Sorting(1)

College of Computer Science, CQU

outline

- **Sorting**
- **Insertion Sort**
- **Bubble Sort**
- **Selection Sort**

Sorting

- Sorting is, without doubt, the most fundamental algorithmic problem
- Supposedly, 25% of all CPU cycles are spent sorting
- Sorting is fundamental to most other algorithmic problems, for example binary search.
- *Many* different approaches lead to useful sorting algorithms, and these ideas can be used to solve many other problems.

What is sorting?

- Given a set of records

$$r_1, r_2, \ldots, r_n$$

with key values $k_1, k_2, \ldots, k_n$

the Sorting Problem is to arrange the records into any order s such that records

$$r_{s1}, r_{s2}, \ldots, r_{sn}$$

have keys obeying the property

$$k_{s1} \leq ks_2 \leq \ldots \leq k_{sn}$$

What is sorting?

- **In other words,**

The sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

Issues in Sorting

- Increasing or Decreasing Order?
  - The same algorithm can be used by both all we need do is change to in the comparison function as we desire.
- What about equal keys?
  - Does the order matter or not? Maybe we need to sort on secondary keys, or leave in the same order as the original permutations.
- What about non-numerical data?
  - Alphabetizing is sorting text strings, and libraries have very complicated rules concerning punctuation, etc. Is *Brown-Williams* before or after *Brown America* before or after *Brown, John*?
- We can ignore all three of these issues by assuming a *comparison function* which depends on the application. *Compare (a, b)* should return ``<", ``>", or "=".

Applications of Sorting

- One reason why sorting is so important is that once a set of items is sorted, many other problems become easy.
- Searching Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.
- Speeding up searching is perhaps the most important application of sorting.

Applications of Sorting

- **Closest pair**
  - Given n numbers, find the pair which are closest to each other.
  - Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.
- **Element uniqueness**
  - Given a set of n items, are they all unique or are there any duplicates?
  - Sort them and do a linear scan to check all adjacent pairs.
  - This is a special case of closest pair above.
- **Frequency distribution – Mode**
  - Given a set of n items, which element occurs the largest number of times?
  - Sort them and do a linear scan to measure the length of all adjacent runs.
- **Median and Selection**
  - What is the kth largest item in the set?
  - Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking in the kth position of the array.

Sorting Terminology and Notation

- **Internal Sorting**
  - An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer.
- **External sorting**
  - **External sorting** is required when the data being sorted do not fit into the main memory and instead they must reside in the slower external memory (usually a hard drive) . External sorting can handle massive amounts of data.

Sorting Terminology and Notation

- **Stable Sorting**
  - A sorting algorithm is said to be **stable** if it does not change the relative ordering of records with identical key values.
- **Unstable sorting**
  - **Unstable sorting** may change the relative order of records with equal key values.

Sorting Terminology and Notation

- **Original key values:**

  49, 38, 65, 97, 76, 13, 27, 49’

- **After sorting:**

  13, 27, 38, 49, 49’, 65, 76, 97 stable

  13, 27, 38, 49’, 49, 65, 76, 97 unstable

How do you sort?

- There are several different ideas which lead to sorting algorithms:
- *Insertion* - putting an element in the appropriate place in a sorted list yields a larger sorted list.
- *Exchange* - rearrange pairs of elements which are out of order, until no such pairs remain.
- *Selection* - extract the largest element form the list, remove it, and repeat.
- *Distribution* - separate into piles based on the first letter, then sort each pile.
- *Merging* - Two sorted lists can be easily combined to form a sorted list.

