Rekursjon

(Big Java kapittel 13)

Fra *Urban dictionary*:

recursion see *recursion*.

Å beregne fakultet

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

$$1! = 1 = 1$$

$$2! = 1 \times 2 = 2$$

$$3! = 1 \times 2 \times 3 = 6$$

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

$$n! = 1 \times 2 \times \cdots \times n =$$

Beregning av fakultet med løkke

Fakultet1.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));
    }
}
```

```
$ java Fakultet1 6
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

En rekursiv løsning

Vi kan imidlertid skrive denne metoden på en helt annen måte om vi innser at fakultetsfunksjonen kan defineres slik:

$$1! = 1$$
$$n! = n \times (n-1)!$$

Fakultet2.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));
    }
}
```

```
$ java Fakultet2 6
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

Bredden av et heltall

Problem: Hvor mange posisjoner trenger man når man skal skrive ut et gitt heltall?

```
$ java Bredde 3
3 har bredde 1
```

```
$ java Bredde -355
-355 har bredde 4
```

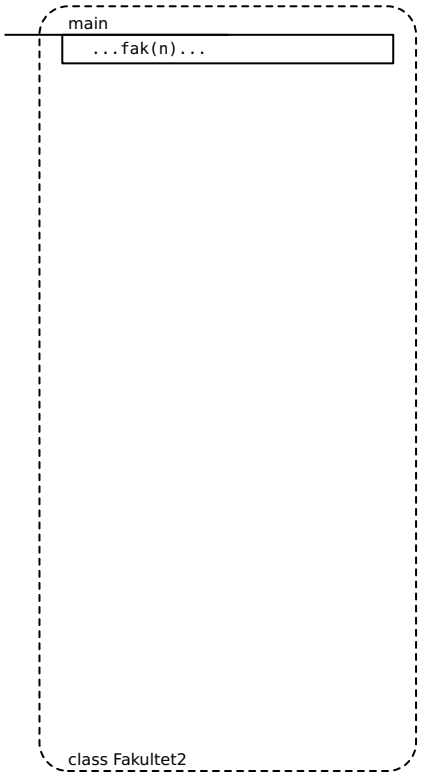
```
class Bredde {
    static int w(int n) {
        if (n < 0) return 1 + w(-n);
        if (n <= 9) return 1;
        return 1 + w(n/10);
    }

    public static void main(String[] arg) {
        int v = Integer.parseInt(arg[0]);
        System.out.println(v + " har bredde " + w(v));
    }
}
```

Rekursjon

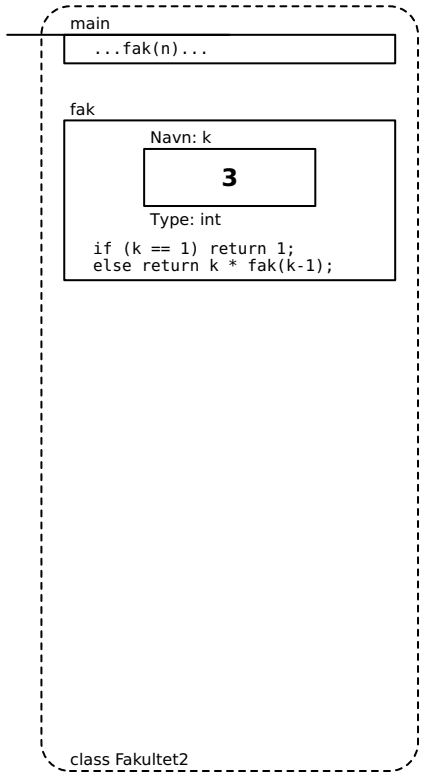
Programkode som kaller seg selv, kalles **rekursiv kode**.

- Noen problemer er rekursive av natur (for eksempel en kompilator eller Hanois tårn)
- Rekursiv programmering er spesielt nyttig for programmer som skal søke ulike veier til en mulig løsning (for eksempel en labyrint eller sjakkspill)
- Noen programmerere foretrekker rekursive løsninger.

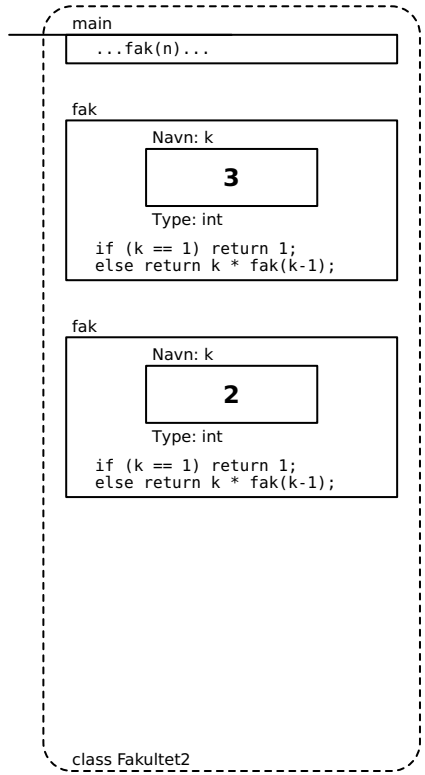
Går det virkelig bra å skrive rekusiv kode?



