Unit Overview
Algorithmic Efficiency & Sorting

Major Topics
- Introduction to Analysis of Algorithms
- Introduction to Sorting
- Comparison Sorts I
- More on Big-O
  - The Limits of Sorting
  - Divide-and-Conquer
  - Comparison Sorts II
  - Comparison Sorts III
- Radix Sort
- Sorting in the C++ STL
Review
Introduction to Analysis of Algorithms [1/2]

**Efficiency**
- General: using few resources (time, space, bandwidth, etc.).
- Specific: fast (time).
  - Also can be qualified, e.g., space efficiency.

**Analyzing Efficiency**
- Measure running time in **steps**.
- Determine how the **size of the input** affects running time.
- **Worst case**: max steps for given input size.

**Scalable**: works well with large problems. Also “**scales well**”.
We say $g(n)$ is $O(f(n))$ if

- There exist constants $k$ and $n_0$ such that
- $g(n) \leq k \times f(n)$, whenever $n \geq n_0$.

Useful: $g(n) =$ max steps used by an algorithm for input of size $n$.

Efficiency categories we will use.

<table>
<thead>
<tr>
<th>Using Big-O</th>
<th>In Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant time</td>
</tr>
<tr>
<td>$O((\log_b n)$, for some $b &gt; 1$</td>
<td>Logarithmic time</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear time</td>
</tr>
<tr>
<td>$O(n \log_b n)$, for some $b &gt; 1$</td>
<td>Log-linear time</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic time</td>
</tr>
<tr>
<td>$O(b^n)$, for some $b &gt; 1$</td>
<td>Exponential time</td>
</tr>
</tbody>
</table>

I will also allow $O(n^3)$, $O(n^4)$, etc.
Review
Introduction to Sorting — Basics, Analyzing

**Sort:** Place a collection of data in order.

**Key:** The part of the data item used to sort.

**Comparison sort:** A sorting algorithm that gets its information by comparing items in pairs.

A **general-purpose comparison sort** places no restrictions on the size of the list or the values in it.

Five criteria for analyzing a general-purpose comparison sort:

- (Time) Efficiency
- Requirements on Data
- Space Efficiency
- Stability
- Performance on Nearly Sorted Data

**In-place** = no large additional space required.

**Stable** = never changes the order of equivalent items.

1. All items close to proper places, OR
2. few items out of order.
Review  
Introduction to Sorting — Overview of Algorithms  

There is no known sorting algorithm that has all the properties we would like one to have.  

We will examine a number of sorting algorithms. Most of these fall into two categories: $O(n^2)$ and $O(n \log n)$.

- **Quadratic-Time** [$O(n^2)$] Algorithms
  - ✓ Bubble Sort
  - ✓ Insertion Sort
    - Quicksort
    - Treesort (later in semester)
- **Log-Linear-Time** [$O(n \log n)$] Algorithms
  - Merge Sort
  - Heap Sort (mostly later in semester)
  - Introsort (not in the text)
- **Special Purpose — Not Comparison Sorts**
  - Pigeonhole Sort
  - Radix Sort
Review
Comparison Sorts I — Bubble Sort: Analysis

(Time) Efficiency 😞
- Bubble Sort is $O(n^2)$.
- Bubble Sort also has an average-case time of $O(n^2)$. 😞

Requirements on Data 😊
- Bubble Sort does not require random-access data.
- It works on Linked Lists.

Space Efficiency 😊
- Bubble Sort can be done in-place.

Stability 😊
- Bubble Sort is stable.

Performance on Nearly Sorted Data 😊/😞
- (1) We can write Bubble Sort to be $O(n)$ if no item is far out of place. 😊
- (2) Bubble Sort is $O(n^2)$ even if only one item is far out of place. 😞
Review
Comparison Sorts I — Insertion Sort: Illustration

Items to left of bold bar are sorted.

Bold item = item to be inserted into sorted section.

A list of size 1 is always sorted

Insert 5

Insert 9

Insert 2

Insert 5

Insert 5

Insert 2

Insert 5

Sorted
Review
Comparison Sorts I — Insertion Sort: Analysis

(Time) Efficiency 😞
- Insertion Sort is $O(n^2)$.
- Insertion Sort also has an average-case time of $O(n^2)$. 😞

Requirements on Data ☺
- Insertion Sort does not require random-access data.
- It works on Linked Lists.*

Space Efficiency ☻
- Insertion Sort can be done in-place.*

Stability ☻
- Insertion Sort is stable.

Performance on Nearly Sorted Data ☻
- (1) Insertion Sort can be written to be $O(n)$ if each item is at most some constant distance from its proper place.*
- (2) Insertion Sort can be written to be $O(n)$ if only a constant number of items are out of place.

