

# ALG 11

**Vícedimenzionální data**

**Řazení vícedimenzionálních dat**

**Zabudované řazení v Javě**

**Experimentální porovnání řadících algoritmů  
na vícedimenzionálních datech**

## Vícedimenzionální data

2.4
1.7
0.2
3.1
-1.1
3.0

$d = 6$

Datový prvek je vektor  
délky  $d$  (= dimenze vektoru)

1	2	3	4	5	6	...
2.4	0.4	-0.1	3.2			
1.7	1.2	-0.1	2.7			
0.2	4.9	-0.2	2.4			
3.1	6.2	3.2	-3.2			
-1.1	9.0	5.6	-0.2			
3.0	-0.1	-1.1	0.9			

Řazené pole obsahuje  
(typicky) vektory stejné délky

2.4	2.4	2.4
1.7	1.7	1.8
0.2	0.2	0.1
3.1	3.2	3.1
-1.1	-5.4	-5.3
3.0	2.9	2.8

$<$        $<$

Dvojici vektorů porovnáváme  
po složkách od začátku,  
první neshodná dvojice složek  
určuje pořadí vektorů.

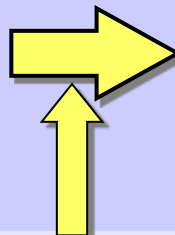
# Přímé a neefektivní řazení vícedimenzionálních dat

## Vstupní pole

1	2	3	4	5	6	7
4	0	4	2	4	3	0
0	1	0	4	3	3	4
4	2	4	0	4	1	4
3	4	4	2	3	3	3

## Seřazené pole

1	2	3	4	5	6	7
0	0	2	3	4	4	4
1	4	4	3	0	3	0
2	4	0	1	4	4	4
4	3	2	3	3	3	4



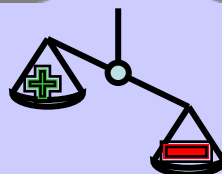
## Řadící algoritmus



Bezprostřední přístup k datům  
je rychlý.



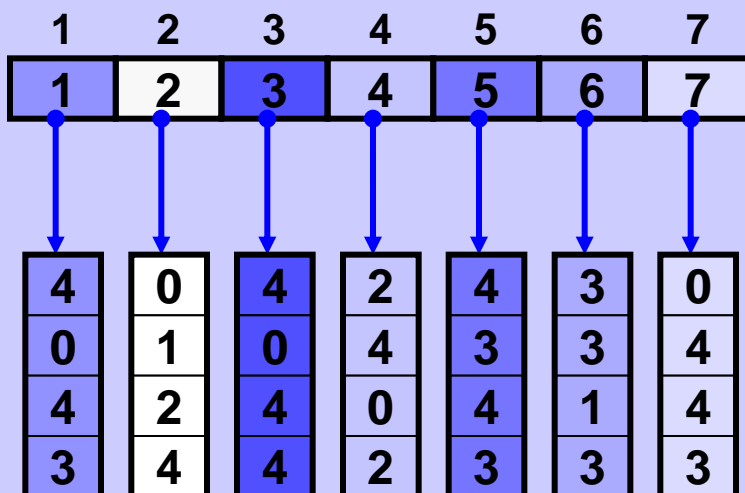
Výměna prvků je pomalá,  
musí se fyzicky přesouvat.



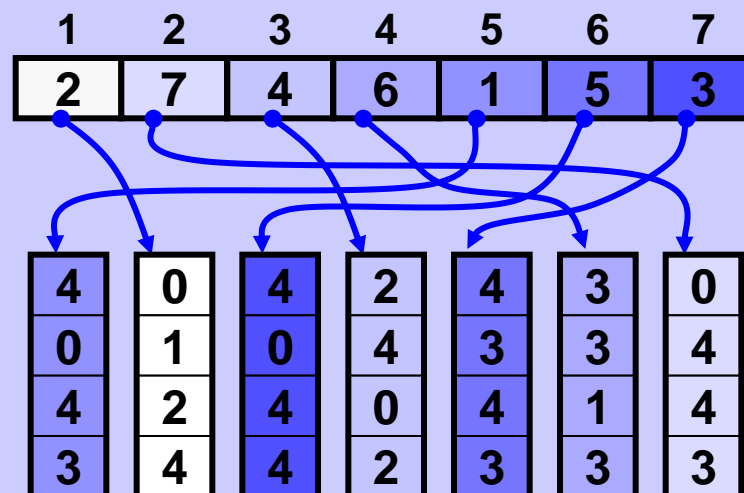
Zápor převažuje

# Řazení vícedimenzionálních dat s pomocnými ukazateli

Pole pomocných ukazatelů  
na datové prvky



Pouze ukazatele seřazeny podle  
velikosti jim odpovídajících dat.

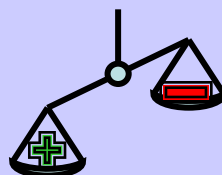


Výměna prvků je rychlá,  
děje se pouze výměnou  
ukazatelů.

Klad převažuje



Přístup k datům pomocí  
ukazatelů je pomalejší.