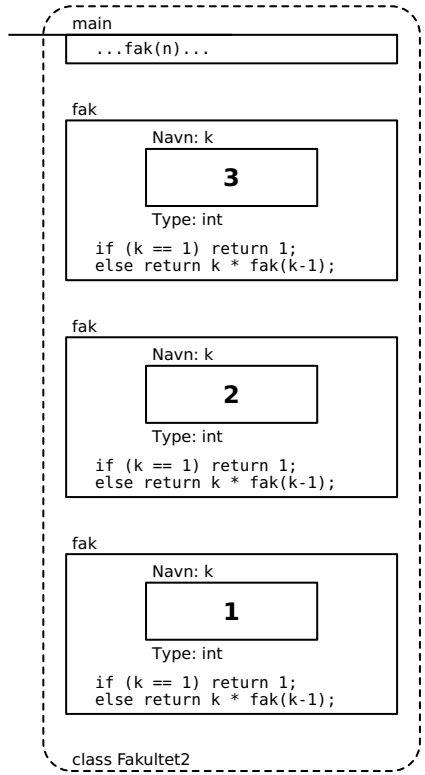
Tilstand 1



Tilstand 2



Tilstand 3



Tilstand 4

Fibonacci-tallene

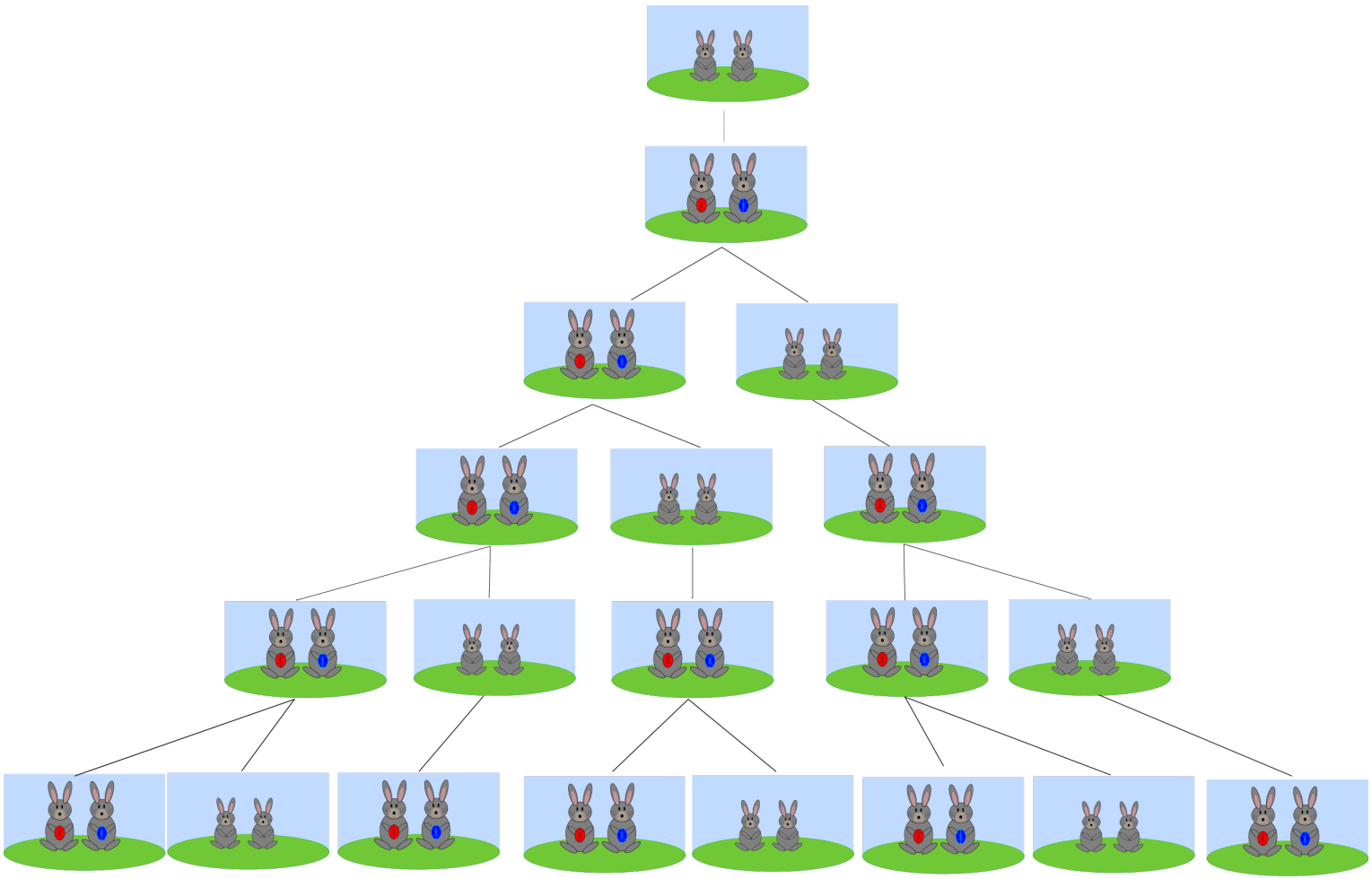
Tallsekvensen

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

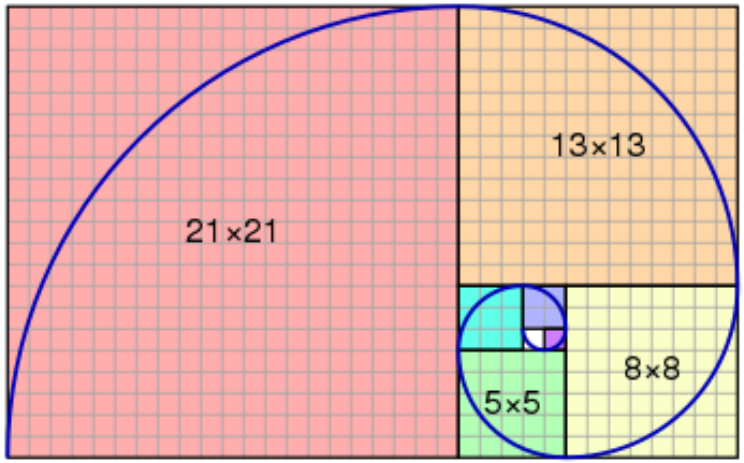
der hvert tall er summen av de to foregående, forekommer usedvanlig ofte i naturen.

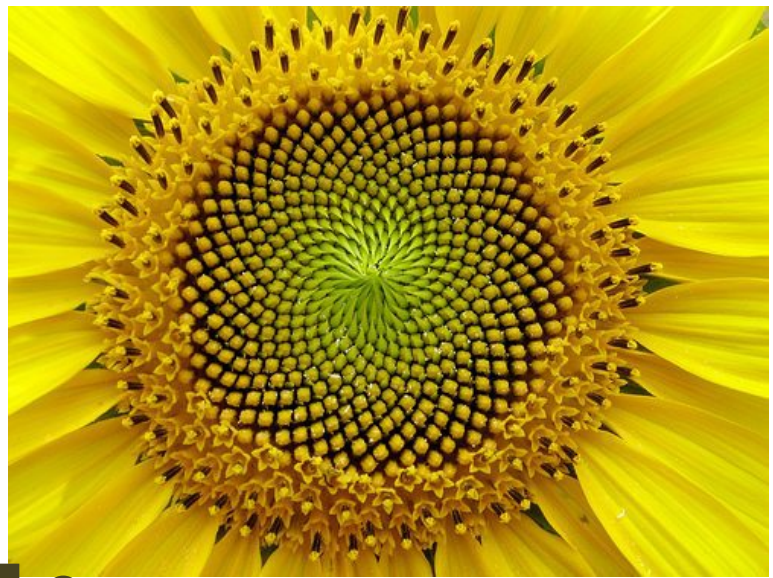
Hvorfor er Fibonacci-tallene interessante

Leonardo Fibonacci (ca 1170–ca 1250) fant denne tallsekvensen som svar på hvordan kaniner formerer seg:



Overraskende ofte finner vi disse tallene i naturen:





Vi skal altså programmere sekvensen gitt ved

$$F_n = F_{n-1} + F_{n-2} \quad \text{og} \quad F_1 = 1$$

Fibonacci.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));
    }
}
```

Dette gikk ikke bra:

```
$ java Fibonacci 10
Fib(1) = 1
Exception in thread "main" java.lang.StackOverflowError
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
        at Fibonacci.fib(Fibonacci.java:4)
              ⋮
```

Problemet oppstår i kallet **fib(2)** som så kaller på

1. fib(1) (går OK)
2. fib(0) som kaller på fib(-1) som . . .

Huskeregler for rekursiv programmering

Når vi programmerer rekursivt, må vi alltid huske på

- Det rekursive kallet må være et kall på et *enklere* problem.
- Vi må sørge for at det *trivielle* problemet håndteres skikkelig.

En riktig versjon av koden:

Fibonacci2.java

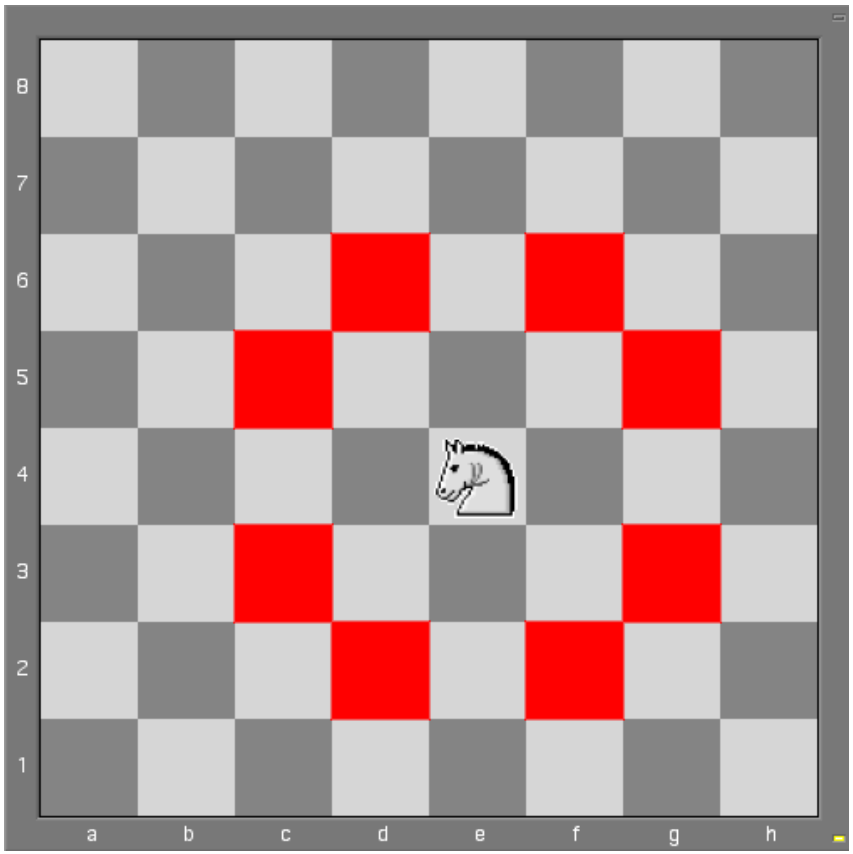
```
class Fibonacci2 {
    static int fib(int n) {
        if (n <= 2) return 1;
        else return fib(n-1) + fib(n-2);
    }

    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));
    }
}
```

En komplett springertur

Et vanlig hobbyproblem i sjakk er:

> *Flytt springeren slik at den er innom alle rutene på et sjakkbrett.*

Dette kan vi løse ved å prøve alle muligheter.



Hvordan flyttes springeren i sjakk?

```
class KnightsTour {
  static int[][] brett;
  static int bredde, bredde2;

  static void flytt(int trekk, int x, int y) {
    if (x < 0 || x >= bredde) return;
    if (y < 0 || y >= bredde) return;
    if (brett[x][y] > 0) return;

    brett[x][y] = trekk;

    if (trekk == bredde2) {
      tegnBrett();  System.exit(1);
    }

    flytt(trekk+1, x+2, y+1);
    flytt(trekk+1, x+2, y-1);
    flytt(trekk+1, x+1, y+2);
    flytt(trekk+1, x+1, y-2);
    flytt(trekk+1, x-1, y+2);
    flytt(trekk+1, x-1, y-2);
    flytt(trekk+1, x-2, y-1);
    flytt(trekk+1, x-2, y-1);
    brett[x][y] = 0;
  }

  static void tegnBrett() {
    for (int iy = bredde-1; iy >= 0; iy--) {
      for (int ix = 0; ix < bredde; ix++) {
        System.out.printf("%3d",brett[ix][iy]);
      }
      System.out.println();
    }
  }

  public static void main(String[] arg) {
    bredde = Integer.parseInt(arg[0]);
    bredde2 = bredde * bredde;
    brett = new int[bredde][bredde];
    flytt(1, 0, 0);
    System.out.println("Ingen løsninger!");
  }
}
```

Da kan vi finne løsningene:

```
$ javac KnightsTour.java
$ java KnightsTour 1
  1

$ java KnightsTour 2
Ingen løsninger!

$ java KnightsTour 3
Ingen løsninger!

$ java KnightsTour 4
Ingen løsninger!

$ java KnightsTour 5
 25 14  9 18  5
  8 19  6 23 10
 13 24 15  4 17
 20  7  2 11 22
  1 12 21 16  3
```

```
$ java KnightsTour 6
 14 27 22 31 16 29
 21 32 15 28 23  4
 26 13 24  5 30 17
 33 20 11  8  3  6
 12 25  2 35 18  9
  1 34 19 10  7 36