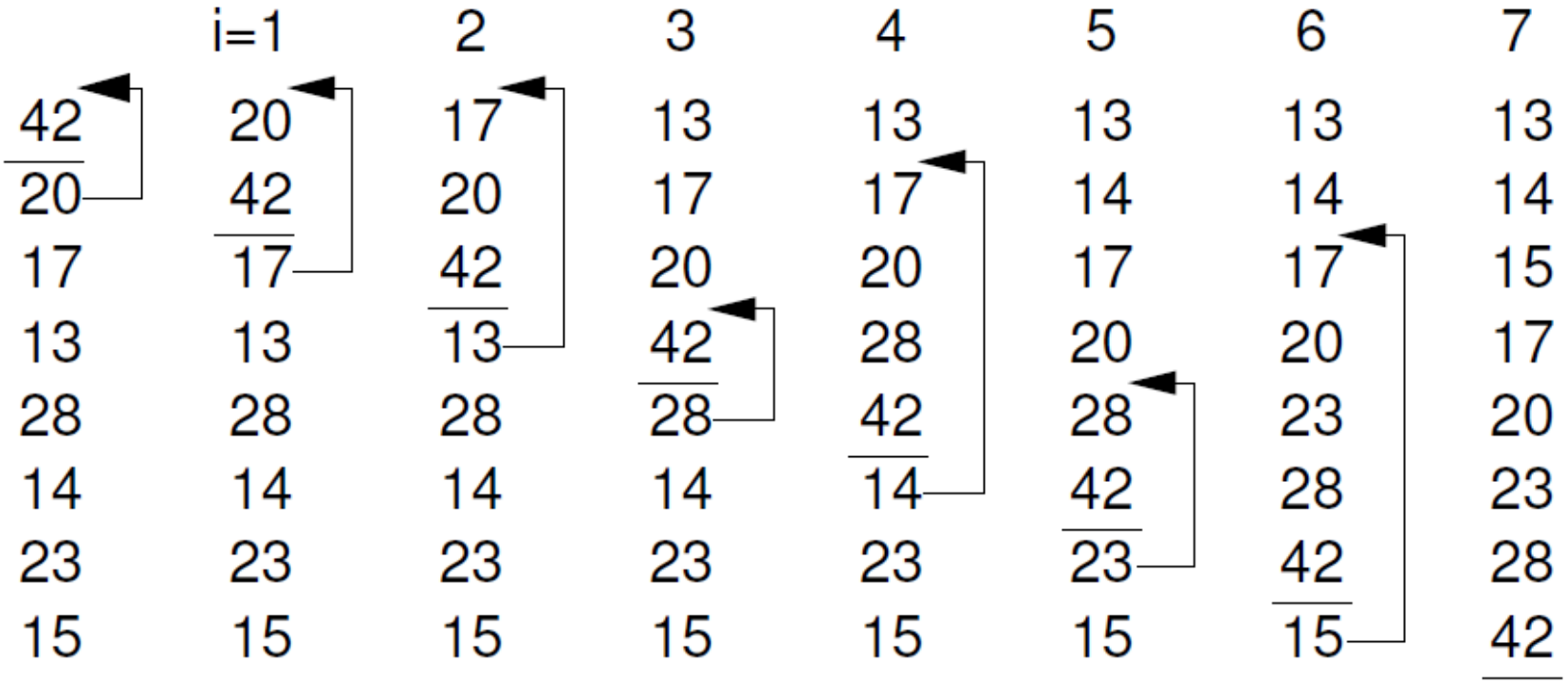
Analyzing sorting algorithms

- **Traditional measure: the number of comparisons made between keys.**
  - This measure is usually closely related to the running time for the algorithm and has the advantage of being machine and datatype independent.
- **Another measure :the number of swap operations performed by the algorithm.**
  - In some cases records might be so large that their physical movement might take a significant fraction of the total running time.
- **In most applications we can assume that all records and keys are of fixed length, and that a single comparison or a single swap operation requires a constant amount of time regardless of which keys are involved.**

Insertion Sort

- Intuitive sorting process:
  - Organizing a stack of phone bills;
  - Organizing cards when play cards
- Basic Idea:
  - Insertion sort iterates through a list of records. Each record is inserted in turn at the correct position within a sorted list composed of those records already processed

Insertion sort: Example



Insertion Sort

- In inserting the element in the sorted section, we might have to move many elements to make room for it.
- If the elements are in an **array**, we scan from bottom to top until we find the *j* such that

$$A[j] \leq x \leq A[j+1]$$

  then move from *j*+1 to the end down one to make room.
- If the elements are in **a linked list**, we do the sequential search until we find where the element goes, then insert the element there. *No other elements need move!*

Insertion Sort: Implementation

```
template <typename E, typename Comp>
void inssort(E A[], int n) { // Insertion Sort
  for (int i=1; i<n; i++)        // Insert i'th record
    for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
      swap(A, j, j-1);
}
```

```
void InsertionSort ( AList &A ) {
  // 对顺序表 A 作直接插入排序。
  for ( i=2; i<=n; ++i )
    if (A[i]< A[i-1]) {
      A[0] = A[i];      // 复制为监视哨
      for ( j=i-1; A[0]< A[j];  -- j )
        A[j+1] = A[j];    // 记录后移
      A[j+1] = A[0];  // 插入到正确位置
    }
} // InsertSort
```

