ÚVODNÍ ZNALOSTI

datové struktury

správnost programů

analýza algoritmů

Datové struktury

základní, primitivní, jednoduché datové typy: int, char, ...

hodnoty: celá čísla, znaky, …

jednoduché proměnné: `int i; char c;`

obsahují hodnoty

třídy: `class Auto {…}`, `class Bod{…}`, **...**

hodnoty: objekty, instance třídy `new Auto()`, **…**

referenční proměnné: `Auto a; Bod b;`

obsahují ukazatele na objekty a ukazatel `null`

Poznámka:

pole: `T[ ]`

hodnoty: objekty, *n*-tice proměnných typu *T*,
`new T[n]`

proměnné: `T[ ]v;`

obsahují ukazatele na objekty a ukazatel `null`

datová struktura –
schéma uložení souvisejících hodnot, objektů

- **sekvenční datová struktura**
- **spojová datová struktura**

ZOO

druhy zvířat

třída `DruhZvirat`

```
class DruhZvirat {

  String druh;      // jmeno druhu zvirat
  int pocet;        // pocet zvirat

  DruhZvirat (String druh, int pocet) {
    this.druh = druh;
    this.pocet = pocet;
  }

  void tisk() {
    System.out.println(druh+" "
      +pocet);
  }

}
```

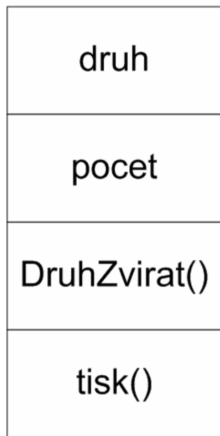
Sekvenční implementace – pole

`z` – pole druhů zvířat v ZOO =
pole objektů třídy `DruhZvirat`

```
DruhZvirat[] z = new DruhZvirat[4];

// uděláme si malou, ale naši ZOO
// nejvíce 4 čtyři druhy zvířat
// pocet počet druhů zvířat

// budeme mít
// 4 opice, 2 slony, 4 žirafy
```

```
class PoleZvirat {
  public static void main(String[] args){

    DruhZvirat[] z = new DruhZvirat[4];
    int pocet = 0;  //pocet druhu zvirat
    tisk(z,pocet);

    z[pocet] = new DruhZvirat("opice",4);
    pocet++;
    tisk(z,pocet);

    z[pocet] = new DruhZvirat("sloni",2);
    pocet++;
    tisk(z,pocet);

    z[pocet] = new DruhZvirat("zirafy",4);
    pocet++;
    tisk(z,pocet);
  }

  static void tisk(DruhZvirat[] z,
                                 int pocet) {
    int i = 1;
    while(i <= pocet) {
      z[i-1].tisk();
      i++;
    }
  }
}
```

DruhZvirat



z

0
1
2
3

pocet 0

druh opice
pocet 4

z

0
1
2
3

pocet 1

datová struktura je dobře

co není dobře ?

data ZOO

```
DruhZvirat[] z;
int pocet;
```

jakož i operace nad nimi jsou oddělené !

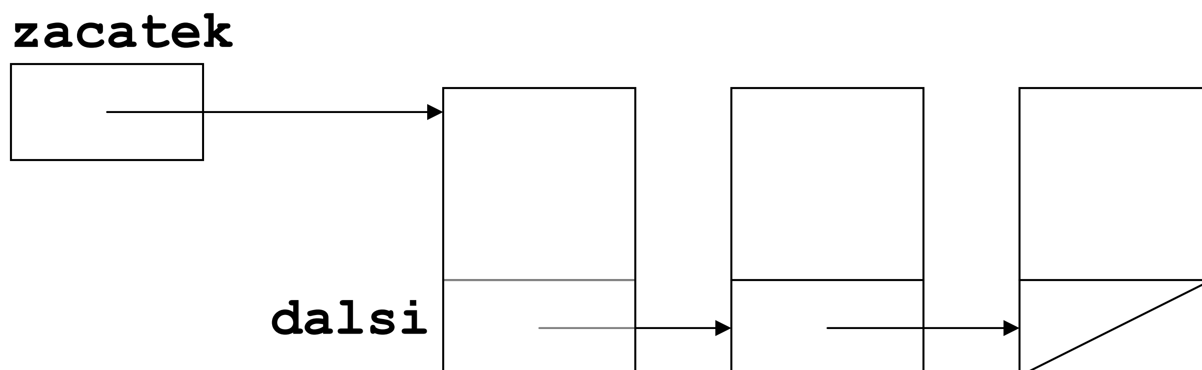
třída ZOO   :-)

```
class ZOO {

  DruhZvirat[] z;
  int pocet;

  ZOO() {
    z = new DruhZvirat[4];
    pocet = 0;
  }

  void pridej (String druh, int kolik) {
    z[pocet] = new DruhZvirat(druh, kolik);
    pocet++;
  }

  void tisk() {
    int i = 1;
    while(i <= pocet) {
      z[i-1].tisk();
      i++;
    }
  }

}
```

