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

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 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

  $f(n) \leq K.g(n)$ 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 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
  while (pole[i]!=x) i++;
  if (i<volny) // hodnota nalezena
    return i;
  else         // hodnota nenalezena
    return -1;
}
```

- Časová složitost je O(*n*), 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 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), časová složitost je O(*log n*)

Binární hledání

- Algoritmus binárního hledání:

```
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;
}
```

Řazení pole

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

  p[*i*-1] <= *p*[*i*] pro *i* = 1,...,počet prvků pole – 1
- Pole *p* v jazyku Java je sestupně seřazené, jestliže platí

  p[*i*-1] >= *p*[*i*] pro *i* = 1,...,počet prvků pole – 1
- Principy některých algoritmů řazení ukážeme na řazení pole prvků typu *int* a budeme je prezentovat jako funkce, které vzestupně seřadí všechny prvky pole daného parametrem

Řazení zaměňováním

- 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]) "výměň a[i] a a[i+1]"
```

- Podrobné řešení (BubbleSort]:

```
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

- 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 (SelectSort):

```
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

- 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 (InsertSort):

```
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

- Problém slučování (merging) lze obecně formulovat takto:
  - ze dvou seřazených (monotonních) 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
- Jsou-li posloupnosti uloženy ve dvou polích, můžeme algoritmus slučování v jazyku Java zapsat jako funkci

  ```
  static int[] slucPole(int[] a, int[] b)
  ```
- Neefektivní řešení:
  - vytvoříme pole, do něhož zkopírujeme prvky *a*, přidáme prvky *b* a pak seřadíme

  To není slučování!

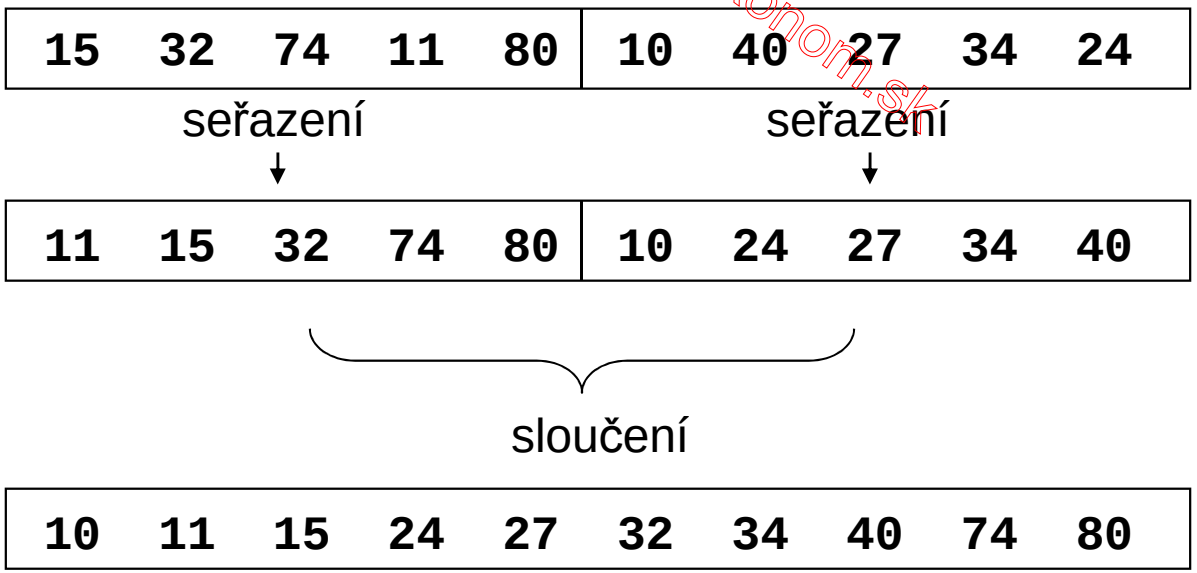
Slučování

- Princip slučování:
  - postupně porovnáváme prvky zdrojových posloupností a do výsledné posloupnosti přesouváme menší z nich
  - 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

- Efektivnější algoritmy řazení mají časovou složitost O(*n log n*)
- Jedním z nich je algoritmus řazení slučováním (MergeSort), 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



Sloučení dvou seřazených úseků pole

- Funkce, která sloučí dva sousední 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;
  int prvniLevy = 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++];
}
```

