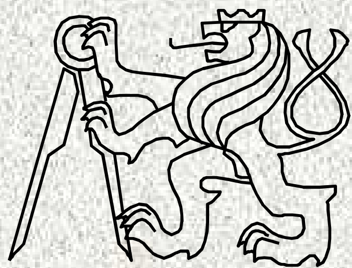
Hodnocení algoritmů, algoritmy řazení a třídění



A0B36PR1-Programování 1

Fakulta elektrotechnická
České vysoké učení technické

Algoritmy – přehled

- **Hledání**
  - Sekvenční hledání
  - Hledání se zarážkou
  - Binary Search – hledání binárním půlením

- **Řazení**
  - BubbleSort
  - SelectSort
  - InsertSort
  - QuickSort
  - MergeSort

Rychlost...

Jeden algoritmus (program, postup, metoda…)
je rychlejší než druhý.

Co ta věta znamená ??

Najdi min and max hodnotu v poli — STANDARD

min	max	a								
3	3	**3**	2	7	10	0	5	–10	4	6

min	max	a								
3	3	3	**2**	7	10	0	5	–10	4	6

```
if (a[i] < min) min = a[i];
if (a[i] > max) max = a[i];
```

min	max	a								
2	3	3	**2**	7	10	0	5	–10	4	6

Příklady

Najdi min and max hodnotu v poli — STANDARD

min	max
2	7

a

3	2	7	10	0	5	–10	4	6

atd...

hotovo

min	max
–10	10

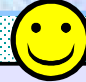
a

3	2	7	10	0	5	–10	4	6

kód

```
min = a[0]; max = a[0];
for ( i = 1; i < a.length; i++) {
   if (a[i] < min) min = a[i];
   if (a[i] > max) max = a[i]; }
```

Příklady



Najdi min and max hodnotu v poli — RYCHLEJI!

min	max
3	3

a

3	2	7	10	0	5	–10	4	6

min	max
3	3

a

3	2	7	10	0	5	–10	4	6

```
if (a[i] < a[i+1]) {
  if (   a[i] < min) min = a[i];
  if (a[i+1] > max) max = a[i+1];}
```

min	max
2	7

a

3	2	7	10	0	5	–10	4	6

Příklady

Najdi min and max hodnotu v poli — RYCHLEJI!

min max a

2	7		3	2	7	10	0	5	–10	4	6

```
if (a[i] < a[i+1]) {
   if (  a[i] < min) min = a[i];
   if (a[i+1] > max) max = a[i+1];}
else {
   if (  a[i] > max) max = a[i];
   if (a[i+1] < min) min = a[i+1];}
```

min max a

0	10		3	2	7	10	0	5	–10	4	6

Příklady

Najdi min and max hodnotu v poli — RYCHLEJI!

hotovo

min	max	a								
–10	10	3	2	7	10	0	5	–10	4	6

kód

```
min = a[0]; max = a[0];
for (i=1; i < a.length-1; i=i+2 ) {
  if (a[i] < a[i+1]) {
    if (  a[i] < min) min = a[i];
    if (a[i+1] > max) max = a[i+1];}
  else {
    if (  a[i] > max) max = a[i];
    if (a[i+1] < min) min = a[i+1];}}
```

Časová složitost algoritmů

- Důležitou vlastností algoritmu je, jakou časovou náročnost mají výpočty provedené podle daného algoritmu
- Časová náročnost výpočtů 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 přitom „měří“ počtem provedených operací, přičemž doba provedení každé operace nezávisí na rozsahu vstupních dat
- 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;
}
```

Časová složitost algoritmů

- Důležitou vlastností algoritmu je, jakou časovou náročnost mají výpočty provedené podle daného algoritmu
- Časová náročnost výpočtů 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 přitom „měří“ počtem provedených operací, přičemž doba provedení každé operace nezávisí na rozsahu vstupních dat
- 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;
}
```

- Budeme-li za operace považovat podtržené konstrukce, pak časová složitost je

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

  kde n je počet prvků pole

Počítání složitosti

Elementární operace

- aritmetická operace
- porovnání dvou čísel
- přesun čísla v paměti

Složitost A

celkový počet elementárních operací

zjednodušení

Složitost B

celkový počet elementárních operací nad daty

Počítání složitosti

Složitost (B) celkový počet elementárních operací nad daty

další zjednodušení

Složitost (C) celkový počet porovnání čísel (znaků) v datech

Takto složitost mnohdy počítáme

Počítání složitosti

Najdi min and max hodnotu v poli

```
min = a[0]; max = a[0];
for ( i = 1; i < a.length; i++){
   if (a[i] < min) min = a[i];
   if (a[i] > max) max = a[i]; }
```

Počítání složitosti

Najdi min and max hodnotu v poli

A

Složitost

Všechny operace

a.length = N

```
min = a[0]; max = a[0];
for ( i = 1; i < a.length; i++){
   if (a[i] < min) min = a[i];
   if (a[i] > max) max = a[i]; }
```

Počty operací: `min =` 1; `max =` 1; `i = 1` 1; `i < a.length` N; `i++` N–1; `a[i] < min` N–1; `min = a[i]` 0...N–1; `a[i] > max` N–1; `max = a[i]` 0...N–1

Případ

nejlepší: 1 + 1 + 1 + N + N–1 + N–1 + 0 + N–1 + 0 = 4N

nejhorší: 1 + 1 + 1 + N + N–1 + N–1 + N–1 + N–1 + N–1 = 6N–2

Počítání složitosti

Najdi min and max hodnotu v poli

složitost (B)

Operace nad daty

a.length = N

```
min = a[0]; max = a[0];
for ( i = 1; i < a.length; i++){
   if (a[i] < min) min = a[i];
   if (a[i] > max) max = a[i]; }
```

Počty operací: min = a[0] … 1; max = a[0] … 1; a[i] < min … N–1; min = a[i] … 0...N–1; a[i] > max … N–1; max = a[i] … 0...N–1

