Složitost

- jen úvod do problematiky

Úvod

- praktická realizace algoritmu = omezení zejména:
  - časem
  - velikostí paměti
- **složitost** = vztah daného algoritmu k daným prostředkům:
- **časová složitost** → každé množině vstupních dat přiřazuje počet operací při výpočtu podle stanoveného algoritmu
  - čas se měří počtem nutně provedených operací (doba provedení operace nezávisí na rozsahu vstupních dat)
  - bývá často podceňována
  - často zrychlení úlohy řešeno rychlejším HW, nebo…
  - dílčím zrychlením
    - programátorský trik (viz. zarážka u třídění)
    - přepis části kódu do assembleru apod.
  - vhodnější zkusit najít rychlejší algoritmus – pokud existuje
- **paměťová složitost** → závislost paměťových nároků na vstupních datech

Přesné zjištění složitosti

- stanovení časové složitosti v závislosti na:
  - konkrétních datech
  - rozsahu dat – mnohem častější
    - obecně → analýzou algoritmu (často velmi složité)
    - u triviálních algoritmů → lze detailní analýzou programu
- příklad č. 1 – součet prvků pole

```
static int SoucetPrvku(int[] pole)
{
      int suma = 0;           // prirazeni p1
      for(int i = 0;          // prirazeni p2
          i < pole.Length;    // porovnani c1
          i++) {              // soucet a prirazeni s1+p3 (i = i+1)
            suma += pole[i];  // soucet a prirazeni s2+p4 …
                              // … (Suma = Suma + Pole[i])
      }
      return suma;
}
```

  - časová složitost pro *n* prvků pole (`pole.Length = n`):

```
C(n) = p1 + p2 + (n + 1) * c1 + n * (s1 + p3) + n * (s2 + p4)
```

- zjednodušení → stejně složité *p*, *c*, *s* → „univerzální“ akce *a*:

```
C(n) = a + a + (n + 1) * a + a * (a + a) + n * (a + a)
     = a * (3 + 5 * n)
```

- pokud akce *a* bude trvat jednotku času → C(n) = 3 + 5n

- příklad č. 2 – součet prvků čtvercové matice

```
static int SoucetMatice(int[,] matice)
{
      int suma = 0;                 // prirazeni p1
      for(int i = 0;                // prirazeni p2
          i < matice.GetLengh(0);   // porovnani c1
          i++) {                    // soucet a prirazeni s1+p3
            for(int j = 0;          // prirazeni p4
                j < matice.GetLengh(1);   // porovnani c2
                j++) {              // soucet a prirazeni s2+p5
                    suma += matice[i][j];
                    // soucet a prirazeni s3+p6
            }
      }
      return suma;
}
```

- časová složitost pro čtvercovou matici řádu *n* (`Rad = n`):

```
C(n) = p1 + p2 + (n + 1) * c1 + n * (s1 + p3)  +
 + n * ( p4 + (n + 1) * c2 + n * (s2 + p5) + n * (s3 + p6))
```

1) po převedení na akce:

```
C(n) = a + a + (n + 1) * a + n * (a + a)  +
       + n * ( a + (n + 1) * a + n * (a + a) + n * (a + a)) =
     = 3a + 5an + n(3a + 5an) =
     = 3a + 5an + 3an +5an^2 =
     = 3a + 8an + 5an^2
```

2) pokud akce *a* bude trvat jednotku času → `C(n) = 3 + 8n + 5n2`

- Vypočtené délky trvání funkcí pro oba příklady pro *a* = 1 ms

n	pole	matice
10	53	583
20	103	2163
40	203	8323
80	403	32643

Odhad složitosti

- přesné vyjádření pro různé *n* nás obvykle nezajímá → používá se pouze tendence růstu počtu operací pro zvětšující se *n*
- pak lze výrazy pro *C*(*n*) zjednodušit zanedbáním:
  1) aditivních konstant (u matice 3)
  2) multiplikativních konstant (u matice 5 a 8)

