

Kapittel 12: Rekursjon

Redigert av:

Khalid Azim Mughal (khalid@ii.uib.no)

Kilde:

Java som første programmeringsspråk (3. utgave)

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen

Cappelen Akademisk Forlag, 2006.

ISBN: 82-02-24554-0

<http://www.ii.uib.no/~khalid/jfps3u/>

(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)

Emneoversikt

- Rekursive definisjoner
- Rekursjon som problemløsingsteknikk
- Utforming av rekursjon
- Nøsting av metodekall
- Rekursive algoritmer
- Metodeaktiviseringer
- Rekursjonsdybde
- Evig rekursjon (rekursjonsbrønn)
- Forskjellige typer rekursjon:
 - Direkte rekursjon
 - Indirekte rekursjon
 - Enkel rekursjon
 - Hale-rekursjon
- Iterasjon kontra rekursjon

Eksempel:

- Fakultet-funksjon
- Binært søk
- Tårn i Hanoi
- Quicksort
- Fibonacci

Rekursjon

To iterate is human; to recurse, divine.

L. Peter Deutsch, Robert Heller

- Rekursjon oppstår når en operasjon benytter seg selv som en del av sin egen utførelse.
- Kan brukes på problemer som kan deles opp i mindre utgaver av seg selv.
- Problem: Finn ut hva fotokromi betyr, ved hjelp av ordbok:
fotokromi: metode til fremstilling av fargetrykk i litografi.
 - Delproblem: Finn ut hva *litografi* betyr, ved hjelp av ordbok:
litografi: grafisk trykkemetode hvorved en tegning eller skrift blir overført på en spesiell slags stein for derpå å trykkes på papir.
 - Del-delproblem: Finn ut hva *grafisk* betyr, ved hjelp av ordbok ...
- Hver deloperasjon må fullføres før operasjonen over kan fullføres.
- For at rekursiv operasjon skal kunne fullføres, må noen deloperasjoner kunne fullføres uten behov for flere deloperasjoner.

Rekursive definisjoner

- *Rekursiv definisjon*: en definisjon som er definert ved hjelp av seg selv.
- Sett i ordliste: *Rekursjon*: se rekursjon.
- Eks: Definisjon på en setning i Java er en rekursiv definisjon: en **for**-løkke kan ha en **for**-løkke i løkke-kroppen, og denne indre løkken kan ha en **for**-løkke i sin løkke-kropp, osv.

```
for (...; ...; ...) {  
    for (...; ...; ...) {  
        for (...; ...; ...) {  
            ...  
        }  
    }  
}
```

- Rekursjon er et velkjent begrep fra matematikken.
 - Faktorial-funksjon kan uttrykkes ved hjelp av en rekursiv definisjon:

$$0! = 1$$

$$n! = n * (n-1)!$$

$$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6$$

Rekursjon som problemløsingsteknikk

- Modellerer problemer som kan deles i mindre, enklere underoppgaver som ligner på hverandre.
 - hver underoppgave kan løses ved å anvende samme teknikk.
- Hele oppgaven blir løst ved å kombinere løsninger til underoppgavene.
 - en form for splitt og hersk.
- I Java modellerer vi dette med metoder som kaller seg selv.
 - *Rekursiv metode*: En metode som er definert med kall til seg selv i metodekroppen.
- Krever et *basistilfelle* for å stoppe rekursjon.

Utforming av rekursjon

- Logikken i metoden må føre til at et basistilfelle blir utført: sekvens av mindre og mindre oppgaver *konvergerer* til et basistilfelle.
 - Dette oppnås vanligvis ved hjelp av en valgsetning, der en betingelse stopper rekursjon.
 - Denne betingelsen er vanligvis avhengig av en parameter til metoden. Et eller flere tilfeller der returverdien beregnes *uten bruk av rekursive kall*.
 - Disse tilfellene kalles for *basistilfeller*.
- Et eller flere tilfeller der returverdien beregnes *ved rekursive kall*.
 - Rekursive kall løser enda mindre versjoner av samme oppgaven.

Fakultet funksjon

Problem: Gitt n parkeringsplasser og n biler, finn antall kombinasjoner (*permutasjoner*) for å parkere bilene.

Iterativ definisjon:

$$n! = n * (n-1) * \dots * 2 * 1$$

Rekursiv definisjon:

$$n! = n * (n-1)!$$

$$0! = 1$$

- Rekursive metodekall definerer funksjonen fakultet på samme måte som den matematiske definisjonen.
 - `fakultet(n) = n!`
 - Basistilfelle: For $n = 0$: `fakultet(n)` gir 1
 - Rekursivt kall: For $n > 0$: `fakultet(n)` gir $(n * fakultet(n-1))$

Eksempel: Iterativ utregning av fakultet

```
// IterativFakultet.java -- Iterativ utregning av fakultet
public class IterativFakultet {
    public static void main(String[] args) {
        System.out.println("5! = " + fakultet(5));
    }

    static int fakultet(int n) {
        int verdi = 1;

        for (int i=2; i<=n; i++)
            verdi *= i;

        return verdi;
    }
}
```

Utskrift ved kjøring:

