Unit Overview
Algorithmic Efficiency & Sorting

Major Topics
- ✔ Analysis of Algorithms
- ✔ Introduction to Sorting
- ✔ Comparison Sorts I
- ✔ Asymptotic Notation
- ✔ Divide and Conquer
- ✔ Comparison Sorts II
- ✔ The Limits of Sorting
(part) ✔ Comparison Sorts III
  - ✔ Non-Comparison Sorts
  - ✔ Sorting in the C++ STL
Review
Useful Rules

- When determining big-O, we can collapse any constant number of steps into a single step.
- **Rule of Thumb.** For nested “real” loops, order is \( O(n^t) \), where \( t \) is the number of nested loops.
- **Addition Rule.** \( O(f(n)) + O(g(n)) \) is either \( O(f(n)) \) or \( O(g(n)) \), whichever is larger. And similarly for \( \Theta \). This works when adding up any fixed, finite number of terms.
Sorting Algorithms Covered

- Quadratic-Time \([O(n^2)]\) Comparison Sorts
  - Bubble Sort
  - Insertion Sort
- (part) Quicksort
- Log-Linear-Time \([O(n \log n)]\) Comparison Sorts
  - Merge Sort
  - Heap Sort (mostly later in semester)
  - Introsort
- Special Purpose—Not Comparison Sorts
  - Pigeonhole Sort
  - Radix Sort
Review
Asymptotic Notation

\( g(n) \) 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 are true—possibly with different values of \( k \).

<table>
<thead>
<tr>
<th>( g(n) )</th>
<th>1</th>
<th>( n )</th>
<th>( n \log n )</th>
<th>( n^2 )</th>
<th>( 5n^2 )</th>
<th>( n^2 \log n )</th>
<th>( n^3 )</th>
<th>( n^4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(n^2) )</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>( \Omega(n^2) )</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>( \Theta(n^2) )</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>YES</td>
<td>YES</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

In an algorithmic context, \( g(n) \) might be:
- The maximum number of basic operations performed by the algorithm when given input of size \( n \).
- The maximum amount of additional space required.

**In-place** means using \( O(1) \) additional space.

\( \Theta \) is very useful!
\( \Omega \) not as much.
A Divide/Decrease-and-Conquer algorithm needs analysis.

- It splits its input into $b$ nearly equal-sized parts.
- It makes $a$ recursive calls, each taking one part.
- It does other work requiring $f(n)$ operations.

To Analyze

- Find $b$, $a$, $d$ so that $f(n)$ is $\Theta(n^d)$—or $O(n^d)$.
- Compare $a$ and $b^d$.
- Apply the appropriate case of the Master Theorem.

The **Master Theorem**

Suppose $T(n) = a \cdot T(n/b) + f(n)$;

- $a \geq 1$, $b > 1$, $f(n)$ is $\Theta(n^d)$.
  - “$n/b$” can be a nearby integer.

Then:

- Case 1. If $a < b^d$, then $T(n)$ is $\Theta(n^d)$.
- Case 2. If $a = b^d$, then $T(n)$ is $\Theta(n^d \log n)$.
- Case 3. If $a > b^d$, then $T(n)$ is $\Theta(n^k)$, where $k = \log_b a$.

We may also replace each “$\Theta$” above with “$O$”. 
Review
Comparison Sorts II — Merge Sort

**Merge Sort** splits the data in half, recursively sorts both, merges.

**Analysis**
- Efficiency: $\Theta(n \log n)$. Avg same. 😊
- Requirements on Data: Works for Linked Lists, etc. 😊
- Space Efficiency: $\Theta(\log n)$ space for recursion. Iterative version is in-place for Linked List. $\Theta(n)$ space for array. 😖/😊/😊
- Stable: Yes. 😊
- Performance on Nearly Sorted Data: Not better or worse. 😊

**Notes**
- Practical & often used.
- Fastest known for (1) stable sort, (2) sorting a Linked List.

See `merge_sort.cpp`. 
The worst-case number of comparisons performed by a general-purpose comparison sort must be $\Omega(n \log n)$.

Reasoning:

- 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.
- Start with all $n!$ orderings. Do comparisons, throwing out orderings that do not match what we know, until just one ordering is left.
- With each comparison, we cannot guarantee that more than half of the orderings will be thrown out.
- How many times must we cut $n!$ in half, to get 1? Answer: $\log_2(n!)$, which is $\Theta(n \log n)$. (Use Stirling’s Approximation.)
- The worst case number of comparisons done by a general-purpose comparison sort must be at least that big. Thus: $\Omega(n \log n)$. 
Review
Comparison Sorts III — Quicksort

Quicksort chooses pivot, does Partition, recursively sorts sublists.

