Rozklad problému na podproblémy

- Postupný návrh programu rozkladem problému na podproblémy
  - zadaný problém rozložíme na podproblémy
  - pro řešení podproblémů zavedeme abstraktní příkazy
  - s pomocí abstraktních příkazů sestavíme hrubé řešení
  - abstraktní příkazy realizujeme pomocí procedur
- Rozklad problému na podproblémy ilustrujme na příkladu hry NIM
- Pravidla:
  - hráč zadá počet zápalek (např. od 15 do 35)
  - pak se střídá se strojem v odebírání; odebrat lze 1, 2 nebo 3 zápalky,
  - prohraje ten, kdo odebere poslední zápalku.
- Dílčí podproblémy:
  - zadání počtu zápalek
  - odebrání zápalek hráčem
  - odebrání zápalek strojem

Hra NIM

- Hrubé řešení:

```
int pocet;
boolean stroj = false;

“zadání počtu zápalek “
do {
   if (stroj) “odebrání zápalek strojem“
   else “odebrání zápalek hráčem“
   stroj = !stroj;
} while (pocet>0);
if (stroj) “vyhrál stroj“
else “vyhrál hráč“
```


- Podproblémy „zadání počtu zápalek“, „odebrání zápalek strojem“ a „odebrání zápalek hráčem“ budeme realizovat procedurami, proměnné *pocet* a *stroj* pro ně budou nelokálními proměnnými

Hra NIM

```
public class Nim {
  static int pocet;     // aktuální počet zápalek
  static boolean stroj; // =true znamená, že bere počítač

  public static void main(String[] args) {
    zadaniPoctu();
    stroj = false;  // zacina hrac
    do {
      if (stroj) bereStroj(); else bereHrac();
      stroj = !stroj;
    } while (pocet>0);
    if (stroj)
      Sys.pln("vyhrál jsem");
    else
      Sys.pln("vyhrál jste, gratuluji");
  }

  static void zadaniPoctu() { … }
  static void bereHrac() { … }
  static void bereStroj() { … }
}
```

Hra NIM

```
static void zadaniPoctu() {
   do {
     Sys.pln("zadejte počet zápalek (od 15 do 35)");
     pocet = Sys.readInt();
   } while (pocet<15 && pocet>30);
 }

 static void bereHrac() {
   int x; boolean chyba;
   do {
     chyba = false;
     Sys.pln("počet zápalek "+pocet);
     Sys.pln("kolik odeberete");
     x = Sys.readInt();
     if (x<1) {
       Sys.pln("prilis malo"); chyba = true;
     }
     else
     if (x>3 || x>pocet) {
       Sys.pln("prilis mnoho"); chyba = true;
     }
   } while (chyba);
   pocet -= x;
 }
```

Hra NIM

- Pravidla pro odebírání zápalek strojem, která vedou k vítězství (je-li to možné):
  - počet zápalek nevýhodných pro protihráče je 1, 5, 9, atd., obecně $4n+1$, kde n je celé nezáporné číslo,
  - stroj musí z počtu p zápalek odebrat x zápalek tak, aby platilo $p - x = 4n + 1$
  - z tohoto vztahu po úpravě a s ohledem na omezení pro x dostaneme $x = (p - 1) \bmod 4$
  - vyjde-li x=0, znamená to, že okamžitý počet zápalek je pro stroj nevýhodný a bude-li protihráč postupovat správně, stroj prohraje.


```
static void bereStroj() {
   Sys.pln("počet zápalek "+pocet);
   int x = (pocet-1) % 4;
   if (x==0) x = 1;
   Sys.pln("odebírám "+x);
   pocet -= x;
 }
```

Rekurzivní algorimus

- Rekurzivní algoritmus v některém kroku volá sám sebe
- Rekurzivní procedura (funkce) v některém příkazu volá sama sebe
- Příklad: nsd(x, y)

  je-li $x = y$, pak $nsd(x,y) = x$

  je-li $x > y$, pak $nsd(x,y) = nsd(x-y,y)$

  je-li $x < y$, pak $nsd(x,y) = nsd(x,y-x)$
- Rekurzivní funkce:

```
static int nsd(int x, int y) {
   if (x==y) return x;
   else if (x>y) return nsd(x-y, y);
   else return nsd(x, y-x);
 }
```

- Jiný příklad – faktoriál:

  $n! = 1$ pro $n \leq 1$

  $n! = n*(n-1)!$ pro $n > 1$
- Rekurzivní funkce:

```
static int fakt(int n) {
   if (n<=0) return 1;
   return n*fakt(n-1);
 }
```