5! = 120

Eksempel: Rekursiv utregning av fakultet

```
// RekursivFakultet.java -- Rekursiv utregning av fakultet
public class RekursivFakultet {
    public static void main(String[] args) {
        System.out.print("Skriv heltall for utregning av fakultet: ");
        Scanner tastatur = new Scanner(System.in);
        int n = tastatur.nextInt();
        System.out.println(n + "!" = " + fakultet(n));
    }

    static int fakultet(int n) {
        if (n == 0)
            return 1; // 0! = 1 (basistilfelle)

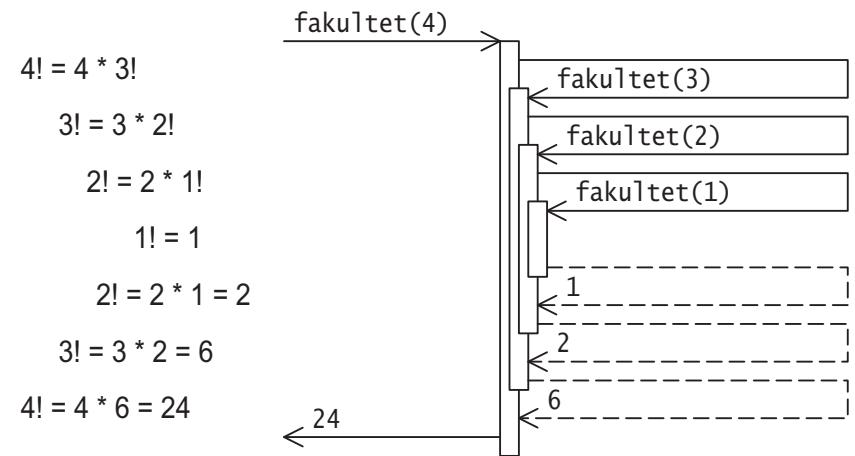
        return n * fakultet(n-1); // n! = n * (n-1)!
    }
}
```

Kjøring av programmet:

Skriv heltall for utregning av fakultet:

12

12! = 479001600



Nøsting av metodekall

- *Rekursive metoder:*
 - Objekter kan sende (forskjellige) meldinger til seg selv, men må nå sende samme melding til seg selv.
 - Metoden må nå også håndtere mindre versjoner av den opprinnelige oppgaven.
- I en rekursiv metode *nøstes* metodekallene til basistilfellet nås.
- Når basistilfellet returnerer, så begynner alle metodekallene å *utnøstes*.
- Hver nøstede utføring av metoden tar vare på sine egne lokale variabler.
 - Disse variablene blir husket når programflyten returnerer fra et rekursivt metodekall.
 - Tilordninger som gjøres på lokale variabler i nøstede metodekall, vil ikke direkte påvirke lokale variabler i ytre metodekall.

Rekursive algoritmer

- Rekursiv algoritme:
 - Har to primære arbeidsmåter: *generelle tilfeller* og *basistilfeller*
 - Implementeres som en rekursiv metode.
- Oppgavens omfang:
 - angis av parameterene til den rekursive metoden.
 - avtar med hvert rekursiv kall.
- *Generelle tilfeller:*
 - Gjør oppgaver mindre ved å dele den opp.
 - Benytter samme algoritme for å løse de mindre oppgavene.
 - Løsningen blir funnet ved hjelp av delløsningene.
- *Basistilfeller:*
 - Gir direkte løsninger for trivielle oppgaver.
 - Ingen nye rekursive kall.
 - Nødvendig for at de rekursive kallene skal slutte.
- Oppgavene *konvergerer* mot basistilfellet.

Metodeaktivisering og rekursjonsdybde

- Konsekvenser av rekursjon:
 - Mange aktiviseringer av en metode kan forekomme samtidig.
 - Hver aktivisering har egne instanser av lokale variabler (inkl. parametere).
 - *Rekursjonsdybde*: antall nøstede metodekall som gjøres før basistilfellet nås.
- *Rekursjonsbrønn*: rekursjon som aldri stopper.
 - Skjer hvis en rekursiv metode ikke konvergerer mot et basistilfelle.
 - Er en logisk programmeringsfeil.
 - Programmer med slike brønner vil som oftest bryte sammen under kjøring på grunn av den uendelige nøstingen.

Huskeregeler for bruk av rekursjon

- Kontroller at *rekursjonsbrønn* ikke forekommer.
 - For hvert rekursive kall må metoden komme et skritt nærmere basistilfellet.
 - Bunnen av rekursjonskallene er nådd når argumentene beskriver et basistilfelle.
- Et rekursivt kall må aldri resultere i at metoden blir kalt på nytt med samme argumenter.
 - Kontroller at hvert *basistilfelle* utfører riktig handling.
 - Kontroller at *for hvert tilfelle som bruker rekursjon, dersom alle rekursive kall utfører sine handlinger riktig, så utføres hele tilfellet riktig.*

Forskjellige typer rekursjon

- Direkte rekursjon:
 - Definisjon av en metode `foo()` inneholder (et eller flere) kall til metode `foo()`.
 - Eksempel: alle eksempler gitt i notatet.
- Indirekte rekursjon:
 - To eller flere metoder samarbeider rekursivt istedenfor bare én.
 - Metoder kan være implementert i samme eller i forskjellige objekter.
 - F.eks. definisjon av en metode `foo()` inneholder (et eller flere) kall til metode `bar()` (i samme eller forskjellige objekter) som i sin tur inneholder kall til metode `foo()`.
- Enkel rekursjon (*simple recursion*):
 - Kall til en metode `foo()` fører til ett rekursivt kall av metode `foo()`, mao. ingen *oppspalting* av rekursjon – hver underoppgave kaller kun én annen underoppgave.
 - Eksempel på enkel rekursjon: Faktorial-funksjon, binært søking.
 - Eksempel på *oppspalting* av rekursjon: Tårn i Hanoi, fibonacci tall.
- Halerekursjon (*tail recursion*):
 - Definisjon av en metode er slik at alle rekursivee kall forekommer som den siste setningen som utføres før retur fra metoden.
 - Eksempel på halerekursjon: binært søking.

Sammenligning av rekursjon og iterasjon

- Rekursjon er vanligvis mer kostbar pga. flere metodekall.
- Rekursjon er mer konsis og elegant for oppgaver som naturlig egner seg til denne teknikken.
- *Enkel rekursjon og iterasjon* er beregningsmessig ekvivalente (jfr. Faktorial-eksempel).

