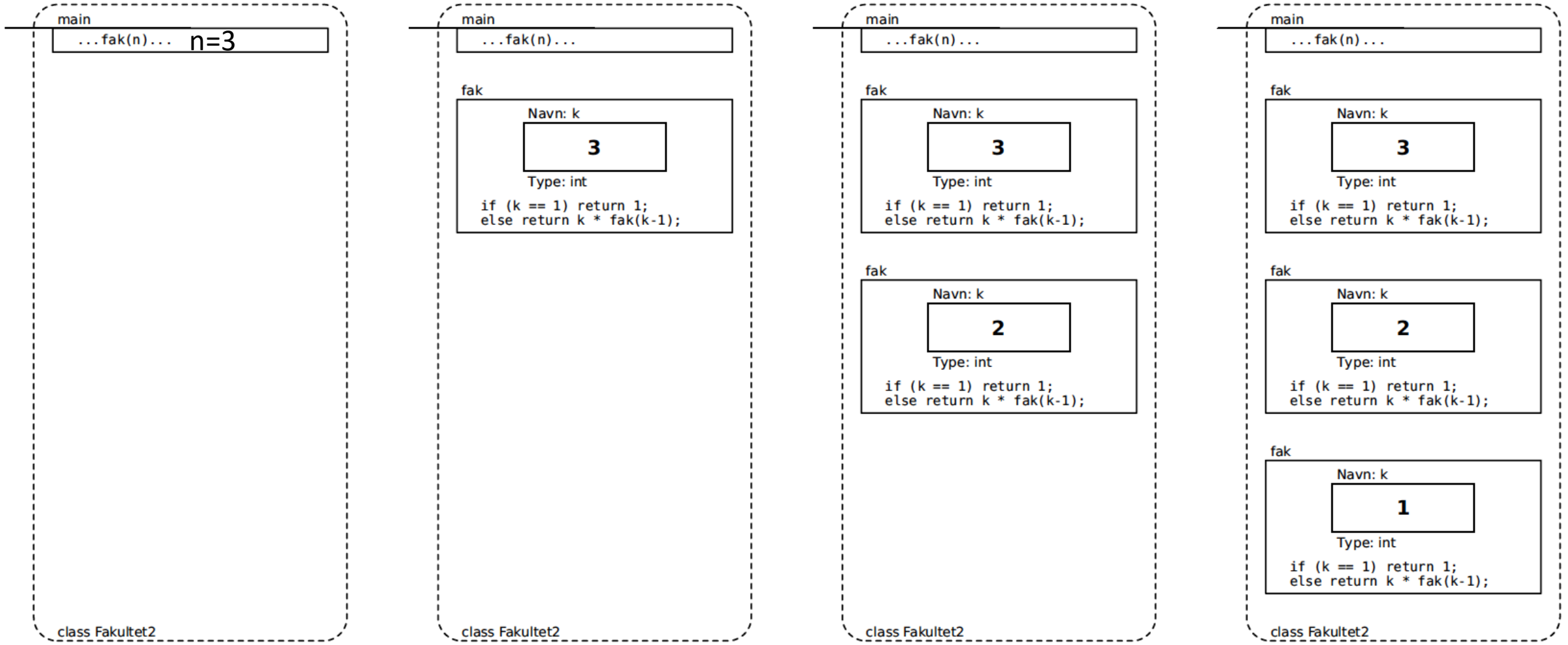

main
...fak(n)...
n=3
class Fakultet2

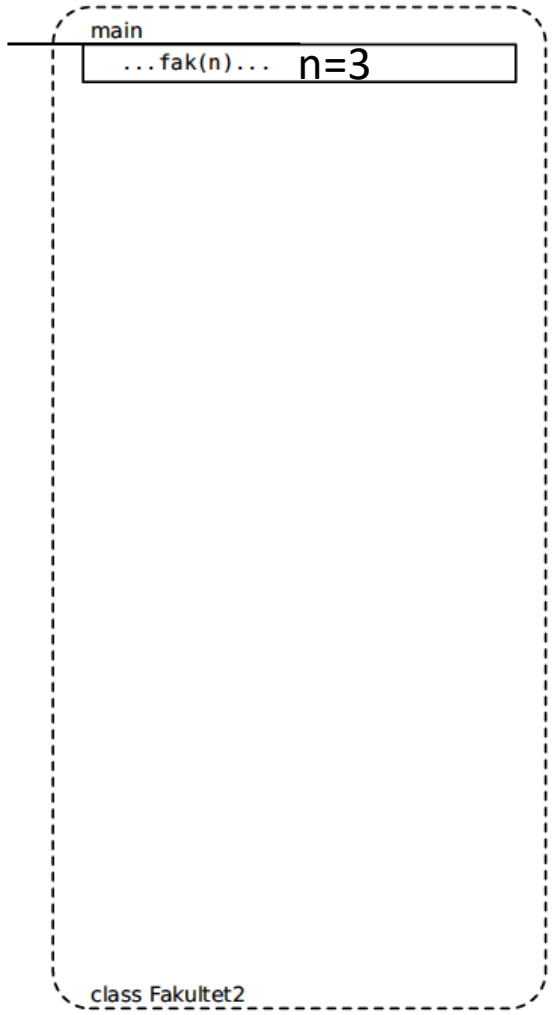

main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2

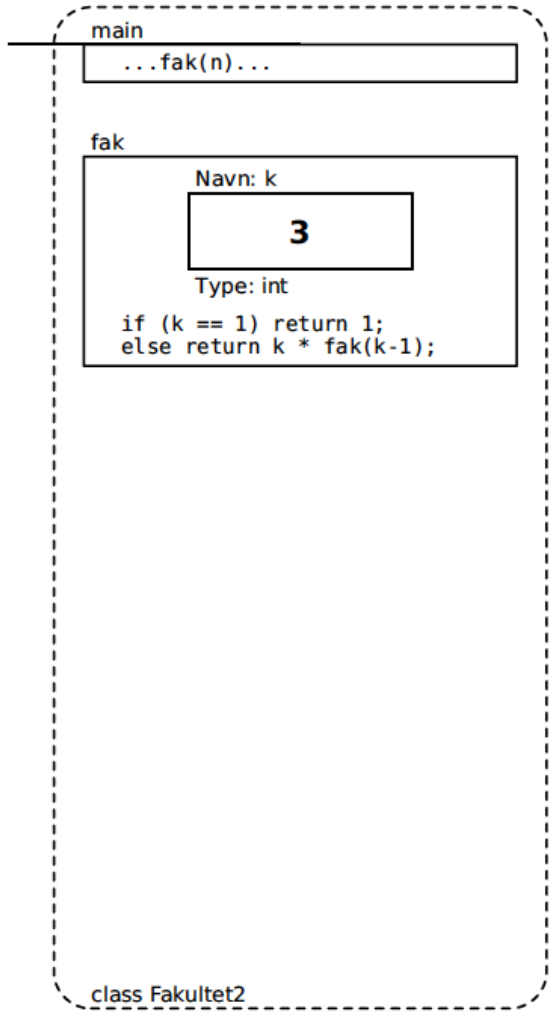

main
...fak(n)...
n=3
class Fakultet2

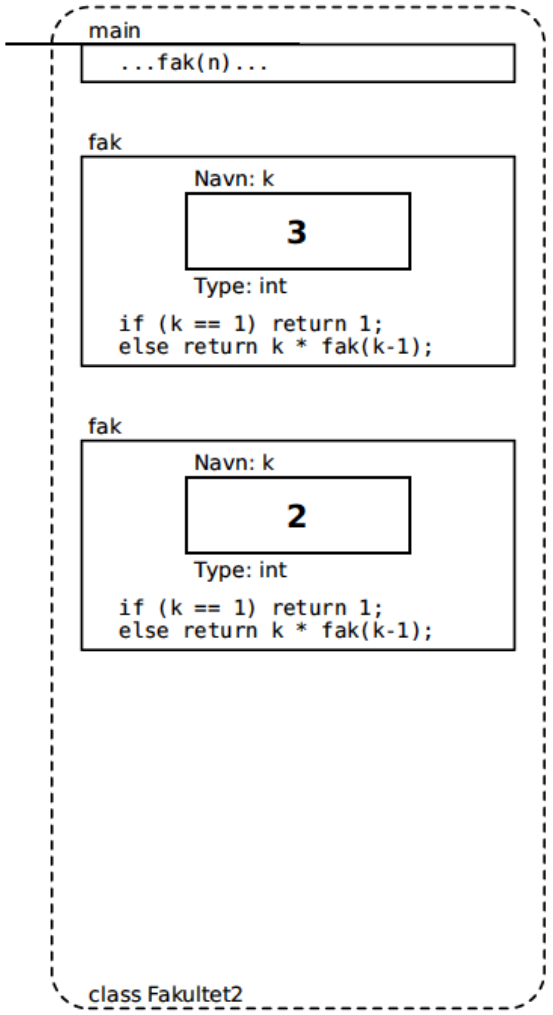

main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2

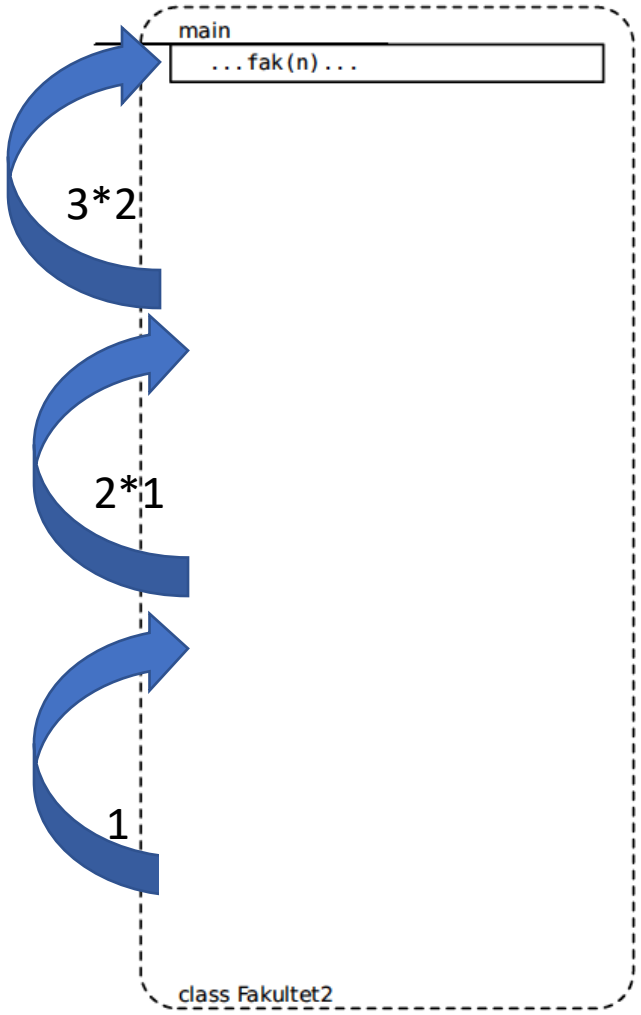

main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
fak
Navn: k
2
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2


main
...fak(n)... n=3
class Fakultet2
main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2
main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
fak
Navn: k
2
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2
main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
fak
Navn: k
2
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
fak
Navn: k
1
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2


main
...fak(n)...
n=3
class Fakultet2


main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2


main
...fak(n)...
fak
Navn: k
3
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
fak
Navn: k
2
Type: int
if (k == 1) return 1;
else return k * fak(k-1);
class Fakultet2


main
...fak(n)...
3*2
2*1
1
class Fakultet2

What is recursion?