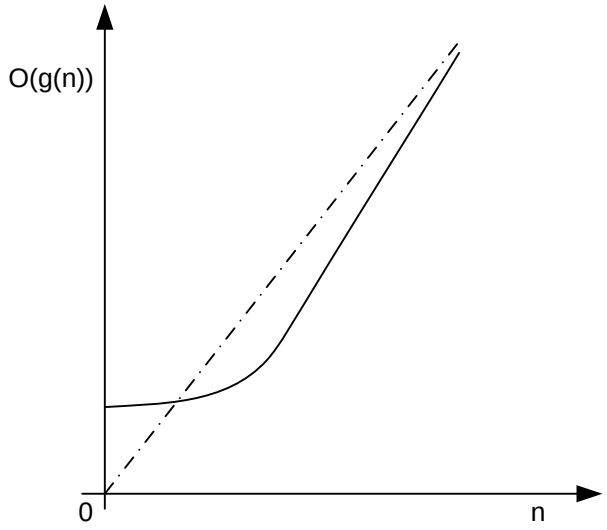
3) všechny složky s nižším řádem růstu než nejvyšším (u matice n)

- u pole je $C(n) = n \rightarrow$ **složitost je lineární**
- u matice je $C(n) = n^2 \rightarrow$ **složitost je kvadratická**
- Vypočtené délky trvání funkcí po zjednodušení

```
n     pole  matice
10    10    100
20    20    400
40    40    1600
80    80    6400
```

Asymptotická složitost

- asymptotické chování funkce = chování pro velká n
- uvažme dvě funkce $t(n)$ a $g(n)$:
  1) nezáporné funkce definované na oboru přirozených čísel
  2) $t(n) \rightarrow$ výpočetní čas algoritmu (obvykle vychází z $C(n)$)
  3) $g(n) \rightarrow$ jednoduchá funkce použitá pro porovnání s $t(n)$
- o $t(n)$ a $g(n)$ můžeme říci: „t roste nejvýše tak rychle jako g“



1) tedy existuje přirozené číslo K: $t(n) \leq K \cdot g(n)$
   - pro součet matice výše je $K = 6$
2) $t(n) = O(g(n))$, $O(g(n))$ je **asymptotická složitost**

- v tomto způsobu zápisu:
  1) součet pole – jeden cyklus – složitost $O(n)$
  2) součet matice – dva cykly – složitost $O(n^2)$

Hrubý odhad složitosti pro triviální algoritmy

- jednoduché → každý vnořený cyklus zvyšuje mocninu složitosti o jednu

- všechny prezentované jednoduché třídící algoritmy mají složitost $O(n^2)$ – zkracování vnitřních cyklů zanedbáváme
- pozor: složitost je odvozena od algoritmu, nikoli od dat! → algoritmy třídění pracovaly nad jednoduchým polem, ale dva vnořené cykly

Třídy složitosti

- Konstantní $O(1)$
  - ideální případ
  - pouze jednoduché matematické operace
- Lineární $O(n)$
  - jeden cyklus v algoritmu
  - např. sekvenční vyhledávání, násobení vektoru skalárem, …
- Kvadratická $O(n^2)$
  - algoritmy se dvěma cykly (vnořenými!!!)
  - např. základní třídící algoritmy, sčítání matic
- Logaritmická $O(\log n)$
  - binární vyhledávání
  - v každém kroku se rozsah dat snižuje na polovinu

    $n \to \frac{n}{2} \to \frac{n}{4} \to \ldots \to 2 \to 1$, kroků půlení intervalů je h

    tedy $n \geq 2^h \to h \leq \log_2 n$
  - místo $\log_2 n \to$ obvykle pouze $\log n$
- Loglineární $O(n \log n)$
  - nebo také linearitmická, supralineární…
  - pro algoritmy „rozděl a panuj“
  - např. algoritmus třídění quick sort, FFT (Fast Fourier Transform – rychlá fourierova transformace)
- existují i další třídy složitosti

Praktický význam složitosti

- doby výpočtu různých tříd složitosti pro různá data; podmínka: akce *a* trvá 1 ns (CPU ~ 1ky GHz)

	10	100	1 000	10 000	100 000
$O(\log n)$	3,3 ns	6,6 ns	10 ns	13,3 ns	16,6 ns
$O(n)$	10 ns	100 ns	1 μs	10 μs	100 μs
$O(n \log n)$	33 ns	660 ns	10 μs	133 μs	1,66 ms
$O(n^2)$	100 ns	10 μs	1 ms	100 ms	10 s

Efektivita algoritmů a data

- většina algoritmů → různý počet operací pro různá data

- nejlepší, nejhorší a průměrný případ
- Nejhorší případ → ohraničuje výpočetní čas shora
- Nejlepší případ → ohraničuje výpočetní čas zdola
- Průměrný případ
  - informace o chování algoritmu na typickém nebo náhodném vstupu
  - není to průměr nejhoršího a nejlepšího případu!!!
  - velmi důležité → některé algoritmy mají průměrnou efektivitu o mnoho lepší než nejhorší případ
- Příklad: sekvenční vyhledávání