	Iterasjon	Rekursjon
<i>Basert på en kontroll-struktur:</i>	<ul style="list-style-type: none">• bruker en løkke-struktur (<code>for</code>, <code>while</code>, <code>do-while</code>)	<ul style="list-style-type: none">• bruker en valg-struktur (<code>if</code>, <code>if-else</code>, <code>switch</code>)
<i>Bruker repetisjon:</i>	<ul style="list-style-type: none">• eksplisitt bruk av en løkke-struktur for repetisjon	<ul style="list-style-type: none">• gjentatte metodekall for å oppnå repetisjon
<i>Stoppkriteriet:</i>	<ul style="list-style-type: none">• stopp når løkkebetingelsen blir usann• kan føre til <i>evig løkke</i>	<ul style="list-style-type: none">• stopp når et basistilfelle blir utført• kan føre til <i>rekursjonsbrønn</i>
<i>Konvergens mot terminering:</i>	<ul style="list-style-type: none">• teller-basert repetisjon konvergerer mot sluttverdi som terminerer løkken	<ul style="list-style-type: none">• stadig enklere versjoner av underoppgaver konvergerer mot et basistilfelle som terminerer rekursjon

Binært søk

- Finner elementer i sorterte tabeller.
- Deler søkerommet i to for hvert skritt.
- Eliminerer halve søkerommet for hvert skritt.
- Kan løses både iterativt og rekursivt:

Rekursiv binærsøk algoritme:

Operasjon binærsøk(målelement, søkerom):

Hvis søkerommet er tomt:

Returner målet-finnes-ikke.

Finn midterste-element i søkerommet.

Hvis midterste-element == målelement:

Returner posisjonen-til-målelementet.

Hvis målelement < midterste-element:

Returner binærsøk(målelement, søkerom-før-midterste-element).

Ellers:

Returner binærsøk(målelement, søkerom-etter-midterste-element).

```
// FinnRekursivt.java -- Rekursivt binærsøk etter primtall
public class FinnRekursivt {
    // Konstant som signaliserer at et tall ikke finnes i tabellen.
    static final int FINNES_IKKE = -1;

    public static void main(String[] args) {
        // Tabell med primtallene mellom 1 og 100 i sortert rekkefølge:
        int[] primtall = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
                          43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
        System.out.print("Skriv heltall mellom 1 og 100: ");
        Scanner tastatur = new Scanner(System.in);
        int tall = tastatur.nextInt();
        if ((tall<1) || (tall>100)) {
            System.out.println("Tallet var ikke mellom 1 og 100");
            return;
        }

        int primtallIndex = finn(primtall, tall);
        if (primtallIndex == FINNES_IKKE) {
            System.out.println(tall + " er ikke et primtall");
        } else {
            System.out.println(tall + " er primtall nummer " +
                               (primtallIndex + 1));
        }
    }
}
```

```

static int finn(int[] sortertTabell, int nøkkel) {
    return binærsoek(sortertTabell, nøkkel, 0, sortertTabell.length - 1);
}

static int binærsoek(int[] sortertTabell, int mål, int nedre, int øvre) {
    if (nedre > øvre) {
        return FINNES_IKKE; // basistilfelle, tom søkerom.
    }

    int midten = (nedre + øvre)/2;
    int midtersteTall = sortertTabell[midten];

    if (mål == midtersteTall) {
        return midten; // Enda et basistilfelle. Målet ble funnet.
    }

    // Generelt tilfelle: søk i enten nedre eller øvre tabellsegment.
    if (mål < midtersteTall) {
        return binærsoek(sortertTabell, mål, nedre, midten-1);
    } else {
        return binærsoek(sortertTabell, mål, midten+1, øvre);
    }
}

```

Java implementasjon av binærsøkalgoritme

- Finner heltall i en sortert tabell.
- Tabellen inneholder alle primtallene fra 1 til 100.
- For angitt heltall, slår opp i tabellen og rapporterer:
 - om tallet er et primtall, med andre ord, finnes i tabellen.
 - hvor langt ut i primtallrekken tallet er.

