Časová složitost algoritmů

- Důležitou vlastností algoritmu je časová náročnost výpočtů provedené podle daného algoritmu
- Ta se nezískává měřením doby výpočtu pro různá data, ale analýzou algoritmu, jejímž výsledkem je časová složitost algoritmu
- Časová složitost algoritmu vyjadřuje závislost času potřebného pro provedení výpočtu na rozsahu (velikosti) vstupních dat
- Čas se však neměří sekundách, ale počtem provedených operací, přičemž trvání každé operace se chápe jako bezrozměrná jednotka
- Příklad: součet prvků pole

```
static int soucet(int[] pole) {
    int s = 0;
    for (int i=0; i<pole.length; i++ ) s = s + pole[i] ;
    return s;
}
```

- Považujme za operace podtržené konstrukce, pak časová složitost je:

  $C(n) = 2 + (n+1) + n + n = 3 + 3n$

  kde n je počet prvků pole

Časová složitost algoritmů

- Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách.
- Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě
- Příklad: sekvenční hledání prvku pole s danou hodnotou

```
static int hledej(int[] pole, int x) {
   for (int i=0; i<pole.length; i++ )
      if ( x==pole[i] ) return i;
   return -1;
}
```

- Analýza:
  - **nejlepší případ: první prvek má hodnotu *x***

Časová složitost algoritmů

- Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách
- Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě
- Příklad: sekvenční hledání prvku pole s danou hodnotou

```
static int hledej(int[] pole, int x) {
   for (int i=0; i<pole.length; i++ )
      if ( x==pole[i] ) return i;
   return -1;
}
```

- Analýza:
  - nejlepší případ: první prvek má hodnotu *x*
    $C_{min}(n) = 3$
  - **nejhorší případ: žádný prvek nemá hodnotu *x***

Časová složitost algoritmů

- Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách
- Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě
- Příklad: sekvenční hledání prvku pole s danou hodnotou

```
static int hledej(int[] pole, int x) {
   for (int i=0; i<pole.length; i++ )
      if ( x==pole[i] ) return i;
   return -1;
}
```

- Analýza:
  - nejlepší případ: první prvek má hodnotu *x*
    $C_{min}(n) = 3$
  - nejhorší případ: žádný prvek nemá hodnotu *x*
    $C_{max}(n) = 1 + (n+1) + n + n = 2 + 3n$
  - **průměrný případ**

Časová složitost algoritmů

- Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách
- Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě
- Příklad: sekvenční hledání prvku pole s danou hodnotou

```
static int hledej(int[] pole, int x) {
   for (int i=0; i<pole.length; i++ )
      if ( x==pole[i] ) return i;
   return -1;
}
```

- Analýza:
  - nejlepší případ: první prvek má hodnotu *x*
    $C_{min}(n) = 3$
  - nejhorší případ: žádný prvek nemá hodnotu *x*
    $C_{max}(n) = 1 + (n+1) + n + n = 2 + 3n$
  - průměrný případ
    $C_{prum}(n) = 2.5 + 1.5n$

Časová složitost algoritmů

- Přesné určení počtu operací při analýze složitosti algoritmu bývá velmi složité
- Zvlášť komplikované, ba i nemožné, bývá určení počtu operací v průměrném případě; proto se většinou **omezujeme jen na analýzu nejhoršího případu.**
- Zpravidla nás nezajímají konkrétní počty operací pro různé rozsahy vstupních dat *n*, ale tendence jejich růstu při zvětšujícím se *n* .
- Pro tento účel lze výrazy udávající složitost zjednodušit: stačí uvažovat pouze složky s nejvyšším řádem růstu a i u nich lze zanedbat multiplikativní konstanty
- Příklad: řád růstu časové složitosti předchozích algoritmů je *n* (časová složitost je lineární).
- Časovou složitost vyjadřujeme pomocí tzv. **asymptotické notace**:

  O dvou funkcích ***f*** a ***g*** definovaných na množině přirozených čísel a s nezáporným oborem hodnot říkáme, že

  f* roste řádově nejvýš tak rychle, jako *g a píšeme

  $f(n) = O(g(n))$

  pokud existují přirozená čísla *K* a n_1 tak, že platí

  $f(n) \leq K.g(n)$ pro všechna $n > n_1$

Časová složitost algoritmů

- Tabulka udávající dobu výpočtu pro různé časové složitosti za přepokladu, že 1 operace trvá 1 μs

	n = 10	20	40	60	500	1000
$\log n$	2,3μs	4,3μs	5μs	5,8μs	9μs	
n	10μs	20μs	40μs	60μs	0,5s	1ms
$n \log n$	23μs	86μs	0,2ms	0,35ms	4,5ms	10ms
n^2	0,1ms	0,4ms	1,6ms	3,6ms	0,25s	1s
n^3	1ms	8ms	64ms	0,2s	125s	17min
n^4	10ms	160ms	2,56s	13s	17h	11,6dní
2^n	1ms	1s	12,7 dní	36000 let		
$n!$	3,6s	77000 let				

Hledání v poli

- Sekvenční hledání v poli lze urychlit pomocí zarážky.
- Za předpokladu, že pole není zaplněno až do konce, uložíme do prvního volného prvku hledanou hodnotu a cyklus pak může být řízen jedinou podmínkou.
- **Sekvenční hledání se zarážkou**:

```
static int hledejSeZarazkou(int[] pole, int volny, int x){
  int i = 0;
  pole[volny] = x;                  // uložení zarážky
  while ( pole[i] != x ) i++;
  if ( i<volny ) return i;          // hodnota nalezena
  else return -1;                   // hodnota nenalezena
}
```

- Tak sice **ušetříme n=volny testů indexu**, avšak časová složitost zůstane O(*n*) a nejde tedy o významné urychlení.