$ java KnightsTour 7
 39 24 43 28 41  8 19
 44 27 40 25 20  5 10
 23 38 29 42  9 18  7
 30 45 26 21  6 11  4
 37 22 35 32  3 14 17
 46 31  2 15 48 33 12
  1 36 47 34 13 16 49
```

Hanois tårn

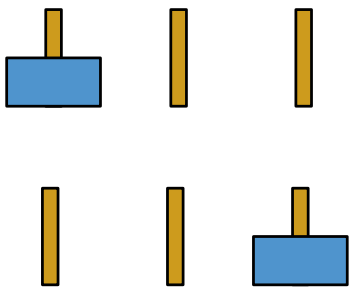
Den oppdiktete bakgrunnen for spillet fra 1883 er:

I et tempel i Hanoi står tre staker, og på den venstre ligger 60 ringer av ulik størrelse, sortert med den minste øverste.

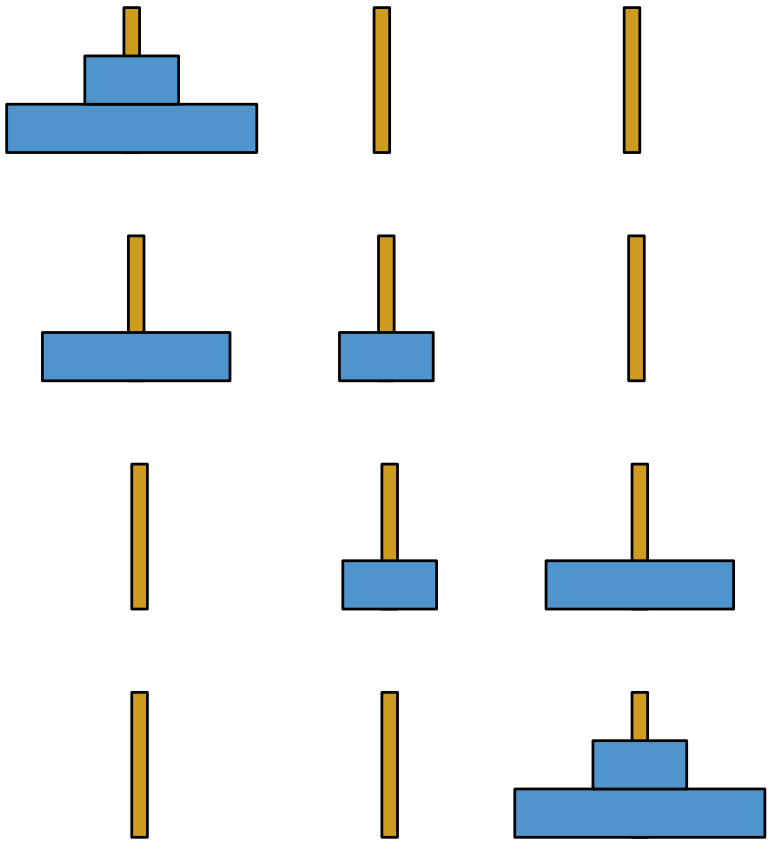
Oppgaven er å flytte ringene til den høyre staken, men

- Ringene må flyttes én og én.
- En ring må aldri legges oppå en mindre ring.
- Den midtre stolpen kan benyttes underveis.

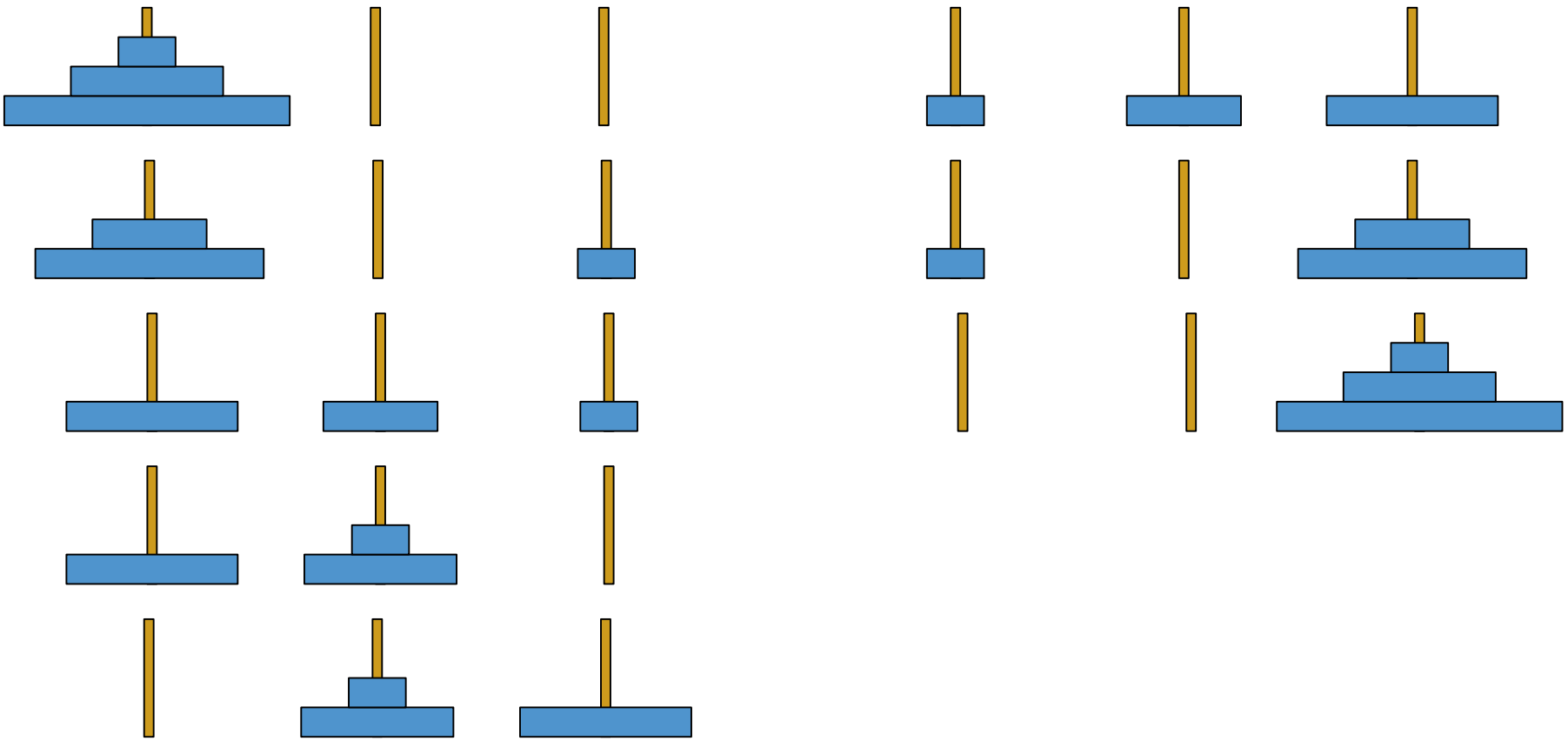
Hanois tårn med 1 ring



Hanois tårn med 2 ringer



Hanois tårn med 3 ringer



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.

Hanoi.java