Exp:


0 1 2 3 4 5 6 7 8
49 38 65 97 76 13 27 52
j
j=i-1
插入位置
插入38
0 1 2 3 4 5 6 7 8
38 49 65 97 76 13 27 52
j=i-1
插入65
0 1 2 3 4 5 6 7 8
12 27 38 49 65 76 97 52
插入52
j
i=i-1
插入位置
0 1 2 3 4 5 6 7 8
12 27 38 49 52 65 76 97

Insertion Sort : Complexity

- Since we do not necessarily have to scan the entire sorted section of the array, the best, worst, and average cases for insertion sort all **differ**!
- ***Best case***: when the array or list is already sorted
  - The element always gets inserted at the end,
  - don't have to move anything,
  - only compare against the last sorted element.
  - have (*n*-1) insertions, each with exactly one comparison and no data moves per insertion!
  - Cost: $\Theta(n)$

Worst Case Complexity

- **Worst case:** the array is sorted in reverse order.
  - the element always gets inserted at the front,
  - all the sorted elements must be moved at each insertion.
  - The ith insertion requires (i-1) comparisons and moves so:

$$\sum_{i=2}^{n} i \approx n^2/2 = \Theta(n^2)$$

- cost: $\Theta(n^2)$

Average Case Complexity

- ***Average Case***: If we were given a random permutation, the chances of the ith insertion requiring 0,1,2,…,(i-1) comparisons are equal, and hence 1/*i*.
- The expected number of comparisons is for the *i* th insertion is:

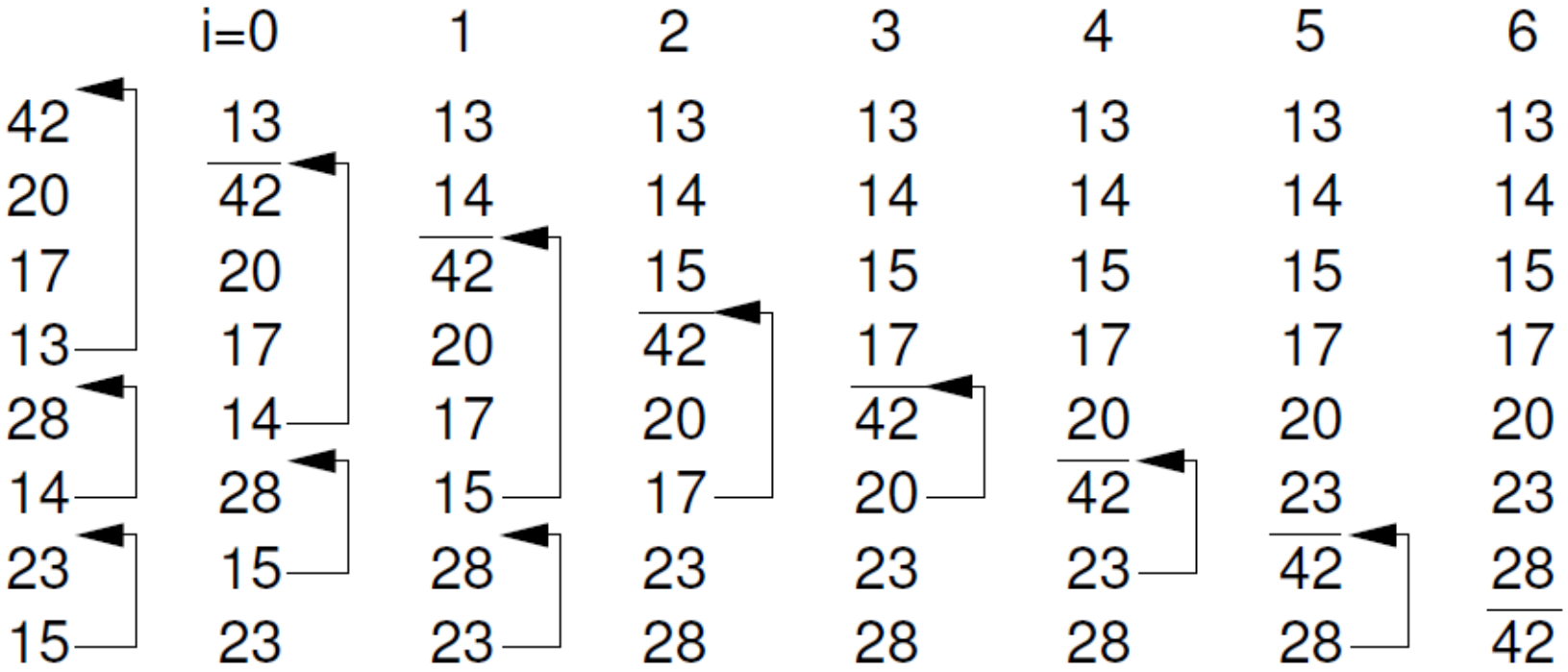
$$\sum_{k=1}^{i} \frac{(k-1)}{i} = \frac{1}{i} \sum_{k=1}^{i-1} k = \frac{1}{i} \times \frac{i(i-1)}{2} = \frac{i-1}{2}$$

