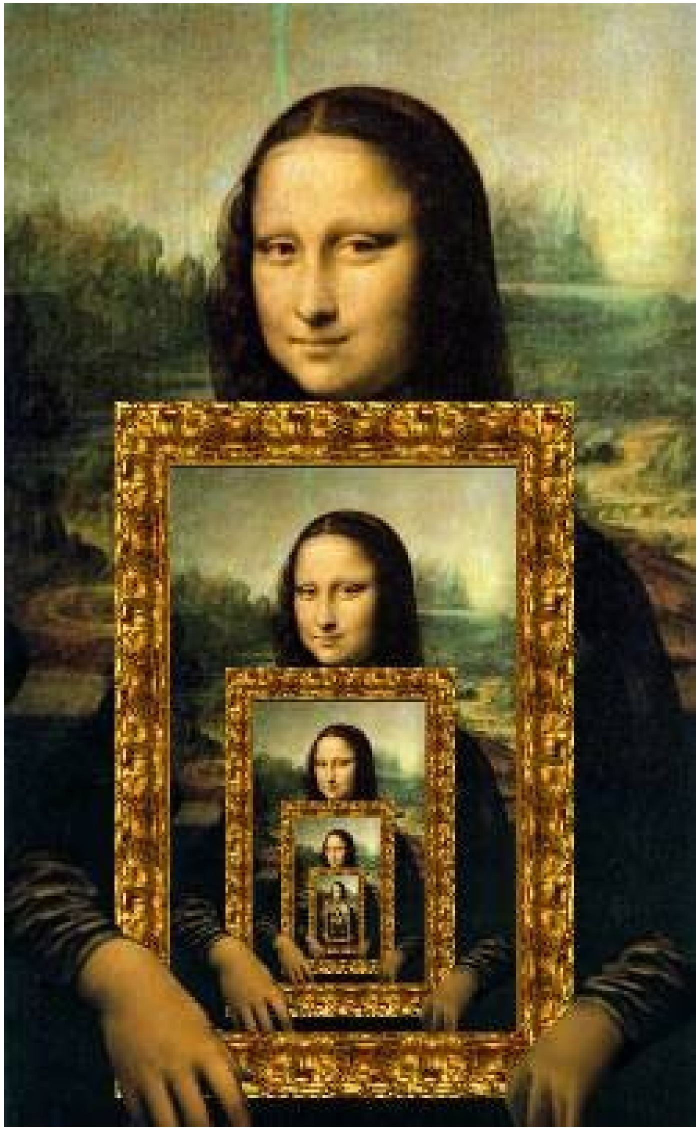
Rekursiv programmering

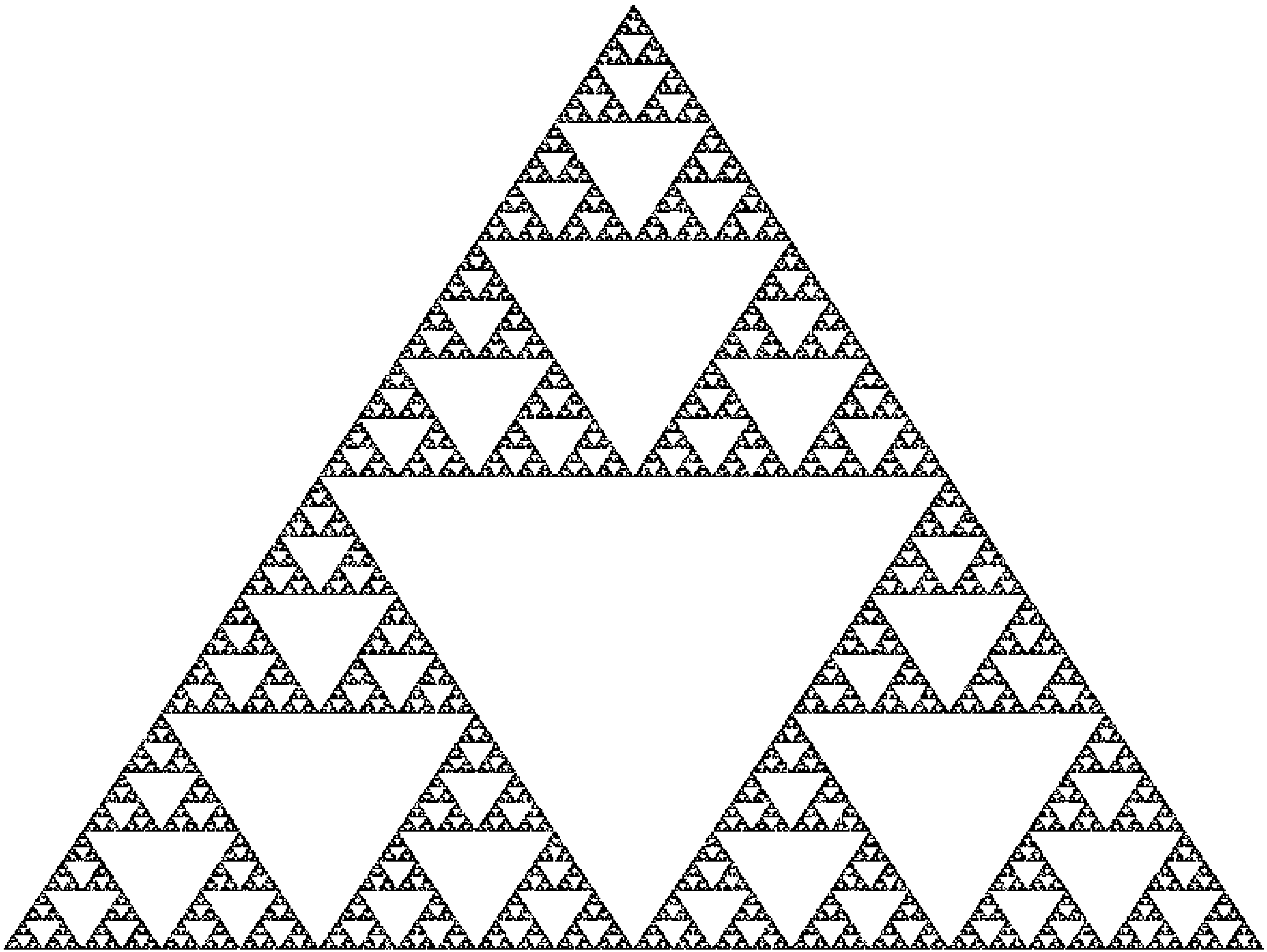


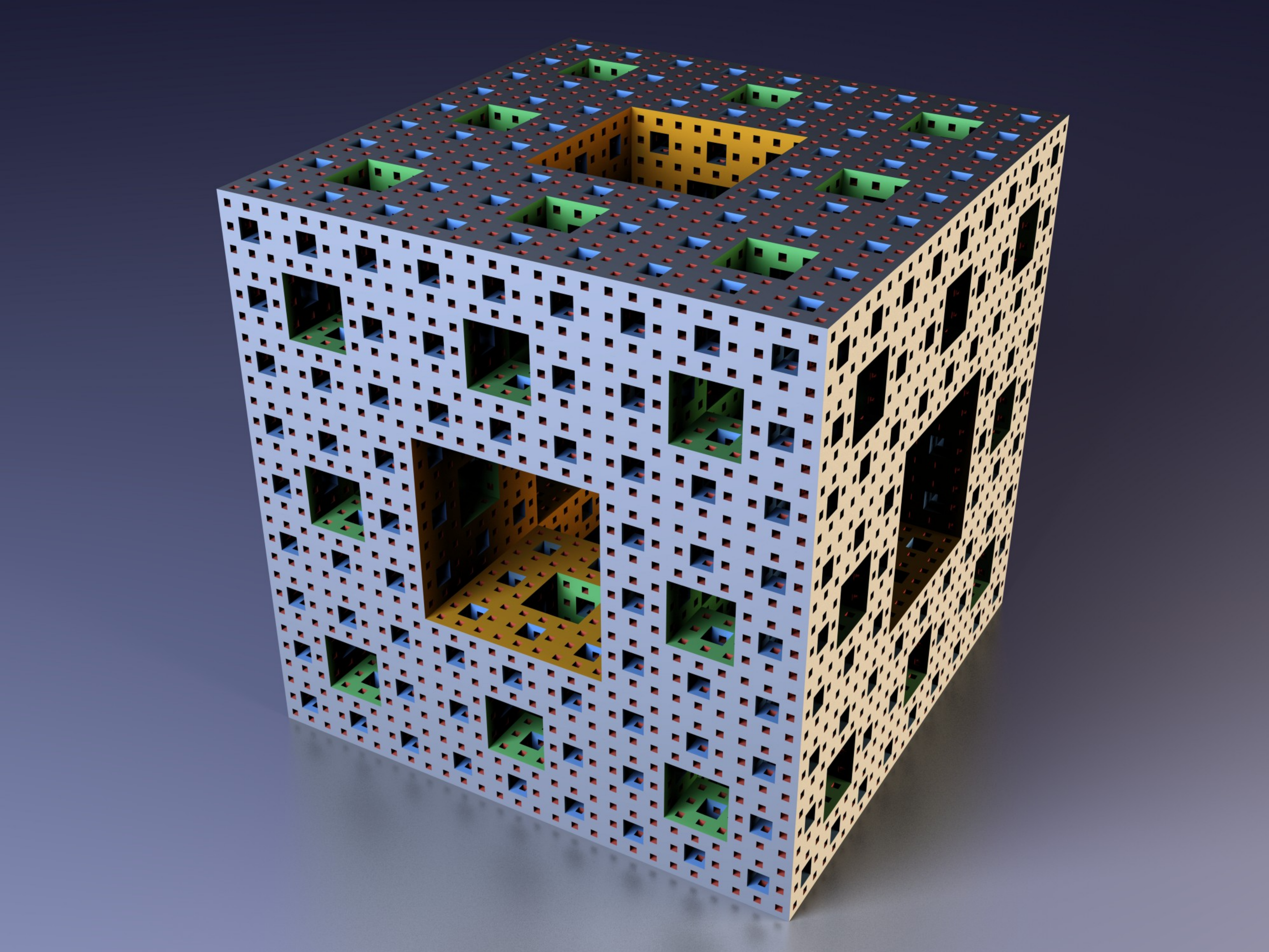
Babushka-dukker

- En russisk Babushka-dukke er en sekvens av like dukker *inne i hverandre*, som kan åpnes

- Hver gang en dukke åpnes er det en *mindre* utgave av dukken inni, inntil man kommer til den *minste* dukken, som ikke kan åpnes

