Sort Intro

CSE 373 - Data Structures

May 6, 2002

Readings and References

- Reading
  - › Sections 7.1-7.4, *Data Structures and Algorithm Analysis in C*, Weiss
- Other References

Sorting

- Input
  - › an array A of data records
  - › a key value in each data record
  - › a comparison function which imposes a consistent ordering on the keys
- Output
  - › reorganize the elements of A such that
    - For any i and j, if $i < j$ then $A[i] \leq A[j]$

Consistent Ordering

- The comparison function must provided a consistent *ordering* on the set of possible keys
  - › You can compare any two keys and get back an indication of a < b, a > b, or a == b
  - › The comparison functions must be consistent
    - If **`compare(a,b)`** says a<b, then **`compare(b,a)`** must say b>a
    - If **`compare(a,b)`** says a=b, then **`compare(b,a)`** must say b=a
    - If **`compare(a,b)`** says a=b, then **`equals(a,b) and equals(b,a)`** must say a=b

Why Sort?

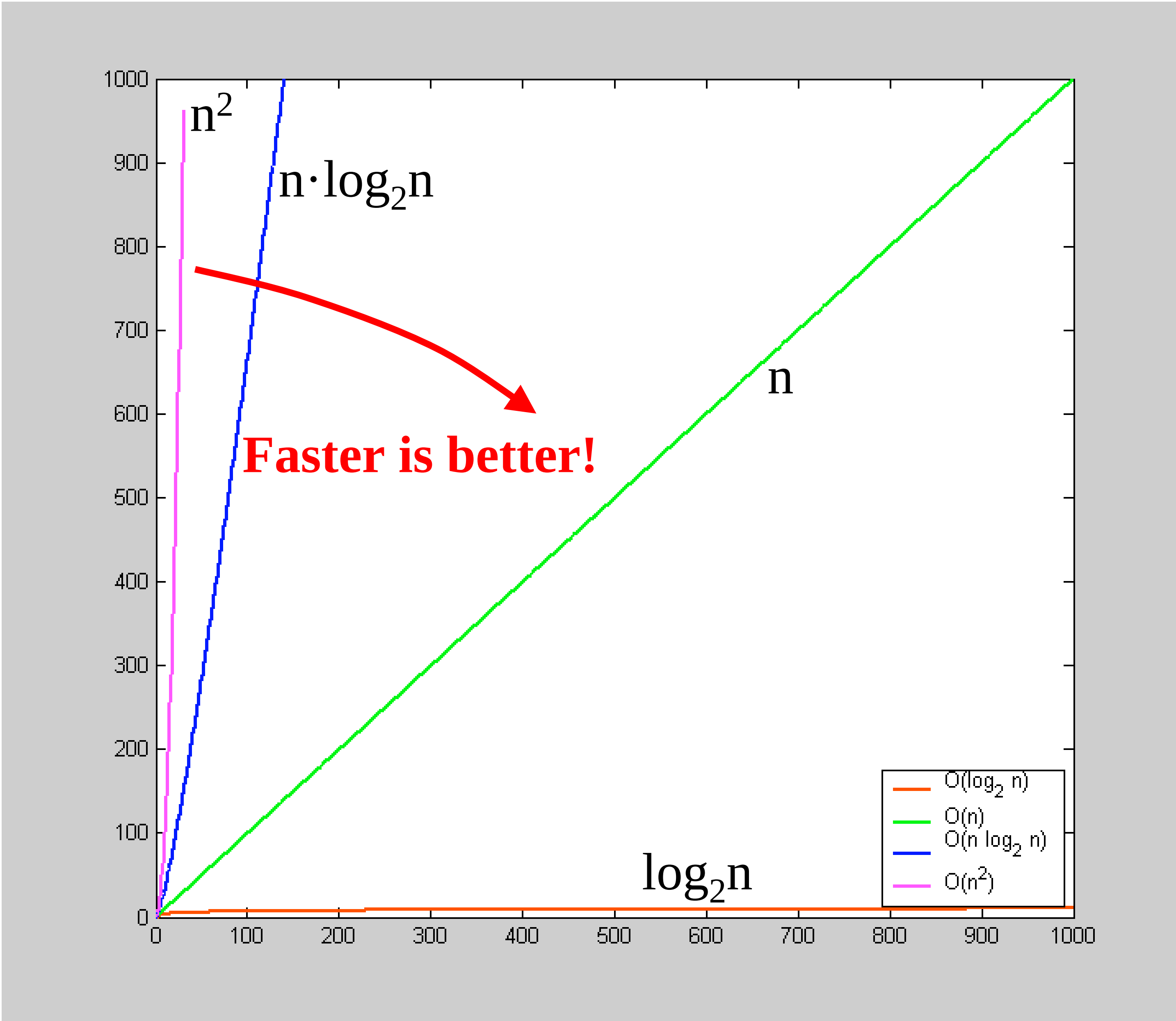
- Allows binary search of an N-element array in O(log N) time
- Allows O(1) time access to *k*th largest element in the array for any *k*
- Allows easy detection of any duplicates
- Sorting algorithms are among the most frequently used algorithms in computer science

