Comparison Sorts III continued

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, October 16, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

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
- (part) Comparison Sorts III
- Radix Sort
- Sorting in the C++ STL

Review
Introduction to Analysis of Algorithms

Efficiency

- General: using few resources (time, space, bandwidth, etc.).
- Specific: fast (time).

Analyzing Efficiency

- Determine how the **size of the input** affects running time, measured in **steps**, in the **worst case**.

Scalable: works well with large problems.

Using Big-O	In Words
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(b^n)$, for some $b > 1$	Exponential time

Cannot read all of input ↑

Probably not scalable ↓

Faster ↑

Slower ↓

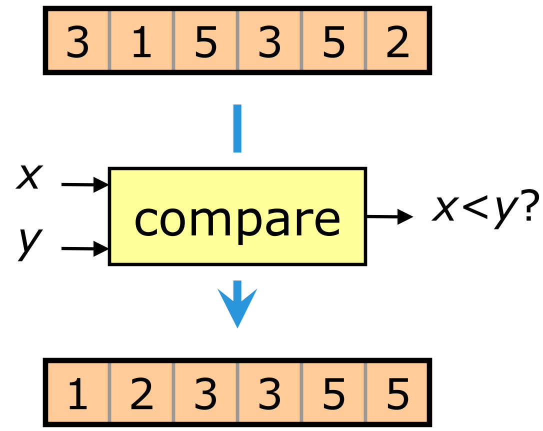
Review
Introduction to Sorting — Basics, Analyzing

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

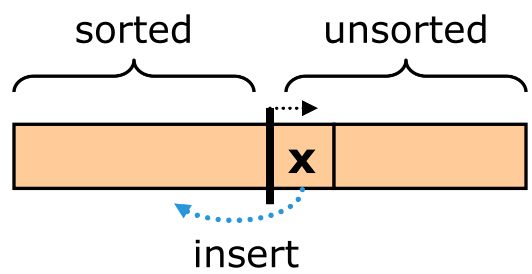
There is no *known* sorting algorithm that has all the properties we would like one to have.

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
  - (part) 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 — Insertion Sort

Insertion Sort repeatedly does this:



Analysis

- Efficiency: $O(n^2)$. Average case same. ☹
- Requirements on data: Works for Linked Lists, etc. ☺
- Space Efficiency: In-place. ☺
- Stable: Yes. ☺
- Performance on Nearly Sorted Data: $O(n)$ for both kinds. ☺

Notes

- Too slow for general-purpose use.
- Works well on **nearly sorted data** and **small lists**.
- Thus, used as **part of other algorithms**.

Review
More on Big-O, The Limits of Sorting

Three ways to talk about how fast a function grows.

$g(n)$ is:

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

In an algorithmic context, $g(n)$ might be the max number of steps required by some algorithm when given input of size n, or the max amount of additional space required.

Fact. Every general-purpose comparison sort does $\Omega(n \log n)$ comparisons in the worst case.

Review
Divide-and-Conquer

A common algorithmic strategy is called **divide-and-conquer**: split the input into pieces, and handle these with recursive calls.

If an algorithm using divide-and-conquer splits the input into **nearly equal-sized parts**, then we can analyze it using the **Master Theorem**.

Important!

