

Vyhledávání a řazení

Jazyk JAVA



České vysoké učení technické Fakulta elektrotechnická

Obsah

- Časová složitost algoritmů
- Sekvenční hledání
- Binární hledání (Binary Search)
- Řazení pole
- Řazení zaměňováním (Bubble Sort)
- Řazení výběrem (Select Sort)
- Řazení vkládáním (Insert Sort)
- Slučování (Merging)
- Řazení slučováním (Merge Sort)
- Sloučení dvou seřazených úseků pole
- Rekurzivní MergeSort
- Nerekurzivní MergeSort
- Přihrádkové řazení (Case sort)
- Problém pro zájemce

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

Sekvenční hledání

- 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

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

Časová složitost algoritmů

- Tabulka udávající dobu výpočtu pro různé časové složitosti za předpokladu, že 1 operace trvá 1 μ s

	n					
	10	20	40	60	500	1000
$\log n$	2,3 μ s	4,3 μ s	5 μ s	5,8 μ s	9 μ s	
n	10 μ s	20 μ s	40 μ s	60 μ s	0,5s	1ms
$n \log n$	23 μ s	86 μ s	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				

Sekvenční hledání se záložkou

- Sekvenční hledání v poli lze urychlit pomocí záložky
- 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 záložkou:

```
static int hledejSeZarazkou(int[] pole, int volny, int x)
{
    int i = 0;
    pole[volny] = x;                // uložení záložky
    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

Binární hledání (Binary Search)

- 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 (Binary Search)

- 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
$$p[i-1] \leq p[i] \quad \text{pro } i = 1, \dots, \text{počet prvků pole} - 1$$
- Pole p v jazyku Java je sestupně seřazené, jestliže platí
$$p[i-1] \geq p[i] \quad \text{pro } i = 1, \dots, \text{počet prvků pole} - 1$$
- 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 (Bubble Sort)

- 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 (Select Sort)

- 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 (Insert Sort)

- 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 (Merging)

- 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!

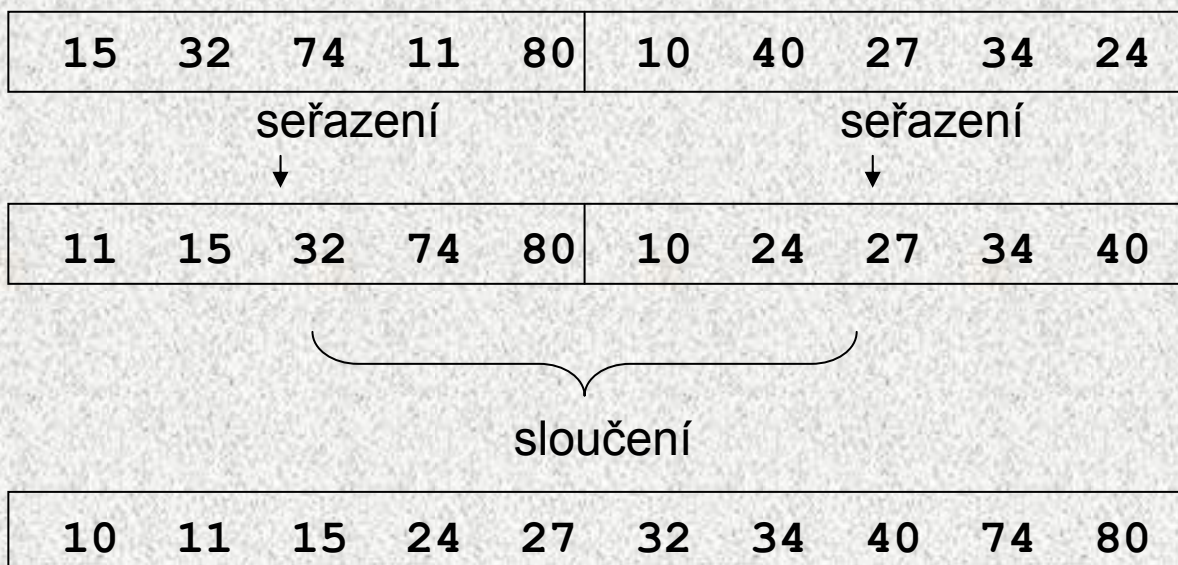
Slučování (Merging)

- 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 (Merge Sort)

- 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 rekurzivně:
 - 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 (první < 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);
}
```

Nerekurzivní MergeSort

- Rekurzivní MergeSort se volá rekurzivně 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.
- Nerekurzivní (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*
- Ilustrujme tento postup na příkladu

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

Nerekurzivní 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];
}
```

Přihrádkové řazení (Case sort)

- 1. průchod 223 131 458 193 756 812 027 579 283 200
rozdělení podle poslední číslice

			283						
			193						
200	131	812	223			756	027	458	579
0	1	2	3	4	5	6	7	8	9
- 2. průchod 200 131 812 223 193 283 756 027 458 579
rozdělení podle druhé číslice

		027			458				
200	812	223	131		756		579	283	193
0	1	2	3	4	5	6	7	8	9
- 3. průchod 200 812 223 027 131 756 458 579 283 193
rozdělení podle první číslice

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

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

Přihrádkové řazení (Case sort)