Sierpinskis trekant





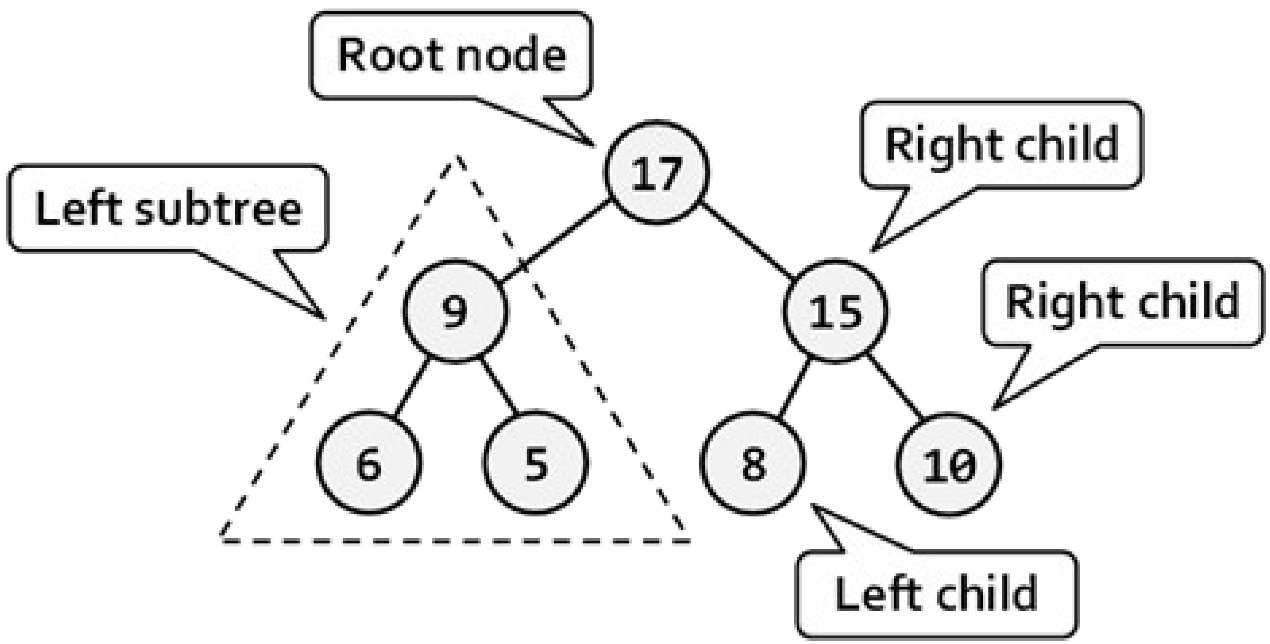

Rekursive beskrivelser og definisjoner

- Et objekt er rekursivt hvis det er beskrevet med *mindre* versjoner av seg selv
- Både matematiske formler og metoder i programmer kan også defineres rekursivt
- En rekursiv definisjon må alltid bestå av to deler:
    1. Et minste tilfelle (“base case”) som *ikke* kan beskrives med mindre utgaver av seg selv
    2. En rekursiv del som beskriver hvorledes formelen eller programmet kan settes sammen med mindre utgaver av seg selv

En rekursiv datastruktur: Binært tre

- Datastruktur der hver node kan ha *to* etterfølgere, som kalles venstre og høyre *barn* (kapittel 10 i læreboken)
- Et binært tre er en struktur med et endelig antall noder som enten:
  1. Ikke inneholder noen noder (er tomt), eller:
  2. Består av en *rotnode*, med et venstre subtre som er et binært tre og et høyre subtre som er et binært tre
- Minste tilfelle / “base case”: Null noder
- Rekursiv del: En rotnode og to binære trær som begge har en node *mindre*