```
class Hanoi {
    static void flytt(int n, char fra, char via, char til) {
        if (n == 1) {
            System.out.println("Flytt " + fra + " til " + til);
        } else {
            flytt(n-1, fra, til, via);
            flytt(1, fra, via, til);
            flytt(n-1, via, fra, til);
        }
    }

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

Slik løser vi problemet

```
$ 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
```

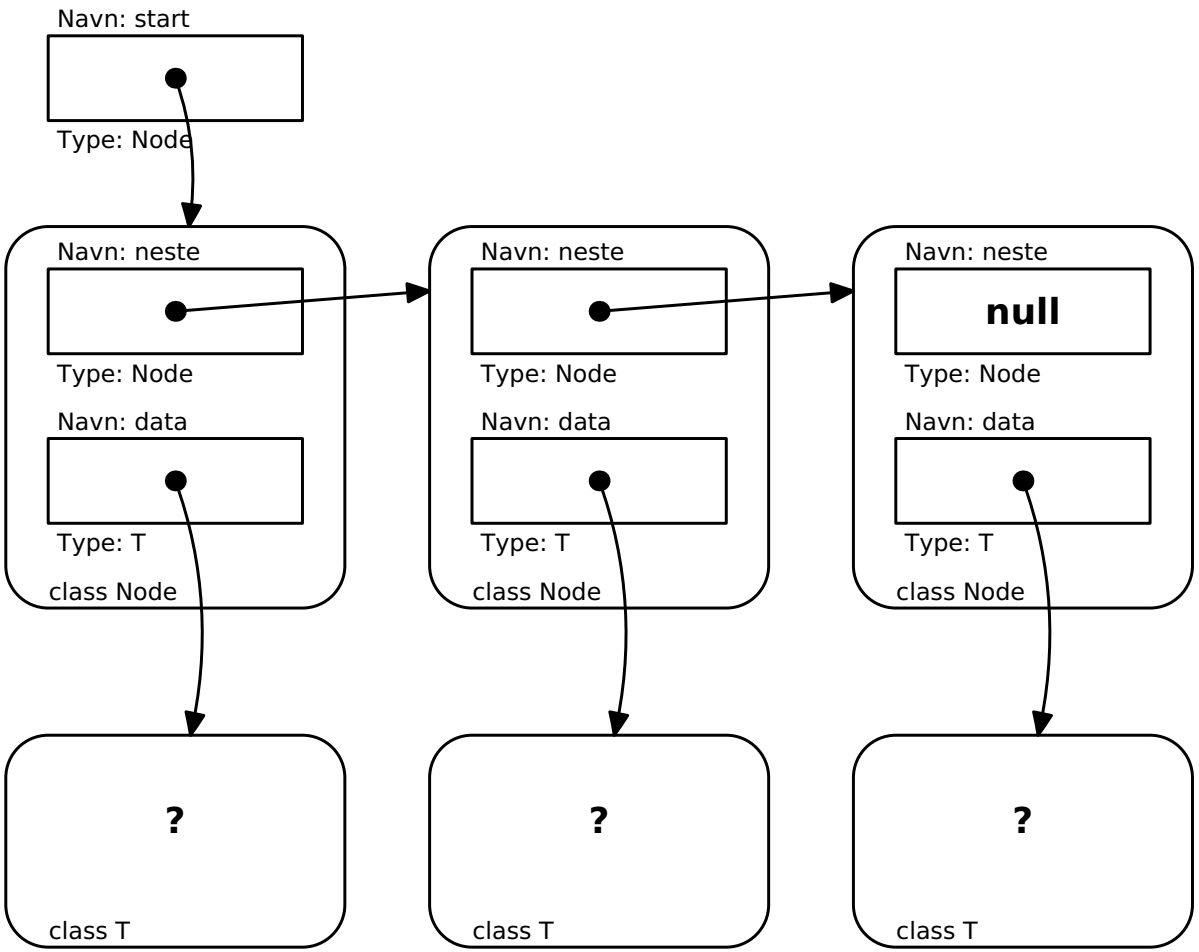
Hvor mange trekk trenger man?