vytvoříme objekt(y) třídy ZOO

```
class ZOOHlavni {

  public static void main(String[] args){

    ZOO plzen = new ZOO();

    plzen.pridej("opice",4);
    plzen.tisk();

    plzen.pridej("sloni",2);
    plzen.tisk();

    plzen.pridej("zirafy",4);
    plzen.tisk();

  }

}
```

Spojové datové struktury

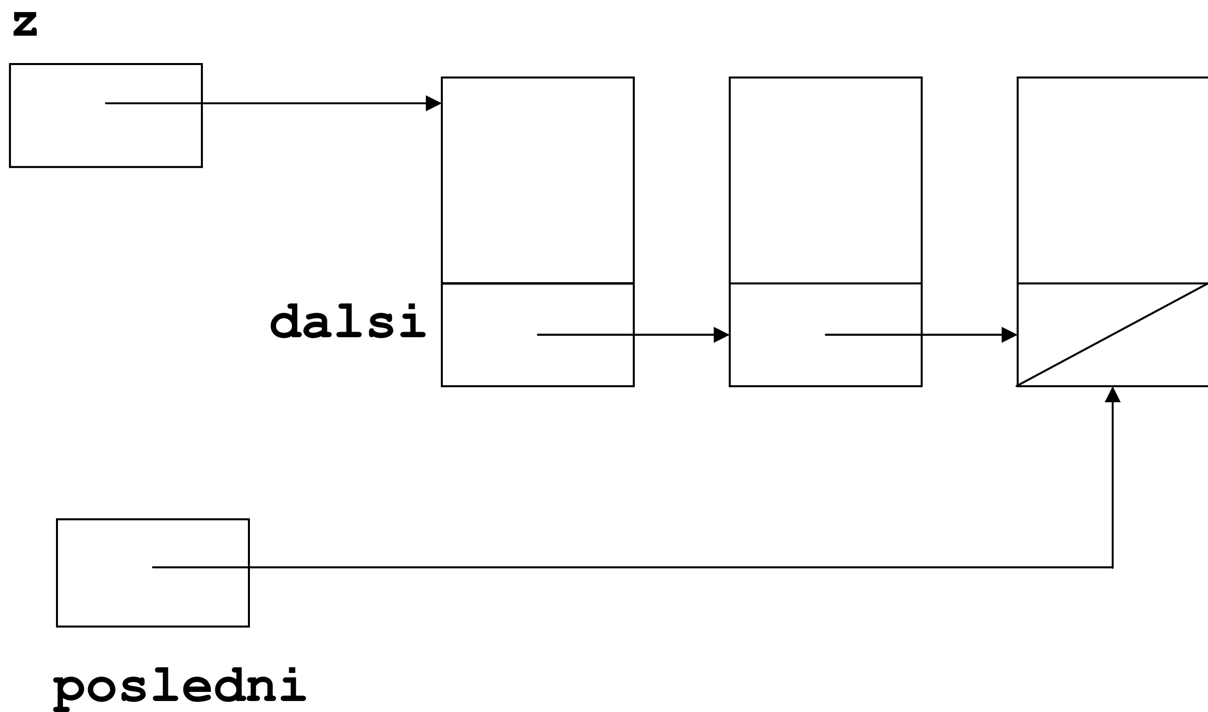


obecně data obsahují ukazatel, pointer, referenci, odkaz, směrník, ... na další data

