Datastrukturer & Algoritmer, Dat C
Forelæsning 1/10-2002
Henning Christiansen

Rekursion og "dynamisk programmering"

"at en procedure kalder sig selv" — eller et antal procedurer kalder hinanden rekursivt.

Meget elegant til ting, som er rekursive af natur!

Eksempel: Binær søgning

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

Tidsforbrug: log n.

- Implementation ved stak (tager compileren sig af)
- Optimeringer ved bl.a. halerekursion (compiler producerer samme kode som var det skrevet med "for" eller "while".

Eksempler hvor rekursion er elegant

—rekursivt definerede datastrukturer,

f.eks. et træ (eksempel om et øjeblik)

—compilere og sprog

<sætning> ::= if(<betingelse>) <sætning> ;

| while(<betingelse>) <sætning> ;

| ...

Datastrukturen er syntakstræer;
Parser, evt. hele compiler (Pascal, men ikke Java)
er en samling rekursive procedurer, én for hver syntaktisk kategori.

—matematiske definitioner ofte rekursive

Træer: rekursiv datastruktur og tilhørende rekursive algoritmer
anvendelser: søgetræer, syntakstræer,...

Eksempel: Simple træer med tal

```
public class TreesWithNumbers
{  private TreesWithNumbers left; private TreesWithNumbers right;
   private int value;

   public TreesWithNumbers(int n) {value = n;};

   public TreesWithNumbers(TreesWithNumbers left, int n,
                                                TreesWithNumbers right)
   {this.left=left; value=n; this.right=right;};


 ....
   public static void main( String [ ] args )

    {  TreesWithNumbers t =
          new TreesWithNumbers( new TreesWithNumbers(5),
                                10,
                                new TreesWithNumbers( new TreesWithNumbers(8),
                                                      6,
                                                      new TreesWithNumbers(17)
                                                     )
                              );
       System.out.println("Træet " + t + " har sum " + t.sum());
       t.pp();
    }
}
```

Rekursive metoder til rekursive strukturer.

Det er fristende at skrive

```
 public String toString()
   {return   "(" + left.toString() + "[" + value + "]" + right.toString() + ")";};
```

Denne her er bedre:

```
 public String toString()
   {return   "("
          + (left==null ? "" : left.toString())
          + "[" + value + "]"
          + (right==null ? "" : right.toString())
          + ")";};
```

(([5])[10](([8])[6]([17])))

At summe værdierne sammen:

Det er fristende at skrive

```
public int sum()
{return left.sum() + value + right.sum();};
```

Denne her er bedre

```
public int sum()
{return   (left==null ? 0 : left.sum())
        + value
        + (right==null ? 0 : right.sum());};
```

Matematiske definitioner,
f.eks. n! = 1 x 2 x ... x (n-1) x n

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

   public static void main( String [ ] args )
   {for( int i = 1; i <= 10; i++ )
         System.out.println( factorial( i ) );}}
```

Eksempel: Fibonaccital

0, 1, 1, 2, 3, 5, 8, 13, ... hvad er systemet?

```
public static int fib1(int n)
{if(n<=1) return n;
 else return fib1(n-1) + fib1(n-2);}
```

Elegant, smukt, ubrugeligt
... eksempel på ”del-og.hersk” når det ikke virker

... vi vender tilbage til eksemplet!

En programmeringsteknik: Del og hersk

Generelt mønster:

```
public static Løsning løs(Problem p)
  {  if(simpel(p)) return løsningen-på-det-simple-problem;
     split p op i p1, ..., pn;
     L1 = løs p1;
     L2 = løs p2;
     . . .
     Ln = løs pn;
     return kombiner(L1, ;2, ..., Ln); }
```

Det bedste eksempel: Binær søgning:

```
   private static int binarySearch( Comparable [ ] a, Comparable x,
                                    int low, int high )
   {   if( low > high ) return -1;

       int mid = ( low + high ) / 2;

       if( a[ mid ].compareTo( x ) < 0 )
           return binarySearch( a, x, mid + 1, high );
       else if( a[ mid ].compareTo( x ) > 0 )
           return binarySearch( a, x, low, mid - 1 );
       else
           return mid;
   }
```

Tidsforbrug: log n.

Andre eksempler: Sortering, f.eks. mergesort

Generel formel til vurdering af tidsforbrug