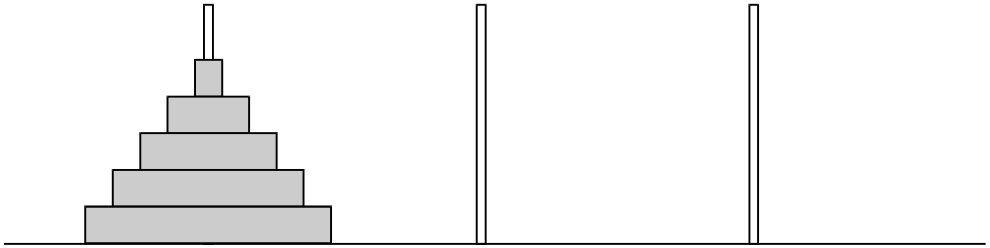
Rekurze a rozklad problému na podproblémy

- Program, který přečte posloupnost čísel zakončenou nulou a vypíše ji obráceně
- Rozklad problému:
  - zavedeme abstraktní příkaz *přečti a obraťposloupnost*
  - příkaz rozložíme do tří kroků:
    - *přečti číslo*
    - if (*přečtené číslo není nula*) *přečti a obraťposloupnost*
    - *vypiš číslo*
- Řešení:

```
public static void main(String[] args) {
   obrat();
 }

static void obrat() {
   int x = Sys.readInt();
   if (x!=0) obrat();
   Sys.pln(x);
 }
```

Hanojské věže



- Úkol: přemístit disky na druhou jehlu s použitím třetí pomocné jehly, přičemž musíme dodržovat tato pravidla:
  - v každém kroku můžeme přemístit pouze jeden disk, a to vždy z jehly na jehlu (disky nelze odkládat mimo jehly),
  - není možné položit větší disk na menší.
- Zavedeme abstraktní příkaz
  přenes_věž(n,a,b,c)
  který interpretujeme jako
  "přenes *n* disků z jehly *a* na jehlu *b* s použitím jehly *c*".
- Pro *n*>0 lze příkaz rozložit na tři jednodušší příkazy

```
přenes_věž(n-1,a,c,b)
"přenes disk z jehly a na jehlu b",
přenes_věž(n-1,c,b,a)
```

Hanojské věže

- Řešení:

```
package alg5;
import sugar.Sys;

public class Hanoj {
  public static void main(String[] args) {
    int pocetDisku = Sys.readInt();
    prenesVez(pocetDisku, 1, 2, 3);
  }

  static void prenesVez(int vyska, int odkud, int kam,
                        int pomoci) {
    if (vyska>0) {
      prenesVez(vyska-1, odkud, pomoci, kam);
      Sys.pln("přenes disk z "+odkud+" na "+kam);
      prenesVez(vyska-1, pomoci, kam, odkud);
    }
  }

}
```

Obecně k rekurzivitě

- Rekurzivní funkce (procedury) jsou přímou realizací rekurzivních algoritmů
- Rekurzivní algoritmus předepisuje výpočet „shora dolů“ v závislosti na velikosti (složitosti) vstupních dat:
  - pro nejmenší (nejjednodušší) data je výpočet předepsán přímo
  - pro obecná data je výpočet předepsán s využitím téhož algoritmu pro menší (jednodušší) data
- Výhodou rekurzivních funkcí (procedur) je jednoduchost a přehlednost
- Nevýhodou může být časová náročnost způsobená např. zbytečným opakováním výpočtu

Příklad: Fibonacciho posloupnost:

$f_0 = 0$

$f_1 = 1$

$f_i = f_{i-1} + f_{i-2}$ pro $i > 1$

```
static int fib(int i) {
  if (i==0) return 0;
  if (i==1) return 1;
  return fib(i-1)+fib(i-2)
}
```

Od rekurze k iteraci

- Řadu rekurzívních algoritmů lze nahradit iteračními, které počítají výsledek „zdola nahoru“, tj, od menších (jednodušších) dat k větším (složitějším)

```
static int fakt(int n) {
  int f = 1;
  while (n>1) {
    f *= n; n--;
  }
  return f;
}

static int fib(int n) {
  int i, fn = 0, fp, fpp;
  if (n>0) {
    fp = fn; fn = 1;
    for (i=2; i<n; i++) {
      fpp = fp; fp = fn; fn = fp + fpp;
    }
  }
  return fn;
}
```


- Pokud algoritmus výpočtu „zdola nahoru“ nenajdeme (např. při řešení problému Hanojských věží), lze rekurzivitu odstranit pomocí tzv. zásobníku