Java – třída obsahuje referenční proměnnou `dalsi`

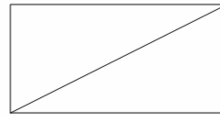
```
class DruhZvirat {

  String druh;
  int pocet;
  DruhZvirat dalsi;

  DruhZvirat (String druh, int pocet) {
    this.druh = druh;
    this.pocet = pocet;
    // dalsi bude inicalizovano na null
  }

  void tisk() {
    System.out.println(druh+" "+pocet);
  }

}
```

// nasledující program je ilustrací spojování objektů
// `z` zvířata – jejich začátek

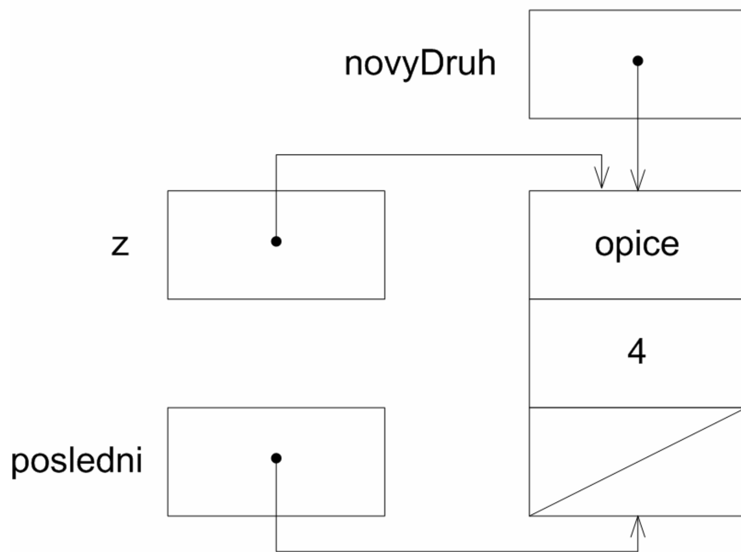


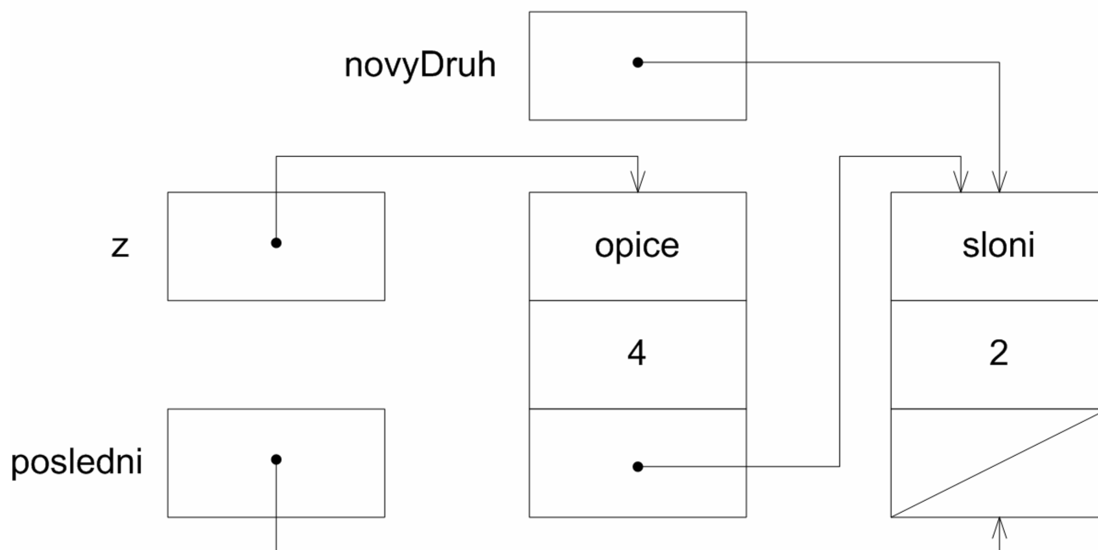
```java
class SeznamZvirat {
  public static void main(String[] args){

    DruhZvirat z = null; //zacatek
    DruhZvirat posledni;
    tisk(z);

    DruhZvirat novyDruh;

    novyDruh = new DruhZvirat("opice",4);
    z = novyDruh;
    posledni = novyDruh;
    tisk(z);

    novyDruh = new DruhZvirat("sloni",2);
    posledni.dalsi = novyDruh;
    posledni = novyDruh;
    tisk(z);

    novyDruh = new DruhZvirat("zirafy",4);
    posledni.dalsi = novyDruh;
    posledni = novyDruh;
    tisk(z);
  }
  static void tisk(DruhZvirat z) {
    DruhZvirat x = z;
    while (x != null) {
      x.tisk();
      x = x.dalsi;
    }
  }
}
```