```
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<16; 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<16; c++)  
            while (!prihradky[c].jePrazdna())  
                pole[j++] = prihradky[c].odeber();  
    }  
}
```

Problém pro zájemce

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

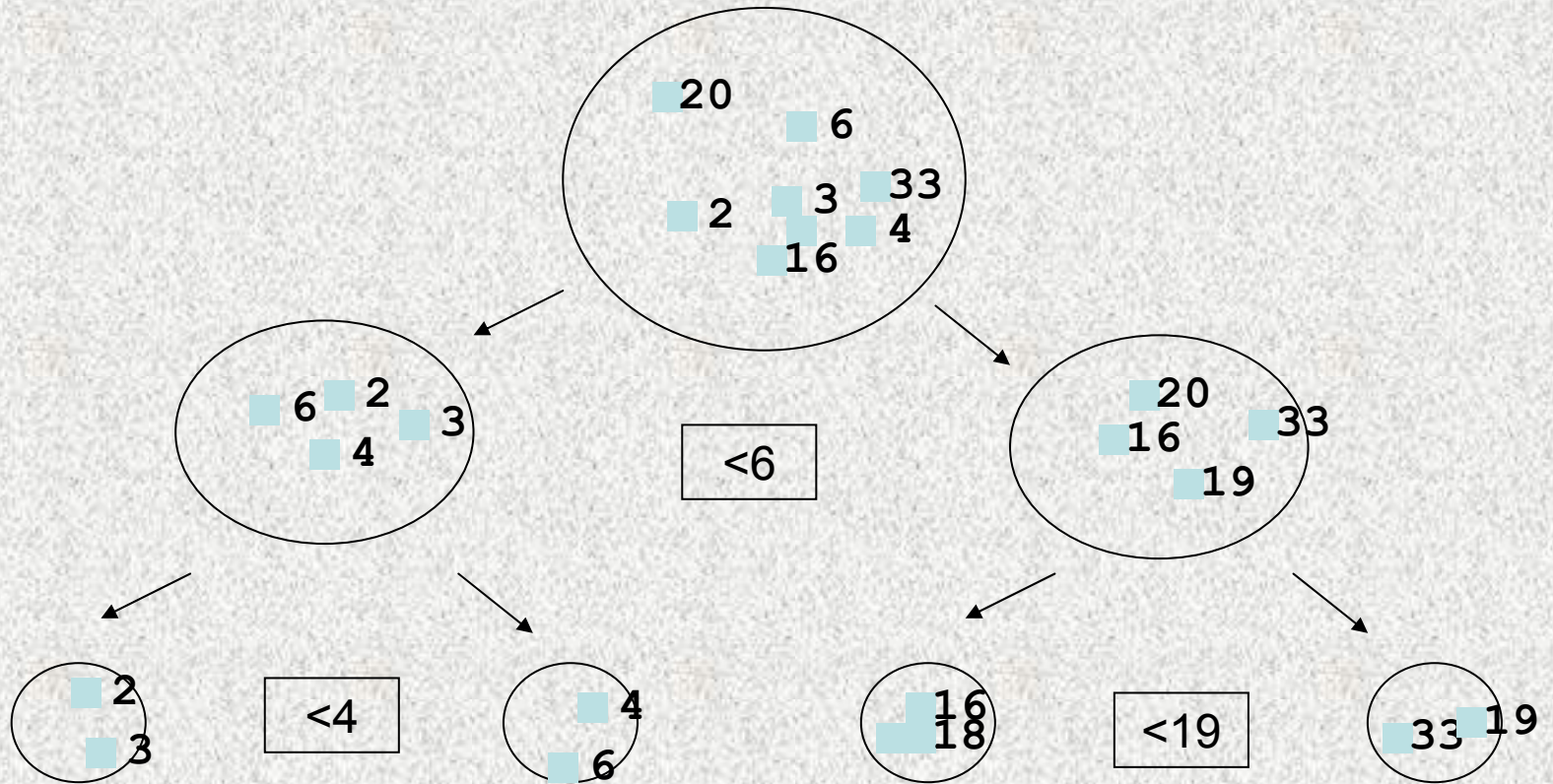
Řešení s lineární složitostí

- 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 ($sumMax$), zaznameneáme 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 $j1$, kde $j1 > j$, nemůže mít součet větší, než je zaznamenan; 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;  
}
```


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: $O(n \log n) - O(n^2)$
- Prostorová složitost algoritmu QuickSort: $O(\log n) - O(n)$
- Viz <http://en.wikipedia.org/wiki/Quicksort>

Vyhledávání a řazení

Jazyk JAVA

Konec

