Sorting Introduction

CSE 373
Data Structures
Lecture 13
Reading

• Reading
  › Sections 7.1-7.5,
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 \( \text{compare}(a,b) \) says \( a<b \), then \( \text{compare}(b,a) \) must say \( b>a \)
    - If \( \text{compare}(a,b) \) says \( a=b \), then \( \text{compare}(b,a) \) must say \( b=a \)
    - If \( \text{compare}(a,b) \) says \( a=b \), then \( \text{equals}(a,b) \) and \( \text{equals}(b,a) \) 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?
  - Is copying needed
  - In-place sorting – no copying – $O(1)$ additional space.
  - External memory sorting – data so large that does not fit in memory
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?
Faster is better!
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
Example

Stable Sort

Unstable Sort
Bubble Sort

- “Bubble” elements 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

\[
\text{bubble}(A[1..n]: \text{integer array, } n : \text{integer}): \{ \\
i, j : \text{integer}; \\
\text{for } i = 1 \text{ to } n-1 \text{ do} \\
\quad \text{for } j = 2 \text{ to } n-i+1 \text{ do} \\
\quad \quad \text{if } A[j-1] > A[j] \text{ then SWAP}(A[j-1],A[j]); \\
\}
\]

\[
\text{SWAP}(a,b) : \{ \\
t : \text{integer}; \\
t := a; a := b; b := t; \\
\}
\]
Put the largest element in its place

larger value? →

1 2 3 8 8

swap

1 2 3 7 8 9 10 12 23 18 15 16 17 14

9 10 12 23 23

swap

1 2 3 7 8 9 10 12 23 18 15 16 17 14

swap

1 2 3 7 8 9 10 12 18 23 15 16 17 14

swap

1 2 3 7 8 9 10 12 18 15 23 16 17 14

swap

1 2 3 7 8 9 10 12 18 15 16 23 17 14

swap

1 2 3 7 8 9 10 12 18 15 16 17 23 14

swap

1 2 3 7 8 9 10 12 18 15 16 17 14

1 2 3 7 8 9 10 12 18 15 16 17 14

swap

1 2 3 7 8 9 10 12 18 15 16 17 23 14

swap

1 2 3 7 8 9 10 12 18 15 16 17 14

23
Put 2nd largest element in its place

larger value? → 2 3 7 8 9 10 12 18 18

1 2 3 7 8 9 10 12 18
swap

1 2 3 7 8 9 10 12 15
swap

1 2 3 7 8 9 10 12 15 16
swap

1 2 3 7 8 9 10 12 15 16 17
swap

1 2 3 7 8 9 10 12 15 16 17 14
swap

1 2 3 7 8 9 10 12 15 16 17 14 23

Two elements done, only n-2 more to go ...
Bubble Sort: Just Say No

- “Bubble” elements 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 $i=1$ to $n$ (i.e., $n$ times)
- Each bubblization is a loop that makes $n-i$ 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

InsertionSort(A[1..N]: integer array, N: integer) {
    j, P, temp: integer;
    for P = 2 to N {
        temp := A[P];
        j := P-1;
        while j > 1 and A[j-1] > temp do
        A[j] = temp;
    }
}

• Is Insertion sort in place? Stable? Running time = ?
• Do you recognize this sort?
  › Similar to percolate up.
Example
Example

1 2 3 7 8 9 10 12 15 16 18 17 23 14

1 2 3 7 8 9 10 12 15 16 17 18 23 14

1 2 3 7 8 9 10 12 15 16 17 18 14 23

1 2 3 7 8 9 10 12 15 16 17 14 18 23

1 2 3 7 8 9 10 12 15 16 14 17 18 23

1 2 3 7 8 9 10 12 15 14 16 17 18 23

1 2 3 7 8 9 10 12 14 15 16 17 18 23
Insertion Sort Characteristics

• In place and Stable
• Running time
  › Worst case is $O(N^2)$
    • reverse order input
    • must copy every element every time
• Good sorting algorithm for almost sorted data
  › Each item is close to where it belongs in sorted order.
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
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
Heap Sort

- We use a Max-Heap
- Root node = A[1]
- Keep track of current size N (number of nodes)

N = 5

<table>
<thead>
<tr>
<th>value</th>
<th>7</th>
<th>5</th>
<th>6</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

11/13/02 Sort Intro - Lecture 13 26
Using Binary Heaps for Sorting

- Build a **max-heap**
- Do N **DeleteMax** operations and store each Max element as it comes out of the heap
- Data comes out in largest to smallest order
- Where can we put the elements as they are removed from the heap?
1 Removal = 1 Addition

- Every time we do a DeleteMax, the heap gets smaller by one node, and we have one more node to store
  - Store the data at the end of the heap array
  - Not "in the heap" but it is in the heap array

<table>
<thead>
<tr>
<th>value</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>7</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ N = 4 \]
Repeated DeleteMax

\[
\begin{array}{ccccccc}
5 & 2 & 4 & 6 & 7 & & \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\(N = 3\)

\[
\begin{array}{ccccccc}
4 & 2 & 5 & 6 & 7 & & \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\(N = 2\)
Heap Sort is In-place

- After all the DeleteMaxs, the heap is gone but the array is full and is in sorted order

<table>
<thead>
<tr>
<th>value</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

N = 0
Heapsort: Analysis

- **Running time**
  - time to build max-heap is $O(N)$
  - time for $N$ DeleteMax operations is $N \cdot O(\log N)$
  - total time is $O(N \log N)$

- Can also show that running time is $\Omega(N \log N)$ for some inputs,
  - so *worst case* is $\Theta(N \log N)$
  - *Average case* running time is also $O(N \log N)$

- Heapsort is in-place but not stable