ALG 14

Vícedimenzionální data

Řazení vícedimenzionálních dat

Experimentální porovnání řadících algoritmů
na vícedimenzionálních datech

Vícedimenzionální data

2.4
1.7
0.2
3.1
-1.1
3.0

d = 6

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			

Datový prvek je vektor délky d (= dimenze vektoru)

Řazené pole obsahuje (typicky) vektory stejné délky

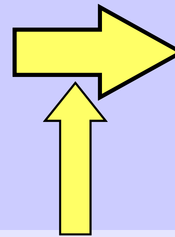
2.4		2.4		2.4
1.7		1.7		1.8
0.2	<	0.2	<	0.1
3.1		3.2		3.1
-1.1		-5.4		-5.3
3.0		2.9		2.8

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

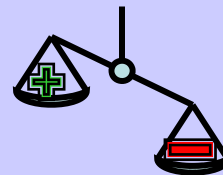
Řadící algoritmus

Klad

Bezprostřední přístup k datům je rychlý.

Zápor

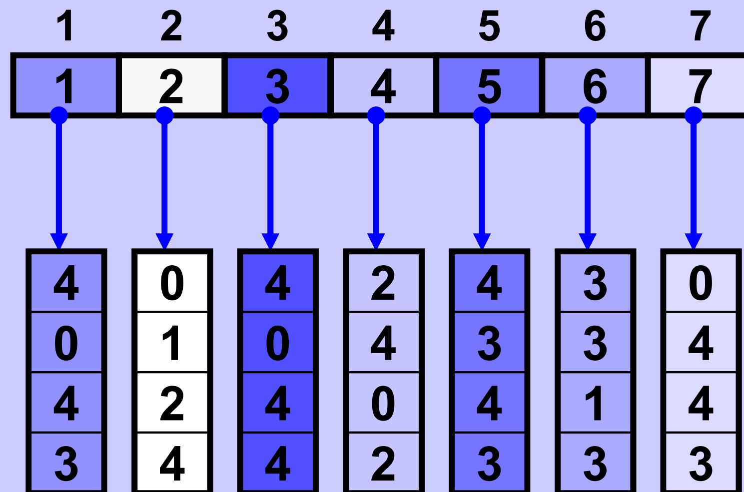
Výměna prvků je pomalá, musí se fyzicky přesouvat.



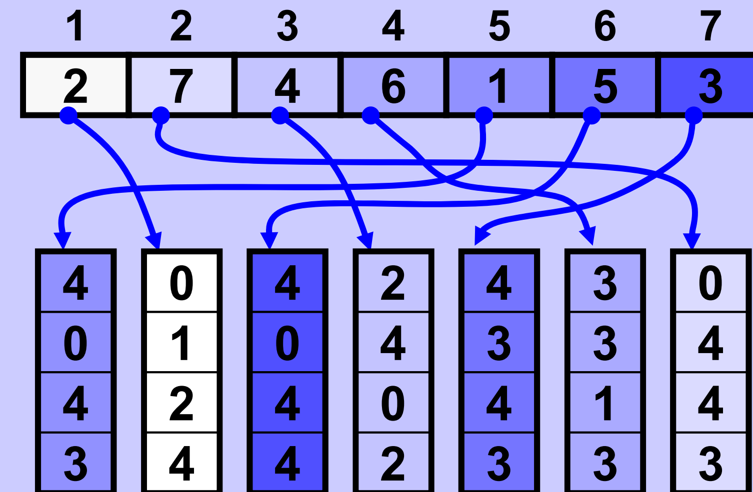
Zápor převažuje

Řazení vícedimenzionálních dat s pomocnými ukazateli

Pole pomocných ukazatelů na datové prvky



Řadíme pouze ukazatele podle velikosti jim odpovídajících dat.



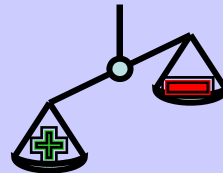
Klad

Výměna prvků je rychlá, děje se pouze výměnou ukazatelů.

Zápor

Přístup k datům pomocí ukazatelů je pomalejší.

