COMP 355
Advanced Algorithms

Sorting and Selection Review
Sorting

Given n elements, rearrange in ascending order.

Obvious sorting applications.
- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.
- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

...
Sorting Algorithms

Usually divided into two classes,

• *internal sorting algorithms*, (assume that data is stored in an array in main memory)

• *external sorting algorithm* (assume that data is stored on disk or some other device that is best accessed sequentially.

We will only consider internal sorting.
Standard Sorting Algorithms

Acceptable for small lists (< 20 elements)
Run in $\Theta(n^2)$ time
- InsertionSort
- SelectionSort
- BubbleSort (easiest to remember), widely viewed as worst of the 3

Canonical efficient comparison-based sorting algorithms
Run in $\Theta(n \log n)$ time
- MergeSort
- QuickSort
- HeapSort
Two Important Properties

• **In-place:** The algorithm uses no additional array storage, and hence (other than perhaps the system’s recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

• **Stable:** A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.
Comparison-Based Sorting Algorithms

Fig. 1: Common $O(n \log n)$ comparison-based sorting algorithms.
MergeSort

Classic divide-and-conquer algorithm

- Recursively sort each half.
- Divide array into two halves.
- Merge two halves to make sorted whole.

\[
\begin{array}{cccccccccc}
A & L & G & O & R & I & T & H & M & S \\
A & L & G & O & R & & I & T & H & M & S \\
A & G & L & O & R & & H & I & M & S & T \\
A & G & H & I & L & M & O & R & S & T
\end{array}
\]

- Advantage: the only *stable* sorting algorithm of these three.
- Disadvantage: it is *not in-place*.
Merging

• Merge.
  – Keep track of smallest element in each sorted half.
  – Insert smallest of two elements into auxiliary array.
  – Repeat until done.
Merging

• Merge.
  – Keep track of smallest element in each sorted half.
  – Insert smallest of two elements into auxiliary array.
  – Repeat until done.

A G L O R

H I M S T

auxiliary array
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

![Diagram of merging process]

Auxiliary array
• Merge.
  – Keep track of smallest element in each sorted half.
  – Insert smallest of two elements into auxiliary array.
  – Repeat until done.
Merging

• Merge.
  – Keep track of smallest element in each sorted half.
  – Insert smallest of two elements into auxiliary array.
  – Repeat until done.

• Auxiliary array

1/10/2014

COMP 355: Advanced Algorithms
Spring 2014
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

```
AGLOR

HIMST

AGHILM0
```

auxiliary array
Merging

• Merge.
  – Keep track of smallest element in each sorted half.
  – Insert smallest of two elements into auxiliary array.
  – Repeat until done.
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

```
AGLOR
HIMST
```

```
AGHILMORS
```

auxiliary array
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

```
AGLOR
HIMST
```

auxiliary array

---

Merge.
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.
Useful Recurrence Relation

• **Def.** $T(n) = \text{number of comparisons to mergesort an input of size } n.$

• **Mergesort recurrence.**

\[
T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lceil n/2 \right\rceil \right) + n & \text{otherwise}
\end{cases}
\]

• **Solution.** $T(n) = \Theta(n \log n).$
QuickSort

- QuickSort
  - Selects a “pivot value” from the array
  - Partitions the array into elements that are less than and greater than the pivot.
  - Recursively sorts each part.
- Widely regarded as the fastest of the fast sorting algorithms
- 2 Versions available
  - In-place and not stable
  - Not in-place, but stable
- $\Theta(n \log n)$ in the expected case, and $\Theta(n^2)$ in the worst case
HeapSort

Based on a data structure called a heap.

- Builds the heap - $\Theta(n)$
- Repeatedly extracts the largest element - $\Theta(\log n)$

- *In-place* sorting algorithm
- Not *stable*
**HeapSort**

- The left child of index $i$ is at index $2i+1$
- The right child of index $i$ is at index $2i+2$
- Example: the children of node $3$ (19) are $7$ (18) and $8$ (14)
Removing and Replacing the Root

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...

0 1 2 3 4 5 6 7 8 9 10 11 12

25 22 17 19 22 14 15 18 14 21 3 9 11

0 1 2 3 4 5 6 7 8 9 10 11 12

11 22 17 19 22 14 15 18 14 21 3 9 25

...And pretend that the last element in the array no longer exists—that is, the “last index” is 11 (9)
Selection

• Related problem to sorting is selection
• Given an array A of n numbers (not sorted) and an integer k, where 1 ≤ k ≤ n, return the kth smallest value of A.