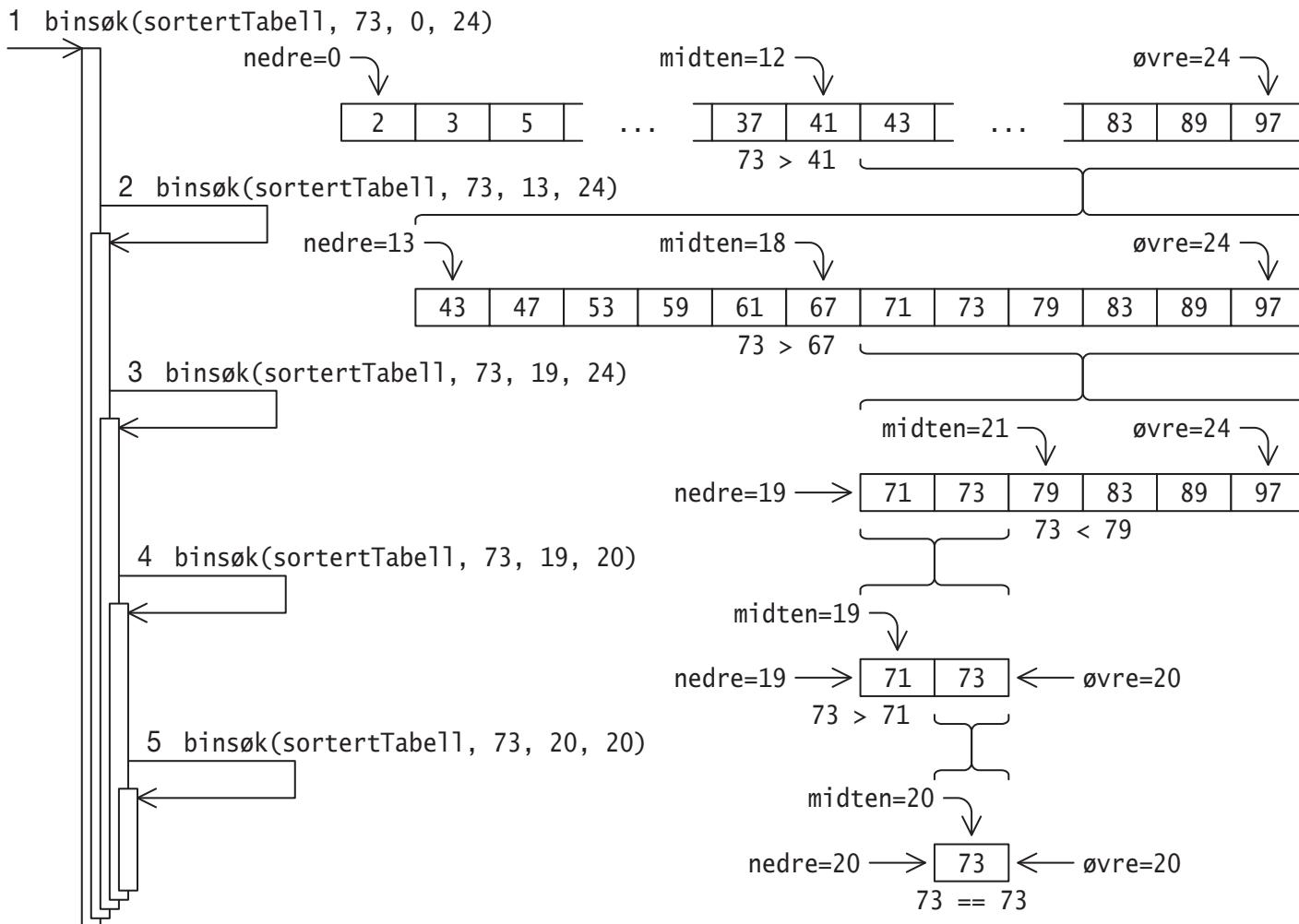
Kjøring av programmet:

```
Skriv heltall mellom 1 og 100: 73  
73 er primtall nummer 21
```

- Parameterene spesifiserer nedre og øvre grensene til søkerommet.
- Søkerommet halveres for hvert rekursivt kall.
- Binærsøkalgoritmen bruker halerekursjon.

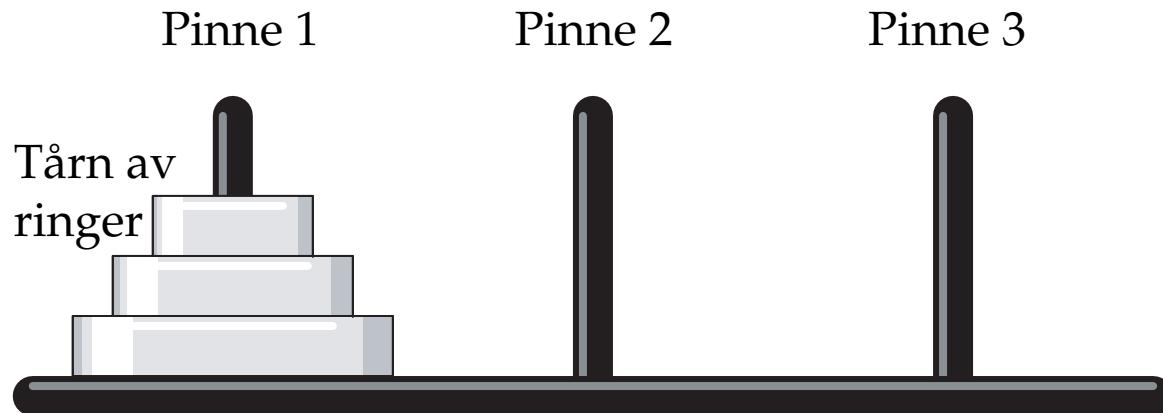
Søke eksempel

Søking etter tallet 73 ved kjøring:



Tårn i Hanoi

- Spill oppfunnet av den franske matematiker Edouard Lucas i 1883.
 - Tre pinner
 - Ringer av forskjellige størrelser



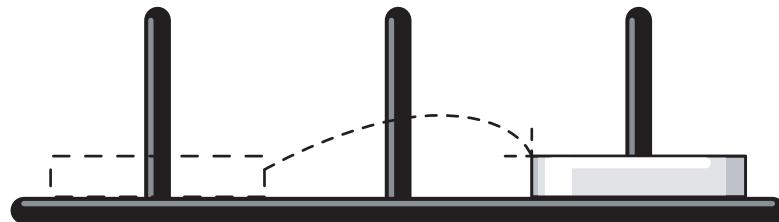
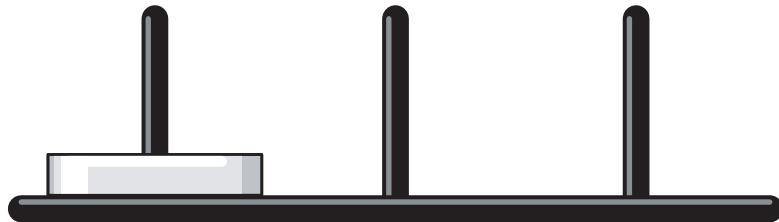
- Mål: flytt tårn med n ringer av forskjellige størrelser fra pinne 1 til pinne 3.
- Regel 1: Ringer kan ikke plasseres oppå mindre ringer.
- Regel 2: Kun én ring kan flyttes om gangen.

Tårn i Hanoi (en ring)

- Spillet kan spilles med et hvilket som helst antall ringer.