DruhZvirat
druh
pocet
dalsi
DruhZvirat()
tisk()


z

posledni


novyDruh
z
opice
4
posledni


novyDruh
z
opice
sloni
4
2
posledni

datová struktura dobře, ale ... - třída ZOO

```
class ZOO {

  DruhZvirat z; //zacatek seznamu zvirat
  DruhZvirat posledni;

  ZOO() {
    z = null;
  }

  void pridej (String druh, int kolik) {
    DruhZvirat novyDruh;
    novyDruh = new DruhZvirat(druh,kolik);
    if (z == null)
      z = novyDruh;
    else
      posledni.dalsi = novyDruh;
    posledni = novyDruh;
  }

  void tisk() {
    DruhZvirat x = z;
    while (x != null) {
      x.tisk();
      x = x.dalsi;
    }
  }
}
```

klient `ZOOHlavni` třídy `ZOO` je stejný

obě implementace třídy `ZOO` poskytují „stejné“ metody a klient používá pouze tyto metody

ZOO
z
pocet
ZOO()
pridej()
tisk()

ZOO
z
posledni
ZOO()
pridej()
tisk()

Obecnější spojová datová struktura

Řád plemenné knihy arabského koně

. . .

7.2. Plemenná kniha se člení:

- plemenná kniha hřebců

- plemenná kniha klisen

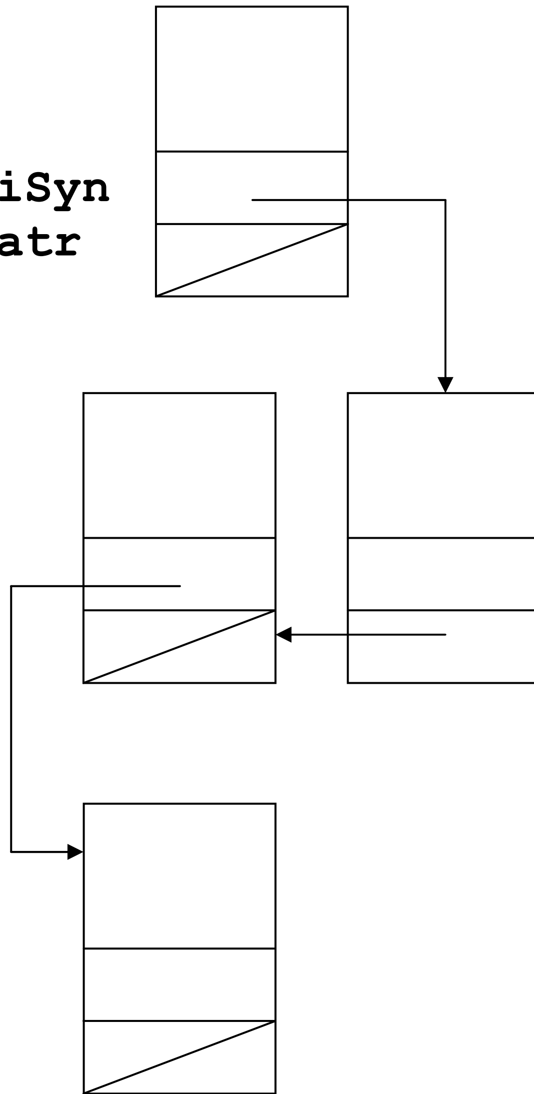
. . .

14.6. Pojmenovávání koní.

Plemenní hřebci - dnem platnosti PK obdrží každý hřebec nově zapsaný do PK jméno, které má stejné začáteční písmeno jako linie k níž po otci přísluší. ...

