11. přednáška

- Obsah – složitost algoritmů
  - Mergesort
  - Problém hledání optimální podposloupnosti

Slučování (merging)

- **Problém slučování** lze obecně formulovat takto:
  - ze dvou seřazených (monotonních) posloupností *a* a *b* máme vytvořit novou posloupnost obsahující všechny prvky z *a* i *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

  Ale to není slučování!

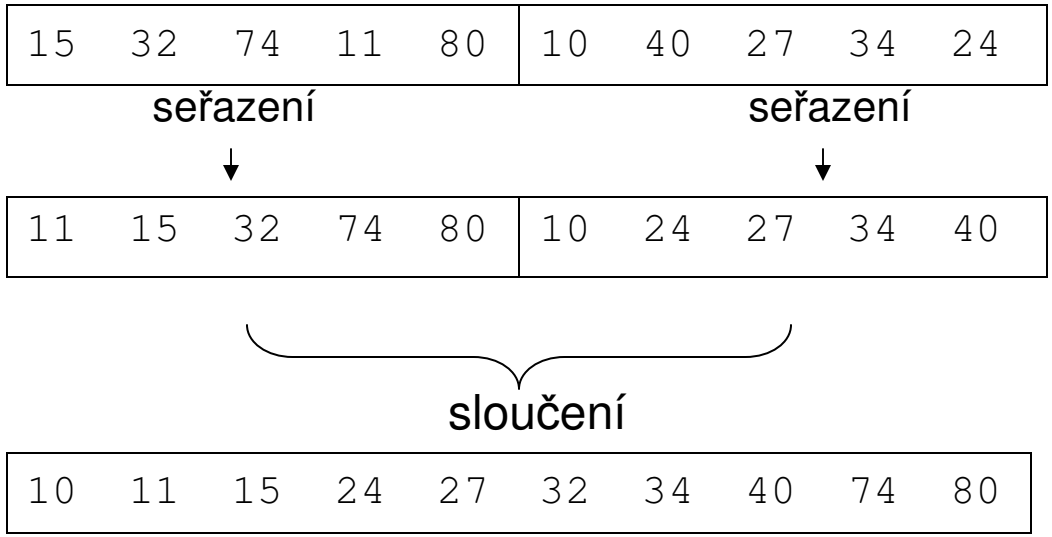
Slučování

- **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];    // vysledné pole
   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

- **Metoda slučující dva sousední seřazené úseky pole *a*, výsledek uloží do pole *b*.**
  Úseky mohou mit i nestejnou délku.

```
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 )
    b[i++] = (a[levy]<a[pravy])? a[levy++] :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 ) {                    // dokud je co dělit
    int stred = (prvni+posl)/2;          // rozděl
    mergeSort(a, pom, prvni, stred);     // a panuj
    mergeSort(a, pom, stred+1, posl);
    merge(a, pom, prvni, stred+1, posl); // sluč mezivýsledky
    for (int i=prvni; i<=posl; i++) a[i] = pom[i];
  }
}
```

