Last Time

• Mergesort
  – If we have two sorted lists, we can merge them together into one big sorted list easily
  – This allows us to recursively build a sorted list by breaking down a string into smaller pieces first

• We can use a recurrence relation to calculate the runtime of Merge Sort
Top-Down Mergesort
Top-Down Mergesort Improvements

• Now we can sort in $N \times \log(N)$ time – a substantial improvement over the $N^2$ time of Insertion and Selection Sorts.

• We know that Insertion Sort is efficient for small arrays, so if we switch to Insertion Sort once the problem is broken down below some threshold, we can improve Mergesort by 10-15%.

• We can reduce the run time to be linear for arrays that are already sorted, by adding a test to skip the call to `merge()` if $a[mid] \leq a[mid+1]$. 
private static Comparable[] aux;

public static void sort(Comparable[] a) {
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz+sz) {
        for (int lo = 0; lo < N-sz; lo += sz+sz) {
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
        } //for
    } //for
} //sort
Bottom-Up Mergesort

• Rather than follow a tree-like structure, instead let’s merge all of the 1-length arrays into 2s in one pass, then all of the 2-length arrays into 4s, and repeat.
  – The second subarray may be smaller than the first in the last pass, but that’s no problem for `merge()` as we designed it.
Bottom-Up Mergesort Visual

<table>
<thead>
<tr>
<th>sz = 1</th>
<th>merge(a, 0, 0, 1)</th>
<th>merge(a, 2, 2, 3)</th>
<th>merge(a, 4, 4, 5)</th>
<th>merge(a, 6, 6, 7)</th>
<th>merge(a, 8, 8, 9)</th>
<th>merge(a, 10, 10, 11)</th>
<th>merge(a, 12, 12, 13)</th>
<th>merge(a, 14, 14, 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
<td>MERGESORTEXAMPLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 2</th>
<th>merge(a, 0, 1, 3)</th>
<th>merge(a, 4, 5, 7)</th>
<th>merge(a, 8, 9, 11)</th>
<th>merge(a, 12, 13, 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EGMRDERTXAMPLE</td>
<td>EGMRDERTXAMPLE</td>
<td>EGMRDERTXAMPLE</td>
<td>EGMRDERTXAMPLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 4</th>
<th>merge(a, 0, 3, 7)</th>
<th>merge(a, 8, 11, 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EEEMORRESSATEXELMP</td>
<td>EEEMORRESSATEELMP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 8</th>
<th>merge(a, 0, 7, 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AEELMPXTX</td>
</tr>
</tbody>
</table>
Bottom-Up Mergesort Evaluation

• Did our number of compares and array accesses change with this new implementation?
  – Well, we still have $N \times \log(N)$ compares ($\log(N)$ passes through as we increase sz, and between $N/2$ and $N$ items compared on each pass) and $6N$ array accesses per compare.
  – The exact number may have changed, but the order of growth has not.

• Bottom-up is better for linked lists. (why?)
### Sorting Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>Compares</th>
<th>Exchanges/Array Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insertion Sort (Worst)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort (Average)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort (Best)</td>
<td>$O(n)$</td>
<td>0</td>
</tr>
<tr>
<td>Shell Sort (Worst)</td>
<td>$O(n^{3/2})$</td>
<td>$O(n^{3/2})$</td>
</tr>
<tr>
<td>Shell Sort (Average)</td>
<td>It depends</td>
<td>It depends</td>
</tr>
<tr>
<td>Top-Down Mergesort (Worst)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Top-Down Mergesort (Average)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Bottom-Up Mergesort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
</tbody>
</table>
Compare-Based Algorithms

• Sorting is a compare-based algorithm.
  – Makes decisions about items only on the basis of comparing keys.
  – Can do an arbitrary amount of calculation between compares, but cannot get any information about a key except by comparing it with another one.

• Theorem: No compare-based sorting algorithm can guarantee to sort $N$ items with fewer than $O(n \times \log(n))$ compares.
Maximum Sort

• **Theorem:** No compare-based sorting algorithm can guarantee to sort $N$ items with fewer than $O(n \times \log(n))$ compares.

• **Proof:** We will use a binary tree to describe the sequence of compares necessary to sort an item.
  
  – Each node in the tree is either a leaf $(i_0..i_{n-1})$ that indicates that a solution has been found and the inputs should be ordered as $a[i_0].a[i_{n-1}]$, ...
  
  – …or the node could be an internal node $(i: j)$ that corresponds to a compare operations between $a[i]$ and $a[j]$.
  
  – Each path from the root to a leaf corresponds to the sequence of compares that the algorithm uses to establish the ordering given in the leaf.
Maximum Sort
Maximum Sort

• Observations:
  – The tree must have at least $n!$ leaves, because there are $n!$ different permutations of $n$ distinct keys.
  – The length of the longest path in the tree (the tree height) is the worst-case number of compares used by the algorithm.
  – We know that a binary tree of height $h$ has no more than $2^h$ leaves.
  – Combining these facts, we know that any compare-based algorithm corresponds to a compare tree of height $h$ with $n! \leq \text{number of leaves} \leq 2^h$.
  – The value of $h$ is the worst-case number of compares, so $\log(n!) \leq \text{number of compares} \leq \log(2^h)$. 
So then, what is $\log(n!)$?

- $\log(n!) = \log(1 \times 2 \times 3 \times 4 \times \cdots \times n)$
- $\log(n!) = \log(1) + \log(2) + \log(3) + \cdots + \log(n)$
- $\log(n!) = n \times \log(\text{anything}) = n \times \log(n)$ ■
Conclusions

• No sorting algorithm can guarantee to use fewer than $O(n \times \log(n))$ compares on all inputs.

  – **Corollary:** Mergesort is an *asymptotically optimal* compare based sorting algorithm.

  • “Both the number of compares used by Mergesort in the worst case and the minimum number of compares that any compare-based sorting algorithm can guarantee are $O(n \times \log(n))$.”
Conclusions

• So... are we done with sorting algorithms, now that we found the best we can do?
  – Mergesort is not optimal with respect to space usage.
  – The worst case may not be likely in practice.
  – Operations other than compare may be more important in practice.
  – Can we sort data with using *any* compares?
Any Questions?