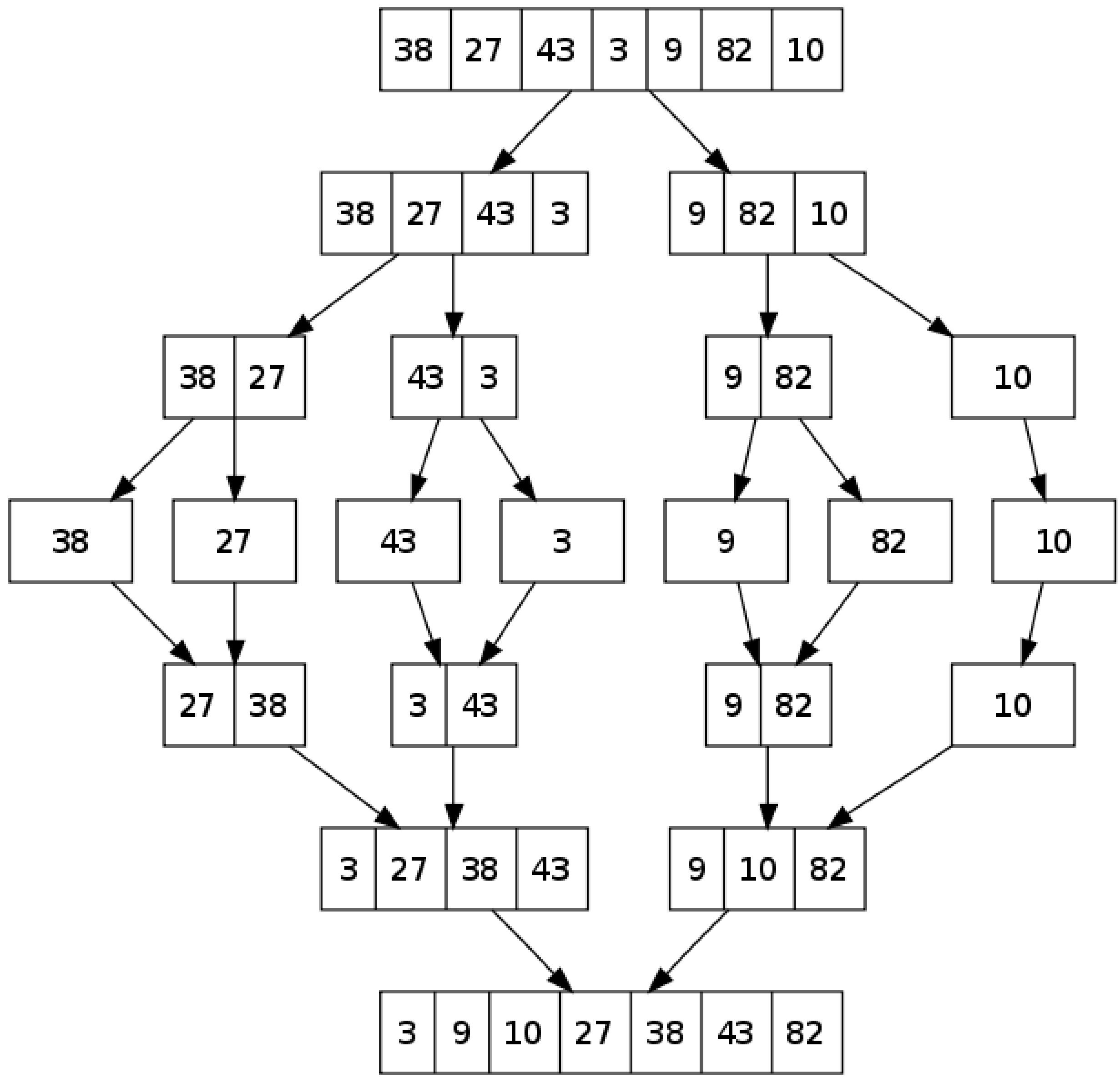
Binært tre


Root node
Right child
Left subtree
Right child
Left child
17
9
15
6
5
8
10

Rekursiv algoritme: Flettesortering

- Eksempel på “splitt og hersk” - algoritme for å sortere en array (kapittel 9 i læreboken)
- Base case: Bare ett element i arrayen
- Rekursiv del:
  - Del arrayen med *n* elementer “på midten”, i to mindre arrayer med *n*/2 elementer i hver
  - Sorter hver av de to halvdelene *rekursivt* med flettesortering
  - *Flett* deretter de to sorterte halvdelene sammen til en hel sortert array

Flettesortering / Merge sort



Rekursive algoritmer

- Må alltid ha minst ett “base case”/minste tilfelle som kan løses *uten* rekursjon
- Må ha et steg som *reduserer* problemet som skal løses til et *mindre* problem med samme struktur
- Problemet må ha et *mål* for problemstørrelsen som flytter seg *mot* base case i hvert rekursive steg
- Dette sikrer at algoritmen til slutt når frem til base case (bunnen i rekursjonen)
- I rekursiv programmering må vi anta at rekursjonen *alltid* virker – og da vil den virke! (magi?)

Klassisk eksempel: Fakultet

- Iterativ definisjon, programmeres enkelt med en for-løkke:

$$n! = n \cdot \underbrace{(n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1}_{(n-1)\,!}$$

- Rekursiv definisjon:

$n! = 1$, hvis $n = 1$

$n! = n \cdot (n-1)\,!$, hvis $n > 1$

- Kan programmeres "rett frem" med et *rekursivt metodekall*

Rekursivt fakultet i Java

```
long factorial(int n)
{
  if (n <= 1)
    return 1;

  return n * factorial(n - 1);
}
```

- Java-kode med testprogram: `factorialDemo.java`

Rekursive kall: `factorial(6)`

```
factorial(6) = 6 * factorial(5)
             = 6 * (5 * factorial(4))
             = 6 * (5 * (4 * factorial(3)))
             = 6 * (5 * (4 * (3 * factorial(2))))
             = 6 * (5 * (4 * (3 * (2 * factorial(1)))))
             = 6 * (5 * (4 * (3 * (2 * 1))))
             = 6 * (5 * (4 * (3 * 2)))
             = 6 * (5 * (4 * 6))
             = 6 * (5 * 24)
             = 6 * 120
             = 720
```

Og hva skjer “bak kulissene”?

- Rekursjon er “mystisk” (til å begynne med) for oss...
- Men, for datamaskinen og kompilatoren er det *ingen* forskjell på et rekursivt metodekall og et vanlig metodekall
- Når en metode kaller en annen metode:
  - Alle data for kallende metode legges unna på en “memory stack” og en “execution stack”
  - Når metoden som ble kalt er *ferdig* med å eksekvere, tar kallende metode over igjen, lokale data hentes fra stackene og eksekveringen fortsetter fra rett *etter* det ferdige metodekallet

Kallstacken i Java: "Vanlige" metodekall

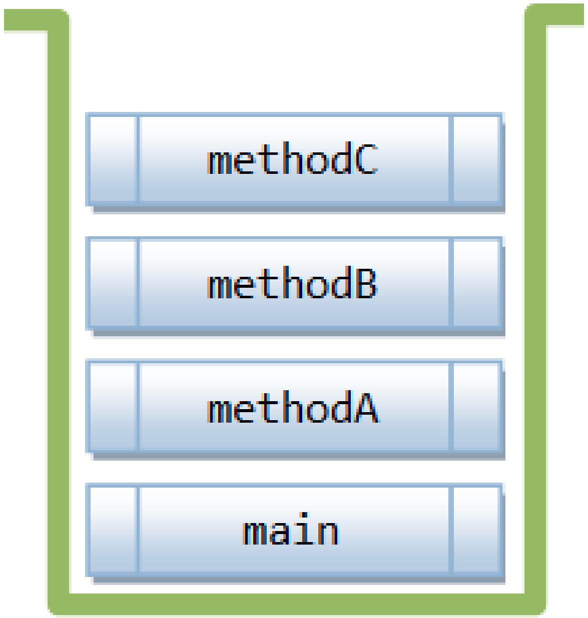
```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit  main()");
   }

   void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit  methodA()");
   }

   void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit  methodB()");
   }

   void methodC() {
      System.out.println("Enter methodC()");
      System.out.println("Exit  methodC()");
   }
}
```