*For one-way sequential-access data (e.g., Linked Lists) we give up EITHER in-place OR $O(n)$ on type (1) nearly sorted data.
Insertion Sort is too slow for general-purpose use. However, Insertion Sort is useful in certain special cases.

- Insertion Sort is fast (linear time) for **nearly sorted data**.
- Insertion Sort is also considered fast for **small lists**.

Insertion Sort often appears as part of another algorithm.

- Most good sorting methods call Insertion Sort for small lists.
- Some sorting methods get the data nearly sorted, and then finish with a call to Insertion Sort. (More on this later.)
Review
More on Big-O

Three ways to talk about how fast a function grows. $g(n)$ is:

- $O(f(n))$ if $g(n) \leq k \times f(n)$ ... 
- $\Omega(f(n))$ if $g(n) \geq k \times f(n)$ ...
- $\Theta(f(n))$ if both of the above are true.
  - Possibly with different values of $k$.

Useful: Let $g(n)$ be the max number of steps required by some algorithm when given input of size $n$.

Or: Let $g(n)$ be the max amount of additional space required when given input of size $n$. 
The Limits of Sorting
Introduction

We have mentioned that most sorting algorithms fall into one of two categories:

- **Slow**: $O(n^2)$.
- **Fast**: $O(n \log n)$.
  - We have not discussed any of these fast algorithms yet, however.

Can we do even better?

- No, not with a general purpose comparison sort.
- Writing a general purpose comparison sort that lies in any time-efficiency category faster than $O(n \log n)$ is **impossible**.
  - Remember: worst-case analysis.
- More precisely: We can **prove** that the **worst-case number of comparisons** performed by a general purpose comparison sort must be $\Omega(n \log_2 n)$.
The Limits of Sorting
Thinking about Sorting

Sorting is determining the ordering of a list. Many orderings are possible. Each time we do a comparison, we find the relative order of two items. Say \( x < y \); we can throw out all orderings in which \( y \) comes before \( x \). We cannot stop until only one possible ordering is left.

Example

- Bubble Sort the list 2 3 1.

### Bubble Sort

<table>
<thead>
<tr>
<th>Pass 1</th>
<th>Pass 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="diagram1.png" alt="Pass 1 Diagram" /></td>
<td><img src="diagram2.png" alt="Pass 2 Diagram" /></td>
</tr>
</tbody>
</table>

### Comparisons

- \( 3 \not< 2 \) No
- \( 1 \not< 3 \) Yes
- \( 1 \not< 2 \) Yes

### Alternate View

**Possible orderings:**
- 123 132
- 213 231
- 312 321

**Possible orderings:**
- 123 132
- 123 132
- 123 132
- 123 132

**Possible orderings:**
- 123 132
- 123 213
- 123 312
- 123 321

12 Oct 2009

CS 311 Fall 2009
The Limits of Sorting
Proof Idea

Based on the idea in the previous slide, we can prove that the worst-case number of comparisons done by a general purpose comparison sort must be $\Omega(n \log_2 n)$.

Outline of the Proof

- We are given a list of $n$ items to be sorted.
- There are $n! = n \times (n-1) \times \ldots \times 3 \times 2 \times 1$ orderings of $n$ items.
- We start with all $n!$ orderings. We do comparisons, throwing out orderings that do not match our new information.
- With each comparison, we cannot force more than half of the orderings to be thrown out. (Remember: worst case.)
- How many times must we cut $n!$ in half, to get 1? Answer: $\log_2(n!)$, or a little more, since we deal only with integers. The worst case uses at least that many comparisons.
- And $\log_2(n!)$ turns out to be close to $n(\log_2 n - \log_2 e) + \frac{1}{2} \log_2(2\pi n)$.
  - We will not verify this step. Look up “Stirling’s Approximation” for info.
Divide-and-Conquer
Introduction

The “obvious” way to search is Sequential Search. But we have seen how to do better on sorted data: Binary Search. Binary Search splits its input into parts and handles them recursively. This last idea is called **divide-and-conquer**.

- A common way to design fast algorithms.

Questions

- How do we analyze the efficiency of algorithms that use divide-and-conquer?
- Can we use divide-and-conquer to come up with an improved sorting algorithm? One that is $O(n \log n)$? (We have not seen any, yet.)
Divide-and-Conquer
The Master Theorem [1/3]

Suppose we are analyzing an algorithm.
- It takes input of size $n$.
- The number of steps it requires is at most $T(n)$.
  - $T$ is a (mathematical) function.
- We want to know what $T(n)$ is, roughly.

Suppose our algorithm uses divide-and-conquer:
- It splits the input into $b$ nearly equal-sized parts.
- It makes $a$ recursive calls each taking one of the parts as input.
  - Write $a = b^k$, for some $k \geq 0$.
- It does some other work requiring $f(n)$ operations.