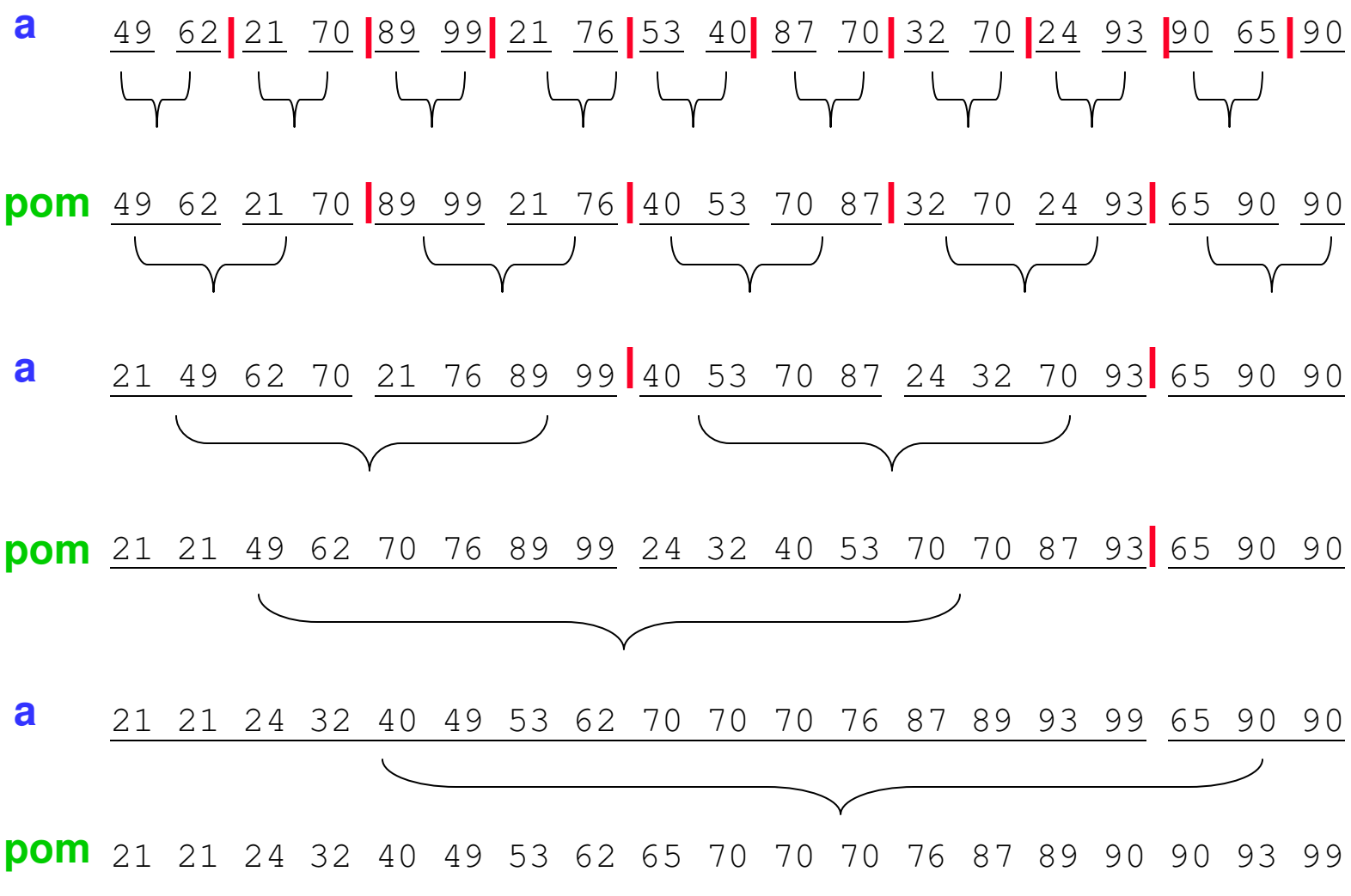
- **Výsledná funkce**:

```
public static void mainMergeSort(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; 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*

Příklad řazení slučováním zdola



Nerekurzívní MergeSort

```
public static void mergeSort(int[] a) {
    int[] pom = new int[a.length],odkud = a,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 podposloupnosti

- Je dána posloupnost čísel { a_0, a_1, ... a_{n-1} }.

  Máme najít zleva první její podposloupnost { a_i , ... a_j } kde i<=j, jejíž prvky dávají největší **kladný** součet ze všech ostatních podposloupností.

  (Nuly zleva ani z prava do podposloupnosti nezapočítávejme.)

- Příklad:

pro	je výsledek
{ -2, 11, -4, 13, -5, 2 }	*í*=1, *j*=3, *součet*=20
{ 1, -3, 4, -2, -1, 6 }	*í*=2, *j*=5, *součet*=7
{ 0, 1, 0, 1, 0, -3, 1, 1, -3, 2 }	*í*=1, *j*=3, *součet*=2
{ }	neexistuje, *součet*=0
{ -1, -3, 0, 0, -5, -7, -2 }	neexistuje, *součet*=0

- Pro řešení tohoto problému existují různě efektivní algoritmy

pro zájemce

Řešení hrubou silou

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

```
static int maxSum = 0, prvni= -1, posledni = -1 ;
   static int maxSoucet(int[] a) {
      for (int i=0; i<a.length; i++)              // 1.cyklus
        for (int j=i; j<a.length; j++) {          // 2.cyklus
          int sum = 0;
          for (int k=i; k<=j; k++) sum += a[k]; // 3.cyklus
          if (sum > maxSum) {                     // nove max.
            maxSum = sum;  prvni = i; posledni = j;
          }
        }
      return maxSum;
    }
```

- **Časová složitost tohoto algoritmu je O(n^3) (kubická)**

pro zájemce

Řešení s kvadratickou složitostí

- **3. cyklus** s proměnnou *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 maxSum = 0, prvni= -1, posledni = -1 ;
    static int maxSoucet(int[] a) {
      int maxSum = 0;
      for (int i=0; i<a.length; i++) {            // 1.cyklus
        int sum = 0;
        for (int j=i; j<a.length; j++) {          // 2.cyklus
          sum += a[j];
          if (sum > maxSum) {                     // nove max.
            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

pro zájemce

Řešení s lineární složitostí

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

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