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - › Do you need to copy and temporarily store the set or some subset of the keys and data records?
  - › An algorithm which requires O(1) extra space is known as an *in place* sorting algorithm
  - › Is the algorithm designed for in-memory operation (internal) or does it use disk or tape (external)?

Time

- How fast is the algorithm?
  - › The definition of a sorted array A says that for any $i<j$, $A[i] < A[j]$
  - › This means that you need to at least check on each element at the very minimum
    - which is $O(N)$
  - › And you could end up checking each element against every other element
    - which is $O(N^2)$
  - › The big question is: How close to $O(N)$ can you get?



n2
n·log2n
n
log2n
Faster is better!
O(log2 n)
O(n)
O(n log2 n)
O(n2)
0
100
200
300
400
500
600
700
800
900
1000

Stability

- Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
  - E.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
  - Extremely important property for databases
  - A **stable sorting algorithm** is one which does not rearrange the order of duplicate keys

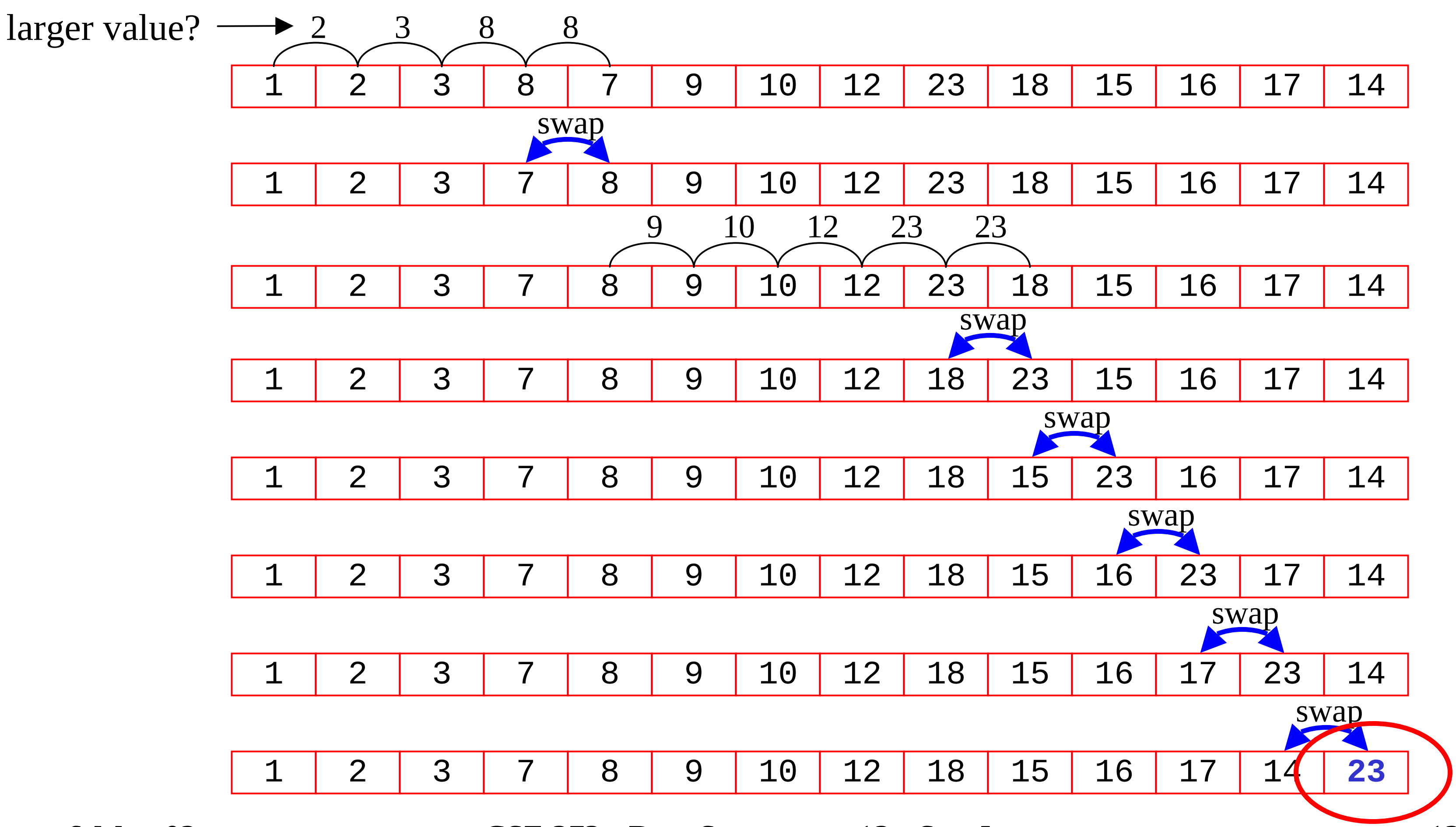
Bubble Sort

- "Bubble" elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