Output:

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit  methodC()
Exit  methodB()
Exit  methodA()
Exit  main()
```

Kallstack: Tre ikke-rekursive metoder

1. JVM invoke the main().
2. main() pushed onto call stack, before invoking methodA().
3. methodA() pushed onto call stack, before invoking methodB().
4. methodB() pushed onto call stack, before invoking methodC().
5. methodC() completes.
6. methodB() popped out from call stack and completes.
7. methodA() popped out from the call stack and completes.
8. main() popped out from the call stack and completes. Program exits.



Method Call Stack
(Last-in-First-out Queue)

Kallstacken i Java: Rekursivt metodekall

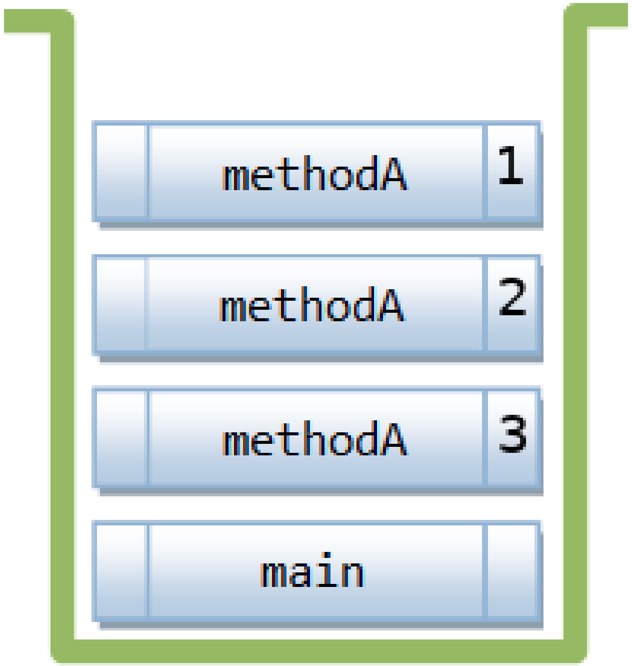
```
public class RecursionCallStackDemo
{
   void methodA(int n)
   {
      System.out.println
       ("Enter methodA(" + n +")");

      if (n > 1)
         methodA(n-1);

      System.out.println
       ("Exit  methodA(" + n +")");
   }

   public static void main(String[] args)
   {
      System.out.println("Enter main()");
      methodA(3);
      System.out.println("Exit  main()");
   }
}
```

Output:

```
Enter main()
Enter methodA(3)
Enter methodA(2)
Enter methodA(1)
Exit  methodA(1)
Exit  methodA(2)
Exit  methodA(3)
Exit  main()
```

Kallstack: Rekursiv metode

1. JVM invoke the main().
2. main() pushed onto call stack, before invoking methodA(3).
3. methodA(3) pushed onto call stack, before invoking methodA(2).
4. methodA(2) pushed onto call stack, before invoking methodA(1).
5. methodA(1) completes.
6. methodA(2) popped out from call stack and completes.
7. methodA(3) popped out from the call stack and completes.
8. main() popped out from the call stack and completes. Program exits.



Method Call Stack

Eksempel: Fibonaccitall

- Fibonaccitallene er en sekvens av heltall:

  $F_1, F_2, F_3, F_4, \ldots,$

  som er slik at:

  $F_1 = 1$ og $F_2 = 1$,

  og hvert av de neste Fibonaccitallene er lik summen av de to foregående:

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

- Fibonaccitall nummer n, for $n > 2$, beregnes som:

  $$F_n = F_{n-1} + F_{n-2}$$

Rekursiv beregning av Fibonaccitall

- Base case:

  $F_1 = 1$ og $F_2 = 1$

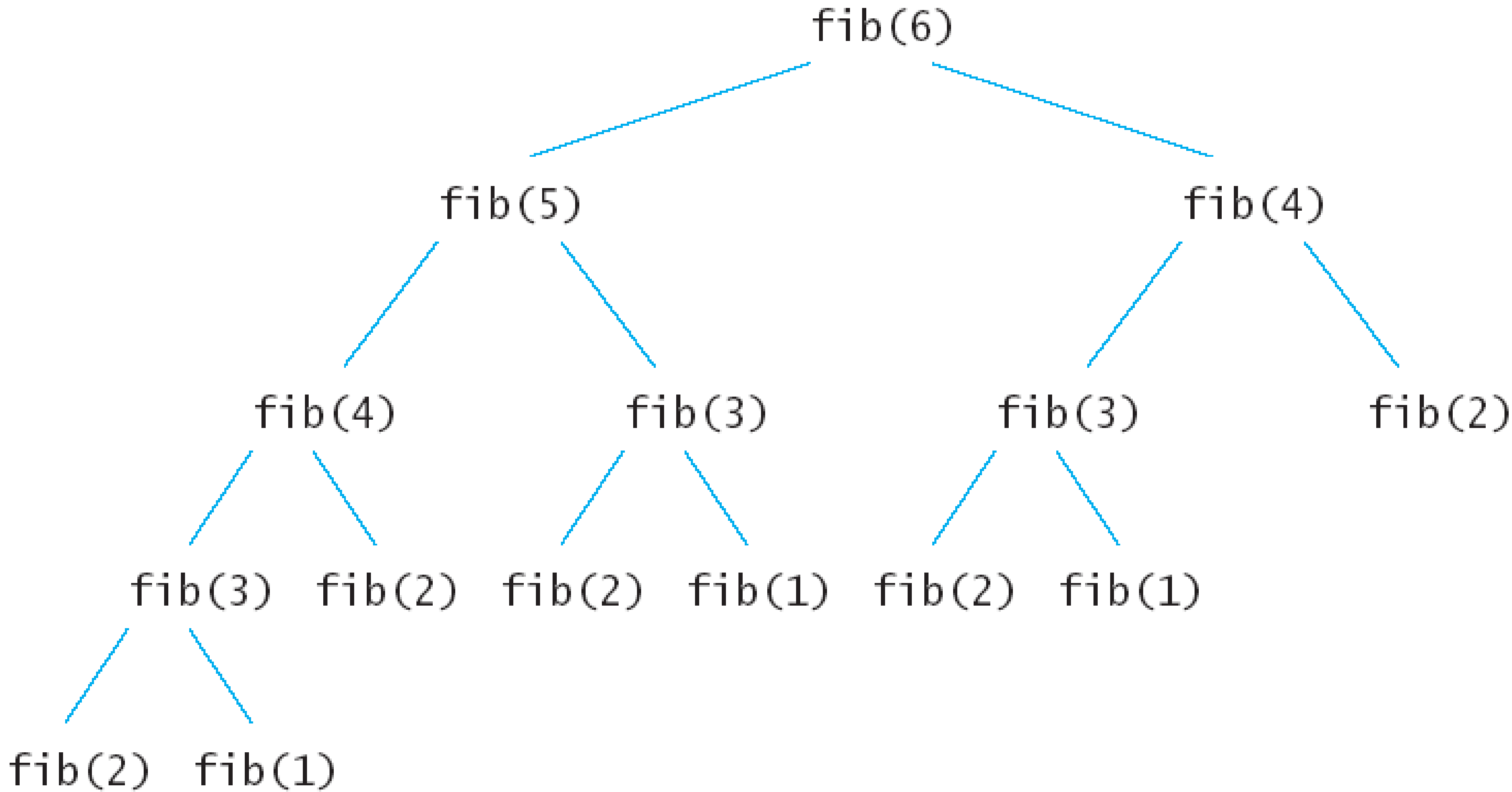
- Rekursiv del:

  $F_n = F_{n-1} + F_{n-2}$, $n > 2$

- Rekursiv Java-metode for å beregne *Fn*:

```
public static long fib(int n)
{
  if (n <= 2)
    return 1;
  return fib(n - 1) + fib(n – 2);
}
```

Kalltre for rekursiv beregning av F_6



Rekursive Fibonacitall: Effektivitet

- Rekursiv beregning av F_n er en dårlig løsning:
  - Gjør svært mange *redundante* beregninger
  - Beregning av f.eks F_6 medfører 5(!) beregninger av F_2
  - Er et eksempel på en *ekstremt* ineffektiv algoritme
  - Kan vises matematisk (med induksjonsbevis) at kjøretiden vokser som F_n, som er *eksponentielt* ($F_n \approx 1.62^n$)
- Lett å lage iterativ versjon med for-løkke som er $O(n)$
- Sammenligning av kjøretider: `fibonacciDemo.java`

Når er rekursjon riktig løsning?

- Rekursjon er vanligvis noe langsommere og krever mer *overhead* enn iterasjon
- Rekursjon kan *alltid* erstattes med iterasjon
- Men, det finnes mange problemer som er rekursive av natur, der en iterativ løsning er mye mer komplisert
- Bruk rekursjon hvis:
  - Det gir en enklere og/eller mer elegant løsning, og:
  - Den rekursive løsningen ikke er vesentlig mer ineffektiv enn den iterative

Eksempel: Søk i en array

- Søking i en usortert array kan gjøres “rett frem” med en metode som bruker en enkel for-løkke
- Alternativt kan vi erstatte forløkken med et rekursivt kall til slutt i koden – “halerekursjon”
- Java-kode med testprogram: `searchArray.java`
- Hvilken variant er mest effektiv?

Øvelse: Rekursiv beregning av kvadrater