Klad převažuje



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 low, int high) {

  int insVal, j;
  for (int i = low+1; i <= high; i++) {

    // find & make place for a[i]
    insVal = a[i];
    j = i-1;
    while ((j >= low) && (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);
}
```

Knihovní funkce řazení

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

```
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) 2.7 GHz, Microsoft Windows 7 Enterprise, SP1, version 6.1
Java: 1.7.0_51; Java HotSpot(TM) 64-Bit Server VM 24.51-b03
L2CacheSize 256 KB, L3CacheSize 4096 KB.

Organizace

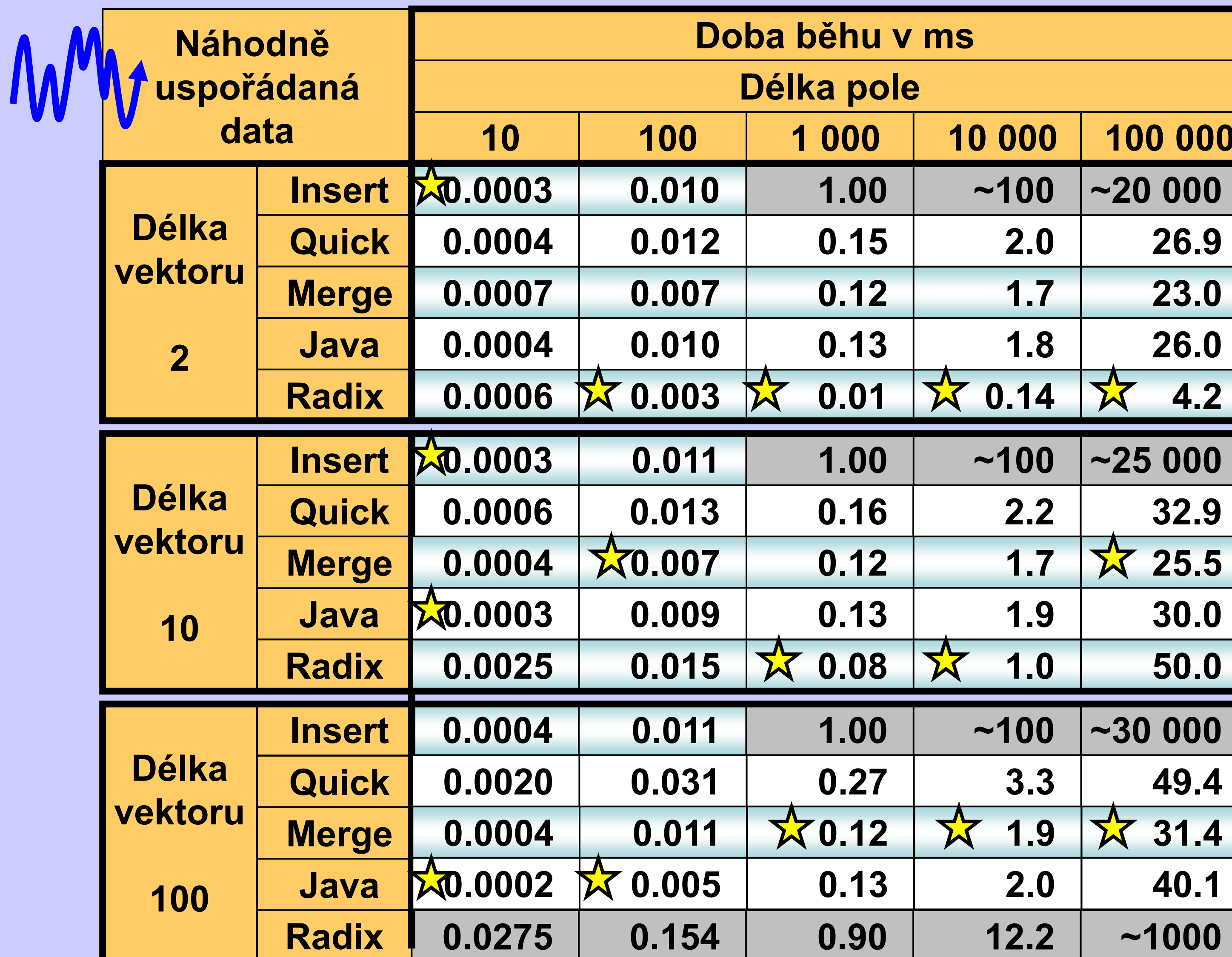
Pole celých čísel z intervalu <0,127> náhodně zamíchána generátorem pseudonáhodných čísel s rovnoměrným rozložením. Výsledky průměrovány přes větší počet běhů.

