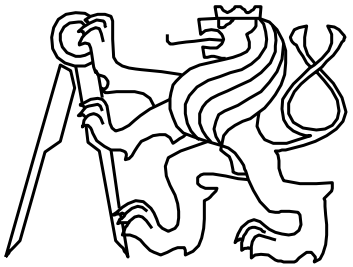
Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.
Praha & EU: Investujeme do vaší budoucnosti

Časová složitost algoritmů, řazení a vyhledávání



BI-PA1 Programování a algoritmizace 1, ZS 2012-2013
Katedra teoretické informatiky
© Miroslav Balík
Fakulta informačních technologií
České vysoké učení technické

Obsah

- časová složitost algoritmů
- algoritmy vyhledávání v poli
- algoritmy řazení pole s kvadratickou složitostí
- algoritmus slučování (merging)
- algoritmus řazení slučováním (MergeSort)
- problém podposloupnosti s největším kladným součtem (pro zájemce)

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

```
int soucet(int pole[], int n) {
  int s = 0, i;
  for (i=0; i<n; 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

```
int hledej(int pole[], int n, int x) {
    int i;
    for (i=0; i<n; 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 (vyhledáváme pouze prvky z pole, vždy najdeme)
    $C_{prum}(n) = 1.5n$

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 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-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

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

Časová složitost algoritmů

- Tabulka udávající dobu výpočtu pro různé časové složitosti za předpokladu, 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	*20*μs	40μs	60μs	0,5*s*	1m*s*
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 (prog10-zarazka.c, přečte posloupnost celých čísel a vypíše různá čísla z posloupnosti):

```
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 */
}
```

- 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:

```
int hledejPulenim(int pole[], int pocet, int x) {
    int dolni = 0; int horni = pocet-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; /* nenalezen */
 }
```

- Vyzkoušejte program prog10-puleni.c

Řazení pole

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

  $a[i-1] <= a[i]$ pro $i = 1 \ldots$ počet prvků pole – 1
- Pole *a* je sestupně seřazené, jestliže platí

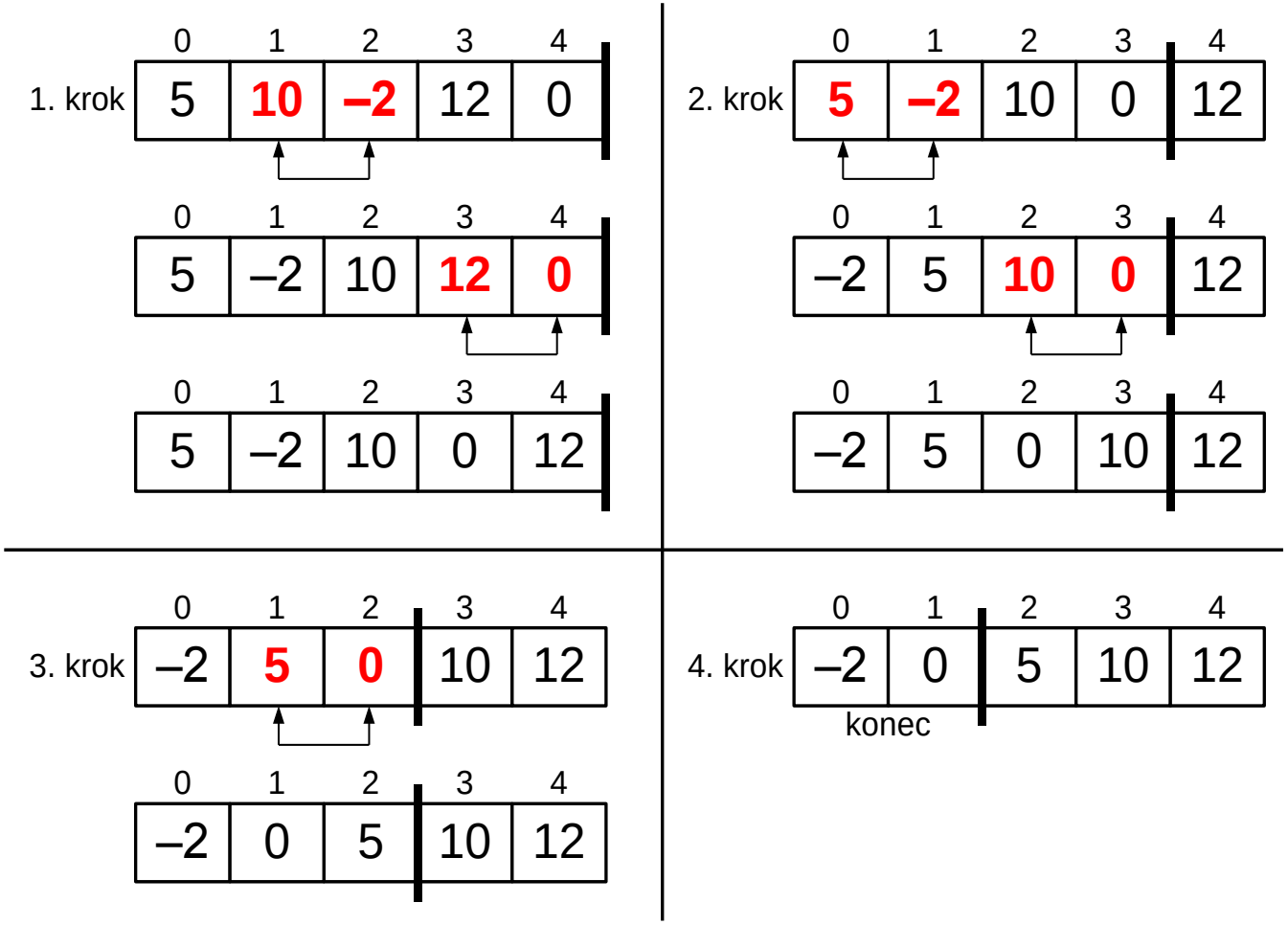
  $a[i-1] >= a[i]$ pro $i = 1 \ldots$ počet prvků pole – 1
- Principy některých algoritmů řazení ukážeme na řazení pole prvků typu *int* a budeme je prezentovat jako funkce, které vzestupně seřadí všechny prvky pole daného prvním parametrem (*a*) a počet prvků je dán druhým parametrem (*n*)
- Všechny procedury jsou uvedeny v programu prog9-razeni.c
- Všechny algoritmy řazení realizované těmito procedurami mají časovou složitost $O(n^2)$
- Efektivnější algoritmy řazení probereme ve 2. semestru