- Definisjon av kvadrat-tallene K_n:
  - Antall ruter i et kvadrat med *n* ruter langs hver sidekant:

    $K_n = n \cdot n = n^2$, $n = 1, 2, 3, 4, ...$

    $K_1 = 1, K_2 = 4, K_3 = 9, K_4 = 16, ...$

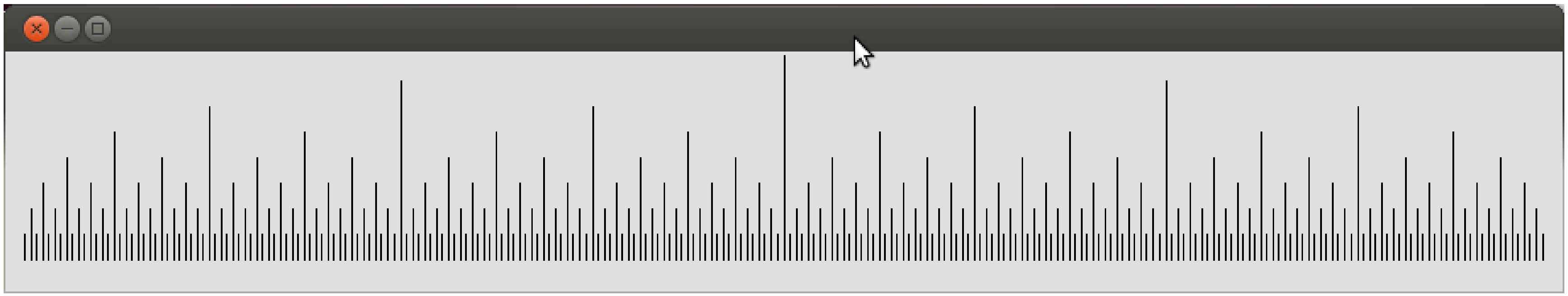
- Rekursiv definisjon:

  $K_1 = 1$

  $K_n = 2 \cdot n + K_{n-1} - 1, \quad i > 1$

- Oppgave:
  - Skriv en funksjon som beregner K_n rekursivt
  - Hva er arbeidsmengden uttrykt med *O*-notasjon?

Eksempel: En rekursiv linjal

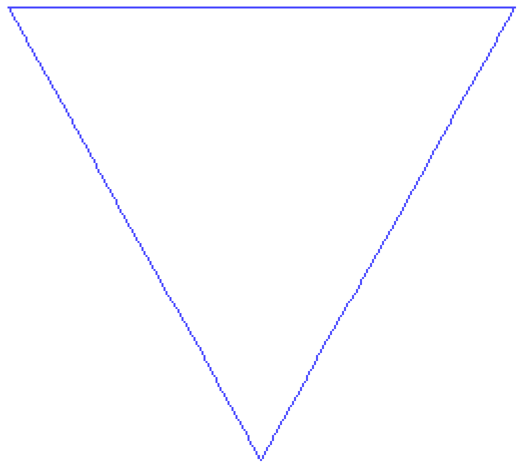


- Skal tegne en "linjal" med *n* nivåer (*n* = 8 ovenfor) og lengde *L*
- Algoritme:
  - Hvis *n* = 0, ingenting å gjøre
  - Hvis *n* > 0
    - Tegn en vertikal strek med høyde *h(n)* i posisjon *L*/2
    - Tegn venstre og høyre halvdel rekursivt, med *n* – 1 nivåer, lengde *L*/2, sentrert i hhv. *L*/4 og 3*L*/4
- Java-kode * : `ruler.java`

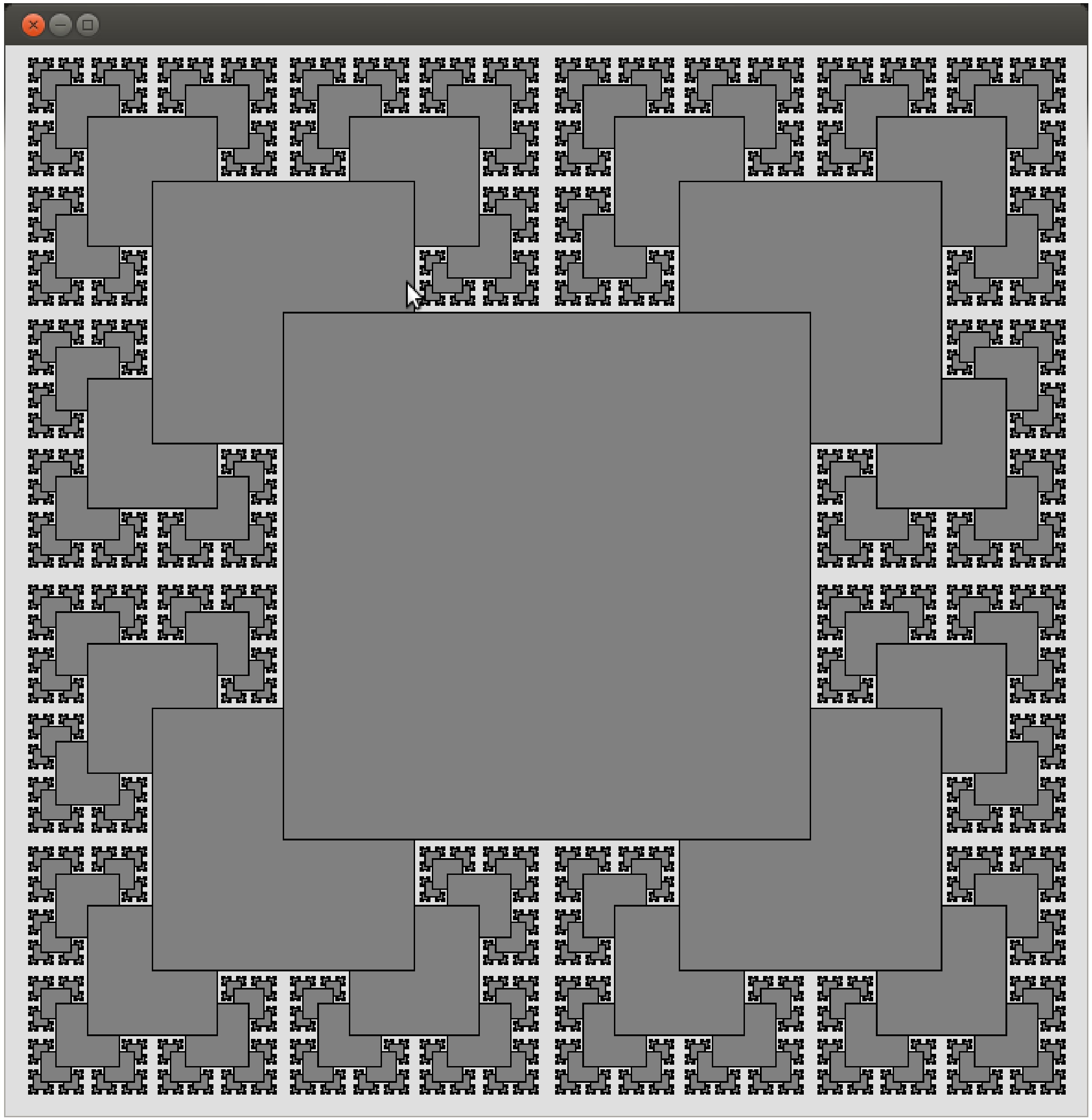
* Grafikk-delen av koden er uvesentlig

Fraktaler

- Selv-repeterende objekter som er likeformet uansett hvor mye vi “zoomer inn”
- Lages oftest med enkle, repeterende operasjoner
- Kan gi opphav til fascinerende mønstre med “uendelig dybde”
- Mest “berømte” fraktal: Mandelbrotmengden



En enkel fraktal



Algoritme for å tegne fraktalen

- Hele fraktalen har senter i (0,0)

- Rekursiv algoritme for å tegne fraktal med senter i (x, y), der største kvadrat i midten har sidekant med lengde L :

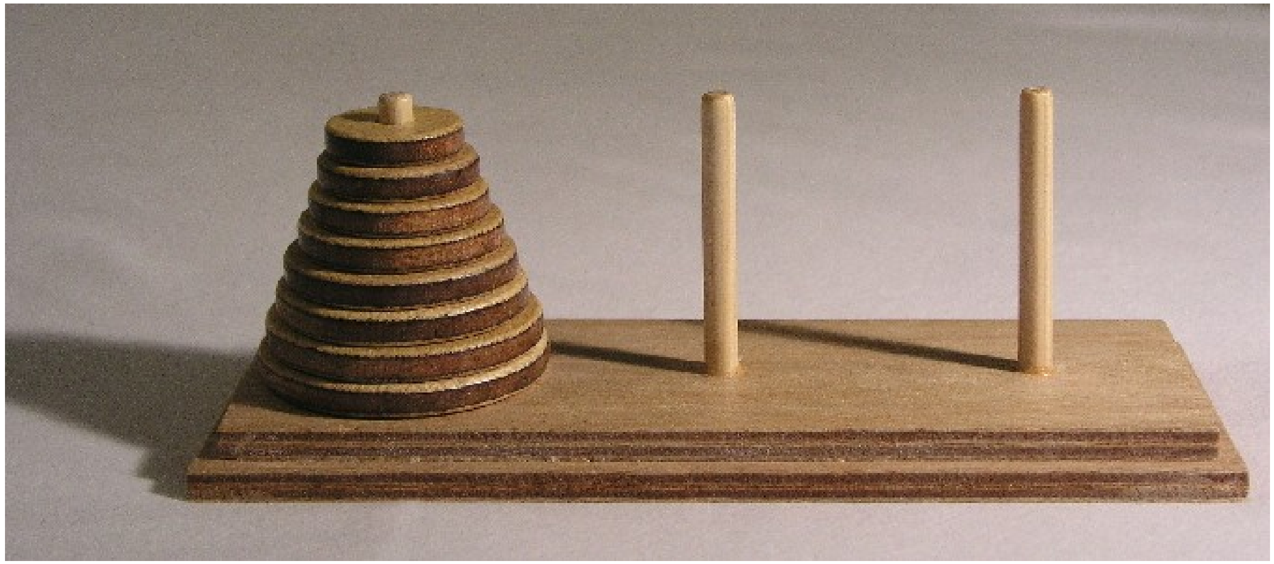