Plemenné klisny - pojmenování klisen je libovolné a přísluší chovateli s požadavkem zabránění duplicity.

```
class Hrebec {

  String jmeno;
  Hrebec nejmladsiSyn;
  Hrebec starsiBratr;

  Hrebec (String jmeno) {
    this.jmeno = jmeno;
  }

  String getJmeno( ) {
    return jmeno;
  }
}
```


nejmladsiSyn
starsiBratr

```
class Linie
{

  Hrebec zakladatel;

  Linie(String jmeno)
  {
  zakladatel = new Hrebec(jmeno);
  }

  void pridejSyna(Hrebec otec, String jmeno)
  {
    Hrebec novy = new Hrebec(jmeno);
    novy.starsiBratr = otec.nejmladsiSyn;
    otec.nejmladsiSyn = novy;
  }

}
```

```
class HrebciHlavni {

  /* toto je pouze ilustrace obecnejsi spojove
  struktury */

  public static void main(String[] args) {

    Linie linieB = new Linie ("Bosbar");
    System.out.println("\n"+"zakladatelem linie je "+
        linieB.zakladatel.getJmeno()+"\n");

    linieB.pridejSyna(linieB.zakladatel, "Borax");
    linieB.pridejSyna(linieB.zakladatel, "Bertik");

    System.out.println("nejmladsi syn je "+
       linieB.zakladatel.nejmladsiSyn.getJmeno()+"\n");

    System.out.println("druhy nejmladsi syn je "+
        linieB.zakladatel.nejmladsiSyn.starsiBratr.getJmeno()
        +"\n");

    linieB.pridejSyna
        (linieB.zakladatel.nejmladsiSyn.starsiBratr,
        "Brist"+"\n");

    System.out.println("(nejmladsi) syn Boraxe je "+

    linieB.zakladatel.nejmladsiSyn.starsiBratr.nejmladsiSyn.
        getJmeno()+"\n");

  }
}
```

http://www.kiv.zcu.cz/~netrvalo/vyuka/ppa2/portal/cviceni/cv03.pdf

Správnost programů

Testováním programu, tj. ověřením, že pro daný vstup získáme správný výstup *nemůžeme* obecně prokázat správnost programu.

Testování *může* prokázat chybu v programu.

Programová logika

ne hodnoty vstupu a výstupu

vztah mezi hodnotami (relevantmích) proměnných před a po vykonání příkazu

tvrzení, logický výraz ($x \neq 0$ před dělením x)

tvrzení před příkazem – *předpoklad (precondition)*
tvrzení před příkazem – *důsledek (postcondition)*

{předpoklad}	***{P}***
příkaz;	**příkaz;**
{důsledek}	***{Q}***

posloupnost příkazů

příkaz1;
příkaz2;

{předpoklad1}
příkaz1;
{důsledek1}

důsledek1 → *předpoklad2* (implikuje)

předpoklad2
příkaz2;
důsledek2

předpoklad1
příkaz1; příkaz2;
důsledek2

přiřazení

{Q(e)}
v = e; // proměnné v přiřadí hodnotu výrazu e
{Q(v)}

// vstup x
```
x  =  x – 1;
y  =  sqrt(x+2);
```
// může být x ≥ 0, důsledek příkazu vstupu ?

{x ≥ -1}
`x = x – 1;` *// x – 1 ≥ -2*
{x ≥ -2}

{x ≥ -2}
`y  =  sqrt(x+2);` *// √(x+2) ≥ 0*
{y ≥ 0}

$x \geq 0 \rightarrow x \geq -1$

alternativa

{P}
if B
  S1;
else
  S2;
{Q}

musí být:

{P && B}	*{P && !B}*
S1;	*S2;*
{Q}	*{Q}*

{ true }

```
if ( x >= y )   // B
  m = x;        // S1
else
  m = y;        // S2
```

{(x >= y && m ==x) || (x<y) && m==y)}   //maximum

pro S1 musí být

{ x>=y }
`m = x;`
{(x >= y && m ==x) || (x<y) && m==y)}

je

{(x >= y && x ==x) || (x<y) && x==y)}
`m = x;`
{(x >= y && m ==x) || (x<y) && m==y)}

protože

(x>=y) → (x >= y && x ==x) || (x<y) && x==y)

skutečně je

{ x>=y }
`m = x;`
{(x >= y && m ==x) || (x<y) && m==y)}

podobně pro S2

cyklus

řazení vkládáním

```
int a[ ] = new int [n];
for (int j=1; j < a.length; j++) {
   int vkladanaHodnota = a[j]; // (n-1)-krát
   int i = j;
   while (i > 0 && a[i-1] > vkladanaHodnota) {
   // pro každé j=1,…,n-1obecně kj-krát
     a[i] = a[i-1];
     --i;
   }
   a[i] = vkladanaHodnota;
 }
```

na začátku každého vykonávání cyklu for, začátek pole s prvky s indexy 0 ... j-1 obsahuje hodnoty původních prvků, jenomže seřazené

tuto vlastnost nazýváme invariantem cyklu

tři vlastnosti invariantu cyklu

Inicializace: Je splněn před vykonáním prvního cyklu.