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	★0.0003	0.010	1.00	~100	~20 000
	Quick	0.0004	0.012	0.15	2.0	26.9
	Merge	0.0007	0.007	0.12	1.7	23.0
	Java	0.0004	0.010	0.13	1.8	26.0
	Radix	0.0006	★0.003	★0.01	★0.14	★4.2
Délka vektoru 10	Insert	★0.0003	0.011	1.00	~100	~25 000
	Quick	0.0006	0.013	0.16	2.2	32.9
	Merge	0.0004	★0.007	0.12	1.7	★25.5
	Java	★0.0003	0.009	0.13	1.9	30.0
	Radix	0.0025	0.015	★0.08	★1.0	50.0
Délka vektoru 100	Insert	0.0004	0.011	1.00	~100	~30 000
	Quick	0.0020	0.031	0.27	3.3	49.4
	Merge	0.0004	0.011	★0.12	★1.9	★31.4
	Java	★0.0002	★0.005	0.13	2.0	40.1
	Radix	0.0275	0.154	0.90	12.2	~1000

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	☆0.0001	0.003	0.10	9.5	~100
	Quick	0.0005	0.010	0.20	2.5	30.0
	Merge	0.0002	0.004	0.06	0.7	9.9
	Java	0.0002	0.004	0.05	0.5	7.9
	Radix	0.0008	☆0.003	☆0.02	☆0.1	☆1.8
Délka vektoru 10	Insert	☆0.0001	0.004	0.11	9.7	~100
	Quick	0.0005	0.015	0.22	2.9	39.1
	Merge	0.0003	☆0.004	0.06	0.8	10.3
	Java	0.0002	0.005	☆0.05	☆0.6	☆9.1
	Radix	0.0027	0.015	0.08	1.0	47.2
Délka vektoru 100	Insert	☆0.0002	☆0.004	0.13	10.5	~100
	Quick	0.0015	0.023	0.35	4.3	73.0
	Merge	0.0005	☆0.004	☆0.06	0.8	12.5
	Java	0.0004	0.008	☆0.06	☆0.7	☆11.5
	Radix	0.0260	0.150	1.36	12.4	~1000

nesoutěží

Výsledky experimentu

Zjištění závislá na stupni uspořádání dat

Náhodně uspořádaná data

Při střední nebo vysoké dimenzionalitě dat je výhodné použít vlastní implementaci Merge sortu, vůči knihovnímu řazení je typicky rychlejší, o jednotky až desítky procent, zejména se zvětšujícími se daty.

Vzestupně uspořádaná data s 10% šumem

**Knihovní řazení detekuje stupeň uspořádání dat a maximálně jej využívá.
V nejlepších případech dosahuje složitost Θ(N).
Při vyšší dimenzionalitě a velkých datech bylo v experimentu mírně rychlejší než vlastní implementace Merge sortu, lze čekat, že při menším šumu v datech bude rozdíl výraznější.**

Výsledky experimentu

Společná zjištění

Náhodně uspořádaná data

Vzestupně uspořádaná data s 10% šumem

Při velmi malém rozsahu dat (nejvýše desítky) nezávisle na dimenzi poskytuje standardně Insert sort nejlepší výkon. Také knihovní řazení Javy detekuje malá data a aplikuje Insert sort, výkon je tu srovnatelný.

Pokud lze použít Radix sort, je to výhodné zejména při nízké dimenzionalitě a velkém objemu dat, zrychlení vůči knihovnímu řazení jsme v tomto případě naměřili cca 5 až 10-i násobné.

Používání Quick sortu nespíše nelze ve většině případů doporučit.