  1. Hvis $L < 1$, ferdig

  2. Tegn rekursivt de fire delene av fraktalen med sentre i $(x - L/2, y + L/2)$, $(x + L/2, y + L/2)$, $(x + L/2, y - L/2)$ og $(x - L/2, y - L/2)$, der største kvadrat har sidekant lik $L/2$

  3. Tegn kvadratet med senter i (x, y) og sidekant lik L

- Java-kode: `fractalStar.java`

En klassiker: Hanois tårn



- *n* ringer, alle av ulik størrelse. 3 pinner.
- Mål: Flytt alle ringer fra venstre til høyre pinne
- Regler:
  - Kun én ring kan flyttes av gangen, til en annen pinne
  - Ingen ring kan legges oppå en ring som er *mindre*

Løsning av Hanois tårn for $n = 4$

For animasjon av løsninger med vilkårlig n, se f.eks **emacs**
(`Ctrl-u n Esc-x hanoi`)

Rekursiv løsning av Hanois tårn

- Tre pinner, A, B og C, *n* ringer ligger på pinne A:

  Hvis *n* = 1

  Flytt ringen fra A til C

  ellers

  Flytt de *n* – 1 øverste ringene rekursivt fra A til B

  Flytt største ring fra A til C

  Flytt *n* – 1 ringer rekursivt fra B til C

- Java-kode: `hanoi.java`

Antall flytt for å løse Hanois tårn

- Med n ringer må ringene flyttes $2^n - 1$ ganger!
- Induksjonsbevis:

1 ring: $2^1 - 1 = 2 – 1 = 1$ flytt (A-C)

2 ringer: $2^2 - 1 = 4 – 1 = 3$ flytt (A-B, A-C, B-C)

n ringer: Flytte $n – 1$ ringer fra A til B: $2^{n-1} – 1$

Flytte største ring fra A til C: 1

Flytte $n – 1$ ringer fra B til C: $2^{n-1} – 1$

Totalt: $1 + 2 \cdot (2^{n-1} – 1) = 1 + 2^n – 2 = \mathbf{2^n – 1}$