Řazení zaměňováním

- Při řazení zaměňováním postupně porovnáváme sousední prvky a pokud jejich hodnoty nejsou v požadované relaci, vyměníme je
- Po prvním průchodu polem se největší hodnota dostane na konec pole (vybublá)
- Pokud se provedla alespoň jedna výměna, průchod opakujeme, stačí však projít úsek pole o jeden prvek kratší

```
void bubbleSort(int a[], int n) {
    int i, vymeneno;
    do {  vymeneno = 0;
          for (i=1; i<n; i++) if (a[i-1]>a[i]) {
                                    vymena(&a[i-1], &a[i]);
                                    vymeneno = 1;
                              }
          n--;
    } while (vymeneno);
}
```

Řazení zaměňováním



Řazení výběrem

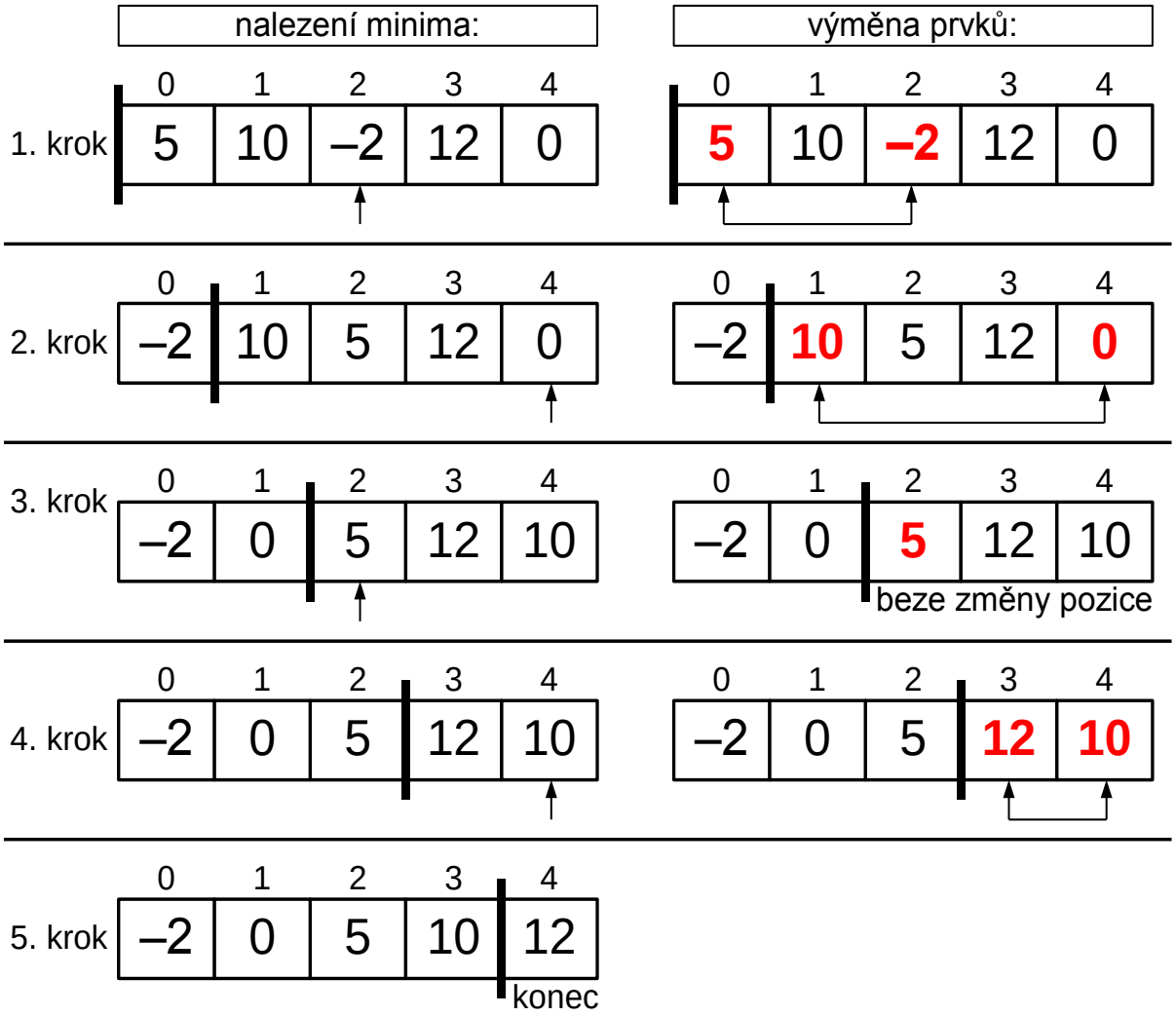
- Při řazení výběrem se opakovaně hledá nejmenší prvek
- Hrubé řešení:

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

- Podrobné řešení:

```
void selectSort(int a[], int n) {
    int i, j, imin;
    for (i=0; i<n-1; i++) {
          imin = i;
          for (j=i+1; j<n; j++) if (a[j]<a[imin]) imin = j;
          if (imin!=i) vymena(&a[imin], &a[i]);
    }
}
```