Případ

nejlepší	1 + 1 + N–1 +0 + N–1 + 0 = 2N
nejhorší	1 + 1 + N–1 + N–1 + N–1 + N–1 = 4N–2

Počítání složitosti

Najdi min and max hodnotu v poli

C

složitost

pouze testy v datech

a.length = N

```
min = a[0]; max = a[0];

for ( i = 1; i < a.length; i++){
   if (a[i] < min) min = a[i];
   if (a[i] > max) max = a[i]; }
```

N–1

N–1

vždy N–1 + N–1 = 2N–2 testů

Složitost algoritmů – doba výpočtu

Doba výpočtu pro různé časové složitosti
s předpokladem, že 1 operace trvá 1 μs (10^{-6} sec)

složitost výpočtu	**Počet operací, které musí výpočet provést**					
	10	**20**	**40**	**60**	**500**	**1000**
$\log_2 n$	3,3 μs	4,3 μs	5 μs	5,8 μs	9 μs	10 μs
n	10 μs	20 μs	40 μs	60 μs	0,5 ms	1 ms
$n \log_2 n$	33 μs	86 μs	0,2 ms	0,35 ms	4,5 ms	10 ms
n^2	0,1 ms	0,4 ms	1,6 ms	3,6 ms	0,25 s	1 s
n^3	1 ms	8 ms	64 ms	0,2 s	125 s	17 min
n^4	10 ms	160 ms	2,56 s	13 s	17 hod	11,6 dnů
2^n	1 ms	1 s	12,7 dnů	36000 let	10^{137} let	10^{287} let
$n!$	3,6 s	77000 let	10^{34} let	10^{68} let	10^{1110} let	10^{2554} let

Průběhy doby výpočtu

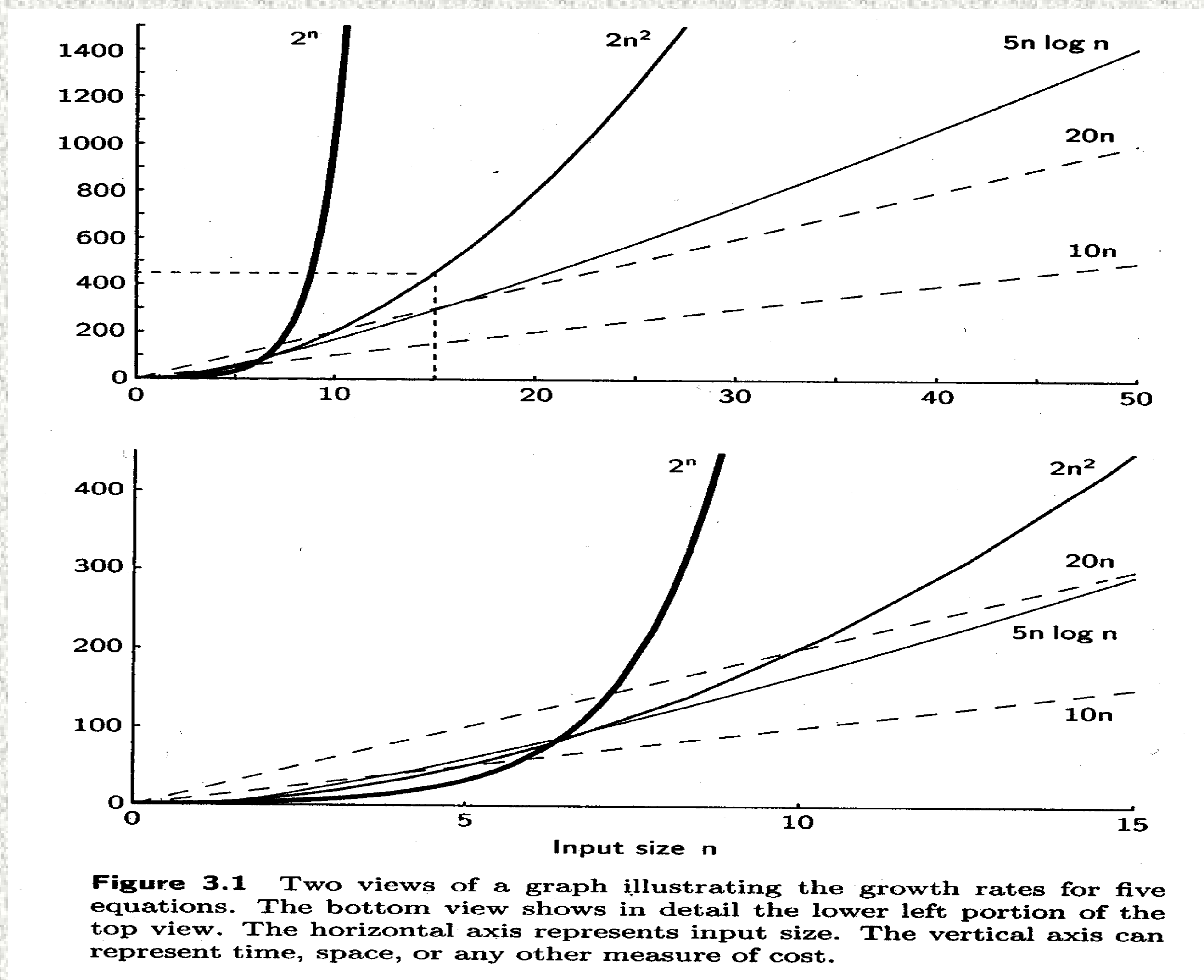


Figure 3.1 Two views of a graph illustrating the growth rates for five equations. The bottom view shows in detail the lower left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

Počítání složitosti

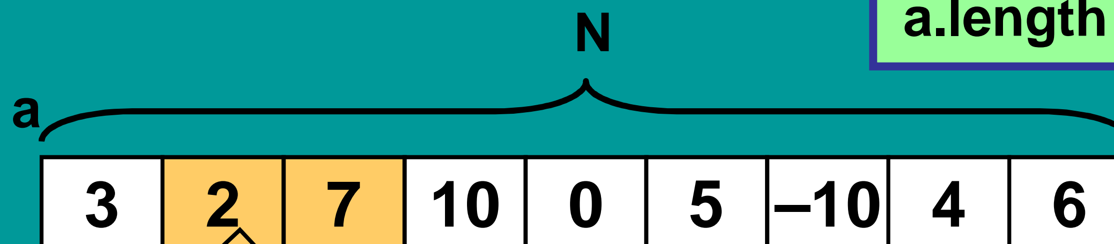
Najdi min and max hodnotu v poli — RYCHLEJI!