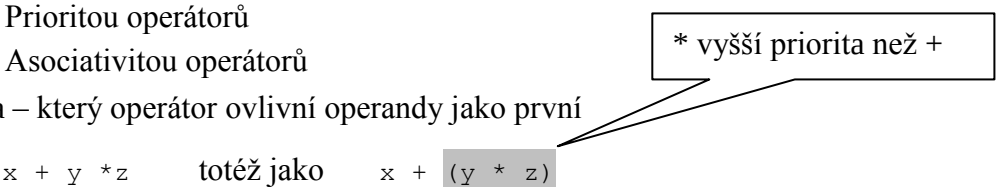
```
static int SeqSearch(int[] pole, int prvek)
{
      int index;

      for(int i=0;i<pole.Length;i++) {
            if (pole[i] == prvek) {
                  index = i;
                  return index;
            }
      }
      return -1;
}
```

  - nejhorší případ → prvek není nalezen (`pole.Length` operací)
  - nejlepší případ → hledaný prvek na první pozici (1 operace)
  - průměrný případ → složitější (statistika)

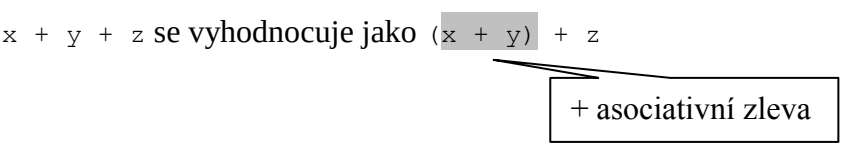
Dodatky k výrazům

Priorita a asociativita operátorů

- (precedence and associativity)
- Pořadí operací ve výrazech dáno
  - Závorkami
  - Prioritou operátorů
  - Asociativitou operátorů
- Priorita – který operátor ovlivní operandy jako první



  - Asociativitou operátorů – směr vyhodnocení operátorů se stejnou prioritou



Category	Operators	associativity
Primary	`x.y f(x) a[x] x++ x-- new typeof checked unchecked`	
Unary	`+ - ! ~ ++x --x (Přetypování)x`	
Multiplicative	`* / %`	
Additive	`+ -`	
Shift	`<< >>`	
Relational and type testing	`< > <= >= is as`	
Equality	`== !=`	
Logical AND	`&`	
Logical XOR	`^`	
Logical OR	`|`	
Conditional AND	`&&`	
Conditional OR	`||`	
Conditional	`?:`	R → L
Assignment	`= *= /= %= += -= <<= >>= &= ^= |=`	R → L

Nejvyšší priorita

Nejnižší priorita

- většina operátorů → zleva doprava (L → R), kromě označených R → L
- doporučení → závorkovat!!!
- příklady:

```
if (A > B && A < C)        totéž co    if ((A > B) && (A < C))
x = A/B*C;    (asociativita) totéž co    x = (A/B)*C;
x = y = 10*C;              totéž co    x = (y = 10*C);
```

Zkrácenené vyhodnocování logických výrazů

- využívá se asociativity log operátorů zleva

```
int a = 5;
int b = 0;
if ((b != 0) && (a / b > 2))
{
      Console.Write("ahoj");
}
```

  - problém → pokud `b = 0` → `a / b` = pád programu
  - zkrácené vyhodnocení → pokud `(b != 0) == false`, k vyhodnocení `(a / b > 2)` nikdy nedojde

Podmíněný (ternární) operátor `?:`

- syntaxe:

```
bVyraz ? vyraz1 : vyraz2;
```

- sémantika

```
if (bVyraz) {
      vyraz1;
} else {
      vyraz2;
}
```

- je to výraz!!!
- typické použití: `vyraz1` i `vyraz2` přiřazují do stejné proměnné
  - Př.: převod čísla na absolutní hodnotu

```
int prom = 10;
prom = (prom >= 0) ? prom : -prom;
```

- rozdíl oproti `if` na příkladu: pokud je `a > b` pak `Cos(a)`, jinak `Cos(b)`
  - `if` je příkaz

```
double x;
if (a > b)
{
      x = Math.Cos(a);
}
else
{
      x = Math.Cos(b);
}
```

  - `? :` je výraz

```
double x = Math.Cos(a > b ? a : b);
```

- další příklady:

```
int a,b,c;
a = (b == 2) ? 1 : 3;
c = (a > b) ? a : b;
```