Řazení výběrem



Řazení vkládáním

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

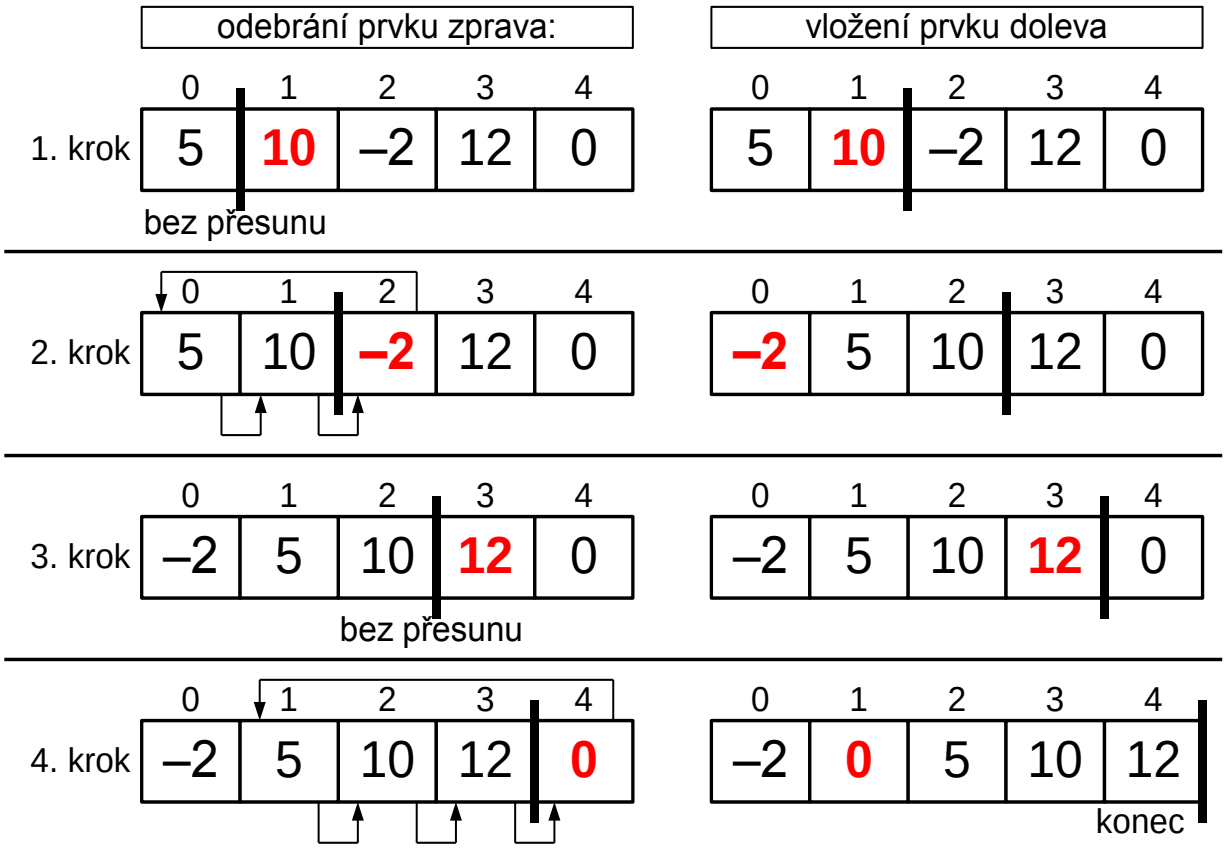
  for (k=1; k<n; k++) {
  “ *úsek pole od* a[0] *do* a[k-1] *je seřazen* ”
  “ *vlož do tohoto úseku délky k hodnotu a*[k] ”
  }

- Podrobné řešení:

```
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;
}

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

Řazení vkládáním



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* 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 *(a* a *b)*, můžeme algoritmus slučování v jazyku C zapsat např. jako funkci