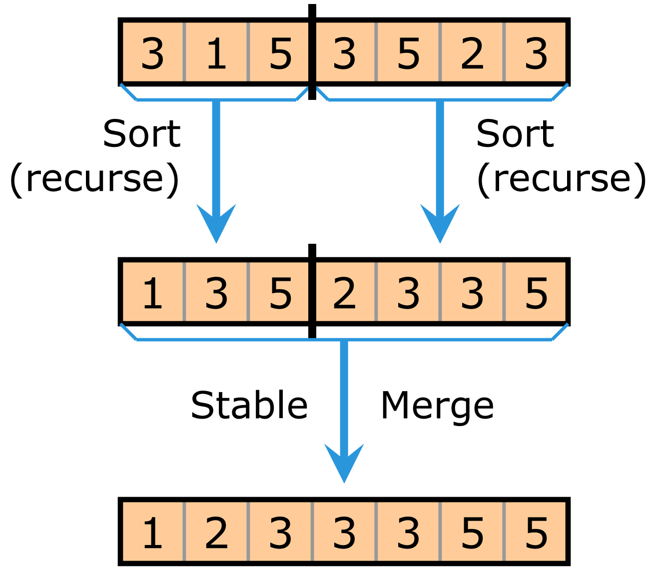
- b is the number of nearly equal-sized parts.
- b^k is the number of recursive calls. Find k.
- $f(n)$ is the amount of extra work done (number of steps).
  - Hopefully, $f(n)$ looks like n raised to some power.
- If the power is less than k, then the algorithm is $O(n^k)$.
- If the power is k, then the algorithm is $\Theta(n^k \log n)$.

Review
Comparison Sorts II — Merge Sort

Merge Sort splits the data in half, recursively sorts each half, and then merges the two.

Stable Merge

- Linear time, stable.
- In-place for Linked List. Uses buffer [*O*(*n*) space] for array.



Analysis

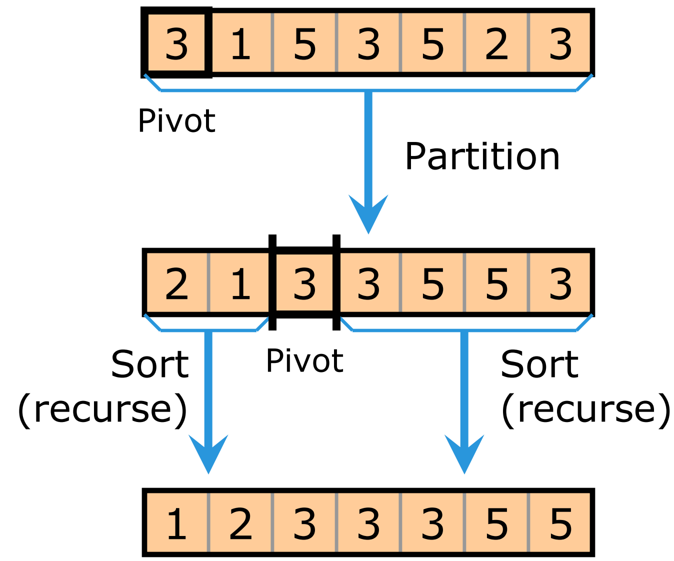
- Efficiency: *O*(*n* log *n*). Average same. ☺
- Requirements on data: Works for Linked Lists, etc. ☺
- Space Efficiency: *O*(log *n*) space for Linked List. Can eliminate recursion to make this in-place. *O*(*n*) space for array. 😐/☺/☹
- Stable: Yes. ☺
- Performance on Nearly Sorted Data: Not better or worse. 😐

Notes

- Practical & often used.
- Fastest known for (1) stable sort, (2) sorting a Linked List.
- Good standard for judging sorting algorithms

Quicksort is another divide-and-conquer algorithm. Procedure:

- Choose a list item (the **pivot**).
- Do a **Partition**: put items less than the pivot before it, and items greater than the pivot after it.
- Recursively sort two sublists: items before pivot, items after pivot.



We did a simple pivot choice: the first item. Later, we improve this.

Fast Partition algorithms are in-place, but not stable.

- Note: In-place Partition does not give us an in-place Quicksort. Quicksort uses memory for recursion.

Review
Comparison Sorts III — Quicksort: Write It

TO DO

- Write Quicksort, with the in-place Partition being a separate function.

Done. See **`quicksort1.cpp`**, *on the web page.*

Review
Comparison Sorts III — Better Quicksort: Problem

Quicksort has a problem.

- In the worst case, the pivot is chosen poorly.
- Thus: linear recursion depth, and so Quicksort is $O(n^2)$. ☹
- And the worst case happens when the list is **already sorted**!