Hanoi-løsning for én ring:

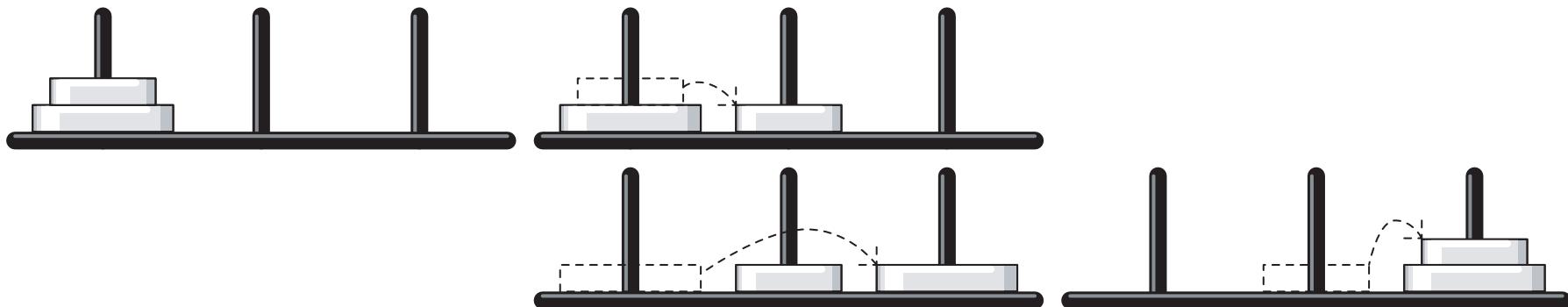
- Løsning er triviell.
 - (*tenk basistilfelle*)
- Ring kan flyttes direkte til målpinnen.



Tårn i Hanoi (to ringer)

Hanoi-løsning for to ringer:

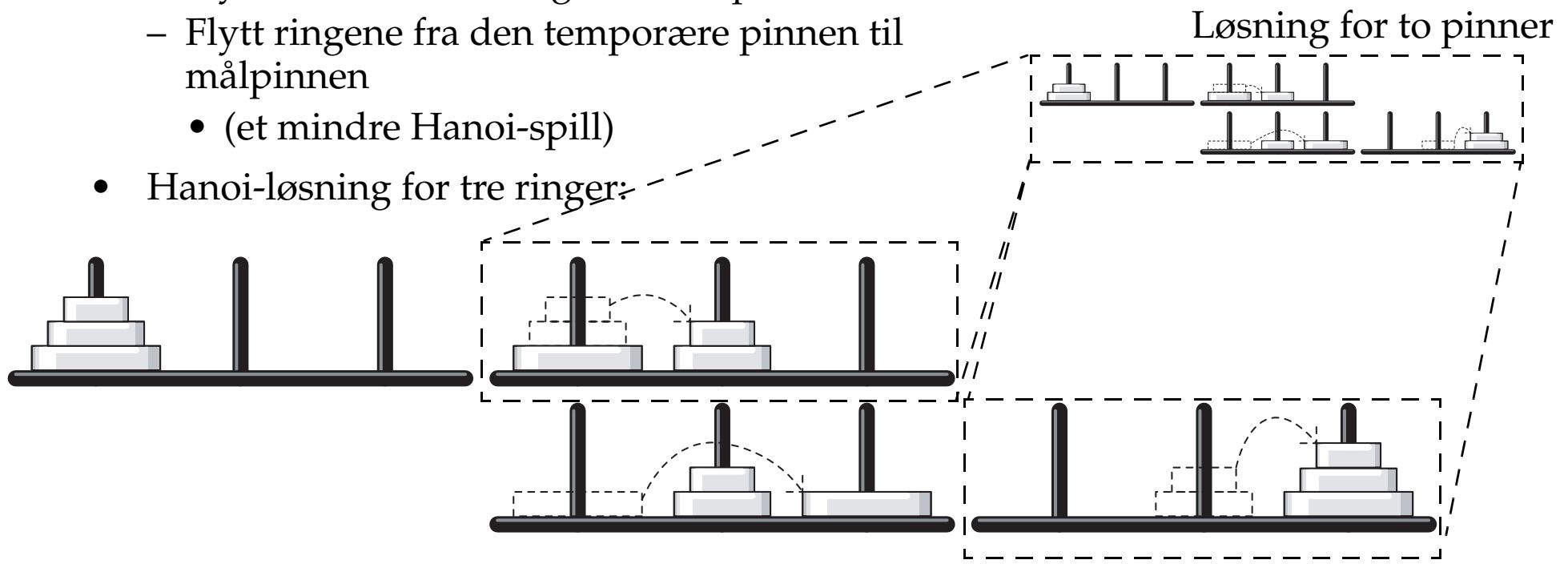
- Nederste ring må havne nederst på målpinnen. (regel 1)
- Ringen som ligger oppå må flyttes vekk først.



- Flytt den lille ringen til en midlertidig pinne.
- Flytt den store ringen til målpinnen.
- Flytt den lille ringen fra den midlertidige pinnen oppå den store ringen på målpinnen.