Fibonacci-sekvensen

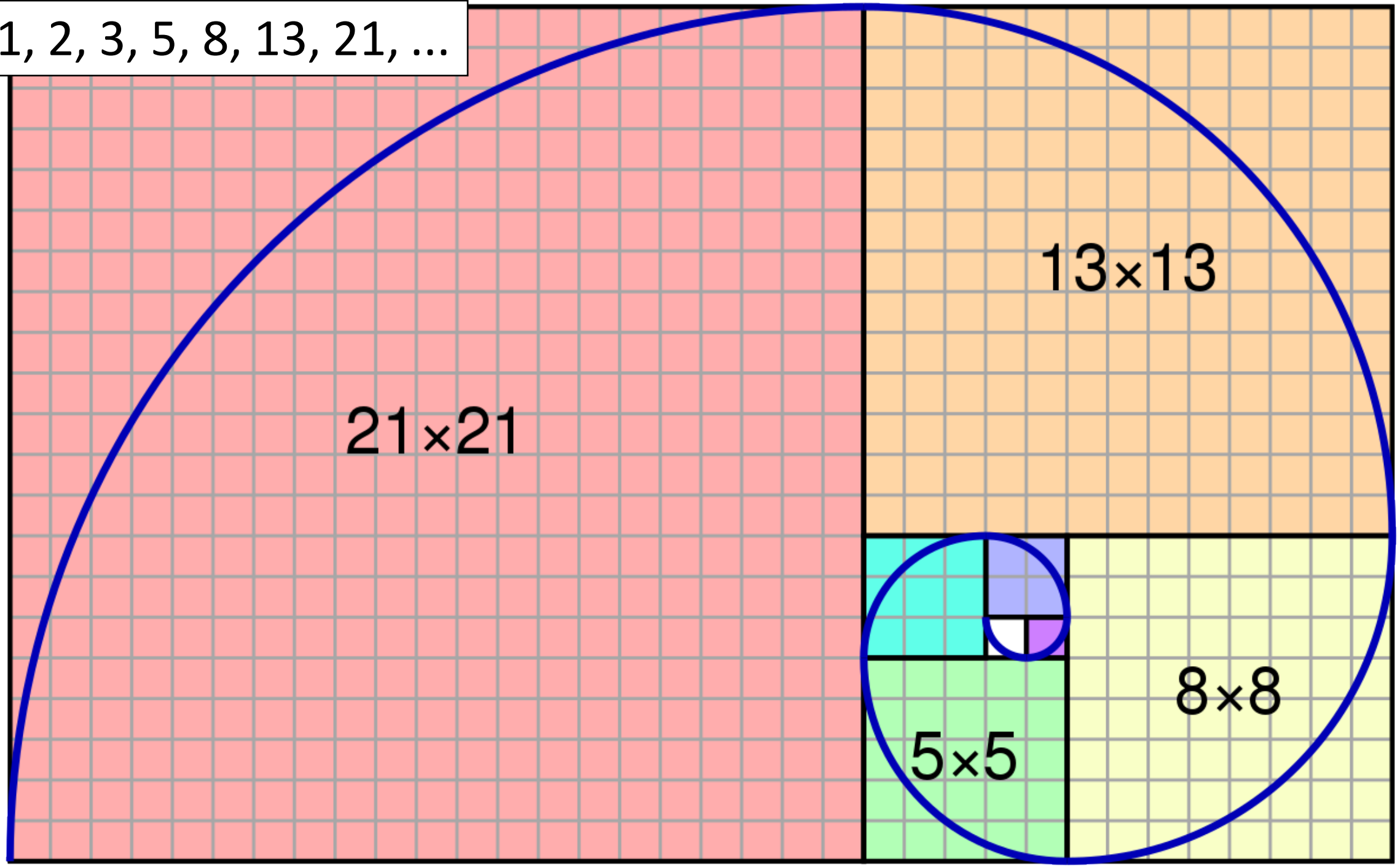
Tallrekken

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

der hvert tall er summen av de to foregående, forekommer overraskende ofte i naturen



1, 1, 2, 3, 5, 8, 13, 21, ...



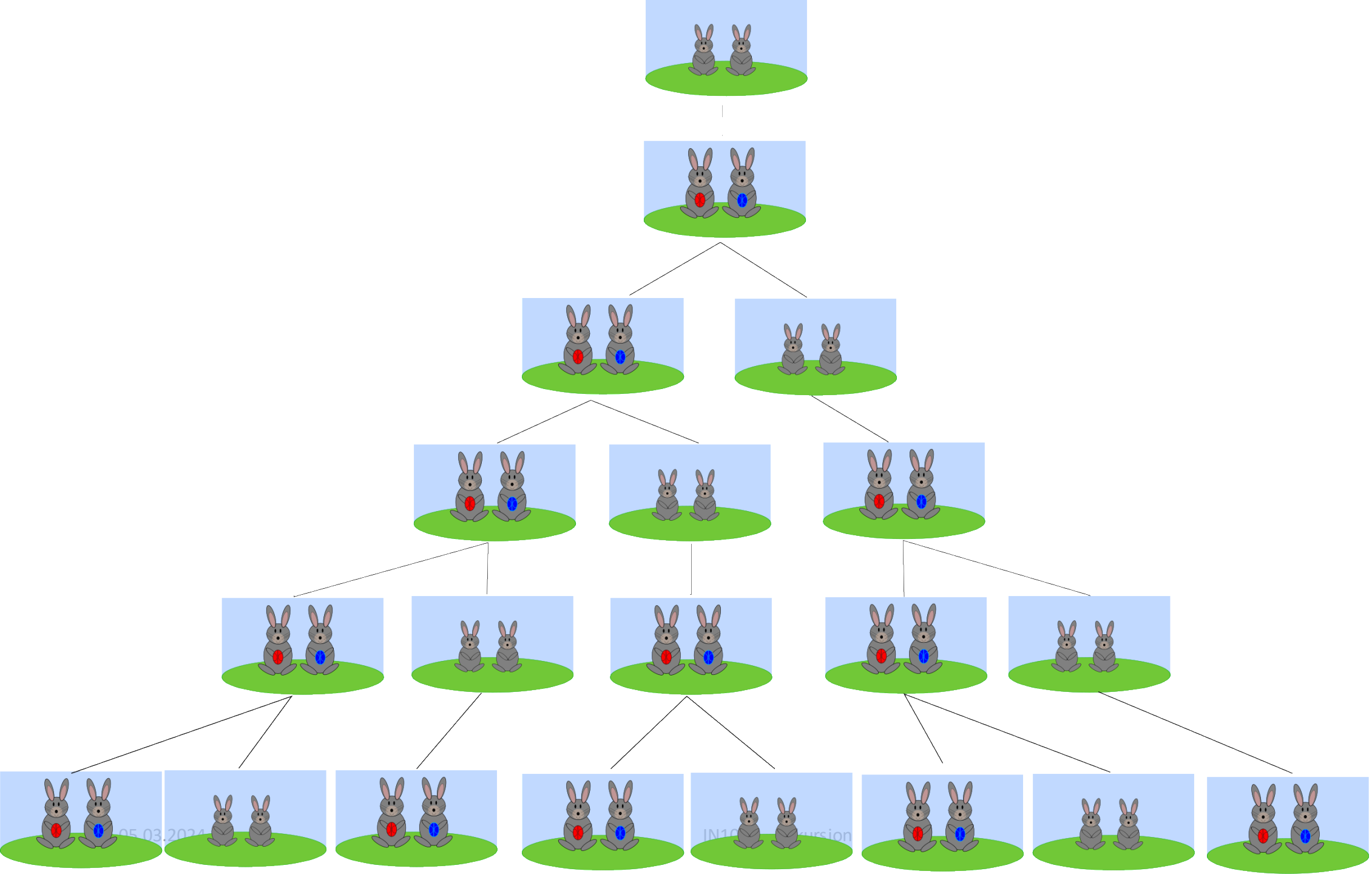


Fibonacci-sekvensen og kaniner

Hvor mange kaninpar vil finnes i en gitt måned? Gitt:

- Kaniner blir kjønnsmodne etter en måned
- Hvert kjønnsmodent par føder et nytt par hver måned
- Kaniner lever evig 😎




1
1
2
3
5
8, 13, 21,
34, 55, . . .

