Sorting - Iterative Algorithms

Problem Definition

Input:

- A sequence of n elements $<a_1, a_2, \ldots, a_n>$

Output:

- A permutation $<a'_1, a'_2, \ldots, a'_n>$ of such elements, so that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Types of Ordering

- Internal Ordering
  - All the elements to be ordered are in main memory
  - Direct access to all elements
- External Ordering
  - Elements cannot be loaded all in memory at the same time
  - It is necessary to act on elements stored on a file
  - Usually, sequential access

3

Practical observations

- Elements to be ordered are usually structures (`struct`) made of many variables (fields)
- The **key** of such structure is usually one field (or a value calculated from one or more fields)
- Remaining fields are additional data but useless for ordering
- Ordering is made for increasing values of the key

4

Example

```
struct student {
     int id;
     char surname[30] ;
     char name[30] ;
     int grade;
} ;

struct student class[100] ;
```

5

Example

```
struct student {
     int id;
     char surname[30]
     char name[30] ;
     int grade;
} ;

struct student class[100] ;
```

Ordering by id

Ordering by name and surname (key = concatenation name and surname)

Ordering by grade(repeated values)

6

Stability

A sorting algorithm is called *stable* whenever, even if there are elements with the same value of the key, in the resulting sequence such elements appear in the *same order* in which they appeared in the initial sequence.

7

Simple Assumption

During the study of sorting algorithms there are usually arrays of n integer values:

```
int A[n];
```

8

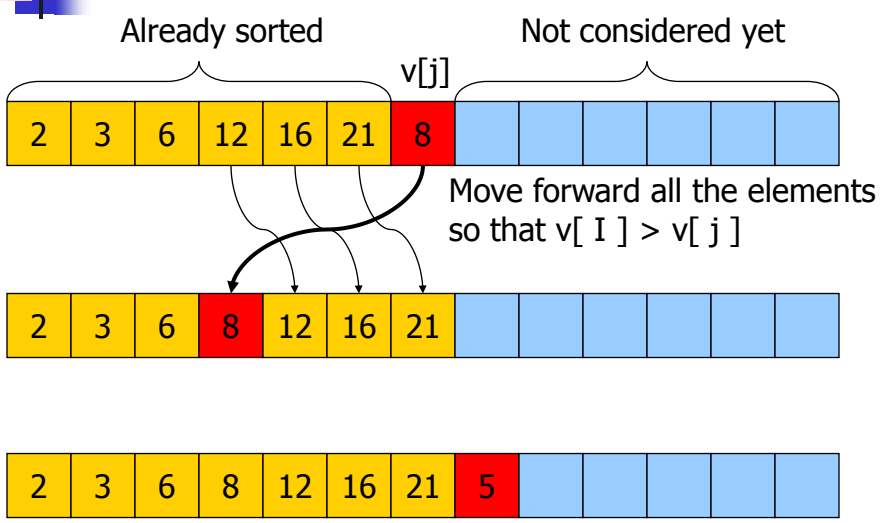
Algorithms

There are many sorting algorithms with different complexity:

- O(n^2): simple, iterative
  - Insertion sort, Selection sort, Bubble sort, …
- O(n): only applicable in particular cases
  - Counting sort, Radix sort, Bin (or Bucket) sort, …
- O($n \log n$): more complex, recursive
  - Merge sort, Quicksort, Heapsort

9

Insertion sort



10

Pseudo-code

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do  key ← A[j]
        ▷ Insert A[ j ] in ordered sequence A [1 .. j-1]
        i ← j – 1
        while i > 0 AND A[i] > key
            do A[i + 1] ← A[i]
               i ← i – 1
        A[i + 1] ← key
```

11

Implementation in C

