Föreläsning 8

Innehåll

Rekursion

- Rekursiv problemlösning
- Binärsökning
- Generiska metoder

Rekursion

- Rekursivt tänkande:

- Rekursiv problemlösning:
  - Dela upp problemet i en eller flera enklare versioner av det ursprungliga problemet.
- Rekursiva metoder
  - är metoder som anropar sig själva.
- Många datastrukturer går att definiera rekursivt.
  - Ex: listor, träd

Rekursiv problemlösning

Kan liknas vid önsketänkande:

- Om jag har lösningen till en (eller flera) mindre instans(er) så kan jag konstruera en lösning för den aktuella instansen genom att ...
  - Ex: Summera de n första talen i en vektor.

    Rekursivt tänkande: Om summan av de n-1 första talen är känd, så får vi lösningen genom att summera denna summa och det sista talet.

- För att det ska fungera krävs basfall där lösningen ges explicit. Basfallen är (oftast) små instanser, t ex 0.
  - I exemplet ovan är ett naturligt basfall n=0. Då är summan 0.

Rekursiv problemlösning

Mönster för rekursiv algoritm

Rekursiv lösning för problem med storlek n:

```
Om problemet går att lösa för aktuellt n-värde:
   Lös det
annars
   Lös problemet (rekursivt) för en eller flera
   mindre värden på n
   Kombinera dessa lösningar till en lösning på
   problemet för n
```

Rekursiv beräkning av n!

Definition

$0! = 1$ (basfall)

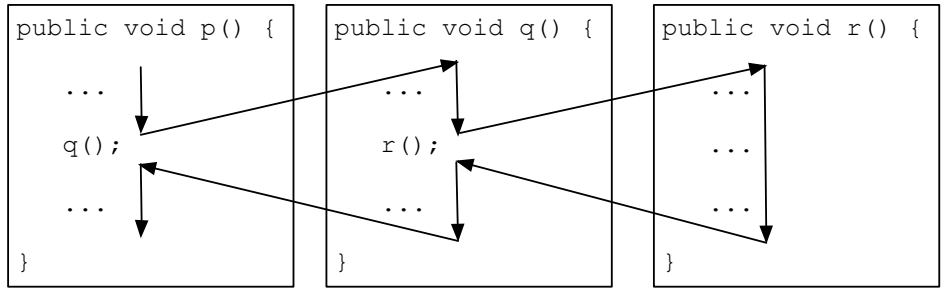
$n! = n * (n-1)!$, n heltal > 0 (rekursiva steget)

```
public static long factorial(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
```


Exekvering och metodanrop

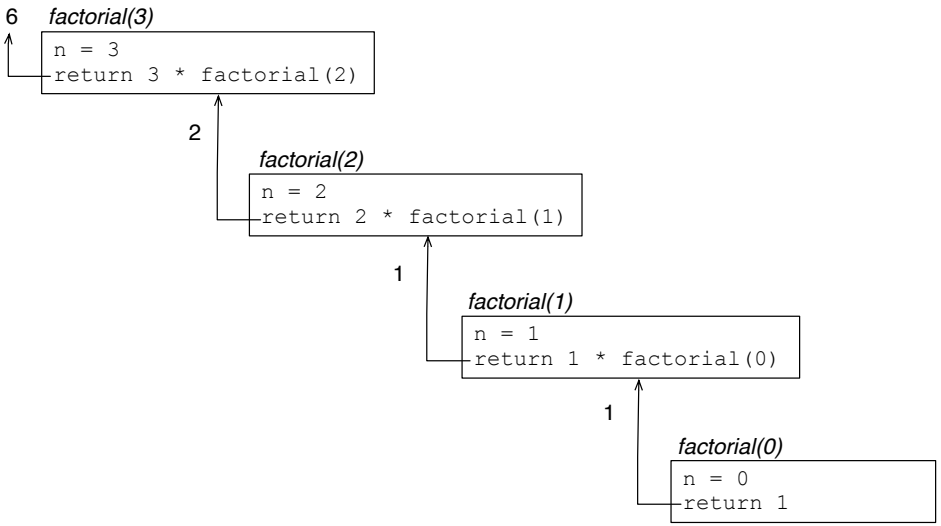
För *alla* metodanrop (rekursiva såväl som icke-rekursiva) gäller:

- Ett anrop av en metod innebär att exekveringen fortsätter med den första satsen i den anropade metoden.
- När den anropade metoden är klar återupptas exekveringen i den metod där anropet gjordes.



Exekvering av rekursiva metoder

Exempel



Exekvering och metodanrop

Vid exekvering av ett metodanrop skapar runtimesystemet en aktiveringspost.