## Pomocná funkce pro porovnání dvou vektorů

```
class L {  
  
    int compare( int[] a, int[] b) {  
        for( int i = 0; i < a.length; i++) {  
            if (a[i] == b[i]) continue;  
            if (a[i] < b[i]) return -1;  
            else return +1;  
        }  
        return 0;  
    }  
    ...  
}
```

## Insert sort pro 1D pole (opakování)

```
void insertSort (int [] a) {  
  
    int insVal, j;  
    for (i = 1; i < n; i++) {  
  
        // find & make place for a[i]  
        insVal = a[i];  
        j = i-1;  
        while ((j >= 0) && (a[j] > insVal)) {  
            a[j+1] = a[j];  
            j--;  
        }  
  
        // insert a[i]  
        a[j+1] = insVal;  
    }  
}
```

## Insert sort pro řazení pole vektorů s využitím ukazatelů

```
void insertSort( int[][] a, int[] ptrs, int low, int high){  
  
    int j;  
    int insValPtr;  
    for (int i = low+1; i <= high; i++) {  
        // find & make place for a[i]  
        insValPtr = ptrs[i];  
        j = i-1;  
        while ((j >= low) &&  
            (L.compare(a[ptrs[j]], a[insValPtr]) == 1)) {  
            ptrs[j+1] = ptrs[j];  
            j--;  
        }  
        // insert a[i]  
        ptrs[j+1] = insValPtr;  
    }  
}
```

## Merge sort pro řazení pole vektorů s využitím ukazatelů I

```
void mergeSortCore (int[][] data, int ptr1[], int ptr2[],  
                    int low, int high){  
    int half = (low+high)/2;  
    int i;  
    if (low >= high) return;           // too small!  
                                         // sort:  
    mergeSortCore(data, ptr2, ptr1, low, half); // left  
    mergeSortCore(data, ptr2, ptr1, half+1, high); // right  
    merge(data, ptr2, ptr1, low, high); // merge halves  
}  
  
void mergeSort (int[][] data, int [] ptr1) {  
    initPtrs(ptr1); // init pointers  
    int [] ptr2 = new int [data.length];  
    initPtrs(ptr2);  
    mergeSortCore(data, ptr1, ptr2, 0, data.length-1);  
}
```



## Merge sort pro řazení pole vektorů s využitím ukazatelů II

```
void merge( byte [][] data, int inptr[], int outptr[],  
            int low, int high) {  
  
    int half = (low+high)/2;  
    int i1 = low;  
    int i2 = half+1;  
    int j = low;  
  
        // compare and merge  
    while ((i1 <= half) && (i2 <= high))  
        if ( compare(data[inptr[i1]], data[inptr[i2]]) <= 0)  
            outptr[j++] = inptr[i1++];  
        else outptr[j++] = inptr[i2++];  
  
        // copy the rest  
    while (i1 <= half) outptr[j++] = inptr[i1++];  
    while (i2 <= high) outptr[j++] = inptr[i2++];  
}
```

## Quick sort pro řazení pole vektorů s využitím ukazatelů II

```

void sortQp( int[][] a, int[] ptrs, int low, int high) {
    int iL = low, iR = high, aux;
    int pivotPtr = ptrs[low];

    do {
        while (L.compare(a[ptrs[iL]], a[pivotPtr]) == -1) iL++;
        while (L.compare(a[ptrs[iR]], a[pivotPtr]) == 1) iR--;
        if (iL < iR) {                                //swap(a, iL, iR)
            aux = ptrs[iL]; ptrs[iL] = ptrs[iR]; ptrs[iR] = aux;
            iL++; iR--;
        }
        else
            if (iL == iR) { iL++; iR--;}

    } while(iL <= iR);

    if (low < iR)  sortQp(a, ptrs, low, iR);
    if (iL < high) sortQp(a, ptrs, iL, high);
}

```

## Knihovni funkce řazení

Příklad použití zabudovaného řazení v Javě  
pro řazení pole celočíselných vektorů libovolné dimenze

```
import java.util.*;
...
class MyCompar implements Comparator {
    public int compare( Object obj1, Object obj2) {
        int [] a = (int []) obj1;
        int [] b = (int []) obj2;

        for( int i = 0; i < a.length; i++) {
            if (a[i] == b[i]) continue;
            if (a[i] < b[i]) return -1;
            else return +1;
        }
        return 0;
    } } // end of class
// usage
int[][] MydataArray = ... ; // any initialization
Arrays.sort(MydataArray, new MyCompar());
```

## **Ilustrační experiment řazení**

### **Prostředí**

**Intel(R) 1.8 GHz, Microsoft Windows XP SP3, jdk 1.6.0\_16.**

### **Organizace**

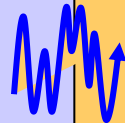
**Pole celých čísel náhodně zamíchána generátorem pseudonáhodných čísel s rovnoměrným rozložením. Výsledky průměrovány přes větší počet běhů.**

### **Závěr**

**Rozhodně neexistuje jedno univerzální řazení, včetně prostředků poskytovaných ve standardních knihovnách jazyka, které by bylo optimální za všech okolností.**