```
void slucPole(int a[], int na, int b[], int nb, int c[])
```

,kde *c* je pole, do kterého bude uložena výsledná (sloučená) posloupnost

- Neefektivní řešení:
  - do pole c zkopírujeme prvky *a*, přidáme prvky *b* a pak seřadíme

Ale to není slučování!

Slučování

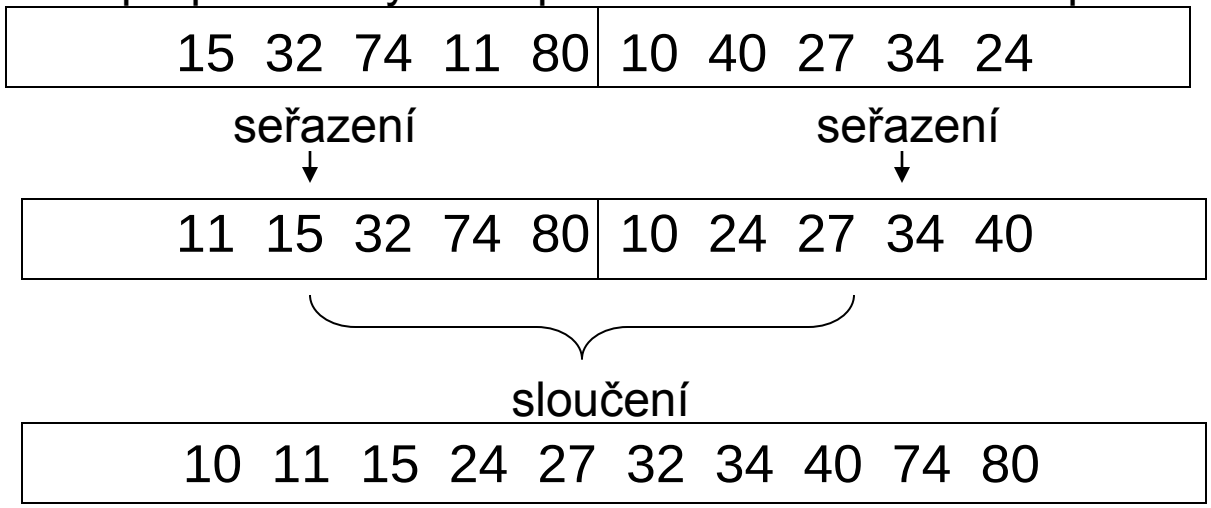
- Princip slučování:
  - postupně porovnáváme prvky zdrojových posloupností a do výsledné posloupnosti přesouváme menší z nich
  - nakonec zkopírujeme do výsledné posloupnosti zbytek první nebo druhé posloupnosti