Tårn i Hanoi: rekursiv algoritme

- Løsninger følger et visst mønster.
 - Blir tydelig når antall ringer øker.
 - Kan beskrives som en rekursiv algoritme.
- *Flytte et tårn fra en pinne til en annen:*
 - Flytt vekk alle ringene bortsett fra den nederste ringen til en temporær pinne.
 - (et mindre Hanoi-spill)
 - Flytt den nederste ringen til målpinnen.
 - Flytt ringene fra den temporære pinnen til målpinnen
 - (et mindre Hanoi-spill)
- Hanoi-løsning for tre ringer:



```

public class Hanoi {
    public static void main(String[] args) {
        int antallRinger = 3;
        System.out.println("Flytter " + antallRinger + " ringer:");
        /* Flytt alle ringene fra pinne 1 til pinne 3 ved å bruke 2 som
           midlertidig lagringsplass. */
        flyttRinger(antallRinger, 1, 3, 2);
    }
    static void flyttRinger(int antall, int fra, int til, int tmp) {
        if (antall < 1)
            return; // Basistilfelle: Ingen ringer å flytte.

        /* Først flyttes de antall-1 nederste ringene over til "tmp"-pinnen.
           Bruk "til"-pinnen som midlertidig lagringsplass: */
        flyttRinger(antall-1, fra, tmp, til);

        // Ringene over er fjernet, og den nederste ringen kan flyttes:
        System.out.println(fra + " ---> " + til);

        /* De antall-1 ringene på "tmp"-pinnen kan nå flyttes til
           "til"-pinnen. Bruk "fra"-pinnen som midlertidig lagringsplass: */
        flyttRinger(antall-1, tmp, til, fra);
    }
}

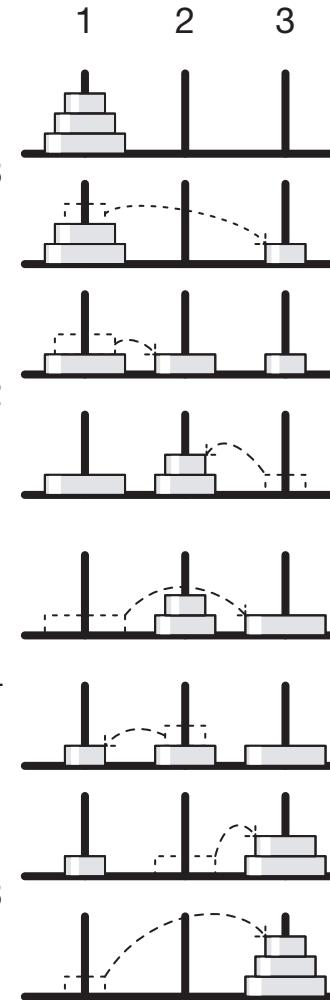
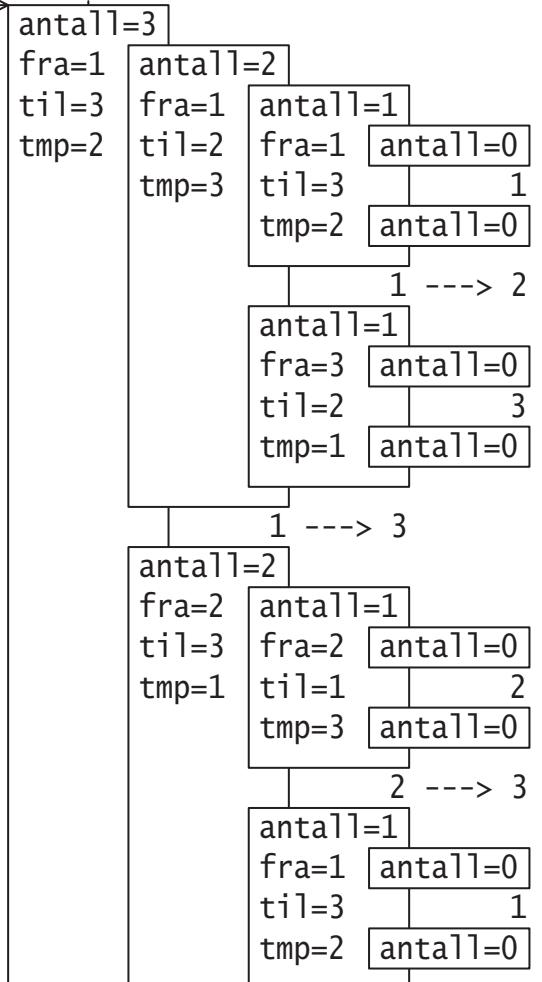
```

Flytter 3
 ringer:
 1 ---> 3
 1 ---> 2
 3 ---> 2
 1 ---> 3
 2 ---> 1
 2 ---> 3
 1 ---> 3

- To rekursive kall per rekursjonsnivå.
- Metoden `flyttRinger()` har følgende lokale variabler:
 - `antall`: antall ringer i tårnet som skal flyttes i denne deloppgaven
 - `fra`: tallet som identifiserer pinnen tårnet skal flyttes vekk fra
 - `til`: tallet som identifiserer pinnen tårnet skal flyttes til
 - `tmp`: tallet som identifiserer pinnen som kan brukes til midlertidig lagring av ringer
- Hver utføring av metoden har sin egen utgave av disse variablene.
 - Tas vare på under rekursjonen.
 - Når programflyten vender tilbake fra et rekursivt metodekall, vil variablene fremdeles være slik de var før det rekursive metodekallet.
- Under metodekall blir lokale variabler lagret i en implisitt stabel. (*programstabel*, eng. *program stack*)
- Hver metodeutføring har sin egen *stabelramme* på programstabelen:
 - Inneholder de lokale variablene.
 - Opprettes på programstabelen når en metode blir kalt.
 - Fjernes når metoden returnerer.

Programstabel som vokser og krymper

flyttRinger(3, 1, 3, 2)



Antall rekursive kall: $2^n - 1$, der n er antall ringer.

Rekursiv sorteringsalgoritme: Quicksort

- En effektiv rekursiv sorteringsalgoritme for tabeller.
- Denne *splitt-og-hersk* algoritmen deler sorteringsoppgaven i to og bruker et rekursivt kall for å løse hver del av oppgaven.
- Mesteparten av arbeidet i Quicksort-algoritmen består i å dele opp oppgaven.
 - En operasjon arbeider på et tildelt tabellsegment som skal sorteres.
 - Tabellsegmentet deles opp og tilpasses slik at oppgaven kan løses ved å sortere hver av delene separat.
 - Et segment deles slik at det største elementet i den første delen er mindre enn alle elementene i den andre delen.