Leketøysvarianten har 8 ringer. Man må da foreta $2^8 - 1 = 255$ flyttinger.

Den originale «legenden» sier at det er 60 ringer. Da trengs $2^{60} - 1 \approx 10^{18}$ flyttinger.

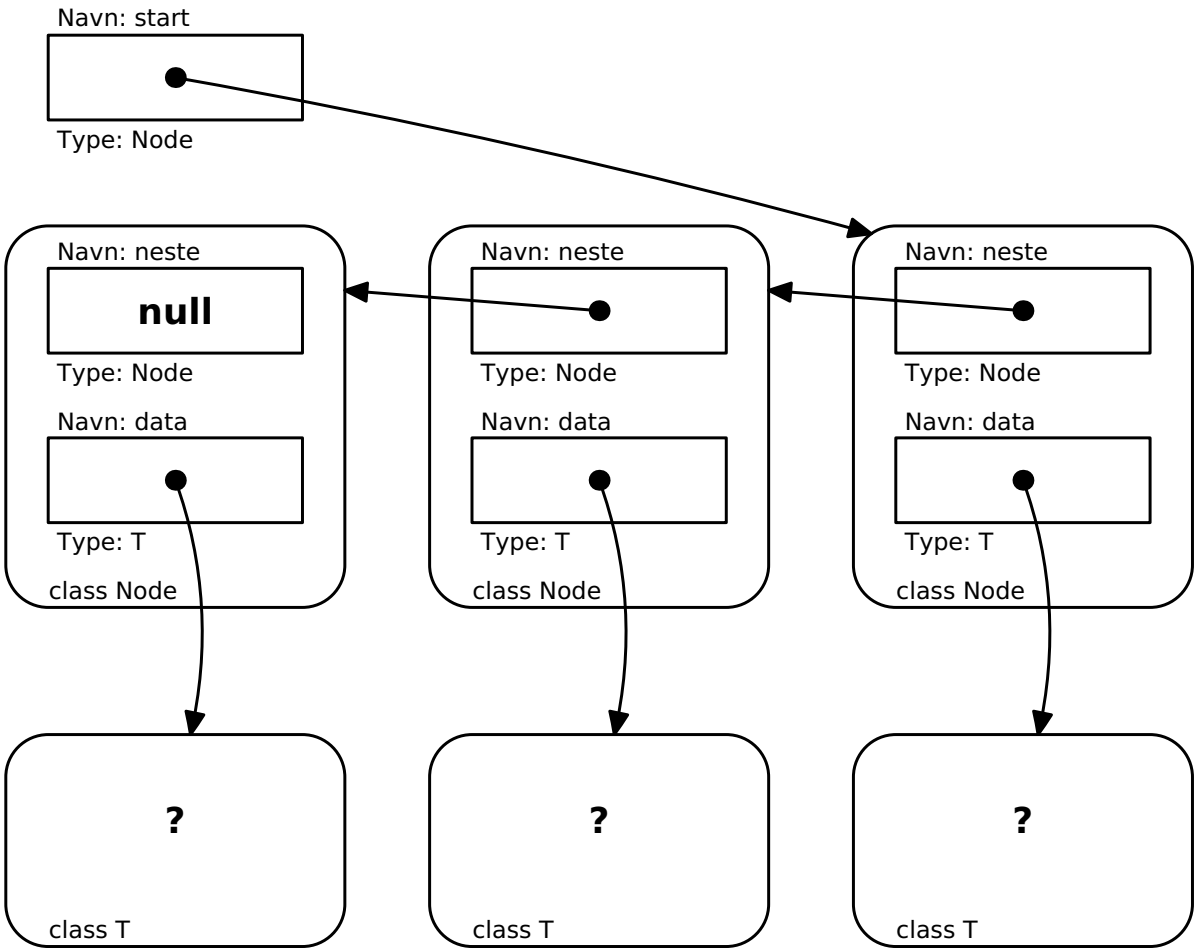
Hvis vi antar at munkene kan flytte én ring hvert sekund, vil det ta omtrent 36 534 658 086 år før verden går under.