```
void slucPole(int a[], int na, int b[], int nb, int c[]) {
    int ia = 0, ib = 0, ic = 0;
    while (ia<na && ib<nb)
                if (a[ia]<b[ib]) c[ic++] = a[ia++];
                else c[ic++] = b[ib++];
    while (ia<na) c[ic++] = a[ia++];
    while (ib<nb) c[ic++] = b[ib++];
}
```

- Viz program prog10-slucovani.c

Řazení slučováním

- Časová složitost předchozích algoritmů řazení je O(n^2)
- 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.* Úseky mohou mít i nestejnou délku.

```
void merge(int a[], int b[], int levy, int pravy, int poslPravy) {
    int poslLevy = pravy-1;
    int i = 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:

```
void mergeSortRek(int a[], int pom[], int prvni, int posl) {
    int i;
    if (prvni<posl) {
        int stred = (prvni+posl)/2;
        mergeSortRek(a, pom, prvni, stred);
        mergeSortRek(a, pom, stred+1, posl);
        merge(a, pom, prvni, stred+1, posl);
        for (i=prvni; i<=posl; i++) a[i] = pom[i];
    }
}
```

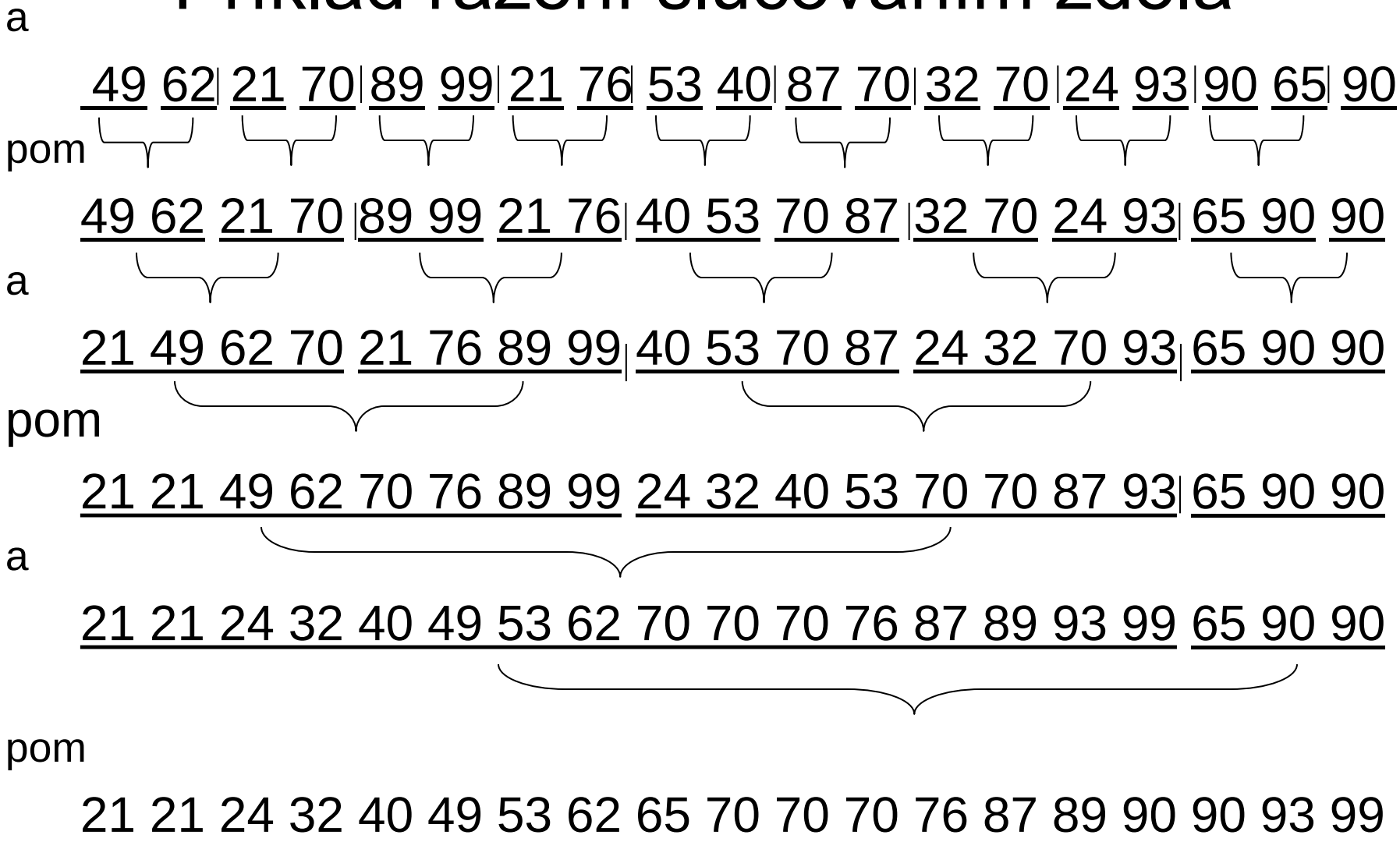
Výsledná funkce (prog10-mergesort1.c):

```
void mergeSort(int a[], int n) {
    int *pom = (int*)malloc(n*sizeof(int));
    mergeSortRek(a, pom, 0, n-1);
    free(pom);
}
```