  - › Bubble every element towards its correct position
    - last position has the largest element
    - then bubble every element except the last one towards its correct position
    - then repeat until done or until the end of the quarter
    - whichever comes first ...

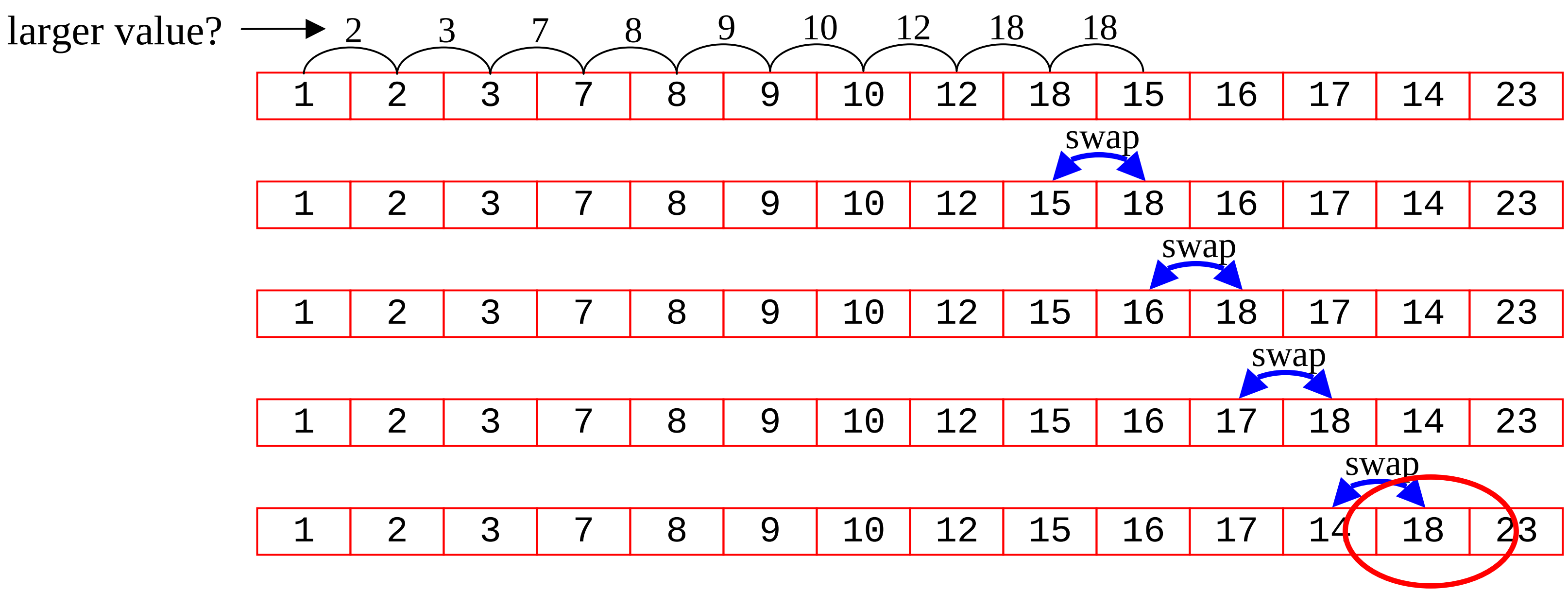
Bubblesort

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }
void bubble( int A[], int n ) {
  int i, j;
  for(i=0;i<n;i++) { /* n passes thru the array */
    /* From start to the end of unsorted part */
    for(j=1;j<(n-i);j++) {
      /* If adjacent items out of order, swap */
      if( A[j-1] > A[j] ) SWAP(A[j-1],A[j]); }
  }
}
```

Put the largest element in its place



Put 2^{nd} largest element in its place



Two elements done, only n-2 more to go ...

Bubble Sort: **Just Say No**

- "Bubble" elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]

- We bubblize for i=0 to n-1 (ie, n times)
- Each bubblization is a loop that makes n-i-1 comparisons
- This is $O(n^2)$

Insertion Sort

- What if first *k* elements of array are already sorted?
  - 4, 7, 12, 5, 19, 16
- We can shift the tail of the sorted elements list down and then *insert* next element into proper position and we get k+1 sorted elements
  - 4, 5, 7, 12, 19, 16

Insertion Sort