Vi skal programmere sekvensen gitt ved $F_n = F_{n-1} + F_{n-2}$ der $F_1 = 1$

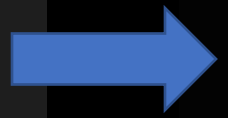
```java
class Fibonacci {
    static int fib(int n) {
        if (n == 1)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] arg) {
        int antall = Integer.parseInt(arg[0]);
        for (int i = 1; i <= antall; i++)
            System.out.println("Fib(" + i + ") = " + fib(i));
    }
}
```

Denne koden har et problem – ser du hva det er?

```java
class Fibonacci {
    static int fib(int n) {
        if (n == 1)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] arg) {
        int antall = Integer.parseInt(arg[0]);
        for (int i = 1; i ⩽ antall; i++)
            System.out.println("Fib(" + i + ") = " + fib(i));
    }
}
```



```
eyvinda$ java Fibonacci 6
Fib(1) = 1
Exception in thread "main" java.lang.StackOverflowError
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
        at Fibonacci.fib(Fibonacci.java:6)
```

Problemet oppstår i kallet fib(2), som kaller på fib(1) 👍 og fib(0) ☠️

Husk: Et rekursivt program består av to hoveddeler
- et rekursivt kall
- og et trivielt (enkelt) tilfelle

Det trivielle tilfellet må håndteres ordentlig!

Hvordan fikser vi programmet?

```java
if(n ⩽ 1)
    return 1;
```

Hvordan løse rekursive problemer?

- Dersom problemet er enkelt, løs det med en gang!
- Hvis ikke, se om du kan redusere problemet til en enklere variant av det samme problemet

- Av og til kan det hjelpe å tenke at *Noen Andre* løser den enklere varianten av problemet



Hanois tårn

I et tempel i Hanoi står tre stolper. På den venstre ligger 64 ringer av ulik størrelse, sortert med den minste øverst. Munkene ved tempelet skal flytte alle ringene fra den venstre til den høyre, etter følgende regler:

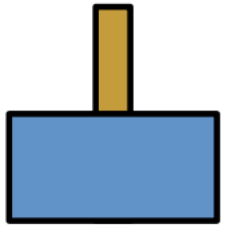
- Ringene må flyttes en og en
- En ring kan aldri legges oppå en mindre ring
- Man kan bruke den midtre stolpen til «mellomlagring» underveis

I følge legenden går verden under når alle ringene er flyttet!