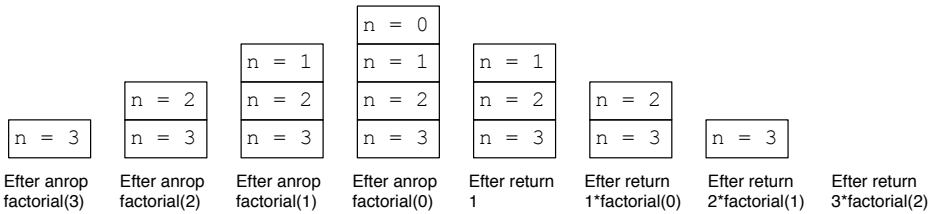
- I aktiveringsposten finns uppgifter om
  - återhoppsadress (där anropet gjordes)
  - parametrarnas värden
  - lokala variabler och deras värden
  - ...
- Aktiveringsposterna placeras i en stack.
  - Den metod som exekverar har sin post överst på stacken.
  - Aktiveringsposten tas bort när metoden exekverat klart.

Exekvering av rekursiva metoder

Rekursiva metoder anropar sig själva.

- Det kommer att på stacken finnas aktiveringsposter för alla oavslutade upplagor av metoden.

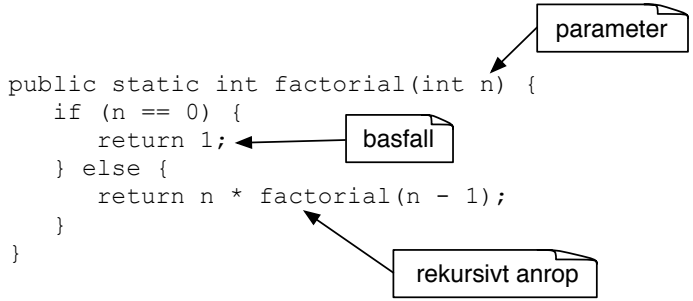
Stackens utseende under exekvering av ett anrop av `factorial(3)`:
Endast parmeterns värde visas för varje upplaga.



Rekursiv metod

En rekursiv metod måste ha:

- En eller flera **parametrar** som bestämmer problemets storlek
- Ett eller flera **basfall** som löses direkt.
- Ett eller flera **rekursiva anrop**. De rekursiva anropen måste leda till att ett basfall så småningom nås.



Övning

Skriv en rekursiv metod som beräknar x^n. n är ett positivt heltal.

Definition

$x^0 = 1$
$x^n = x * x^{n-1}$, n heltal > 0

Exempel: Skriv ut ett tal baklänges

Algoritm

Problem: Givet ett heltal n ≥ 0. Skriv ut siffrorna i omvänd ordning.
Exempel: Talet 257 ska ge utskriften 752

Basfall:

- Talet har bara en siffra. Skriv ut denna enda siffra.

Rekursiva steget:

- Skriv ut sista siffran.
- Skriv därefter ut de övriga siffrorna i omvänd ordning.

Exempel: Skriv ut ett tal baklänges

Rekursiv metod

```
public static void reverse(int n) {
   if (n < 10) {
      System.out.print(n);
   } else {
      System.out.print(n % 10);
      reverse(n / 10);
   }
}
```


Exempel: Skriv ut ett tal baklänges

Negativa tal

- Metoden klarar inte och ska inte klara negativa tal
- Det bör kontrolleras. Vi inför därför en metod som först kontrollerar parametern och därefter anropar den rekursiva metoden:

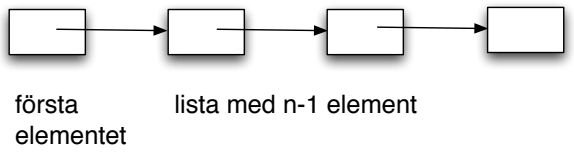
```
public static void printReverse(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("Argument < 0");
    }
    reverse(n);
    System.out.println();
}
```

- Den rekursiva metoden bör nu göras privat.