Udržování: Je-li splněn pře vykonáním cyklu, zůstává splněn i po něm.

Skončení: Když cyklus skončí, invariant nám ukáže správnost algoritmu.

(Obdoba matematické indukce, kromě skončení.)

řazení vkládáním

Inicializace: Po začátečním přiřazení j = 1 je pole s prvky s indexy 0 ... j-1 triviálně seřazeno.

Udržování: Cyklus for posouvá prvky a[j-1],a[j-2], ... o jednu pozici doprava dokud se nenajde správné místo pro a[j].

Skončení: Cyklus for skončí když j=n, kde n je počet prvků pole.

Dosazením do textu invariantu cyklu dostáváme, že prvky s indexy 0 ... n-1 obsahují seřazené původní prvky pole.

jenom teorie?

ne – příkaz **assert**

```
assert výraz1 : výraz2 ;
```

výraz1 – tvrzení, logický výraz

výraz2 – chybová zpráva

```
// použijte -ea v příkazu java
import java.util.Scanner;
public class Assert {
  public static void main( String args[] ) {
    Scanner input = new Scanner( System.in );
    System.out.print( "Zadejte cislo mezi 0 a 10: ");
    int cislo = input.nextInt();
    // tvrdime, ze cislo musi byt >= 0 a <= 10
     assert ( cislo >= 0 && cislo <= 10 ) : "Chybne
              cislo: " + cislo;
    System.out.printf( "Zadali jste %d\n", cislo );
  }
}
```

Analýza algoritmů

Analýza algoritmu *určuje požadované prostředky na jeho vykonání.*

Nejčastěji nás zajímá *čas výpočtu*

Označme $c_{operace}$ počet elementárních kroků potřebných pro vykonání určité operace

Je-li vykonána n-násobně, přispěje k času vykonání
$n*c_{operace}$

Splnitelnost výrokových formulí

if (*booleovský výraz*)

**   *příkaz;***

Nechť *booleovský výraz* pozůstává z
- **n proměnných logického typu v1, v2, …, vn**
- **m logických operátorů not, and, …**
- **závorek**

c_h - čas přiřazení hodnoty proměnné

c_o - čas každé logické operace

vyhodnocení podmínky (výrokové formuly)

$T(n,m) = n\ c_h + m\ c_o$

Výroková formule (podmínka) je splnitelná existuje-li přirazení hodnot true a false proměnným v1, v2, …, vn takové, že hodnota formule (logického výrazu) je true.

Úkol – zjistit zda-li je zadaná formule splnitelná

Algoritmus – pro každé možné přiřazení hodnot proměnným v1, v2, …, vn formuli vyhodnoť

Počet různých přiřazení je 2^n a čas výpočtu algoritmu je

$$T(n,m) = 2^n(n\, c_h + m\, c_o)$$

Násobení matic

```
int[ ][ ] a = new int[n][n];
   int[ ][ ] b = new int[n][n];
   int[ ][ ] c = new int[n][n];
   for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++) {
       c[i][j] = 0;
       for (int k = 0; k < n; k++)
         c[i][j] = c[i][j] + a[i][k]*b[k][j];
       }
```

Čas výpočtu:

n-krát vnější cyklus (

n-krát cyklus pro j ($c_{nulování}$ +

n-krát vnitřní cyklus

($c_{sčítání}$ + $c_{násobení}$)))

$$T(n)=n^2(c_{nulování} + n(c_{sčítání} + c_{násobení})) =$$

$$n^3(c_{sčítání} + c_{násobení}) + n^2 c_{nulování} = an^3 + bn^2$$

Co z $an^3 + bn^2$ dostatečně charakterizuje chování algoritmu, tedy míru růstu času výpočtu v závislosti na velikosti problému n?

Důležité je chování pro velká n.