Hanois tårn med 1 ring

Steg

1



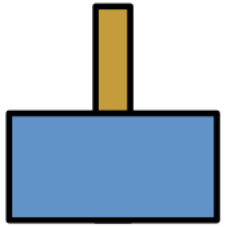




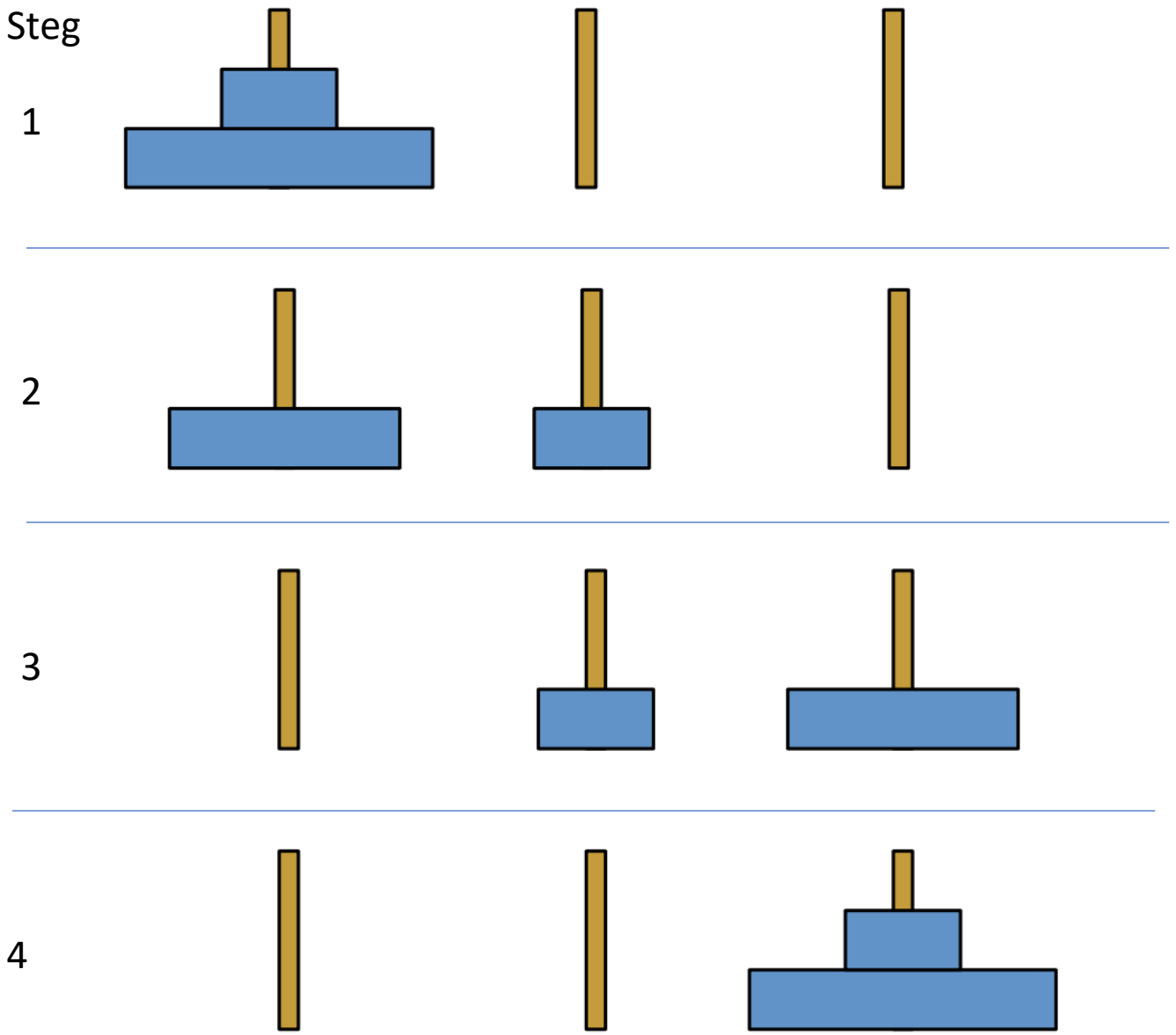
2







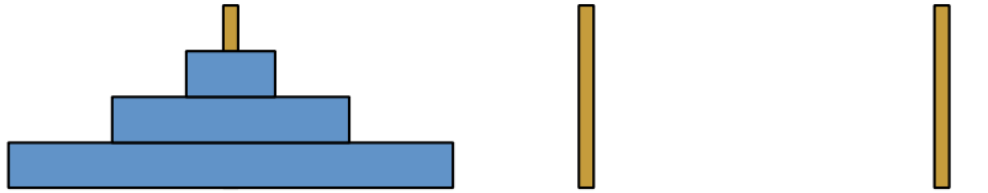
Hanois tårn med 2 ringer



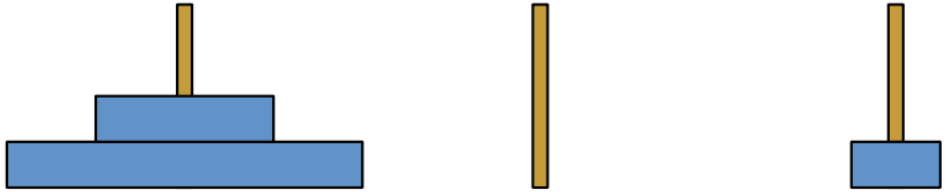
Hanois tårn med 3 ringer

Steg

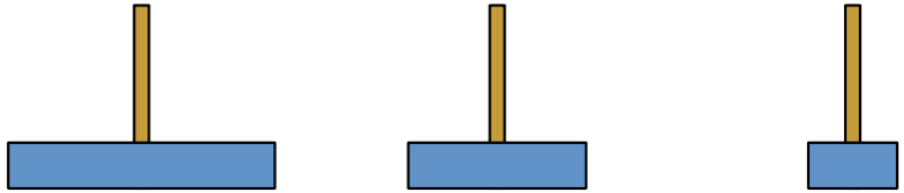
1



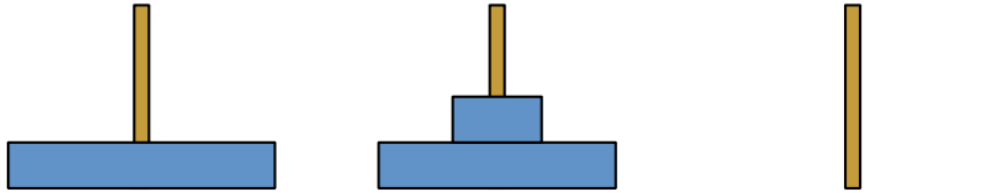
2



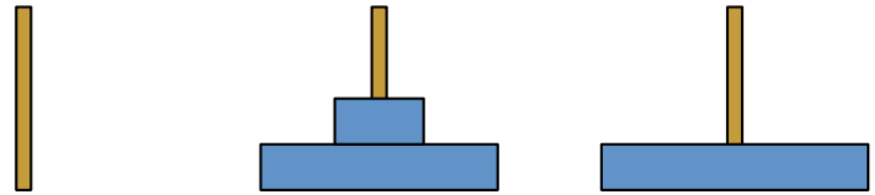
3



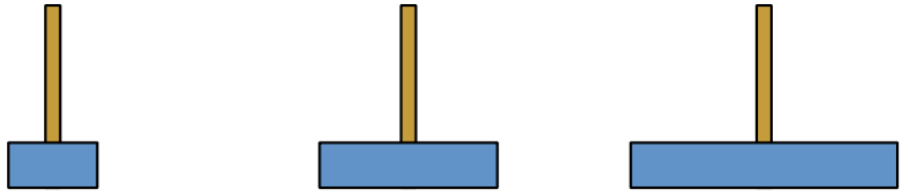
4



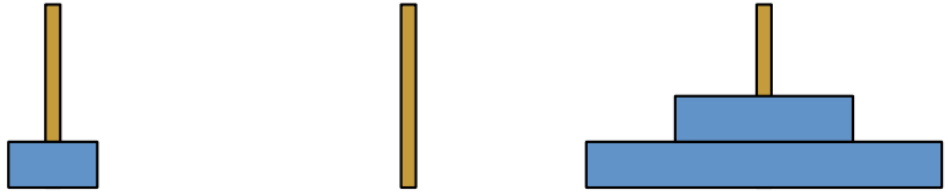
5



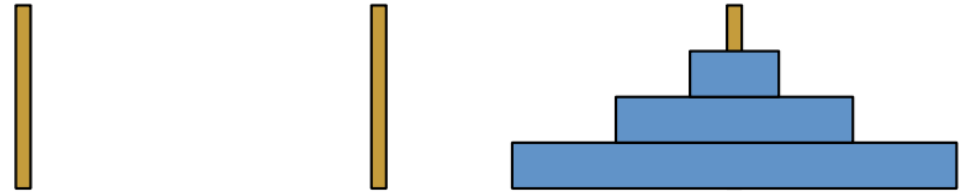
6



7



8



Programmere Hanois tårn

- Vi kaller de tre stolpene **A**, **B** og **C**.
- Vi skal flytte alle ringene fra **A** til **C**.
- For å kunne flytte den største ringen, må vi først flytte alle de andre ringene til hjelpestolpen **B**.
- Så kan vi flytte den største ringen fra **A** til **C**.
- Etterpå må vi flytte alle ringene fra hjelpestolpen **B** til **C** (oppå den største ringen).