This gives us a recurrence relation:
- $T(n) = b^k \cdot T(n/b) + f(n)$.

Given such a recurrence, we can often use the Master Theorem.
The Master Theorem*

- Suppose $T(n) = b^k T(n/b) + f(n)$, where $b > 1$ and $k \geq 0$.
  - “$n/b$” means the next integer going up or down, as appropriate.

- Case 1
  - If $f(n)$ is $O(n^{k-\varepsilon})$, for some $\varepsilon > 0$, then $T(n)$ is $\Theta(n^k)$.

- Case 2
  - If $f(n)$ is $\Theta(n^k)$, then $T(n)$ is $\Theta(n^k \log n)$.

*The Master Theorem, as it is usually stated, actually says a little more than this.

- There is a Case 3, which is more complex, but essentially says that, if $f(n)$ is large, then $T(n)$ is of the same order as $f(n)$.
- However, Case 1 and Case 2 are all we will use in this class.
Divide-and-Conquer
The Master Theorem [3/3]

How the Master Theorem is applied to analyze a recursive algorithm:

- An algorithm takes input of size $n$. It splits its input into nearly equal-sized parts, and makes recursive calls, each call handling one of the parts.
  - $b$ is the number of nearly equal-sized parts.
  - $b^k$ is the number of recursive calls. Find $k$.
  - $f(n)$ is the amount of extra work done, in each function call.
- Case I: $f(n)$ is $O(n^{\text{less than } k}) \rightarrow$ Algorithm is $\Theta(n^k)$.
- Case II: $f(n)$ is $\Theta(n^k) \rightarrow$ Algorithm is $\Theta(n^k \log n)$.

Find $f(n)$, hopefully involving a power of $n$.

Here, the exponent of $n$ is $k$.

Here, the exponent of $n$ is less than $k$. 
Sequential Search is easily seen to be $O(n)$.

Analyze Binary Search

- Find $b$, $k$, $f$
  - Binary Search splits its input into 2 nearly-equal-sized parts.
    - Thus $b = 2$.
  - Binary Search makes 1 recursive call.
    - Thus $b^k = 1$, and so $k = 0$.
  - In addition, Binary Search does a comparison: constant time.
    - So $f(n)$ is $O(1)$, and, in fact, $\Theta(1)$.

- Which Case?
  - $1 \equiv n^0 = n^k$. The exponent of $n$ is $k$, and so we are in Case 2.

- Conclusion
  - By the Master Theorem, Case 2, $T(n)$ is $\Theta(n^k \log n)$.
  - Simplify: Binary Search is $\Theta(\log n)$, and therefore $O(\log n)$. 


Divide and Conquer is a common way to get algorithms with order $O(\log n)$ or $O(n \log n)$.

Earlier we said that the base of the logarithm does not matter. Why is this?

- Suppose (for example) that an algorithm takes $3 \log_2 n$ steps.
- Clearly, this algorithm is $O(\log_2 n)$.
- Is it also $O(\log_{10} n)$?
  - Yes!
  - $3 \log_2 n = 3(\log_2 10 \times \log_{10} n) = (3 \log_2 10) \times \log_{10} n$.
- Fact: If $a$ and $b$ are greater than 1, $O(\log_a n)$ and $O(\log_b n)$ mean the same thing.
- Thus we generally leave off the base and say “$O(\log n)$”.
- Similarly, we say “$O(n \log n)$”, etc.
Divide-and-Conquer Thoughts

We can use the Master Theorem “backwards”.

- We have been saying, “Here is the order of $f(n)$; what is the order of the algorithm as a whole?”
- Instead, we can say, “We want an algorithm with a certain order; how large is $f(n)$ allowed to be?”

How it works: Suppose we use divide-and-conquer.

- We split our input into $b$ nearly equal-sized parts.
- We make $b^k$ recursive calls.
- The Master Theorem says: To be efficient $[O(n^k \log n)]$, we can only do additional work requiring $O(n^k)$ steps.

Getting Logarithmic and Log-Linear Time

- If an algorithm is $O(\log n)$ or $O(n \log n)$, then it there is a good chance that it uses some form of divide-and-conquer.
- So divide-and-conquer might get us a fast sorting algorithm.
Comparison Sorts II
Merge Sort — Introduction

How can we use divide-and-conquer to build a better sorting algorithm?

- Suppose we split a list into two equal-sized (or nearly so) pieces, and sort each piece recursively.
- Then **merge** the two parts into a single sorted list.
  - Do this in a stable manner: **Stable Merge**.
- The resulting sorting algorithm is called **Merge Sort**.

If we want an $O(n \log n)$ sort, how long can a Stable Merge operation take?