C

složitost

pouze testy v datech

a.length = N



Jedna dvojice — 3 testy

(N–1)/2 dvojic

vždy 3(N–1)/2 = (3N – 3)/2 testů

Počítání složitosti

Velikost pole **N**	Počet testů STANDARDNÍ **2 (N – 1)**	Počet testů RYCHLEJŠÍ **(3N – 3)/2**	poměr STD./RYCHL.
11	20	15	1.33
21	40	30	1.33
51	100	75	1.33
101	200	150	1.33
201	400	300	1.33
501	1 000	750	1.33
1 001	2 000	1 500	1.33
2 001	4 000	3 000	1.33
5 001	10 000	7 500	1.33
1 000 001	2 000 000	1 500 000	1.33

Tab. 1

Příklady

data

pole a:

1	–1	0	–2	5	1	0

pole b:

4	2	4	3	4	2	7

úloha

Kolik prvků pole b je rovno součtu prvků pole a?

řešení

pole a:

1	–1	0	–2	5	1	0

součet = 4

pole b:

4	2	4	3	4	2	7

výsledek = 3

Příklady

funkce

```
int sumArr(int[] a) { /*snadné*/ }
```

POMALÁ metoda

```
count = 0;
for (int i = 0; i < b.length; i++)
   if (b[i]==sumArr(a)) count++;
return count;
```

RYCHLÁ metoda

```
count = 0;
sumOf_b = sumArr(a);
for (int i = 0; i < b.length; i++)
   if (b[i]==sumOf_b) count++;
return count;
```

Počítání složitosti

POMALÁ metoda

a.length == n
b.length == n

pole a:	1	–1	0	–2	5	1	0
pole b:	4	2	4	3	4	2	7

≈ n x n = n^2 operací

Kvadratická složitost

Prvek pole b se porovnává se součtem celého pole a, který se počítá pokaždé znovu

Počítání složitosti

RYCHLÁ metoda

pole a:	1	–1	0	–2	5	1	0

součet a: 4

a.length == n
b.length == n

≈ 2 x n operací

Lineární složitost

pole b:	4	2	4	3	4	2	7

Prvek pole b se porovnává se součtem celého pole a, který byl spočítán pouze jednou.

Počítání složitosti

Velikost pole **N**	POMALÁ metoda operací **N^2**	RYCHLÁ metoda operací **2N**	poměr POMALÁ/RYCHLÁ
11	121	22	5.5
21	441	42	10 .5
51	2 601	102	25 .5
101	10 201	202	50 .5
201	40 401	402	100 .5
501	251 001	1 002	250 .5
1 001	1 002 001	2 002	500 .5
2 001	4 004 001	4 002	1 000 .5
5 001	25 010 001	10 002	2 500 .5
1 000 001	1 000 002 000 001	2 000 002	1 000 000 .5

Tab. 2

Počítání složitosti

Velikost pole **N**	Poměr rychlostí řešení 1. úlohy	Poměr rychlostí řešení 2. úlohy
11	1.33	5.5
21	1.33	10 .5
51	1.33	25 .5
101	1.33	50 .5
201	1.33	100 .5
501	1.33	250 .5
1 001	1.33	500 .5
2 001	1.33	1 000 .5
5 001	1.33	2 500 .5
1 000 001	1.33	1 000 000 .5

Tab. 3

Příklady

Hledání v seřazeném poli — lineární, POMALÉ

dané pole

Seřazené pole: → **velikost = N**

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

najdi 993 !

testů: N ☹

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

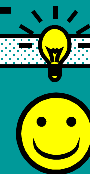
najdi 363 !

testů: 1 ☺

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

Příklady

Hledání v seřazeném poli — binární, RYCHLÉ



najdi 863 !

363	369	388	603	638	693	803	833	836	839	860	863	938	939	966	968	983	993

363	369	388	603	638	693	803	833

839	860	863	938	939	966	968	983	993

2 testy

839	860	863	938	939	966	968	983	993

2 testy

839	860	863	938

966	968	983	993

839	860	863	938

2 testy

839

863	938

1 test

863	938

Exponent, logarimus a půlení intervalu

k — N — $N = 2^k$

…

3 — 8 — $8 = 2^3$

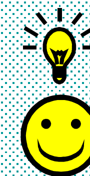
2 — 4 — $4 = 2^2$

1 — 2 — $2 = 2^1$

0 — 1 — $1 = 2^0$

$$N = 2^k \Rightarrow$$

$$k = \log_2(N)$$



Počítání složitosti

Velikost pole	Počet testů: lineární hledání — případ: nejlepší	nejhorší	průměrný	binární hledání nejhorší případ	poměr
5	1	5	3	5	0.6
10	1	10	5.5	7	0.79
20	1	20	10.5	9	1.17
50	1	50	25.5	11	2.32
100	1	100	50.5	13	3.88
200	1	200	100.5	15	6.70
500	1	500	250.5	17	14.74
1 000	1	1000	500.5	19	26.34
2 000	1	2000	1000.5	21	47.64
5 000	1	5000	2500.5	25	100.02
1 000 000	1	1 000 000	500 000.5	59	8 474.58