- Summing up over all *n* keys, $\sum_{i=1}^{n} \frac{i-1}{2} = \frac{1}{2}\left(\frac{n(n-1)}{2} + \frac{n}{2}\right) = \frac{n^2}{4} + O(n)$
- So we do half as many comparisons/moves on average!
- Cost: $\Theta(n^2)$

Bubble Sort

- Bubble Sort consists of a simple double for loop.
- The first iteration of the inner for loop moves through the record array from bottom to top, comparing adjacent keys. If the lower-indexed key's value is greater than its higher-indexed neighbor, then the two values are swapped. Once the smallest value is encountered,this process will cause it to "bubble" up to the top of the array.
- The second pass through the array repeats this process. However, because we know that the smallest value reached the top of the array on the first pass, there is no need to compare the top two elements on the second pass.

Bubble Sort: Example



Bubble Sort: Implementation

```
template <typename E, typename Comp>
void bubsort(E A[], int n) { // Bubble Sort
  for (int i=0; i<n-1; i++)     // Bubble up i'th record
    for (int j=n-1; j>i; j--)
      if (Comp::prior(A[j], A[j-1]))
        swap(A, j, j-1);
}
```

Bubble Sort: Complexity

- One traversal = move the minimum element at the start
- Traversal #i : n – i + 1 operations
- Cost: $\sum_{i=1}^{n} i \approx n^2/2 = \Theta(n^2)$
- Bubble Sort's running time is roughly the same in the best, average, and worst cases.

Bubble Sort

- Alternative algorithm for Bubble sort
- **Idea:**
  - 1. Set flag = false
  - 2. Traverse the array and compare pairs of two elements
    - 2.1 If E1 $\leq$ E2 - OK
    - 2.2 If E1 > E2 then