Algoritme: Quicksort

Operasjon quicksort(segment):

Hvis segment er tomt:

Returner.

skillelement = første element i segment

Utfør partisjonering av segment i to deler: del1 og del2

quicksort(del1)

quicksort(del2)

Operasjon partisjonering(segment):

skillelement = et valgt element i segmentet

Bytt rundt på elementene i segmentet slik at:

Alle elementene mindre enn skillelement kommer før skillelement

Alle elementene større enn skillelement kommer etter skillelement

Quicksort

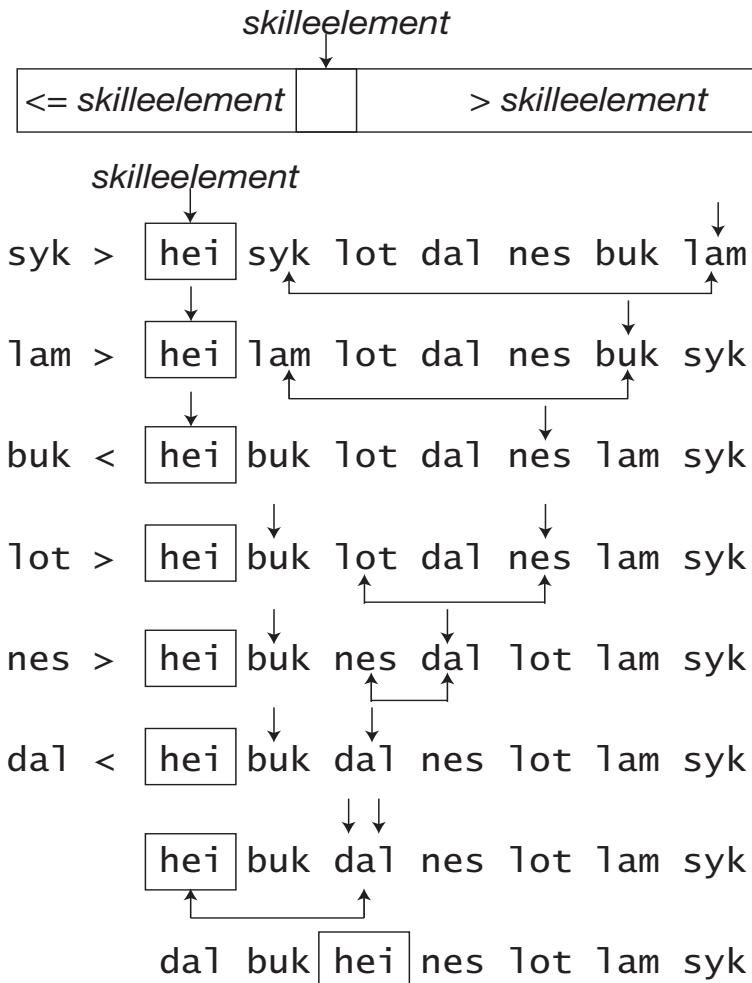
```
static <T extends Comparable<T>> void quicksort(T[] elementer,
                                                int første, int siste, int nivå) {
    // Basistilfelle: 0 eller ett element, ingen sortering nødvendig.
    if (første >= siste) {
        System.out.println(blanker(nivå) + nivå + "> basistilfelle");
        return;
    }

    // Generelt tilfelle: Partisjoner segmentet.
    skrivSegment(blanker(nivå) + nivå + "> segment før deling: ",
                 elementer, første, siste);
    int skillePosisjon = partisjoner(elementer, første, siste, nivå);
    System.out.println(blanker(nivå) + nivå + "> skillelementet: " +
                       elementer[skillePosisjon]);
    skrivSegment(blanker(nivå) + nivå + "> segment etter deling: ",
                 elementer, første, siste);

    // Sorter elementene som var mindre eller lik skillelementet:
    skrivSegment(blanker(nivå) + nivå + "> sorter: ",
                 elementer, første, skillePosisjon-1);
    quicksort(elementer, første, skillePosisjon-1, nivå+1);
    skrivSegment(blanker(nivå) + nivå + "> sortert: ",
                 elementer, første, skillePosisjon-1);
```

```
// Sorter elementene som var større enn skilleelementet:  
skrivSegment(blanker(nivå) + nivå + "> sorter: ",  
    elementer, skillePosisjon+1, siste);  
quicksort(elementer, skillePosisjon+1, siste, nivå+1);  
skrivSegment(blanker(nivå) + nivå + "> sortert: ",  
    elementer, skillePosisjon+1, siste);  
}
```

Partisjonering



```

static <T extends Comparable<T>> int partisjoner(T[] elementer,
                                              int første, int siste, int nivå) {
    /* Generelt tilfelle: Del opp elementene i dem som er mindre
     * eller lik skilleelementet, og dem som er større enn
     * skilleelementet. */
    T skilleElement = elementer[første];
    int sisteLite = første;
    int sisteUdelte = siste;
    // Utfør så lenge det er flere elementer som ikke har blitt delt.
    while (sisteLite < sisteUdelte) {
        // skrivSegment(blanker(nivå) + nivå + "> deling: ",
        //               elementer, første, siste);
        int førsteUdelte = sisteLite + 1;
        T testElement = elementer[førsteUdelte];
        boolean større = testElement.compareTo(skilleElement) > 0;
        if (større) {
            // Bytt om for å plassere elementet etter de udelte.
            elementer[førsteUdelte] = elementer[sisteUdelte];
            elementer[sisteUdelte] = testElement;
            --sisteUdelte;
        } else {
            // Elementet er på riktig side av de udelte allerede.
            ++sisteLite;
        }
    }
}

```

```
/* Ingen udelte igjen. Plassér skilleelementet mellom de store og
   små elementene ved å bytte plass med det siste av de små. */
int skillePosisjon = sistelite;
elementer[første] = elementer[skillePosisjon];
elementer[skillePosisjon] = skilleElement;
// skrivSegment(blanket(nivå) + nivå + "> deling: ",
//               elementer, første, sist);
return skillePosisjon;
}
```