Rekursion och listor

Rekursiva datastrukturer

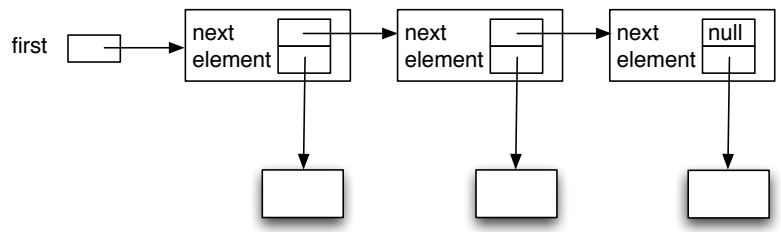
- Begreppet lista med n element kan definieras rekursivt:
  - om n == 0 är listan tom (basfall)
  - annars består den av ett första element följt av en lista med $n - 1$ element



Rekursion och listor

Klassen SingleLinkedList från tidigare föreläsning

```
public class SingleLinkedList<E> {
   private ListNode<E> first;
   ...
   private static class ListNode<E> {
      E element;
      ListNode<E> next;
      ...
   }
}
```



Diskutera

Rekursion och listor

Antag att vi vill skriva ut innehållet i den enkellänkade listan i omvänd ordning med rekursiv teknik.

Börja med att tänka ut

- basfall.
- enklare delproblem (rekursiva steget).

Skissa sedan på algoritmen i pseudokod.

Rekursion och listor

Exempel

Problem: Skriv ut innehållet i listan i omvänd ordning.

```
public void printReverse() {
   printReverse(first);
}

private void printReverse(ListNode<E> p) {
   if (p != null) {
      printReverse(p.next);
      System.out.println(p.element);
   }
}
```

Vad skulle behövas ändras i koden för att skriva ut listan i vanlig ordning?

Rekursion och listor

Kommentarer

- Basfallet i `printReverse` "syns inte". Basfallet är den tomma listan (`p == null`) då ingenting görs.
- Metoden har tidskomplexitet $O(n)$ – det görs n metodanrop och i varje upplaga görs ett arbete som tar konstant tid.
- Ett alternativ är att söka efter sista, näst sista o.s.v. Det blir krångligare kod och högre tidskomplexitet.
- I `printReverse` vi utnyttjat att vi har tillgång till den interna datastrukturen. Om man använder någon av `Collection`-klasserna i Java får man lösa det på något annat sätt. Man kan t.ex. använda en stack.

Beräkna storleken rekursivt

Ett annat listexempel

Problem: beräkna antalet element i en lista rekursivt.

```
public int size() {
   return size(first);
}

private void size(ListNode<E> p) {
   if (p == null) {
      return 0;
   } else {
      return 1 + size(p.next);
   }
}
```

Beräkna storleken rekursivt
Kommentarer

- Att beräkna antalet listelement rekursivt är inte nödvändigtvis lättare än att beräkna det iterativt. Däremot underlättar rekursion vid beräkning över andra länkade datastrukturer såsom träd, vilket vi kommer att se senare i kursen.
- Att räkna antal element i metoden `size` är dessutom olämpligt. En metod som `size` bör ha konstant tidskomplexitet:

```
public int size() {
   return nbrElements;  // attribut som uppdateras vid
}                       // insättning och borttagning
```


Binärsökning
Algoritm

Om en vektor är sorterad i växande ordning och vi söker ett element x, finns en effektiv algoritm:

Basfall:

- 0 element. Sökt element finns ej.
- Jämför x med mittelementet i vektorn. Om likhet, avbryt.

Rekursiva steget:

1. Om x är *mindre* än mittelementet, fortsätt sökningen i *vänster halva* av vektorn.
2. Om x är *större* än mittelementet, fortsätt sökningen i *höger halva* av vektorn.

Binärsökning - parametrar

- I den rekursiva lösningen måste varje upplaga av metoden känna till vilken del av vektorn den arbetar med. Det kommer därför att behövas parametrar för **"första index" och "sista index"** i den rekursiva metoden.
- Användare ska dock inte behöva anropa med två index. Det räcker att de anger vad som söks och vilken vektor det ska sökas i.
- Den **publika metoden** som användare får tillgång till:

```
/** Returns the index of x if found in the array
    a, otherwise -1. The array must be sorted.*/
public static int indexOf(int[] a, int x) {
   return binarySearch(a, x, 0, a.length - 1);
}
```

Den **rekursiva hjälpmetodens** rubrik:

```
private static int binarySearch(int[] a, int x, int first,
                                                int last);
```


Binärsökning
Privat metod

```
private static int
binarySearch(int[] a, int x, int first, int last) {
   if (first > last) {
      return -1;
   } else {
      int mid = first + ((last - first) / 2);
      if (x == a[mid]) {
         return mid;
      } else if (x < a[mid]) {
         return binarySearch(a, x, first, mid - 1);
      } else {
         return binarySearch(a, x, mid + 1, last);
      }
   }
}
```

- Linjärsökning är $O(n)$ i värsta fall.
- Binärsökningen är mycket effektivare.
  - I värsta fall finns inte x i vektorn.
  - Vi börjar med vektorstorlek n, därefter $n/2, n/4, ...n/2^k, ..., 1, 0$.
  - Det krävs $^2\log(n)$ halveringar i värsta fall.
  - I varje upplaga krävs konstant arbete.
  - Hela algoritmen blir därför $O(logn)$ i värsta fall.