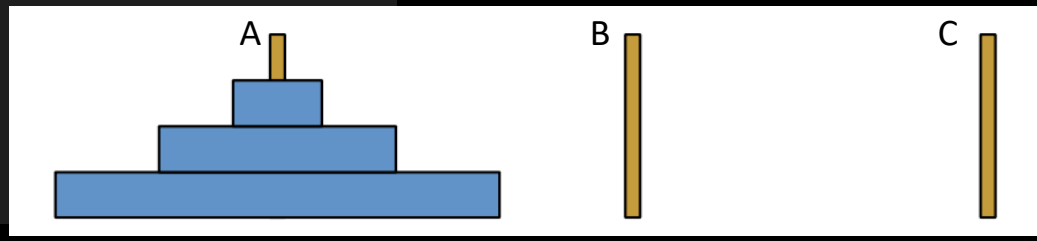
Og da er jobben gjort!

```java
class Hanoi {
    static void flytt(int n, char fra, char via, char til) {
        if (n == 1) {
            // Det trivielle tilfellet: flytt 1 ring direkte:
            System.out.println("Flytt " + fra + " til " + til);
        } else {
            // Det rekursive tilfellet:
            //
            // Flytt de små ringene til hjelpestolpen:
            flytt(n - 1, fra, til, via);
            // Flytt den største ringen fra 'fra'-stoplen til 'til'-stoplen:
            flytt(1, fra, via, til);
            // Flytt ringene fra hjelpestolpen til 'til'-stoplen igjen:
            flytt(n - 1, via, fra, til);
        }
    }

    public static void main(String[] arg) {
        int antall = Integer.parseInt(arg[0]);
        flytt(antall, 'A', 'B', 'C');
    }
}
```

Men… Hvordan fungerer dette da?

Det største «trikset» med rekursjon: løs det enkle problemet, og reduser vanskelige problemer til enkle problemer stegvis!



Eksempelkjøringer

```
$ java Hanoi 1
Flytt A til C

$ java Hanoi 2
Flytt A til B
Flytt A til C
Flytt B til C

$ java Hanoi 3
Flytt A til C
Flytt A til B
Flytt C til B
Flytt A til C
Flytt B til A
Flytt B til C
Flytt A til C
```

```
$ java Hanoi 4
Flytt A til B
Flytt A til C
Flytt B til C
Flytt A til B
Flytt C til A
Flytt C til B
Flytt A til B
Flytt A til C
Flytt B til C
Flytt B til A
Flytt C til A
Flytt B til C
Flytt A til B
Flytt A til C
Flytt B til C
```



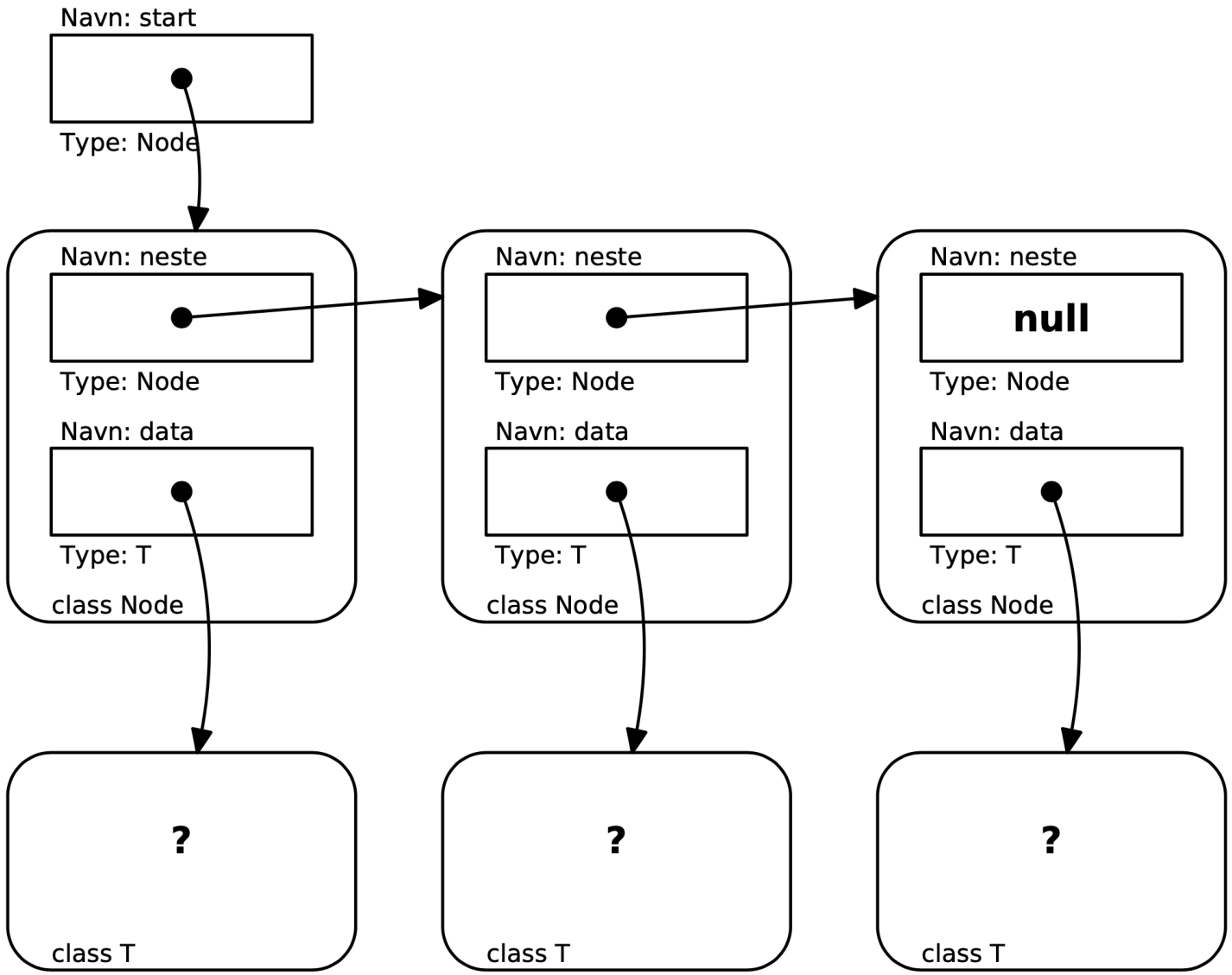
Så, hvor lenge har vi igjen?

Antall trekk som trengs er 2^{64} -1.