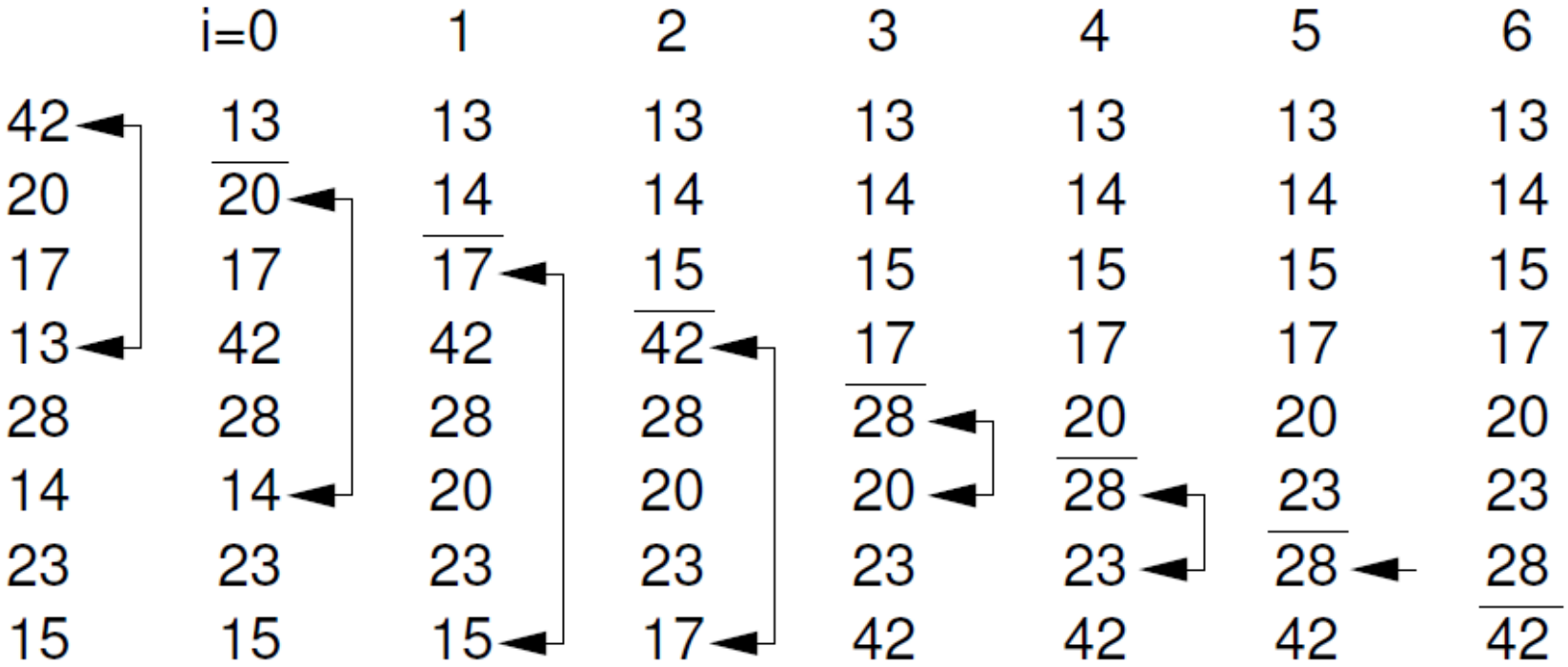
      Switch(E1, E2) and set flag = true
  - 3. If flag = true goto 1.
- **Complexity:**
  - Best case: O(n) when the array is sorted by ascending order, the number of comparisons.

Bubble Sort: Implementation2

```
void  bubbleSort (AList &L, int n )  //起泡排序算法
{ for ( int i = 0, change =1;  i<n-1 &&change; i++)
    {   change =0;
        for (j=n-1;j>i; j--)
            if ( L[j]< L[j-1] )
                { Swap (L[j -1], L [ j] );        //发生逆序
                  change = 1;   //做“发生了交换”标志
                }
    }
}
```

Selection Sort

- In my opinion, the most natural and easiest sorting algorithm is *selection sort*, where we repeatedly find the smallest element, move it to the front, then repeat...



Selection Sort : Implementation

```
template <typename E, typename Comp>
void selsort(E A[], int n) { // Selection Sort
  for (int i=0; i<n-1; i++) {    // Select i'th record
    int lowindex = i;            // Remember its index
    for (int j=n-1; j>i; j--)    // Find the least value
      if (Comp::prior(A[j], A[lowindex]))
        lowindex = j;            // Put it in place
    swap(A, i, lowindex);
  }
}
```

- If elements are in an array, swap the first with the smallest element- thus only one array is necessary.
- If elements are in a linked list, we must keep two lists, one sorted and one unsorted, and always add the new element to the back of the sorted list.

Selection Sort : Complexity

- *best case, worst case, and average cases are all the same!*
- Intuitively, we make *n* iterations, each of which "on average" compares *n*/2, so we should make about $n^2/2$ comparisons to sort *n* items.
- To do this more precisely, we can count the number of comparisons we make.
- To find the largest takes (*n*-1) steps, to find the second largest takes (*n*-2) steps ... to find the last largest takes 0 steps.
- The number of comparisons is still $\Theta(n^2)$, but the number of swaps is much less than that required by bubble sort.

Summary

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

- Swapping adjacent records is called an exchange.
- Thus, these sorts are sometimes referred to as **exchange sorts.**

Why exchange sorts are slow?

- The cost of any exchange sort can be at best the total number of steps that the records in the array must move to reach their "correct" location (i.e., the number of **inversions** 倒置 for each record).
- Inversion: a pair (i, j) such that i<j but Array[i] > Array[j]
- Array of size n can have $\Theta(n^2)$ inversions
  - average number of inversions in a random set of elements is n(n-1)/4
- Exchange Sort only swaps adjacent elements
  - only removes 1 inversion!

References

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**
  - P.231-239

-End-