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

- This is $O(n \log n)$ and typically significantly faster than Merge Sort.

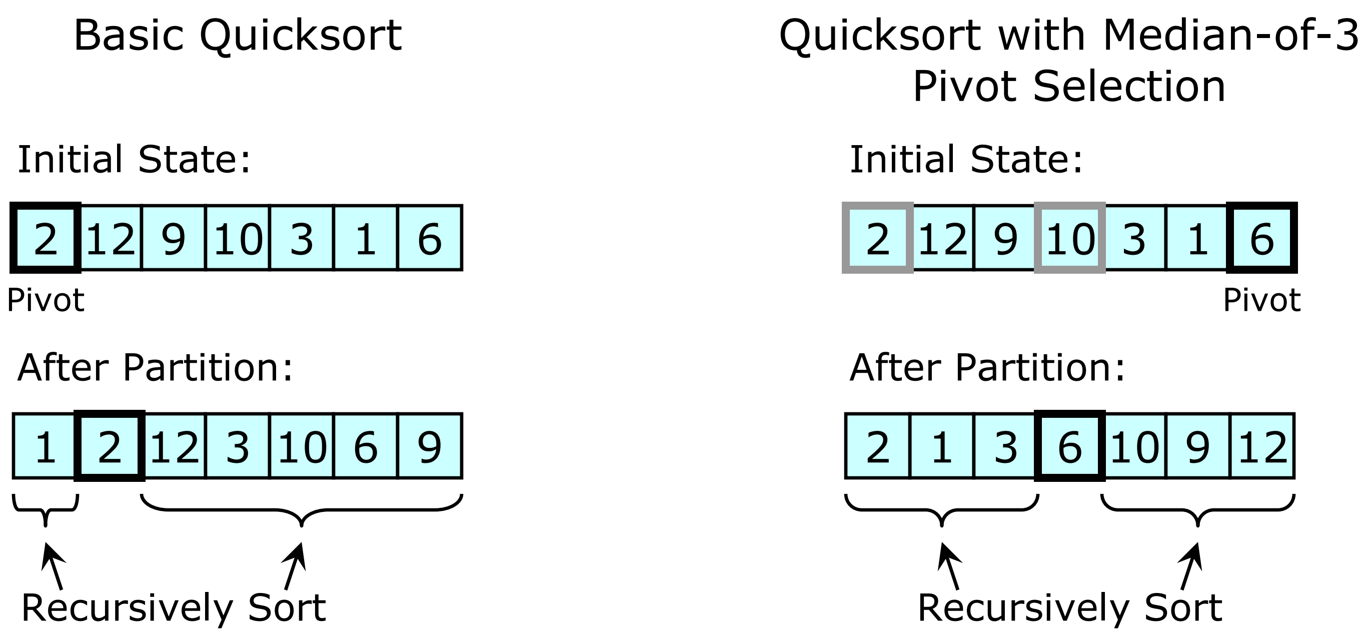
Quicksort is *usually* very fast; thus, people want to use it.

- So we try to figure out how to make it better.
- We look at three of the best optimizations …

Review
Comparison Sorts III — Better Quicksort: Opt. 1 = Pivot Selection

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 acceptable performance on most nearly sorted data.

- But it is still $O(n^2)$.

Review
Comparison Sorts III — Better Quicksort: Opt. 2 = Tail-Rec. Elim.

Quicksort uses space for recursion.

- Its additional space usage is proportional to its recursion depth …
- … which is linear. Additional space: $O(n)$. ☹

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: $O(\log n)$. 😐
  - And this additional space need not hold any data items.

Comparison Sorts III continued
Better Quicksort — Optimization 3: Finishing with Insertion Sort

Another 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.
- This is generally faster*, but still $O(n^2)$.
- Note: This is not the same as using Insertion Sort for small lists.