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*

Příklad řazení slučováním zdola



Nerekurzívní MergeSort

```
void mergeSort(int a[], int n) {
    int *pom = (int*)malloc(n*sizeof(int)), *odkud = a, *kam = pom;
    int delkaUseku = 1;
    int posl = n-1, i;
    while (delkaUseku<n) {
        int levy = 0;
        int *p;
        while (levy<=posl) {
            int pravy = levy+delkaUseku;
            merge(odkud, kam, levy, min(pravy, n),
                    min(pravy+delkaUseku-1, posl));
            levy = levy+2*delkaUseku;
        }
        delkaUseku = 2*delkaUseku;
        p = odkud; odkud = kam; kam = p;
    }
    if (odkud!=a)  for (i=0; i<n; i++) a[i] = pom[i];
    free ( pom );
}
```

Podposloupnost s největším součtem

- 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*

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

```
int prvni, posledni;
    int maxSoucet3(int a[], int n) {
    int maxSum = 0, sum, i, j, k;
    for (i=0; i<n; i++)
            for (j=i; j<n; j++) {
                    sum = 0;
                    for (k=i; k<=j; k++) sum += a[k];
                    if (sum>maxSum) {
                            maxSum = sum; prvni = i; posledni = j;
                    }
            }
    return maxSum;
}
```

maxSum = 0;
Co když pole obsahuje pouze záporné prvky?

- 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 $S_{i,j} = S_{i,j-1}+ a[j]$

```
int prvni, posledni;
int maxSoucet2(int a[], int n) {
    int maxSum = 0, sum, i, j;
    for (i=0; i<n; i++) {
            sum = 0;
            for (j=i; j<n; 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 největší součet; hodnotu proměnné *i* je proto možné nastavit na *j*+1 a *sum* na 0

Řešení s lineární složitostí

```
int prvni, posledni;
int maxSoucet1(int a[], int n) {
    int maxSum = 0, sum = 0, i = 0, j;
    for (j=0; j<n; j++) {
                sum += a[j];
                if (sum>maxSum) {
                        maxSum = sum;
                        prvni = i;
                        posledni = j;
                } else if (sum<0) {
                        i = j + 1;
                        sum = 0;
                }
    }
    return maxSum;
}
```