Enveis lister



Hvordan kan vi enklest snu en slik liste?

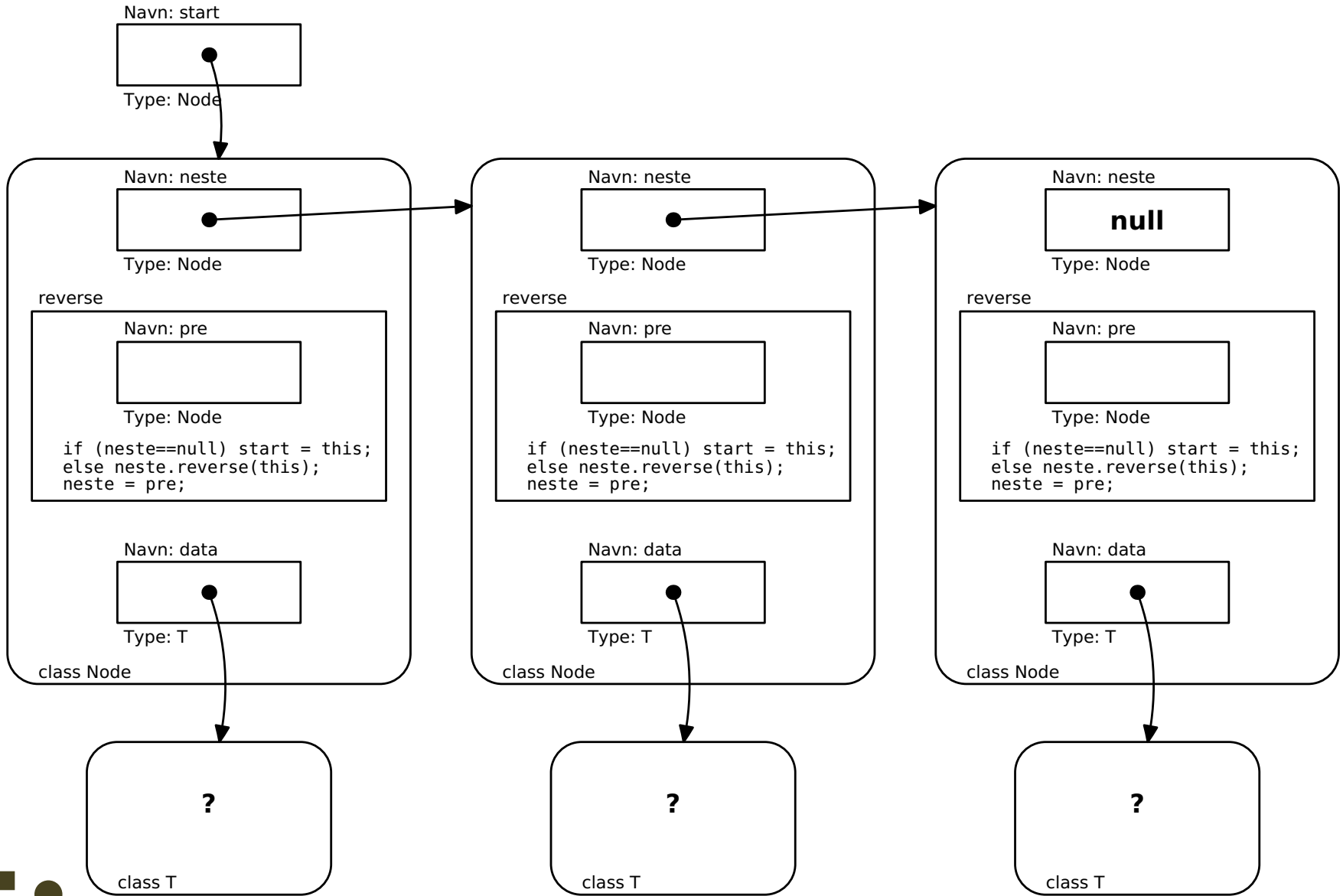
Oppgave: Skriv en metode **reverse** som snur listen til



Lenkeliste.java

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

        Node(T x) { data = x; }

        void reverse(Node pre) {
            if (neste == null) start = this;
            else neste.reverse(this);
            neste = pre;
        }
    }
    private Node start = null;

    public void reverse() {
        if (start != null)
            start.reverse(null);
    }
```

Metoden reverse



Oppsummering

Sitat fra læreboken

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

Men husk allikevel alltid på at

- den rekursive metoden må kalle på en *enklere* versjon av problemet
- vi må håndtere det *trivielle* problemet.