• Easy algorithm:
  – Sort the array (Θ (n log n))
  – Return kth element

• Harder algorithm
  – O(n)
  – Variant of QuickSort
Lower Bounds for Comparison-Based Sorting

• $O(n \log n)$ sorting algorithms have been the fastest algorithms for many years.

• Can we sort faster?

• **Theorem**: Any comparison-based sorting algorithm has worst-case running time $\Omega(n \log n)$.
Linear-Time Sorting

- The $\Omega(n \log n)$ lower bound implies that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone.

- **Counting Sort**: assumes each integer in range from 1 to $k$.

- **Radix Sort**: only practical for very small ranges of integers.

- **Bucket Sort**: works for floating-point numbers, but should only be used if numbers are roughly uniformly distributed over some range.
Counting Sort

• Each input is an integer between 1 and k
• $\Theta(n + k)$ time
• If $k = O(n)$ -> algorithm runs in $\Theta(n)$
• Requires $\Theta(n + k)$ working storage (not in-place)
• Stable
Figure 8.2 The operation of COUNTING-SORT on an input array A[1..8], where each element in A is a non-negative integer no larger than k = 5. a) The array A and the auxiliary array C after line 5. b) The array C after line 8. c-e) The output array B and the auxiliary array C after one, two and three iterations of the loop in lines 10-12. Only the lightly shaded elements in B have been filled in. f) The final sorted output array B.
Radix Sort

- Main shortcoming of CountingSort is that it is only practical for a very small ranges of integers.
- RadixSort sorts one digit or one byte at a time.
- $\Theta(d(n + k))$ where $d$ is the number of digits in each value, $n$ is the length of the list, and $k$ is the number of distinct values each digit may have
- $\Theta(n + k)$ space needed
- Stable
- Not in-place
Radix Sort

• Sort repeatedly
  – Start at the lowest order digit
  – Finish with the highest order digit

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>576</td>
<td></td>
<td></td>
<td></td>
<td>176</td>
</tr>
</tbody>
</table>
BucketSort

• Works for floating-point numbers
• Assumption: all floating-point numbers are roughly uniformly distributed over some range
• Sorting time is $\Theta(n)$, where $n$ is the number of buckets
• Stable
• Not in-place
BucketSort

- Suppose that the numbers to be sorted range over some interval, say $[0, 1)$
- Subdivide this interval into $n$ subintervals
- Use the floor function to map values to their bucket index

```
A
.42 .71 .10 .14 .86 .38 .59 .17 .81 .56
```

```
  0  1  2  3  4  5  6  7  8  9
B  
  .10 .14 .17
  .38 .42 
  .56 .59 
  .71 
  .81 .86 
```
Comparison-Based Sorting Algorithms: A stable sorting algorithm preserves the relative order of equal elements. An in-place sorting algorithm uses no additional array storage (although $O(\log n)$ additional space is allowed for the recursion stack).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Stable</th>
<th>In-place</th>
</tr>
</thead>
<tbody>
<tr>
<td>BubbleSort</td>
<td>$\Theta(n^2)$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>$\Theta(n^2)$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MergeSort</td>
<td>$\Theta(n \log n)$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$\Theta(n \log n)$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>QuickSort*</td>
<td>$\Theta(n \log n)$</td>
<td>Yes/No</td>
<td>No/Yes</td>
</tr>
</tbody>
</table>

*There are two versions of QuickSort, one which is stable but not in-place, and one which is in-place but not stable.

Non-Comparison-Based Sorting Algorithms: All of these algorithms are stable, but not in-place.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Assumptions</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountingSort</td>
<td>Integers over $[0..k]$</td>
<td>$\Theta(n + k)$</td>
<td>$\Theta(n + k)$</td>
</tr>
<tr>
<td>RadixSort</td>
<td>Integers over $[0..n^d]$</td>
<td>$\Theta(d(n + k))$</td>
<td>$\Theta(n + k)$</td>
</tr>
<tr>
<td>BucketSort</td>
<td>Integers uniformly distributed</td>
<td>$\Theta(n)$ (Expected)</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>