Tab. 4

Průběhy doby výpočtu

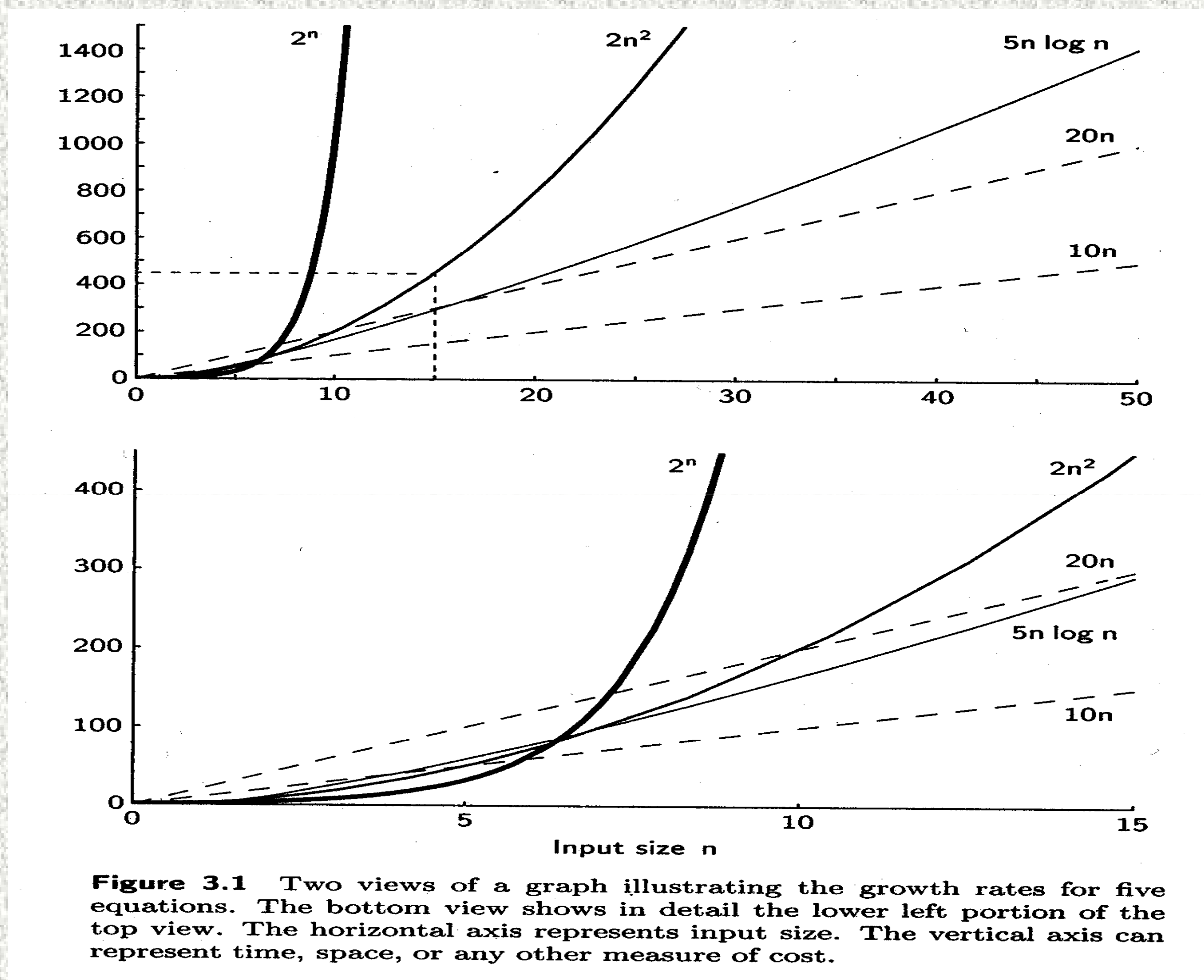


Figure 3.1 Two views of a graph illustrating the growth rates for five equations. The bottom view shows in detail the lower left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

Časová složitost algoritmů II

- Přesné určení počtu operací při analýze složitosti algoritmu je často velmi složité
- Zvlášť komplikované (nebo i nemožné) bývá určení počtu operací v průměrném případě; proto se většinou omezujeme jen na analýzu nejhoršího případu
- Zpravidla nás ani 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 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

$$f(n) \le K.g(n) \quad \text{pro všechna } n > n_1$$

Asymptotická složitost hledání sekvenčně

Postupné prohledávání

```
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
    $C_{prum}(n) = 2.5 + 1.5n$

Asymptotická složitost je O(n), tj. čas pro zpracování n položek je přímo úměrný počtu položek n

Asymptotická složitost sekvenční se zarážkou

- Sekvenční hledání v poli lze urychlit pomocí zarážky
- Za předpokladu, že pole není zaplněno až do posledního prvku, 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, hledaná hodnota
   while (pole[i]!=x){ // místo (pole[i]!=x && i<volny)
     i++;
   }
   if (i<volny) // hodnota nalezena
     return i;
   else         // hodnota nenalezena, došly jsme až k
   zarážce
     return -1;
}
```

- Asymptotická složitost je O(*n*), nejde tedy o významné urychlení

Asymptotická složitost binárního hledání

- 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 prohledávaného úseku pole
  - jestliže hledaná hodnota $x = y$, je prvek nalezen
  - jestliže $x < y$, pokračujeme v hledání v levém úseku
  - jestliže $x > y$, pokračujeme v hledání v pravém úseku

- Hledání prvku pole půlením se nazývá též binární hledání (binary search), asymptotická složitost je O(*log n*)

Binární půlení

- Algoritmus hledání binárním půlením:

```
static int hledejBinarne(int[] pole, int 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;
    else if (x>pole[stred]) dolni = stred +1;
    else return stred;
  }
  return -1;
}
```