Kjøring av programmet:

```
Tabell usortert: hei syk lot dal nes buk lam
1> segment før deling: hei syk lot dal nes buk lam
1> skillelementet: hei
1> segment etter deling: dal buk hei nes lot lam syk
1> sorter: dal buk
    2> segment før deling: dal buk
    2> skillelementet: dal
    2> segment etter deling: buk dal
    2> sorter: buk
        3> basistilfelle
    2> sortert: buk
    2> sorter:
        3> basistilfelle
    2> sortert:
1> sortert: buk dal
1> sorter: nes lot lam syk
    2> segment før deling: nes lot lam syk
    2> skillelementet: nes
    2> segment etter deling: lam lot nes syk
    2> sorter: lam lot
        3> segment før deling: lam lot
        3> skillelementet: lam
        3> segment etter deling: lam lot
        3> sorter:
            4> basistilfelle
        3> sortert:
        3> sorter: lot
            4> basistilfelle
        3> sortert: lot
    2> sortert: lam lot
    2> sorter: syk
        3> basistilfelle
    2> sortert: syk
1> sortert: lam lot nes syk
Tabell sortert: buk dal hei lam lot nes syk
```

Enhetstest for Quicksort

```
import junit.framework.TestCase;

public class TestSort extends TestCase {

    String[] array;

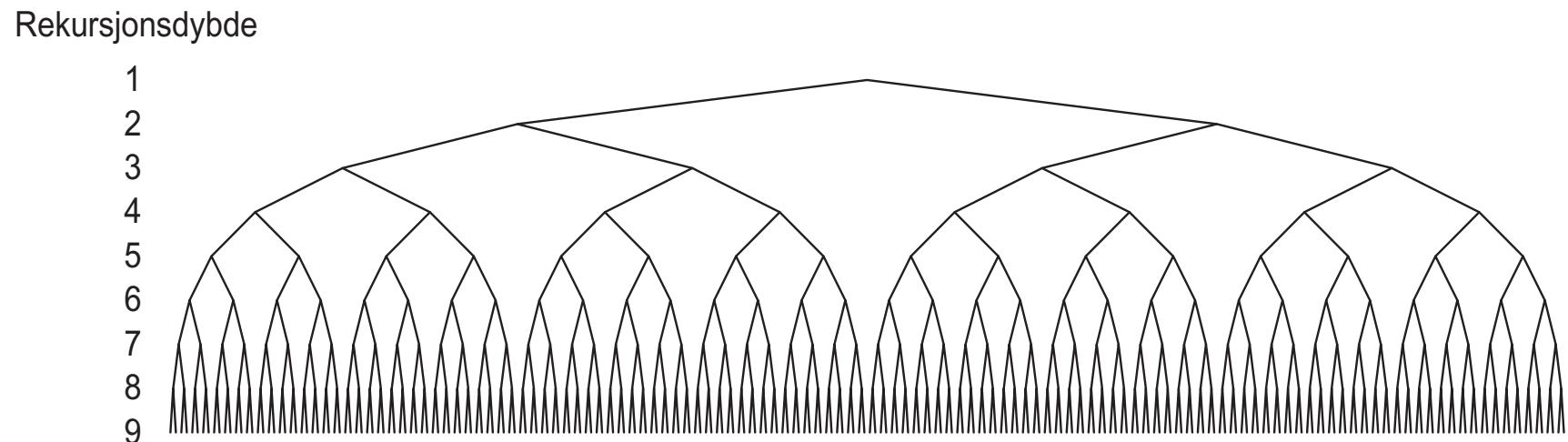
    public TestSort(String testMethodName) {
        super(testMethodName);
    }

    public void setUp() {
        array = new String[] { "This", "is", "not", "so", "difficult" };
    }

    public void testOrder() { // array[0] <= array[] <= ... <= array[n-1]
        Quicksort.sorter(array);
        for (int i = 1; i < array.length; i++) {
            int status = array[i-1].compareTo(array[i]);
            assertTrue("Not sorted. Check index: " + i, status <= 0);
        }
    }
}
```

Eksponentiell økning i antall metodekall

- Dersom oppgaven deles opp i to eller flere rekursive metodekall, er ikke rekursjonsdybden lik antall ganger metoden kalles.
 - I Quicksort-algoritmen, dobles det totale antall metodekall for hvert rekursjonsnivå.



- Resultat av mer enn ett rekursivt kall per rekursjonsnivå.
- Gjør at rekursive algoritmer ikke alltid er beste løsning.

Fibonacci-serien

- Tallrekke $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 510, 787\dots$
- *Matematisk definisjon:*

$$f_1 = 1, f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

- *Java implementasjon:*

```
static int fibonacci(int n) {  
    if (n < 3)  
        return 1;  
    return fibonacci(n-2) + fibonacci(n-1);  
}
```

- Utregning av f_n vil kreve utregning av f_{n-1} og f_{n-2} .
 - begge disse krever utregning av f_{n-3}
 - utregningen vil skje separat for både f_{n-1} og f_{n-2}
- Mengden med dobbeltarbeid øker eksponentielt, med n .

Fibonacci-tall	Antall metodekall
f_3	3
f_4	5
f_5	9
f_6	15

- Iterativ utregning er bedre.

```
static int fibonacci(int n) {  
    int nestSiste = 1;  
    int siste = 1;  
    for (int i=3; i<=n; i++) {  
        int neste = siste + nestSiste;  
  
        // Forskyv variablene:  
        nestSiste = siste;  
        siste = neste;  
    }  
  
    return siste;  
}
```

- Regner ut f_n ved å regne ut verdiene fra f_1 til f_{n-1} i sekvens.