- Vi vill ha en generell metod, d.v.s. en metod som kan söka efter ett element i en vektor av godtycklig typ (och inte bara heltal).
  - Lösning: Låt metoden vara generisk, dvs deklarera en typparameter (E) i metodrubriken.

    ```
    public static <E> int indexOf(E[] a, E x)
    ```

- Vi måste dock kräva att elementen är av en typ för vilka jämförelse är definierad eftersom vektorn ska vara sorterad.
  - Lösning: Kräv att den klass som ersätter E implementerar interfacet Comparable.

    ```
    public static <E extends Comparable<E>> int ...
    ```


- Den **publika metoden** som användare får tillgång till:

```
/** Returns the index of x if found in the array
    a, otherwise -1. The array must be sorted.*/
public static <E extends Comparable<E>> int
indexOf(E[] a, E x) {
   return binarySearch(a, x, 0, a.length - 1);
}
```

- Den **rekursiva hjälpmetodens** rubrik:

```
private static <E extends Comparable<E>> int
binarySearch(E[] a, E x, int first, int last);
```

- Implementering följer, men först en liten utvikning om generiska metoder.

- Man kan deklarera generiska metoder, genom att "parametrisera" metoden med en eller flera typparametrar. Exempel:

```
public class Utilities {
   ...
   /* Fyller alla platser i a med elementet x */
   public static <T> void fill(T[] a, T x) {
      for (int i = 0; i < a.length; i++) {
         a[i] = x;
      }
   }
   ...
}
```

- Typparameter (en eller flera) anges inom "<" och" >" före metodens returtyp.

Generiska metoder

- Generiska metoder kan anropas utan att man explicit anger vad typen `T` är:

```
Integer[] nbrs = new Integer[10];
Utilities.fill(nbrs, -1);

String[] a = new String[5];
Utilities.fill(a, "abc");
```

- För det första anropet fastställer kompilatorn typen `T` till `Integer` och i det andra till `String`.

Den generiska metoden `indexOf`

```
public static <E extends Comparable<E>> int
indexOf(E[] a, E x) {
   return binarySearch(a, x, 0, a.length - 1);
}
```

- Har en typparameter `E` med inskränkning:
  - `E` måste vid anrop vara en klass som implementerar interfacet `Comparable<E>`
- Har returtyp `int`:
  - Returtypen skrivs efter typparametrarna
- Har två parametrar som använder `E` i sin deklaration:
  - `E[] a` och `E x`

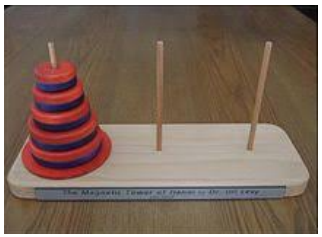
Binärsökning - generisk metod

Privat metod

```
private static <E extends Comparable<E>> int
binarySearch(E[] a, E x, int first, int last) {
   if (first > last) {
      return -1;
   } else {
      int mid = first + ((last - first) / 2);
      int compResult = x.compareTo(a[mid]);
      if (compResult == 0) {
         return mid;
      } else if (compResult < 0) {
         return binarySearch(a, x, first, mid - 1);
      } else {
         return binarySearch(a, x, mid + 1, last);
      }
   }
}
```


Hanois torn

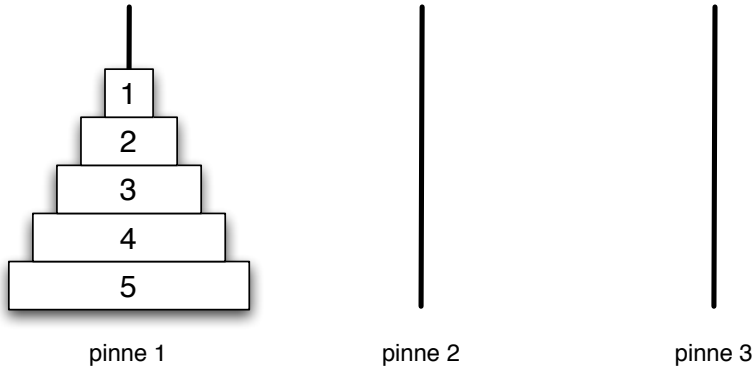
Bonusexempel



- Hanois torn är ett matematiskt pussel som går ut på att flytta ett antal skivor från en pinne till en annan enligt vissa regler.
- Minimala antal drag som behövs för att lösa pusslet för n skivor är $2^n - 1$.
- Här finns mer att läsa: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Hanois torn

- n skivor finns i avtagande storlek på en pinne (`start`).