Asymptotická složitost je **O(*log n*)**

Zajímavý problém

- Je dána posloupnost celých čísel A_1, A_2, ... A_n. Máme najít takovou její podposloupnost A_i až A_j , jejíž prvky dávají největší kladný součet ze všech ostatních podposloupností
- Příklad:
  - pro {-2, **11**, **-4**, **13**, -5, 2} je výsledkem *i*=2, *j*=4, *soucet*=20
  - pro {1, -3, **4**, **-2**, **-1**, **6**} je výsledkem *i*=3, *j*=6, *soucet*=7
  - pro {-1, -3, -5, -7, -2} je výsledkem *i*=0, *j*=0, *soucet*=0
- Pro řešení tohoto problému lze sestavit několik, méně či více efektivních algoritmů
- Poznámka: čísla A_1, A_2, ... A_n budou uložena v prvcích pole *a*[0], *a*[1], ... *a*[*n*-1], kde *n* je velikost pole *a*

Zajímavý problém - úvaha

Příklad

- 2 2 2 -5 3 3 3 i=1 j=7 součet=10
- 2 2 2 -6 3 3 3 i=1 j=7 součet=9
- 2 2 2 -7 3 3 3 i=5 j=7 součet=9
- 2 2 2 -8 3 3 3 i=5 j=7 součet=9

Řešení hrubou silou

- Nejjednodušší (a nejméně efektivní) je algoritmus, který postupně probere všechny možné podposloupnosti, zjistí součet jejich prvků a vybere tu, která má největší součet

```
static int maxSoucet(int[] a) {
   int maxSum = 0;
   for (int i=0; i<a.length; i++)
     for (int j=i; j<a.length; j++) {
       int sum = 0;
       for (int k=i; k<=j; k++)
         sum += a[k];
       if (sum>maxSum) {
         maxSum = sum;
         prvni = i;
         posledni = j;
       }
     }
   return maxSum;
 }
```

- Poznámka: proměnné *prvni* a *posledni* jsou nelokálními proměnnými
- Časová složitost tohoto algoritmu je **O(n^3) (kubická)**

Řešení s nápadem č. 1

- Vnitřní cyklus (proměnná *k*) počítající součet $S_{i,j} = a[i]+...+a[j]$ je zbytečný: známe-li součet $S_{i,j-1} = a[i]+...+a[j-1]$, pak $S_{i,j} = S_{i,j-1}+ a[j]$

```
static int maxSoucet(int[] a) {
  int maxSum = 0;
  for (int i=0; i<a.length; i++) {
    int sum = 0;
    for (int j=i; j<a.length; j++) {
      sum += a[j];
      if (sum>maxSum) {
        maxSum = sum;
        prvni = i;
        posledni = j;
      }
    }
  }
  return maxSum;
}
```

- **Časová složitost tohoto algoritmu je O(n^2)**

Řešení s nápadem č. 2

- Řešení lze sestavit s použitím jediného cyklu s řídicí proměnnou *j*, která udává index posledního prvku podposloupnosti
- Proměnná *i* udávající index prvního prvku podposloupnosti se bude měnit takto:
  - počáteční hodnotou *i* je 0
  - postupně zvětšujeme *j* a je-li součet podposloupnosti od *i* do *j* (sum) větší, než doposud největší součet (*sumMax*), zaznamenáme to (*prvni=i, posledni=j, sumMax=sum*)
  - vznikne-li však zvětšením *j* podposloupnost, jejíž součet je záporný, pak žádná další podposloupnost začínající indexem *i* a končící indexem *j*1, kde *j*1>*j*, nemůže mít součet větší, než je zaznamenán; hodnotu proměnné *i* je proto možné nastavit na *j*+1 a *sum* na 0

Řešení s lineární složitostí

```
static int maxSoucet(int[] a) {
    int maxSum = 0;
    int sum = 0, i = 0;
    for (int j=0; j<a.length; j++) {
      sum += a[j];
      if(sum > maxSum) {
        maxSum = sum;
        prvni = i;
        posledni = j;
      }
      else if(sum < 0) {
        i = j + 1;
        sum = 0;
      }
    }
    return maxSum;
  }
```

Řešení s lineární složitostí

```
static int maxSoucet(int[] a) {
    int maxSum = 0;
    int sum = 0, i = 0;
    for (int j=0; j<a.length; j++) {
      sum += a[j];
      if(sum > maxSum) {
        maxSum = sum;
        prvni = i;
        posledni = j;
      }
      else if(sum < 0) {
        i = j + 1;
        sum = 0;
      }
    }
    return maxSum;
  }
```

Řazení zaměňováním (BubbleSort - probublávání)

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]";
```

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 **$O(n^2)$**, tj. čas pro zpracování n položek je úměrný kvadrátu počtu položek n

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]";
  }
```

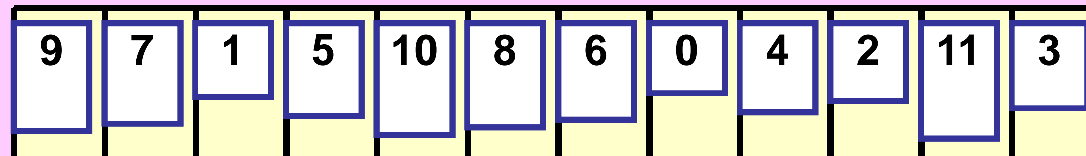
Podrobné řešení

```
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;
  }}}
```

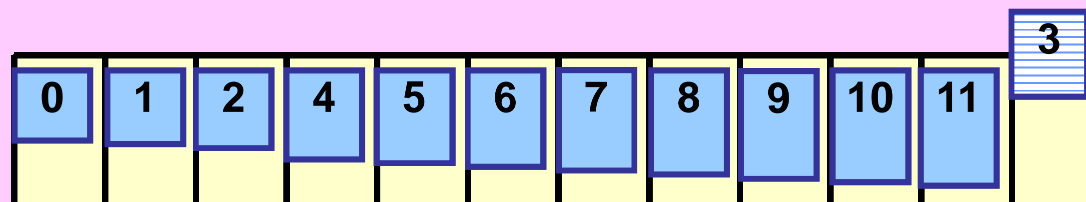
Časová složitost algoritmu SelectSort: $O(n^2)$