Princip opakovaného půlení

- Pro některé problémy lze sestavit algoritmus založený na **principu opakovaného půlení**:
  - základem je cyklus, v němž se opakovaně zmenšuje rozsah dat na polovinu
  - časová složitost takového cyklu je logaritmická (dělíme-li n opakovaně 2, pak po $\lceil log_2(n) \rceil$ krocích dostaneme číslo menší nebo rovno 1)
- Při hledání prvku pole lze použít princip opakovaného půlení **v případě, že pole je seřazené**, tj. hodnoty jeho prvků tvoří monotonní posloupnost.
- **Hledání půlením** ve vzestupně seřazeném poli:
  - zjistíme hodnotu y prvku ležícího uprostřed zkoumaného úseku pole
  - je-li hledaná hodnota $x = y$, je prvek nalezen
  - je-li $x < y$, budeme hledat v levém úseku
  - je-li $x > y$, budeme hledat v pravém úseku
- Takovéto hledání se nazývá též **binární hledání** (binary search), **časová složitost je O(*log n*)**.

Binární hledání

- **Algoritmus binárního hledání**:

```
static int hledejBinarne(int[] pole, int x) {
   // metoda vrací index hledaného prvku x
   int dolni = 0;
   int horni = pole.length-1;
   int stred;
   while ( dolni<=horni ) {
      stred = (dolni+horni)/2;
      if ( x<pole[stred] ) horni = stred-1;     // nalevo
      else if (x>pole[stred]) dolni = stred +1;// napravo
      else return stred;                        // nalezen
   }
   return -1;   // nenalezen
}
```

Řazení pole

- Algoritmy řazení pole jsou algoritmy, které přeskupí prvky pole tak, aby upravené pole bylo seřazené.
- **Pole *p* je vzestupně seřazené**, jestliže platí

  $p[i-1] <= p[i]$ pro $i = 1 \ldots$ počet prvků pole – 1

  Pole *p* je sestupně seřazené, jestliže platí

  $p[i-1] >= p[i]$ pro $i = 1 \ldots$ počet prvků pole – 1

- Principy některých algoritmů řazení ukažme na řazení pole prvků typu *int.*

  Ukážeme si následující metody řazení pole:
  - `bubbleSort( )`
  - `selectSort( )`
  - `insertSort( )`
  - `mergeSort( )`

Řazení zaměňováním (BubbleSort)

- Při řazení zaměňováním postupně porovnáváme sousední prvky a pokud jejich hodnoty nejsou v požadované relaci, vyměníme je; to je třeba provést několikrát.
- **Hrubé řešení**:

```
for ( n=a.length-1; n>0; n-- )
   for ( i=0; i<n; i++ )
      if ( a[i]>a[i+1] ) // vyměň a[i] a a[i+1]
```

Jak budou seřazeny prvky pole?

- **Podrobné řešení**:

```
static void bubbleSort(int[] a) {
   int pom, n, i;
   for ( n=a.length-1; n>0; n-- )
      for ( i=0; i<n; i++ )
         if ( a[i]>a[i+1] )
            pom = a[i]; a[i] = a[i+1]; a[i+1] = pom;
}
```

- **Časová složitost je O(n^2)**

Řazení výběrem (SelectSort)

- **Při řazení výběrem se opakovaně hledá nejmenší prvek**
- **Hrubé řešení:**

```
for (i=0; i<a.length-1; i++) {
   "najdi nejmenší prvek mezi a[i] až a[a.length-1]";
   "vyměň hodnotu nalezeného prvku s a[i]";
}
```

- **Podrobné řešení:**

```
public static void selectSort(int[] a) {
   int i, j, imin, pom;
   for (i=0; i<a.length-1; i++) {
      imin = i;
      for (j=i+1; j<a.length; j++)
         if (a[j]<a[imin]) imin = j;
         if (imin!=i) {
            pom = a[imin]; a[imin] = a[i]; a[i] = pom;
      }
   }
}
```

- **Časová složitost algoritmu SelectSort: O(n^2)**

Řazení vkládáním (InsertSort)

- **Pole lze seřadit opakovaným vkládání prvku do seřazeného úseku pole**
- **Hrubé řešení:**

```
for (n=1; n<a.length; n++) {
   " úsek pole od a[0] do a[n-1] je seřazen "
   " vlož do tohoto úseku délky n hodnotu a[n] "
}
```

- **Podrobné řešení:**

```
private static void vloz(int[] a, int n, int x) {
   int i;
   for (i=n-1; i>=0 && a[i]>x; i--) a[i+1]=a[i]; // odsun
      a[i+1] = x; // vlozeni
}
public static void insertSort(int[] a) {
   for (int n=1; n<a.length ; n++)
      vloz(a, n, a[n]);
}
```

- **Časová složitost algoritmu InsertSort: O(n^2)**