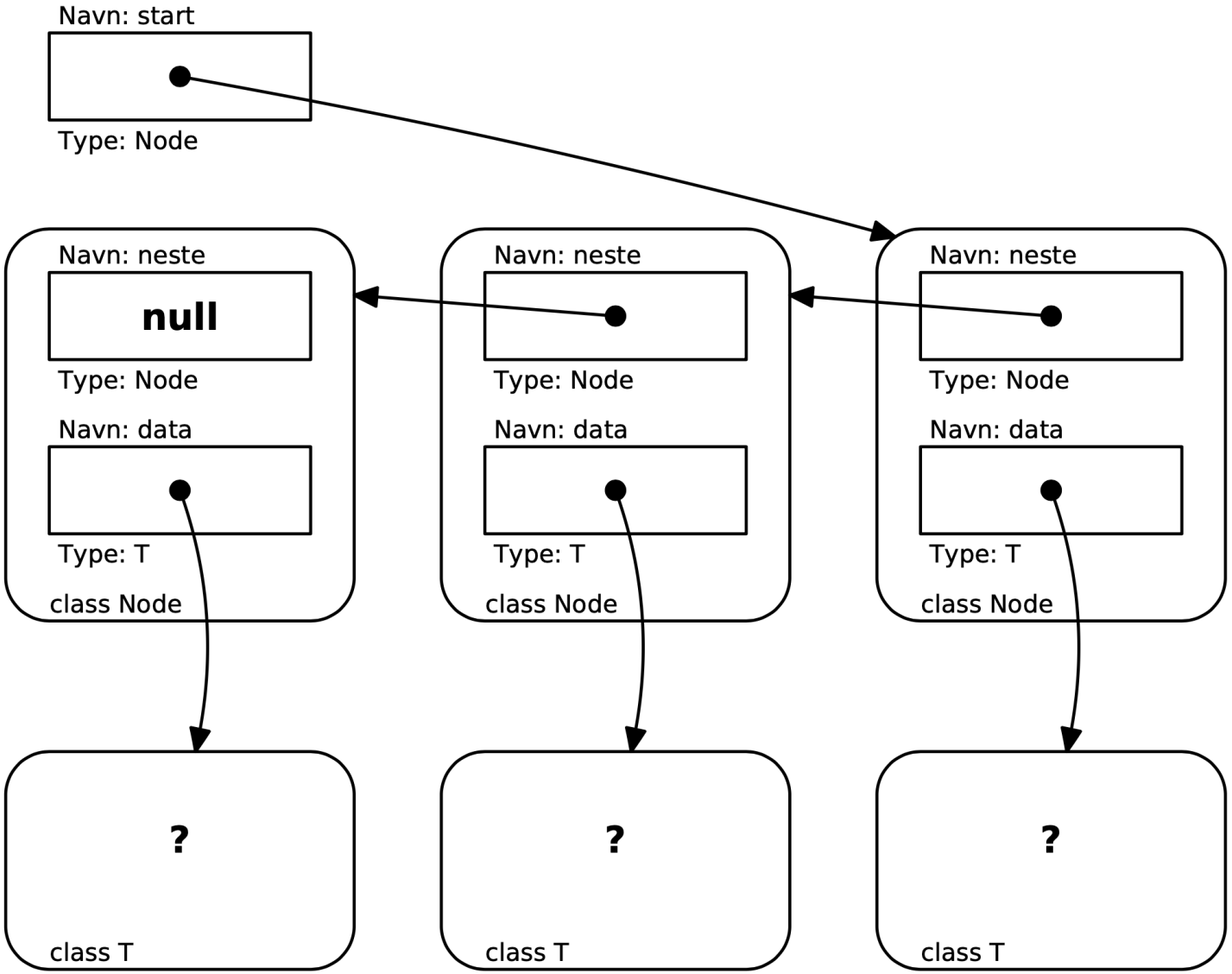
Dersom vi antar at munkene kan flytte 1 ring per sekund, så tar det omtrent **585 milliarder år**, eller omtrent 42 ganger universets alder…

https://en.wikipedia.org/wiki/Tower_of_Hanoi

Enveis lenkede lister



Vi skal skrive en metode **reverse**, som snur listen:



```
class Lenkeliste<T> implements Liste<T> {
    Node start = null;

    class Node {
        Node neste = null;
        T data;

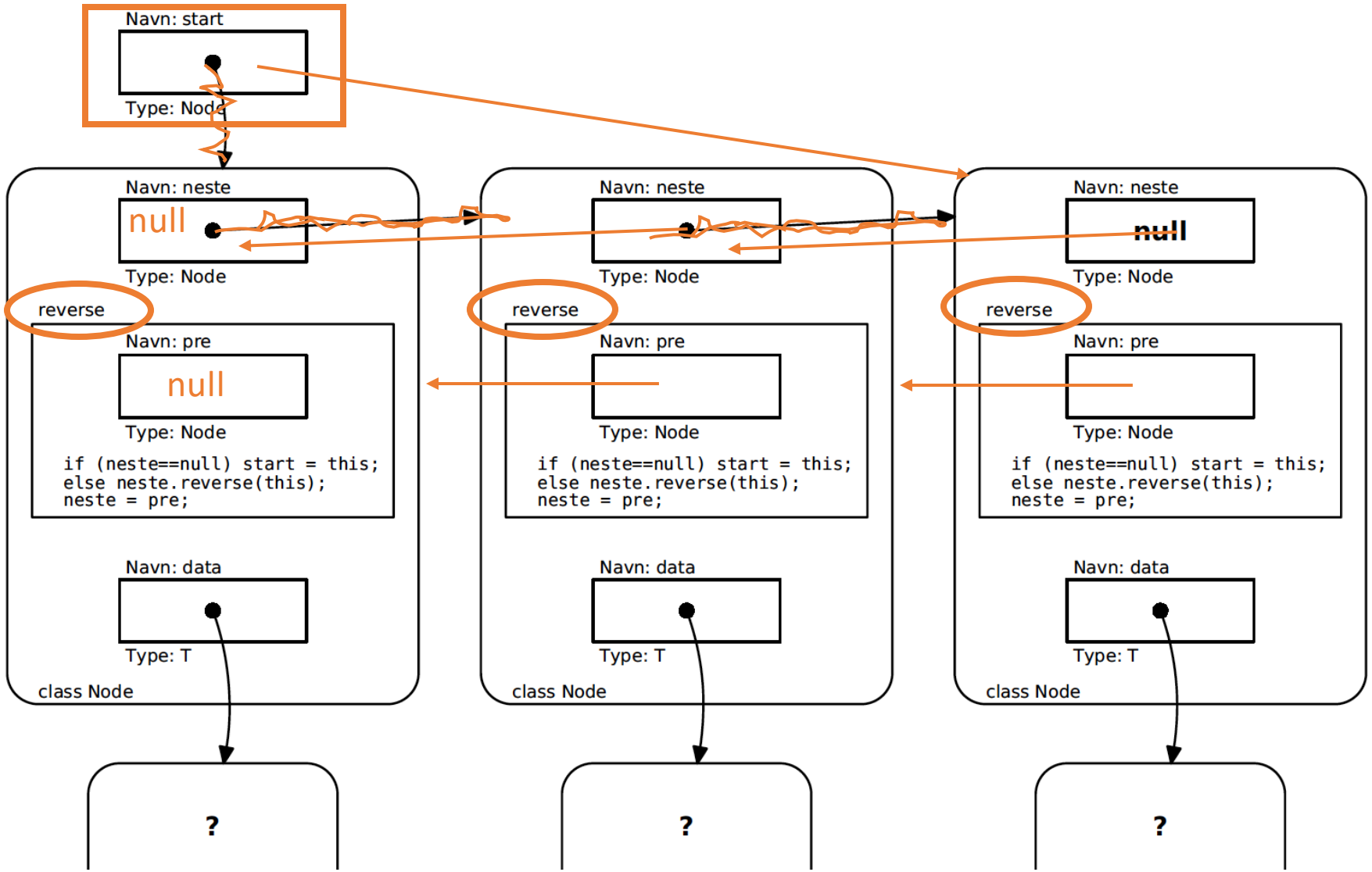
        Node(T x) {
            data = x;
        }

        void reverse(Node pre) {
            if (neste == null)
                // dersom neste er null, er vi på slutten av lista som skal snus,
                // og starten blir dermed denne noden:
                start = this;
            else
                // ellers reverserer vi lista rekursivt:
                neste.reverse(this);

            // neste node er den som tidligere var før oss i kjeden:
            neste = pre;
        }
    }
}
```