Ilustrace řazení vkládáním

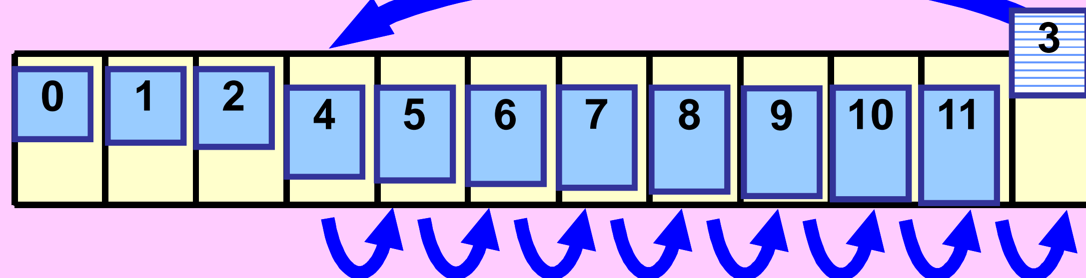
start



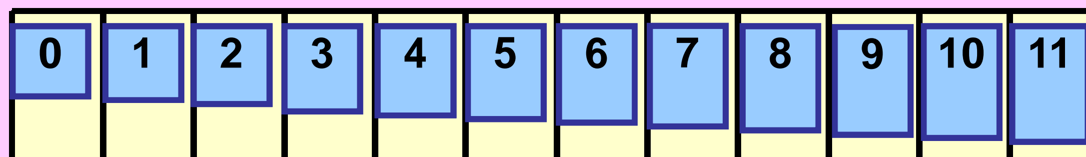
úvaha



vložení



výsledek



Řazení vkládáním (InsertSort)

Pole lze seřadit opakováním algoritmu vložení 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];
   a[i+1] = x;
}
```

```
public static void insertSort(int[] a) {
   for (int n=1; n<a.length ; n++)
      vloz(a, n, a[n]);
}
```

Časová složitost algoritmu InsertSort: $O(n^2)$

Slučování (Merging)

- Problém slučování lze obecně formulovat takto:
  - ze dvou seřazených (monotónních) (!) posloupností *a* a *b* máme vytvořit novou posloupnost obsahující všechny prvky z *a* a *b*, která je rovněž seřazená
- Příklad:

a:	2 3 6 8 10 34
b:	3 7 12 13 55
výsledek:	2 3 3 6 7 8 10 12 13 34 55

Poznámka:

Neefektivní řešení: Vytvoříme pole, do něhož zkopírujeme prvky a, přidáme prvky b a pak seřadíme - to není slučování!

Princip

Slučování

- Princip slučování:
  1. postupně porovnáváme prvky zdrojových posloupností a do výsledné posloupnosti přesouváme menší z nich
  2. nakonec zkopírujeme do výsledné posloupnosti zbytek první nebo druhé posloupnosti

```
static int[] slucPole(int[] a, int[] b) {
  int[] c = new int[a.length+b.length];
  int ia = 0, ib = 0, ic = 0;
  while (ia<a.length && ib<b.length)
    if (a[ia]<b[ib])c[ic++] = a[ia++];
    else c[ic++] = b[ib++];
  while (ia<a.length) c[ic++] = a[ia++];
  while (ib<b.length) c[ic++] = b[ib++];
  return c;
}
```

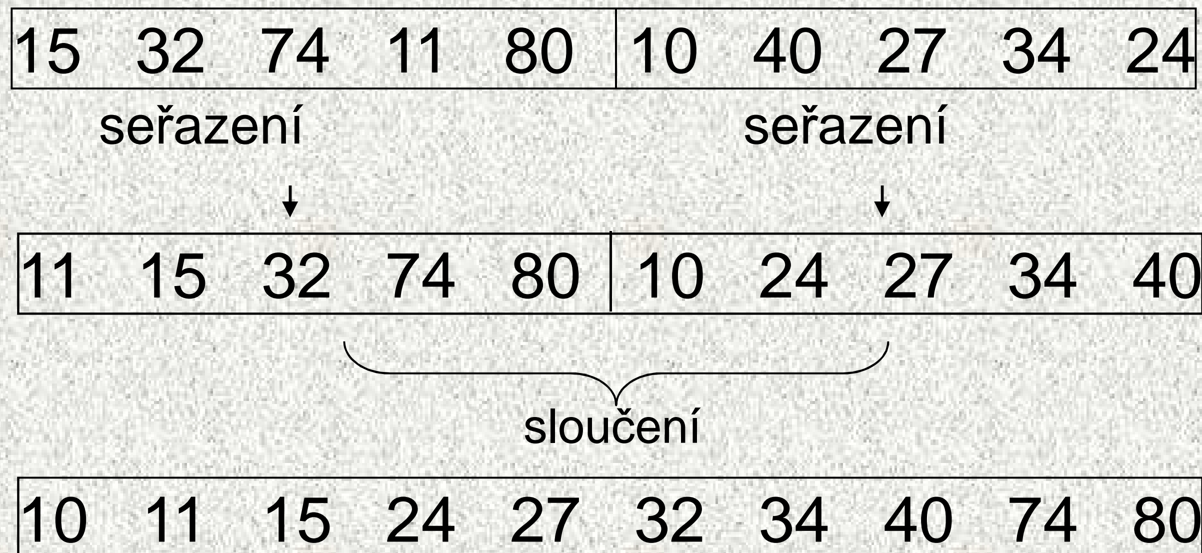
Řazení slučováním (MergeSort)

Efektivnější algoritmy řazení mají časovou složitost O(*n log n*)

Jedním z nich je algoritmus řazení slučováním, který je založen na opakovaném slučování seřazených úseků do úseků větší délky

Lze jej popsat rekurzívně:

- řazený úsek pole rozděl na dvě části
- seřaď levý úsek a pravý úsek
- přepiš řazený úsek pole sloučením levého a pravého úseku



Princip

Sloučení dvou seřazených úseků pole

- Funkce, která sloučí dva sousední již seřazené úseky pole *a* a výsledek uloží do pole *b*:

```
private static void merge(int[] a, int[] b,
                           int levy, int pravy,
                           int poslPravy) {
  int poslLevy = pravy-1;
  int i = levy;
  while (levy<=poslLevy && pravy<=poslPravy)
    if (a[levy]<a[pravy])
      b[i++] = a[levy++];
    else
      b[i++] = a[pravy++];
  while (levy<=poslLevy) b[i++] = a[levy++];
  while (pravy<=poslPravy) b[i++] = a[pravy++];
}
```


Princip

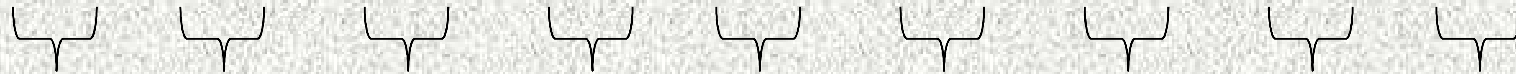
Nerekurzívní MergeSort

- Nerekurzívní (iterační) algoritmus MergeSort postupuje zdola nahoru:
  - pole *a* se rozdělí na dvojice úseků délky 1, které se sloučí do seřazených úseků délky 2 v pomocném poli *pom*
  - dvojice sousedních seřazených úseků délky 2 v poli *pom* se sloučí do seřazených úseků délky 4 v poli *a*
  - dvojice sousedních úseků délky 4 v poli *a* se sloučí do seřazených úseků délky 8 v poli *pom*
  - atd.

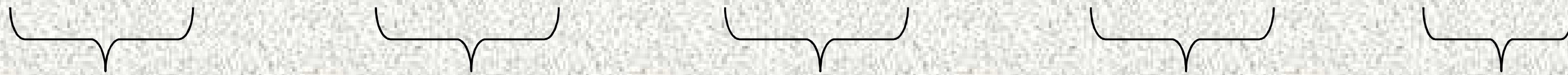
  - tento postup se opakuje, pokud délka úseku je menší než velikost pole
  - skončí-li slučování tak, že výsledek je v pomocném poli *pom*, je třeba jej zkopírovat do původního pole *a*

Příklad řazení slučováním zdola

a 49 62 |21 70 |89 99 |21 76 |53 40 |87 70 |32 70 |24 93 |90 65 |90



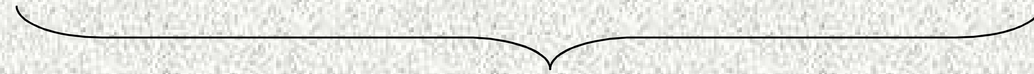
pom 49 62 21 70 |89 99 21 76 |40 53 70 87 |32 70 24 93 |65 90 90



a 21 49 62 70 21 76 89 99 |40 53 70 87 24 32 70 93 |65 90 90

pom 21 21 49 62 70 76 89 99 24 32 40 53 70 70 87 93 |65 90 90

a 21 21 24 32 40 49 53 62 70 70 70 76 87 89 93 99 65 90 90



pom 21 21 24 32 40 49 53 62 65 70 70 70 76 87 89 90 90 93 99

Nerekurzívní MergeSort

```
public static void mergeSort(int[] a) {
int[] pom = new int[a.length];
int[] odkud = a;   int[] kam = pom;
int delkaUseku = 1;   int posl = a.length-1;
while (delkaUseku<a.length) {
 int levy = 0;
 while (levy<=posl) {
  int pravy = levy+delkaUseku;
  merge(odkud, kam, levy,Math.min(pravy, a.length),
                Math.min(pravy+delkaUseku-1, posl));
  levy = levy+2*delkaUseku;
        }
 delkaUseku = 2*delkaUseku;
 int[] p = odkud; odkud = kam; kam = p;
 }
if (odkud!=a)
 for (int i=0; i<a.length; i++) a[i] = pom[i];
  }
```

Rekurzívní MergeSort

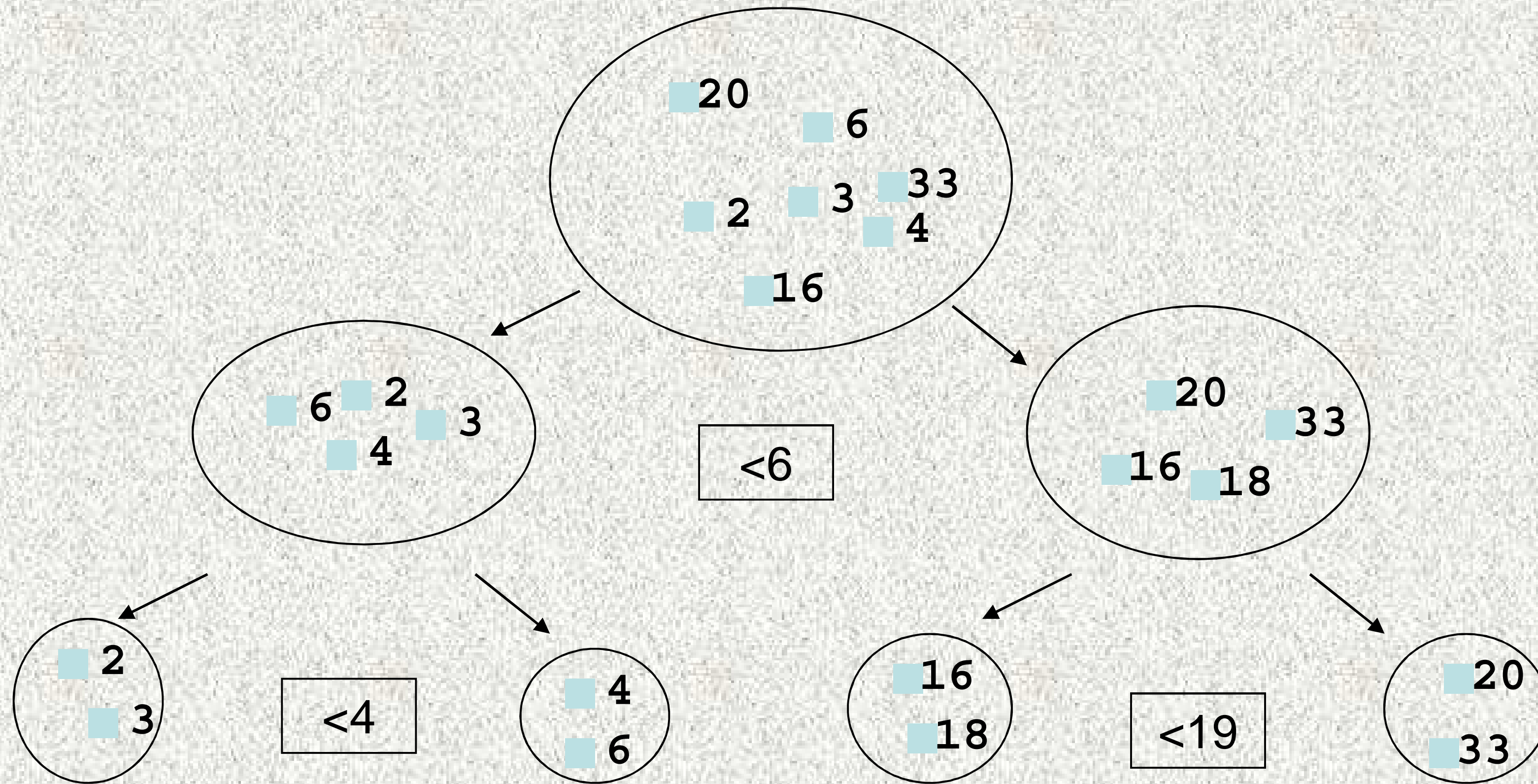
- Rekurzívní (iterační) algoritmus MergeSort postupuje shora dolu:
  - Rekurzívní MergeSort se volá rekurzívně tak dlouho, dokud délka úseku pole není 1
  - Pak začne slučování sousedních úseků do dvakrát většího úseku v pomocném poli, který je třeba zkopírovat do původního pole
  - Rozdělení pole na úseky, které se postupně slučují, je dáno postupným půlením úseků pole shora dolů.

Rekurzivní řazení slučováním MergeSort

- Rekurzivní funkce řazení úseku pole:

```
private static void mergeSort(int[] a, int[] pom, int prvni,
  int posl) {
      if (prvni<posl) {
        int stred = (prvni+posl)/2;
        mergeSort(a, pom, prvni, stred);
        mergeSort(a, pom, stred+1, posl);
        merge(a, pom, prvni, stred+1, posl);
        for (int i=prvni; i<=posl; i++) a[i] = pom[i];
      }
    }
public static void mergeSort(int[] a) {
      int[] pom = new int[a.length];
      mergeSort(a, pom, 0, a.length-1);
    }
```

QuickSort



QuickSort I

```
class Quick {
  static void quickSort (int[] a, int
   left, int right){
    if(left <  right){
      int p = castecne(a, left, right);
      quickSort(a, left, p-1);
      quickSort(a, p+1, right);
    }
  }
```

```
private static int castecne(int[] a, int left, int right){
    int pivot = a[left]; int p = left;
    for (int r = left + 1; r <= right; r++) {
      if(a[r] < pivot){
        a[p] = a[r]; a[r] = a[p+1]; a[p+1]= pivot;
        p++;
      }
    }
    return p;
  }
```

33	60	40	20	10

20	10	33	60	40

QuickSort II

```
public static void main(String[] args) {
    int[] pole={22, 33, 14, 91, 3 , 51 , 6, 24, 7, 44,
  5};
    for (int i = 0; i < pole.length; i++) {
      System.out.print(pole[i] + " ");
    }
    System.out.print ();
      quickSort(pole, 0, 10);
      for (int i = 0; i < pole.length; i++) {
      System.out.print (pole[i] + " ");
    }
}
```

```
22 33 14 91 3 51 6 24 7 44 5
3 5 6 7 14 22 24 33 44 51 91
```

- Časová složitost algoritmu QuickSort:
  - **průměrně(*n* log n), až O(n^2)**
- Prostorová složitost algoritmu QuickSort: **O(n)**
- Viz http://en.wikipedia.org/wiki/Quicksort

Přihrádkové řazení

1. průchod 223 131 458 193 756 812 027 579 283 200

rozdělení podle poslední číslice

0	1	2	3	4	5	6	7	8	9
			283						
			193						
200	131	812	223			756	027	458	579

2. průchod 200 131 812 223 193 283 756 027 458 579

rozdělení podle druhé číslice

0	1	2	3	4	5	6	7	8	9
		027			458				
200	812	223	131		756		579	283	193

Pro zájemce

3. průchod 200 812 223 027 131 756 458 579 283 193

rozdělení podle první číslice

0	1	2	3	4	5	6	7	8	9
		283							
	193	223							
027	131	200		458	579		756	812	

Výstup: 027 131 193 200 223 283 458 579 756 812

Přihrádkové řazení

```
static int hexCislice(int x, int r) {
    return (x >> 4*r) & 0xF;
}
static void caseSort(int[] pole) {
FrontaCisel[] prihradky = new FrontaCisel[16];
int r, j, c;
for ( c=0; c<prihradky.length; c++ )prihradky[c] = new
  FrontaCisel();
for ( r=0; r<8; r++ ) {
  for ( j=0; j<pole.length; j++ )
   prihradky[hexCislice(pole[j],r)].vloz(pole[j]);
   j = 0;
     for ( c=0; c<prihradky.length; c++ )
        while ( !prihradky[c].jePrazdna() )
          pole[j++] = prihradky[c].odeber();
} }
```