Partition

- Place items less than pivot before the pivot, other items after.
- Two common algorithms: Hoare’s, Lomuto’s. We covered Hoare’s.
- Linear time, in place, not stable.
Comparison Sorts III

continued
Comparison Sorts III
Quicksort — CODE

TO DO

- Write Quicksort, with the in-place Partition being a separate function.
  - Use Hoare’s Partition Algorithm, written as a separate function.
  - Require random-access iterators.

Done. See quicksort1.cpp.
Quicksort has a serious problem.

- Try applying the Master Theorem. It does not work, because Quicksort may not split its input into nearly equal-sized parts.
- The pivot *might* be chosen very poorly. In such cases, Quicksort has linear recursion depth and does linear-time work at each step.
- Result: Quicksort is $\Theta(n^2)$. 😞
- And the worst case happens when the list is *already sorted*!

However, Quicksort’s average-case time is very fast.

Quicksort is *usually* very fast, so people want to use it.

In the decades following Quicksort’s introduction in 1961, many people published suggested improvements. We will look at three of the most successful.
Choose the pivot using **Median-of-3**.

- Look at 3 items in the list: first, middle, last.
- Let the pivot be the one that is between the other two (by <).

This gives good performance on most nearly sorted data—as do other similar pivot-selection schemes.

But Quicksort with Median-of-3 (or similar) is slow for *other* data. So: \(\Theta(n^2)\).

Look into “Median-of-3 killer sequences”.

Ideally, our pivot is the *median* of the list.

- If it were, then Partition would create lists of (nearly) equal size, and we could apply the Master Theorem, which would tell us:
  - If we do $O(n)$ extra work at each step, then we get an $O(n \log n)$ algorithm (same computation as for Merge Sort).

Can we find the median of a list in linear time?

- Yes! Use **BFPRT** (the Blum-Floyd-Pratt-Rivest-Tarjan Algorithm).

**Catchy name, eh?**

**It is also called Median of Medians.**

However, this is not a very fast linear time. The resulting sorting algorithm is log-linear time, but much slower than Merge Sort.

<sigh>
Okay, stick with Median-of-3.
How much additional space does Quicksort use?

- Partition is in-place and Quicksort uses few local variables.
- However, Quicksort is recursive.
- Quicksort’s additional space usage is thus proportional to its recursion depth ...
- ... which is linear. Worst-case additional space used: $\Theta(n)$. 😞

We can significantly improve this:

- Do the *larger* of the two recursive calls last.
- Do tail-recursion elimination on this final recursive call.
- Result: Recursion depth & additional space usage: $\Theta(\log n)$. 😊
- And this additional space need not hold data items. (Why is this kinda good?)
A possible speed-up: finish with Insertion Sort

- Stop Quicksort from going to the bottom of its recursion. We end up with a nearly sorted list.
- Finish sorting this list using one call to Insertion Sort.
- Apparently this is generally faster*, but it is still $\Theta(n^2)$.
- Note. This is not the same as using Insertion Sort for small lists.

*I have read that this tends to adversely affect the number of cache hits.
Better Quicksort — CODE

TO DO

- Rewrite our Quicksort to include the optimizations discussed:
  - Median-of-3 pivot selection.
  - Tail-recursion elimination on the larger recursive call.
  - Recursive calls to sort small lists do nothing. End with Insertion Sort of entire list.

Done. See quicksort2.cpp.
Comparison Sorts III
Better Quicksort — What is Needed?

We want an algorithm that:
- Is as fast as Quicksort on average.
- Has good $\Theta(n \log n)$ worst-case performance.

But for over three decades no one found one.

Some said (and some still say), “Quicksort’s bad behavior is very rare; we can ignore it.”

I suggest that this is not a good way to think.
- Sometimes poor worst-case behavior is okay; sometimes it is not.
- Know what is important in your situation.
- Remember that malicious users exist, particularly on the Web.

In 1997, a solution to Quicksort’s big problem was finally published. We will discuss this. But first, we analyze Quicksort.
Comparison Sorts III
Better Quicksort — Analysis of Quicksort

Efficiency 😞

- Quicksort is $\Theta(n^2)$.
- Quicksort has a very good $\Theta(n \log n)$ average-case time. 😊😊

Requirements on Data 😞

- Non-trivial pivot-selection algorithms (Median-of-3 and similar) are only efficient for random-access data.