Rekurzivní 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];
  }
}
```

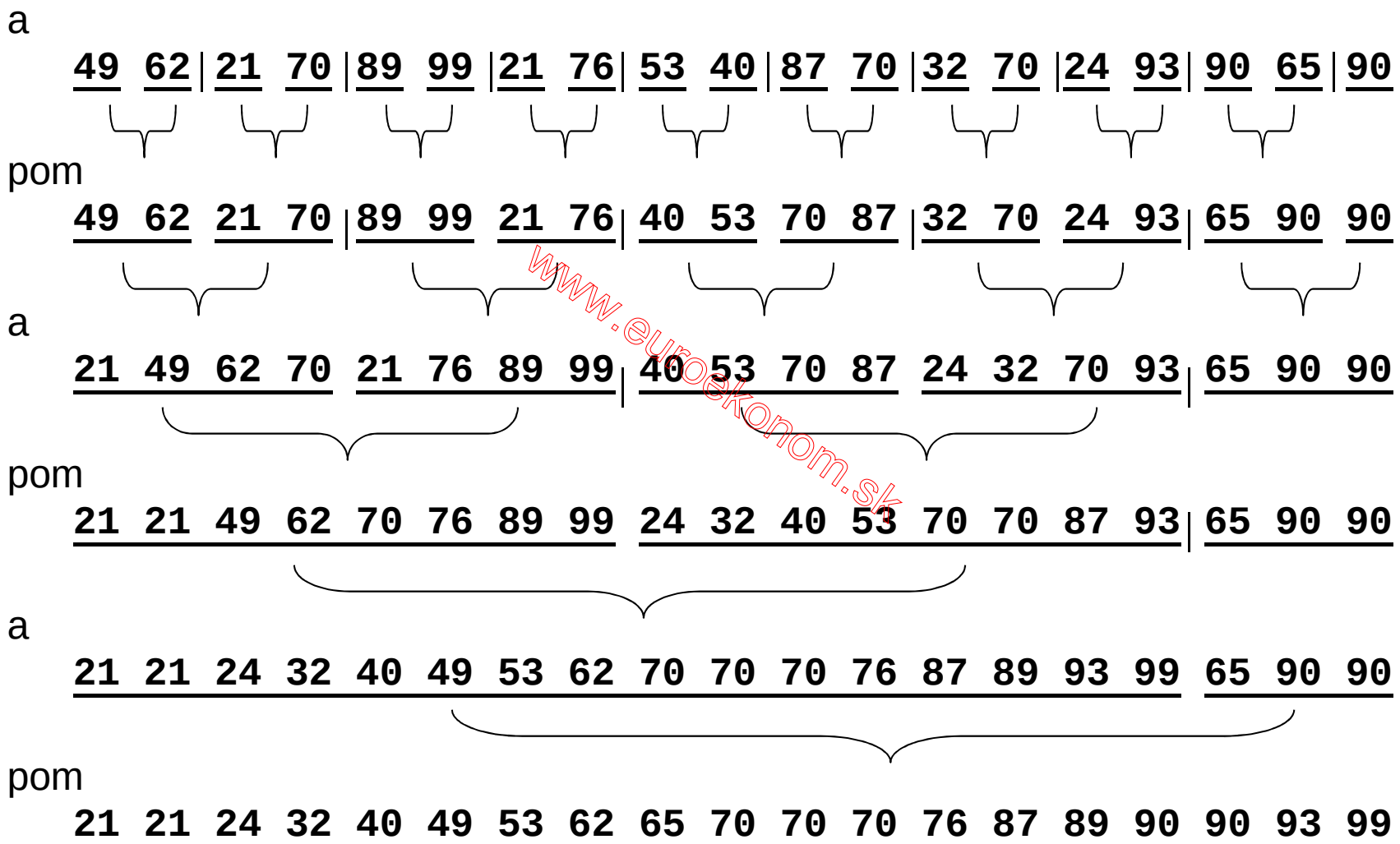
- Výsledná funkce:

```
public static void mergeSort(int[] a) {
  int[] pom = new int[a.length];
  mergeSort(a, pom, 0, a.length-1);
}
```

Nerekurzívní MergeSort

- Rekurzívní MergeSort se volá rekurzívně tak dlouho, dokud délka úseku pole není 1; teprve 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.
- Nerekurzívní (iterační) algoritmus MersgeSort 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*
- Ilustrujme tento postup na příkladu

Příklad řazení slučováním zdola



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];
 }
```

Problém pro zájemce

- 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 *í*=2, *j*=4, *soucet*=20
  - pro {1, -3, **4**, **-2**, **-1**, **6**} je výsledkem *í*=3, *j*=6, *soucet*=7
  - pro {-1, -3, -5, -7, -2} je výsledkem *í*=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*

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 kvadratickou složitostí

- 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 lineární složitostí

- 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;
 }
```