Pro a=b=1 a n=1000 je celková hodnota uvedeného výrazu 1 001 000 000 a člen n^2 (1 000 000) přispívá podílem 0,1%.

n^3 tedy intuitivně charakterizuje růst času výpočtu algoritmu v závislosti na velikosti vstupu.

Řazení vkládáním

```
int a[ ] = new int [n];

for (int j=1; j < a.length; j++) {
    int vkladanaHodnota = a[j]; // (n-1)-krát
    int i = j;
    while (i > 0 && a[i-1] > vkladanaHodnota) {
    // pro každé j=1,…,n-1 obecně kj-krát
      a[i] = a[i-1];
      --i;
    }
    a[i] = vkladanaHodnota;
  }
```

čas výpočtu

$$T(n)=3(n-1)*c_{přiř} + c_{por} \sum_{j=1}^{n-1} k_j + 2c_{přiř} \sum_{j=1}^{n-1}(k_j - 1)$$

Pozn.: **k_j závisí na zadané řazené posloupnosti**
může být různé pro totéž j

v nejlepším případě $\mathbf{k_j = 1}$ a

$\mathbf{T(n)=(n-1)*(3c_{při\check{r}} + c_{por})}$

- růst času výpočtu je lineární - n

v nejhorším případě $\mathbf{k_j = j+1}$ a

$\mathbf{T(n)=3(n-1)*c_{při\check{r}} + (n(n+1)/2-1)c_{por} + n(n-1)c_{při\check{r}}}$

- růst času výpočtu je kvadratický – n^2

v průměrném případě $\mathbf{k_j = (j+1)/2}$ a
- růst času výpočtu je opět kvadratický – n^2

Časová složitost - růst času výpočtu v nejhorším nebo průměrném případě

Asymptotická složitost

Asymptotická složitost vyjadřuje limitní růst času výpočtu, když velikost problému neomezeně roste.

Funkce $g(n)$ je $\Theta(f(n))$ existují-li kladné konstanty c_1, c_2 a n_0 takové, že

$$0 \le c_1 f(n) \le g(n) \le c_2 f(n)$$

pro všechna $n \ge n_0$

Růst času výpočtu uvedeného algoritmu násobení matic jsme intuitivně navrhli charakterizovat členem n^3

Čas výpočtu uvedeného algoritmu je $\Theta(n^3)$

$$c_1 n^3 \le an^3 + bn^2 \le c_2 n^3 \qquad a,b > 0$$

$$c_1 \le a + b/n \le c_2$$

$c_1 = a$, $c_2 = a + b/n_0$, n_0 libovolné kladné

Polynom $a_d n^d + \ldots + a_0$, $a_d > 0$ je $\Theta(n^d)$

Θ notace omezuje funkci shora i zdola

Asymptotické omezení shora

Funkce $g(n)$ je $O(f(n))$, existují-li kladné konstanty c a n_0 takové, že

$$0 \leq g(n) \leq cf(n)$$

pro všechna $n \geq n_0$

$an^3 + bn^2$ je tedy $O(n^3)$, ale i $O(n^4)$, …

Pro algoritmus řazení vkládáním:

- nejhorší případ $O(n^2)$
- každý vstup $O(n^2)$
- nejlepší případ je $O(n)$

Asymptotické omezení zdola

Funkce g(n) je Ω(f(n)), existují-li kladné konstanty c a n_0 takové, že

$$0 \leq cf(n) \leq g(n)$$

pro všechna $n \geq n_0$

Pro algoritmus řazení vkládáním:

-nejlepší případ Ω(n)

-pro každý vstup Ω(n)

-nejhorší případ $\Omega(n^2)$

Rychlost procesoru

Předpokládejme, že rychlost procesoru se zvětšila k-krát.

Byl-li za čas T algoritmem s časem výpočtu $f(n)$ vyřešen problém o velikosti a na pomalejším procesoru, jak velký problém x vyřešíme stejným algoritmem na k-krát rychlejším počítači za stejný čas?

Pro pomalejší počítač

$T=f(a)$

Pro rychlejší počítač

$T=f(x)$

Pro pomalejší počítač

$kT=f(x)$

$x = f^{-1} (kT) = f^{-1} (k.f(a))$

http://www.kiv.zcu.cz/~netrvalo/vyuka/ppa2/portal/cviceni/slozitost.pdf