- Flytta dem så att de kommer i samma inbördes ordning på en av de andra pinnarna (`dest`).
- Även den tredje pinnen (`temp`) får utnyttjas för mellanlagring.

Hanois torn – regler

- Bara en skiva i taget får flyttas.
- En skiva som tas från en pinne måste genast läggas på en av de andra pinnarna.
- Det får aldrig inträffa under flyttningarnas gång att en större skiva hamnar ovanför en mindre (på samma pinne).

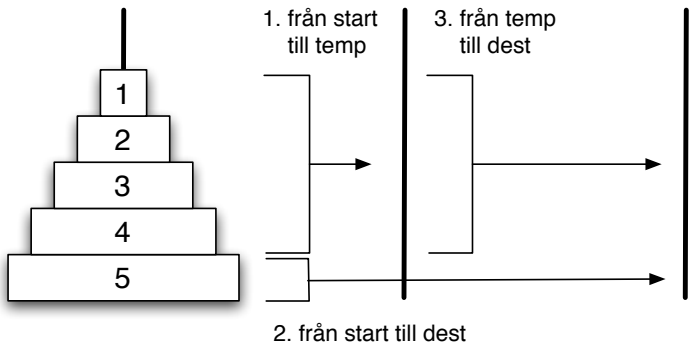
Hanois torn – rekursiv lösning

Basfall:

- Endast en skiva att flytta (n = 1). Flytta skivan från `start` till `dest`.

Rekursiva steget:

1. Flytta först de $n-1$ översta skivorna från `start` till `temp`.
2. Flytta därefter den största skivan från `start` till `dest`.
3. Till sist flyttas de $n-1$ skivorna från `temp` till `dest`.



Hanois torn – rekursiv metod

```
public void move(int n, int start, int dest, int temp) {
   if (n == 1) {
      System.out.println("Move from " + start + " to " + dest);
   } else {
      move(n - 1, start, temp, dest);
      System.out.println("Move from " + start + " to " + dest);
      move(n - 1, temp, dest, start);
   }
}
```

Samband mellan rekursion och induktion

Överkurs

Matematisk induktion används ofta för att bevisa samband som gäller för positiva heltal n. Bevisen görs i två steg:

1. **Basfall**. Visa att sambandet gäller för ett eller flera små värden på n.
2. **Induktionssteg.** Visa att om man gör antagandet att sambandet håller för alla heltal n upp till ett visst värde k (induktionsantagandet), så gäller det även för närmast större värde $k+1$.

Samband mellan rekursion och induktion – exempel

Överkurs

Visa att $1+2+3+\ldots+n = n(n+1)/2$ för alla $n \geq 1$

1. $n = 1$.
   Vänsterledet $= 1$.
   Högerledet $= 1 * 2/2 = 1$. Stämmer.
2. Vi antar nu att $1+2+\ldots+n = n(n+1)/2$ för $1 \leq n \leq k$ (*)

   Visa att $1+2+\ldots+k+k+1 = (k+1)(k+2)/2$.

   Vänsterledet $= (1+2+\ldots+k)+k+1 =$ [enligt (*)]
   $= k(k+1)/2 + k + 1 = (k^2+3k+2)/2$
   Högerledet $= (k+1)(k+2)/2 = (k^2+3k+2)/2 =$ Vänsterledet.

Induktion för att visa att en rekursiv algoritm är korrekt

Överkurs

```
public static long factorial(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
```

1. När n=0 blir resultatet 1 vilket är korrekt enligt def av $n!$
2. Antag att algoritmen ger korrekt resultat för $0 \leq n \leq k$. (*)

   Anrop `factorial(k+1)` ger (eftersom $k+1 > 0$) resultatet
   $(k+1) *$ `factorial(k+1-1)` $= (k+1) *$ `factorial(k)`.
   Enligt induktionsantagandet (*) ger anropet factorial(k) korrekt resultat, dvs $k!$.
   Vi får därför resultatet $(k+1) * k! = (k+1)!$ V.S.B.

Exempel på vad du ska kunna

- Förklara begreppet rekursion.
- Förklara hur rekursion fungerar.
- Formulera rekursiva algoritmer och implementera rekursiva metoder.