Hva er det trivielle (ikke-rekursive) tilfellet her?

Metoden **reverse** under kjøring




```java
class Lenkeliste<T> implements Liste<
    Node start = null;

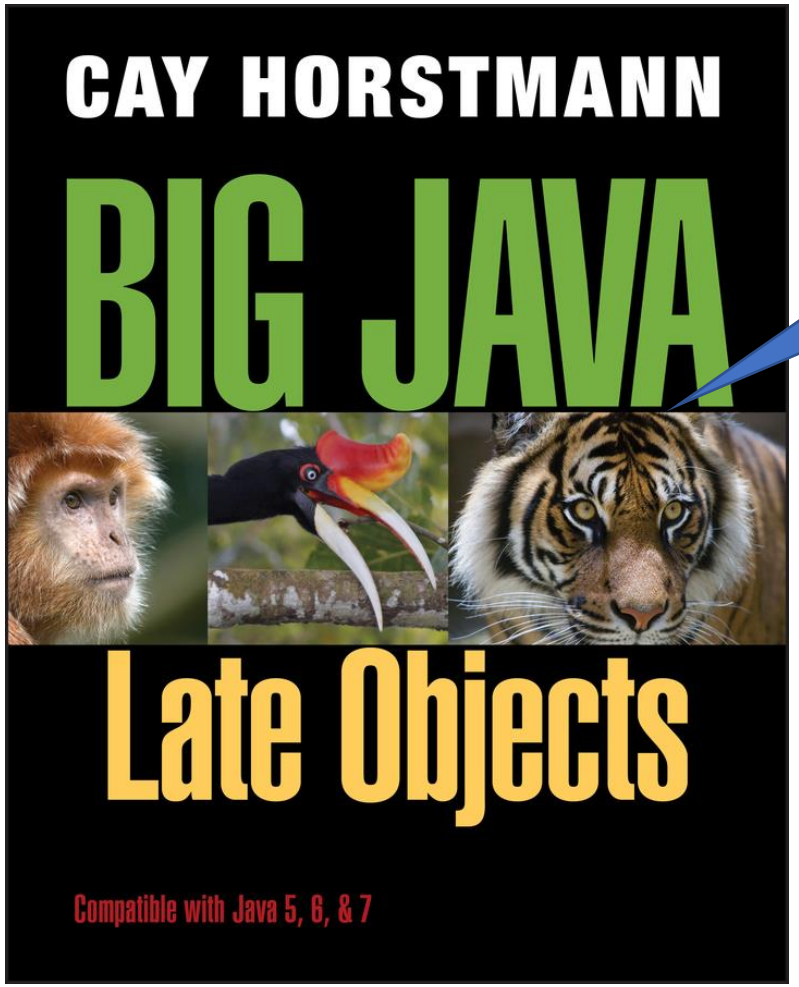
    class Node {
        Node neste = null;
        T data;

        Node(T x) {
            data = x;
        }

        void reverse(Node pre) {
            if (neste == null)
                // dersom neste er nu
                // og starten blir de
                start = this;
            else
                // ellers reverserer
                neste.reverse(this);

            // neste node er den som
            neste = pre;
        }
    }
}
```

Oppsummering



The key to the successfull design of a recursive method is not to think about it.

```java
class Hanoi {
    static void flytt(int n, char fra, char via, char til) {
        if (n == 1) {
            // Det trivielle tilfellet: flytt 1 ring direkte:
            System.out.println("Flytt " + fra + " til " + til);
        } else {
            // Det rekursive tilfellet:
            //
            // Flytt de små ringene til hjelpestolpen:
            flytt(n - 1, fra, til, via);
            // Flytt den største ringen fra 'fra'-stoplen til 'til'-stoplen:
            flytt(1, fra, via, til);
            // Flytt ringene fra hjelpestolpen til 'til'-stoplen igjen:
            flytt(n - 1, via, fra, til);
        }
    }

    public static void main(String[] arg) {
        int antall = Integer.parseInt(arg[0]);
        flytt(antall, 'A', 'B', 'C');
    }
}
```

Oppsummering

- En rekursiv metode er en metode som kaller på seg selv
- Det rekursive kallet må alltid være *enklere*
- Det trivielle tilfellet må håndteres ordentlig