```
void InsertionSort( ElementType A[ ], int N ) {
   int j, P; ElementType Tmp;
   for( P = 1; P < N; P++ ) {
        Tmp = A[ P ];
        for( j = P; j > 0 && A[ j - 1 ] > Tmp;j-- )
               A[ j ] = A[ j - 1 ];
        A[ j ] = Tmp;
   }
}
```

- Is Insertion sort in place? Stable? Running time = ?
- Do you recognize this sort?
  - › This is what we used for percolating binary heap elements.

Insertion Sort Characteristics

- In place and Stable
  - › One extra location for Tmp
- Running time
  - › Worst case is $O(N^2)$
    - reverse order input
    - must copy every element every time
  - › Best case is $\Omega(N)$
    - in-order input
    - copy down stops with first comparison every time

Inversions

- An *inversion* is a pair of elements in wrong order
  - › $i < j$ but $A[i] > A[j]$
- By definition, a sorted array has no inversions
- So you can think of sorting as the process of removing inversions in the order of the elements

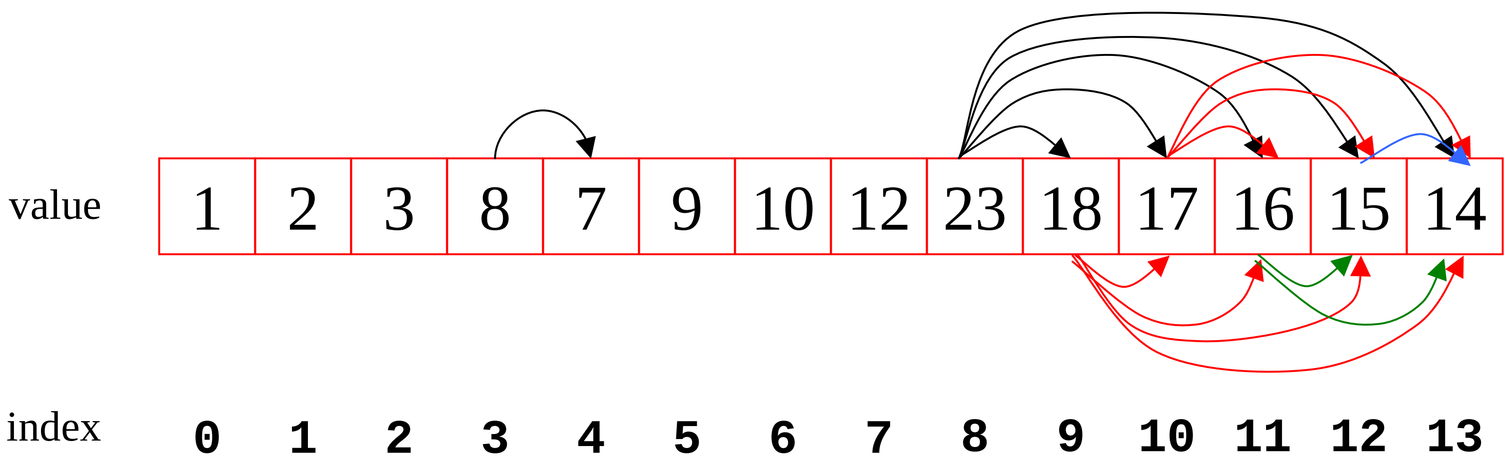
Inversions

- A single value out of place can cause several inversions

value	1	2	3	8	7	9	10	12	23	14	15	16	17	18
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**

Reverse order

- All values out of place (reverse order) causes numerous inversions



Inversions

- Our simple sorting algorithms so far swap adjacent elements (explicitly or implicitly) and remove just 1 inversion at a time
  - › Their running time is proportional to number of inversions in array
- Given N distinct keys, the maximum possible number of inversions is

$$(n-1)+(n-2)+...+1=\sum_{i=1}^{n-1} i=\frac{(n-1)(n)}{2}$$

Inversions and Adjacent Swap Sorts

- "Average" list will contain half the max number of inversions = $\frac{(n-1)(n)}{4}$
  - › So the average running time of Insertion sort is $\Theta(N^2)$
- Any sorting algorithm that only swaps adjacent elements requires $\Omega(N^2)$ time because each swap removes only one inversion