**Hraje roli velikost, dimenzionalita i stupeň předběžného seřazení dat.**

# Výsledky experimentu



Náhodně uspořádaná data		Doba běhu v ms				
		Délka pole				
		10	100	1 000	10 000	100 000
Délka vektoru  2	Insert	★ 0.0010	0.051	4.61	560	~75 000
	Quick	0.0014	✗ 0.027	✗ 0.41	5.66	81
	Merge	0.0014	0.025	0.38	5.36	74
	Java	0.0016	0.025	0.40	✗ 5.69	✗ 87
	Radix	✗ 0.0025	★ 0.007	★ 0.043	★ 0.55	★ 15
Délka vektoru  10	Insert	★ 0.001	0.052	4.75	561	~120 000
	Quick	0.002	0.034	✗ 0.47	✗ 6.3	108
	Merge	0.002	★ 0.026	0.38	5.3	★ 81
	Java	0.003	0.029	0.39	5.7	102
	Radix	✗ 0.014	✗ 0.039	★ 0.22	★ 3.0	✗ 161
Délka vektoru  100	Insert	★ 0.001	0.045	5.10	590	~300 000
	Quick	✗ 0.006	✗ 0.114	✗ 1.38	✗ 13.5	✗ 208
	Merge	★ 0.001	★ 0.012	0.35	★ 5.7	★ 81
	Java	0.002	0.026	★ 0.31	7.9	147
	Radix	0.121	0.377	2.66	33.0	2765

nesoutěží

## Výsledky experimentu

Vzestupně uspoř. data s 10% šumem		Doba běhu v ms				
		Délka pole				
		10	100	1 000	10 000	100 000
Délka vektoru  2	Insert	0.0003	0.008	0.53	56	~6 000
	Quick	0.0012	X 0.050	X 0.81	X 10.4	X 112
	Merge	★ 0.0010	0.017	0.25	3.8	46
	Java	0.0011	0.014	0.24	3.8	78
	Radix	X 0.0023	★ 0.006	★ 0.04	★ 0.45	★ 4
Délka vektoru  10	Insert	0.0003	0.01	0.57	59.5	~6 000
	Quick	0.0020	X 0.057	X 0.85	X 11.8	X 156
	Merge	★ 0.0010	0.025	0.25	3.8	★ 50
	Java	0.0020	★ 0.011	0.22	3.6	79
	Radix	X 0.0120	0.036	★ 0.20	★ 3.1	144
Délka vektoru  100	Insert	★ 0.0009	0.011	0.55	55.1	~20 000
	Quick	0.0010	X 0.119	X 1.57	X 17.7	X 380
	Merge	X 0.0030	★ 0.052	★ 0.51	★ 4.21	★ 101
	Java	X 0.0030	0.080	0.64	8.61	153
	Radix	0.1190	0.370	2.68	34.35	2750

nesoutěží

## Výsledky experimentu

Sestupně uspoř. data s 10% šumem		Doba běhu v ms				
		Délka pole				
		10	100	1 000	10 000	100 000
Délka vektoru  2	Insert	0.0003	0.008	0.53	56	~100 000
	Quick	0.0015	X 0.051	X 0.99	X 12.9	X 141
	Merge	★ 0.0011	0.017	0.25	3.59	44
	Java	0.0016	0.022	0.30	4.38	87
	Radix	X 0.0026	★ 0.006	★ 0.02	★ 0.38	★ 6
Délka vektoru  10	Insert	0.0003	0.001	0.57	59.5	~110 000
	Quick	★ 0.0011	X 0.058	X 1.00	X 14.7	X 186
	Merge	0.0020	★ 0.017	0.26	3.9	★ 53
	Java	0.0030	0.020	0.28	4.7	89
	Radix	X 0.0130	0.037	★ 0.20	★ 3.2	172
Délka vektoru  100	Insert	X 0.009	0.049	9.6	1 031	~420 000
	Quick	0.007	X 0.144	X 1.97	X 21.8	X 453
	Merge	★ 0.004	★ 0.021	★ 0.46	★ 3.4	★ 69
	Java	★ 0.004	0.077	★ 0.45	9.4	200
	Radix	0.117	0.320	2.41	33.5	2 791

nesoutěží

## Výsledky experimentu

**Pokud lze použít Radix sort, je to výhodné zejména při nízké dimenzionalitě a velkém objemu dat, zrychlení vůči zabudovanému řazení jsme v tomto případě naměřili až 20 násobné.**

**Jinak při vysoké dimenzionalitě je výhodné použít vlastní implementaci Merge sortu, vůči jeho zabudované variantě je typicky rychlejší, někdy až o 50%.**

**Při střední dimenzionalitě dat (jedna nebo více desítek) jsou výkony Radix sortu, vlastní i zabudované varianty Merge sortu přibližně srovnatelné.**

**Při velmi malém rozsahu dat (nejvýše desítky) nezávisle na dimenzi je vhodné zvážit řazení pomocí Insert sortu, vůči zabudovanému řazení jsme naměřili až 10 násobné zrychlení.**

**Používání Quick sortu nespíše nelze ve většině případů doporučit.**