- 2 nearly equal-sized parts ($b = 2$).
- 2 recursive calls ($b^k = 2$, and so $k = 1$).
- Case 2. Stable Merge is allowed $O(n^k) = O(n^1) = O(n)$: linear time.
Comparison Sorts II
Merge Sort — Merging in a Linked List

We can do an efficient Stable Merge of a **Linked List** in-place.

- To merge two sorted ranges within a Linked List:
  - Keep two pointers, A and B. A starts at the head, B at the end of range #1.
  - Check whether the item after B’s node is less than the item after A’s node. If so, remove the item after B’s node and re-insert it after A.
    - This requires only pointer operations. We do not actually move any nodes.
  - Advance A and B as appropriate and repeat.
Comparison Sorts II
Merge Sort — General-Purpose Merge

Efficient Stable Merge in an array generally uses a separate buffer.
- Note, this Stable Merge algorithm does not require an array; it works with just about any kind of data.

![Diagram of Merge Sort]

As before, use two pointers. Check which item comes first, and copy that to the buffer. Advance the pointers as appropriate.

![Intermediate stage in Merge operation]

At the end, we may wish to copy the buffer back to the original array.
Comparison Sorts II
Merge Sort — Putting It All Together

Conclusion: For both Linked Lists and arrays, a Stable Merge can be done in linear time.

How do we write the sort itself?

- We find the middle of the list, recurse twice, and Merge.
  - Finding the middle of an array: $O(1)$.
  - Finding the middle of a Linked List: $O(n)$.
- However, we already do the linear-time Merge operation at each step. Adding $O(1)$ or $O(n)$ additional steps only makes this into a slightly slower linear-time operation.
  - $O(1) + O(n) = O(n) + O(n) = O(n)$.
- Conclusion: Merge Sort might be written differently for different types of data. However, it is always $O(n \log n)$.

Now let’s write Merge Sort, and analyze it in detail.
Comparison Sorts II
Merge Sort — Do It

When writing Merge Sort, the only part that takes any work is the Stable Merge operation. Once Stable Merge is done, writing the sort is very easy.

TO DO

- Implement Merge Sort.
  - Use the general-purpose Stable Merge algorithm, and make it a separate function.
- Analyze.
  - See the next slide.

Notes

- We allocated a buffer every time a Stable Merge was done. It would be more efficient to allocate once, in a wrapper function, and then pass a pointer when calling each function.
- We merged to the buffer and then copied the buffer back. This copy-back can often be avoided, but it adds complexity to the code.

Done. See `merge_sort.cpp`, on the web page.
Comparison Sorts II
Merge Sort — Analysis

Efficiency 😊
- Merge Sort is $O(n \log n)$.
- Merge Sort also has an average-case time of $O(n \log n)$.

Requirements on Data 😊
- Merge Sort does not require random-access data.

Space Usage 😊/😊/😊
- Recursive Merge Sort uses stack space: recursion depth $\approx \log_2 n$.
  - An iterative version can avoid this (small) memory requirement.
- For a Linked List, no more is needed: $O(\log n)$ additional space. 😊
  - Or $O(1)$ additional space, for an iterative version. 😊
- General-purpose Merge Sort uses a buffer: $O(n)$ additional space. 😞

Stability 😊
- Merge Sort is stable.

Performance on Nearly Sorted Data 😊
- Merge Sort is still log-linear time on nearly sorted data.

See iterative_merge_sort.cpp, on the web page.
Comparison Sorts II
Merge Sort — Comments

Merge Sort is very practical and is often used.
  - Merge Sort is considered to be the **fastest known** algorithm:
    - When a stable sort is required.
    - When sorting a Linked List.
  - Merge Sort is the usual way to implement two of the six sorting algorithms in the C++ Standard Template Library.

Stable Merge is done differently for different kinds of data.
  - Thus, while the overall structure is the same, different versions of Merge Sort can differ greatly in lower-level details.
  - Merge Sort is *almost* two different algorithms.

I have seen research indicating that one can do a linear-time Stable Merge in an array without an extra buffer.
  - However, even highly regarded Merge Sort implementations still allocate the buffer.
  - C++ Standard Library algorithm `std::stable_sort` tries to allocate a buffer. If this fails, then it is allowed to be $O(n \log n)^2$.
  - I have not quite figured out this issue.
Comparison Sorts II
Merge Sort — Comparing Algorithms

Merge Sort does essentially everything we would like a sorting algorithm to do:

- It runs in $O(n \log n)$ time.
- It is stable.
- It works well with various kinds of data — especially Linked Lists.

Thus, Merge Sort is a good standard by which to judge sorting algorithms.

When evaluating some other sorting algorithm, ask:

- How is this algorithm better than Merge Sort?
  - If it is not better in any way, then use Merge Sort.
- How is this algorithm worse than Merge Sort?
  - If it is better than Merge Sort in some way, then it must also be worse in some way.
- In this application, are the advantages worth the disadvantages?