Space Usage 😞

- Quicksort uses space for recursion.
  - Additional space: $\Theta(\log n)$, if clever tail-recursion elimination is done.
  - Even if all recursion is eliminated, $O(\log n)$ additional space is still used.
  - This additional space need not hold any data items.

Stability 😞

- Efficient versions of Quicksort are not stable.

Performance on Nearly Sorted Data 😞

- An unoptimized Quicksort is slow on nearly sorted data: $\Theta(n^2)$.
- Quicksort + Median-of-3 is $\Theta(n \log n)$ on most nearly sorted data.
In 1997, algorithms researcher David Musser introduced a simple algorithm-design idea.

- For some problems, there are known algorithms with very good average-case performance and very poor worst-case performance.
- Quicksort is the best known of these, but there are others.
- Musser’s idea is that, when such an algorithm runs, it should keep track of its performance. If it is not doing well, then it can switch to a different algorithm that has a better worst-case.
- Musser called this technique **introspection**, since the algorithm is examining itself.

The most important application of introspection is to sorting. We can eliminate the awful worst-case behavior of Quicksort, using **introspective sorting**.
Comparison Sorts III
Introsort — Heap Sort Preview

Here is a preview of a sort we will cover later in the semester. Eventually, we will study *Priority Queues*.

- In a normal **Queue**, we insert items, and then we can remove them in the same order (FIFO = First-In-First-Out).
- In a **Priority Queue**, each item has a **priority**. Items are removed in order of priority: highest to lowest.
- Set priority = value of item. Items come out in descending order.

A **Binary Heap** data structure can work as a Priority Queue.

- To sort: create a Binary Heap containing the data to be sorted. Remove items, one by one, storing them in a list in reverse order.
- This algorithm is called **Heap Sort**.

We study Heap Sort in detail later in the semester. For now:

- Heap Sort is $\Theta(n \log n)$ time.
- Heap Sort is in-place.
- Heap Sort requires random-access data.
- Heap Sort forms part of a fast sort called *Introsort* ...
Recall: Quicksort calls Partition—which is $\Theta(n)$—and then recurses.

- If the recursion depth (ignore tail-recursion elimination!) is logarithmic, then Quicksort does only $\Theta(n \log n)$ basic operations.
- Thus, Quicksort is slow only when the recursion gets too deep.

Apply introspection:

- Do optimized Quicksort, but keep track of the recursion depth.
- If the depth exceeds some threshold—Musser suggested $2 \log_2 n$—then switch to Heap Sort for the current sublist being sorted.

The resulting algorithm is called Introsort \([\text{introspective sort}]\).

Musser’s 1997 paper recommends the optimizations we covered:

- Median-of-3 pivot selection.
- Tail-recursion elimination on one of the recursive calls.
  - But now it does not matter which recursive call.
- Stop the recursion prematurely, and finish with Insertion Sort.
  - *Maybe.* This can adversely affect cache performance.
Comparison Sorts III
Introsort — Diagram

Here is an illustration of how Introsort works.

- In practice, the recursion will be much deeper than this.
- We might not do the Insertion Sort, due to its effect on cache hits.

When the sublist to sort is very small, do not recurse. Insertion Sort will finish the job later.

Recursion Depth Limit
- When the recursion depth is too great, switch to Heap Sort to sort the current sublist.

Here, the list is nearly sorted. Finish with a (linear time!) Insertion Sort.

Tail-recursion elimination on one recursive call. But it still counts toward the “recursion depth”.
Comparison Sorts III
Introsort — Analysis

Efficiency 😊😊
- Introsort is $\Theta(n \log n)$.
- Introsort also has an average-case time of $\Theta(n \log n)$—of course.
  - Its average-case time is just as good as Quicksort. 😊😊

Requirements on Data 😞
- Introsort requires random-access data.

Space Usage 😞
- Introsort uses space for recursion.
  - Additional space: $\Theta(\log n)$—even if all recursion is eliminated.
  - This additional space need not hold any data items.

Stability 😞
- Introsort is not stable.

Performance on Nearly Sorted Data 😞
- Introsort is not significantly faster or slower on nearly sorted data.
Our discussion of Quicksort & Introsort might suggest that their average-case time is significantly better than Merge Sort. Historically, this has been largely the case. But experience shows that, on modern architectures, Merge Sort can be faster.

This is a tricky issue. Relative speed depends on:

- The processor used, and the performance of its cache.
- The type of the data being sorted.
- The data structure used, and its size.

It appears to me [GGC] that, in practice, use of the Quicksort family of algorithms—including Introsort—is fading. For example, the old C Standard Library function `qsort` traditionally used Quicksort (thus the name). But some implementations now use Merge Sort.