```
void InsertionSort(int A[], int n)
{
     int i, j, key ;
     for(j=1; j<n; j++)  {
          key = A[j] ;
          i = j - 1 ;
          while ( i >= 0 && A[i]>key ) {
               A[i+1] = A[i] ;
               i-- ;
          }
          A[i+1] = key ;
     }
}
```

12

From pseudo-code to C

Note well:

- In C, array indexes are from 0 to n−1, while pseudo-code use ranges from 1 to n.
- Indentation of code is useful but remember braces to identify blocks { … }

13

Complexity

Number of comparisons:

- $C_{min} = n-1$
- $C_{avg} = ¼(n^2+n-2)$
- $C_{max} = ½(n^2+n)-1$

Number of data-copies

- $M_{min} = 2(n-1)$
- $M_{avg} = ¼(n^2+9n-10)$
- $M_{max} = ½(n^2+3n-4)$

Best case: array already ordered

Worst case: array ordered inversely

$C = O(n^2)$, $M = O(n^2)$

$T(n) = O(n^2)$

$T(n)$ non è $\Theta(n^2)$

$T_{worst\ case}(n) = \Theta(n^2)$

14

Other quadratic algorithms

		Min	Average	Max
Insertion Sort	$C =$	$n - 1$	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	$M =$	$2(n - 1)$	$(n^2 - 9n - 10)/4$	$(n^2 + 3n - 4)/2$
Selection Sort	$C =$	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	$M =$	$3(n - 1)$	$n(\ln n + 0.57)$	$n^2/4 + 3(n - 1)$
Bubble Sort	$C =$	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	$M =$	0	$(n^2 - n)*0.75$	$(n^2 - n)*1.5$

15

Execution Time (ms)

	Ordered		Random		Inversely Ordered	
Direct Insertion	12	23	366	1444	704	2836
Binary Insertion	56	125	373	1327	662	2490
Direct Selection	489	1907	509	1956	695	2675
Bubble sort	540	2165	1026	4054	1492	5931
Bubble sort with change notification	5	8	1104	4270	1645	6542
Shaker sort	5	9	961	3642	1619	6520
Shell sort	58	116	127	349	157	492
Heap sort	116	253	110	241	104	226
Quick sort	31	69	60	146	37	79
Merge	99	234	102	242	99	232
n =	256	512	256	512	256	512

16

Impact of data

n = 256	Ordinati		Disordinati		Inversamente Ordinati	
Inserimento diretto	12	46	366	1129	704	2150
Inserimento binario	56	76	373	1105	662	2070
Selezione diretta	489	547	509	607	695	1430
Bubblesort	540	610	1026	3212	1492	5599
Bubblesort con segnalatore di scambio	5	5	1104	3237	1645	5762
Shakersort	5	5	961	3071	1619	5757
Shellsort	58	186	127	373	157	435
Heapsort	116	264	110	246	104	227
Quicksort	31	55	60	137	37	75
Fusione *	99	196	102	195	99	187
	2 byte	16 byte	2 byte	16 byte	2 byte	16 byte

17

Counting sort

It cannot be applied in general, as it is based on this hypothesis:

- The n elements to be ordered are integer numbers between 1 and k, with k integer.

With such hypothesis, if k = O(n), then the algorithm's complexity is just O(n).

18

Basic Idea

Find out, for each element x, how many elements of the array are less than x.

Such information allows to put x directly in the final position in the array

19

Data Structure

- Three arrays are needed:
  - Initial array: A[1..n]
  - Final array: B[1..n]
  - Temporary Array: C[1..k]
- Array C keeps track of number of elements of A having a certain value: C[i] is the number of elements of A equals to i.
- Sum of the first i elements of C defines the number of elements of A whose values is <= i

20

Pseudo-code

```
COUNTING-SORT(A, B, k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 2 to k
    do C[i] ← C[i] + C[i − 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

21

Analysis

For each j, C[A[j]] represents the number of elements less than or equals to A[j], and then it is the final position of A[j] in B:

- B[C[A[j]]] = A[j]

The correction C[A[j]] ← C[A[j]] – 1 is needed to handle duplicate elements.

22

Example (n=8, k=6)

A	3	6	4	1	3	4	1	4
C	2	0	2	3	0	1		
C	2	2	4	7	7	8		

```
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
```

```
for i ← 2 to k
    do C[i] ← C[i] + C[i − 1]
```

B							4	
B		1					4	

C	2	2	4	6	7	8
C	1	2	4	6	7	8

```
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

23

Example (2)

```
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

A	3	6	4	1	3	4	1	4

B							4		C	2	2	4	6	7	8	j=8
B		1					4		C	1	2	4	6	7	8	j=7
B		1				4	4		C	1	2	4	5	7	8	j=6
B		1		3		4	4		C	1	2	3	5	7	8	j=5
B	1	1		3		4	4		C	0	2	3	5	7	8	j=4
B	1	1		3	4	4	4		C	0	2	3	4	7	8	j=3
B	1	1		3	4	4	4	6	C	0	2	3	4	7	7	j=2
B	1	1	3	3	4	4	4	6	C	0	2	2	4	7	7	j=1

24

Complexity

- 1-2: Initialization of C: O(k)
- 3-4: Calculate C: O(n)
- 6-7: Sum in C: O(k)
- 9-11: Copy in B: O(n)

Total complexity is O(n+k).

Algorithm is useful only when k=O(n), because the resulting complexity is O(n).

25

Note

The condition of applicability of the algorithm can be extended in this way:

- The key field of n elements to be ordered has a limited number of possible values k.

26

Bubble Sort

- In each cycle compare every couple of consecutive elements and if they are not ordered, then swap (exchange) them.
- Repeat this process N times and all the elements will be ordered
- Complexity is $O(n^2)$
- Optimization: if during last cycle there are no swaps, then the elements are already sorted

27

Bubble sort in C

```
void BubbleSort(int A[], int n)
{
     int i, j, t;
     for(i=1; i<n-1; i++)  {
          for(j=1; j<n-1; j++) {
            if ( A[j]>A[j+1] ) {
               t = A[j] ;
               A[j] = A[j+1];
               A[j+1] = t;
             }
          }
     }
}
```

28

Bubble sort (optimized) in C

```
void BubbleSort2(int A[], int n)  {
     int i, j, t, repeat = 1;
     While (repeat)  {
       repeat=0; /*if no swaps remains 0-> exit while*/
          for(j=1; j<n-1; j++) {
            if ( A[j]>A[j+1] ) {
               t = A[j] ; /* swap elements*/
               A[j] = A[j+1];
               A[j+1] = t;
               repeat=1;
             }
          }
     }
}
```

29