```
public static Løsning løs(Problem p)
  {  if(simpel(p)) return løsningen-på-det-simple-problem;
     split p op i p1, ..., pn;
     L1 = løs p1;
     L2 = løs p2;
     . . .
     Ln = løs pn;
     return kombiner(L1, ;2, ..., Ln); }
```

Parametre:

A, antallet af delproblemer (n ovenfor)
B, mål for relativ størrelse af delproblem (f.eks. B=2 for halvering)
k, bestemt ved "overhead" Theta(N^k) (= prisen for at "kombinere")

$$T(N) = \begin{cases} O(N^{\log_B A}) & \text{hvis } A > Bk \\ O(N^k \log N) & \text{for } A = Bk \\ O(Nk) & \text{for } A < Bk \end{cases}$$

Anvende den på binær søgning ....

Del-og-hersk-version af maksimum-sum-problemet

<fotokopier fra boget s. 261 og 262>

Anvend formel ...

Dynamisk programmering

Princippet: Vi skal løse et problem P(n)
Hvis vi, for at løse P(n) får brug for, direkte eller indirekte, at kende løsningen på P(1), P(2), ..., P(n-1),
så beregner vi dem alle fra en ende af og lægger dem i et array.

Eksempel: Fibonacci ved dynamisk programmering

```
public static int fib2(int n) //assume n>=2
{ int [] fibs = new int [n+1];
  fibs[0] = 0; fibs[1] = 1;
  for(int i=2; i<= n; i++)
    fibs[i] = fibs[i-1]+fibs[i-2];
  return fibs[n];}
```

Eksemplet, at give byttepenge

```
public final class MakeChange
{  public static void makeChange( int [ ] coins, int differentCoins,
              int maxChange, int [ ] coinsUsed )
   {   coinsUsed[ 0 ] = 0;
       for( int cents = 1; cents <= maxChange; cents++ )
       {  int minCoins = cents;
          int newCoin  = 1;
          for( int j = 0; j < differentCoins; j++ )
          {  if( coins[ j ] > cents )   // Cannot use coin j
                continue;
             if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins )
             {  minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                newCoin  = coins[ j ];}}
          coinsUsed[ cents ] = minCoins;}}

   public static void main( String [ ] args )
   { // The coins and the total amount of change
      int numCoins = 5;
      int [ ] coins = { 1, 5, 10, 21, 25 };
      int change = 117;
      int [ ] used = new int[ change + 1 ];
      makeChange( coins, numCoins, change, used );

      System.out.println( "Best is " + used[ change ] + " coins" );}}
```

Afsluttende øvelse: 4 måder at beregne Fibonacci-tal på:

```
// "Divide & conquer" - not recommended
public static int fib1(int n)
{if(n<=1) return n;
 else return fib1(n-1) + fib1(n-2);}
```

```
// "Dynamic programming"
public static int fib2(int n) //assume n>=2
{ int [] fibs = new int [n+1];
  fibs[0] = 0; fibs[1] = 1;
  for(int i=2; i<= n; i++)
    fibs[i] = fibs[i-1]+fibs[i-2];
  return fibs[n];}
```

```
// "Memoization"
 private static HashMap table = new HashMap();

 public static int fib3(int n)
 { Object resultCell;
   if((resultCell = table.get(new Integer(n)))!= null )
      return ((Integer)resultCell).intValue();
   else
      {int result;
       if(n<=1) result=n;
       else result=fib1(n-1) + fib1(n-2);
       table.put(new Integer(n), new Integer(result));
       return(result);};}
```

```
// "Direct algorithm"
// - can be seen as optimized DynProg or memo
// ... however it the long run outperformed by Memo!
public static int fib4(int n)//assume n>=2
{int f0=0; int f1=1; int buf;
 for(int i=2;i<=n;i++)
   f1= f0+(f0=f1);
 return f1;};
```

Afslutning:

— *Rekursion* er elegant for de rigtige problemer
— *Del-og-hersk* er en måde at forstå mange rekursive algoritmer på, f.eks. søgning og sortering
— *"Dynamisk programmering"* er en lignenden, ikke rekursiv teknik, som er god når et problem splittes op i "lignende, men mindre" delproblemer.
— *Memoisering* er kan ses som dyn.prog. med langtidshukommelse, og som kun bberegner det, der er brug for.
  — kræver en tungere datastruktur, men tjener sig ind i længden