Initial State: | 2 | 12 | 9 | 10 | 3 | 1 | 6 |

↓ Modified Quicksort

Stop the recursion when the sublist to be sorted is small.

Nearly Sorted: | 2 | 3 | 1 | 6 | 12 | 9 | 10 |

↓ Insertion Sort

Sorted: | 1 | 2 | 3 | 6 | 9 | 10 | 12 |

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

Comparison Sorts III
Better Quicksort — Rewrite It

TO DO

- Rewrite our Quicksort to do:
  - Median-of-3 pivot selection.
  - Tail-recursion elimination for reduced recursion depth.
  - Finishing with Insertion Sort.

Done. See **`quicksort2.cpp`***,*
on the web page.

Comparison Sorts III
Better Quicksort — Needed?

We want an algorithm that:

- Is as fast as Quicksort on the average.
- Has reasonable [$O(n \log n)$] 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; ignore it.”

- I suggest to you that this is not a good way to think.
- Sometimes bad worst-case behavior is okay; sometimes it is not.
  - Know what is important in the situation you are addressing.
  - Also, understand that your software can end up being used in other situations.
  - Lastly, remember that on the Web, there are **malicious users**.
- From a former version of the Wikipedia article on Quicksort (retrieved 18 Oct 2006; the statements below were removed on 19 Jan 2007):

> The worst-case behavior of quicksort is not merely a theoretical problem. When quicksort is used in web services, for example, it is possible for an attacker to deliberately exploit the worst case performance and choose data which will cause a slow running time or maximize the chance of running out of stack space.

However, in 1997, a solution was finally published. We discuss this shortly. But first, we analyze Quicksort.

Comparison Sorts III
Better Quicksort — Analysis of Quicksort

Efficiency ☹
- Quicksort is $O(n^2)$.
- Quicksort has a **very** good $O(n \log n)$ average-case time. ☺☺

Requirements on Data ☹
- Non-trivial pivot-selection algorithms (median-of-3 and others) are only efficient for random-access data.

Space Usage 😐
- Quicksort uses space for recursion.
  - Additional space: $O(\log n)$, 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.

Unlike Merge Sort

Stability ☹
- Efficient versions of Quicksort are not stable.

Performance on Nearly Sorted Data 😐
- An unoptimized Quicksort is **slow** on nearly sorted data: $O(n^2)$.
- Quicksort + median-of-3 is $O(n \log n)$ on most nearly sorted data.

Comparison Sorts III
Introsort — "Introspection"

In 1997, David Musser introduced a simple algorithmic-design idea.

- For a number of problems, there are known algorithms with very good average-case performance and very bad worst-case performance.
- Quicksort is the best known of these, but there are others.
- Musser suggests keeping track of an algorithm's performance. If it is not doing well, switch to a different algorithm that has reasonably good worst-case performance.
- Musser calls this technique **introspection**, since an algorithm is examining itself.

The most important application is to sorting.

- Now we can eliminate the bad behavior of Quicksort.

Introspection, and its applications to sorting, are not in the text.

Comparison Sorts III
Introsort — Heap Sort Preview

This is a preview of a sorting algorithm to be covered later.

Later in the semester, we will study the "Priority Queues", generally implemented via a data structure known as a "Heap".

- In a normal Queue, we insert items, and then remove them in the same order (first-in-first-out).
- In a **Priority Queue**, each item has a "priority". Items come out in order of their priority.

Set an item's priority equal to its numerical value, and items come out in sorted order.

- So: Make a Heap and then remove all items from it, in numerical order.
- This sorting algorithm is called **Heap Sort**.

Important Facts about Heap Sort

- Log-linear time.
- In-place.
- Requires random-access data.
- Can be modified to handle problems that are more general than simple comparison sorting. For example, we can allow new items to be added during the sorting process.
- Used as part of a very fast sorting algorithm called "Introsort". Read on …

Comparison Sorts III
Introsort — Description

Recall: Quicksort does a linear time operation (Partition), then calls itself recursively.

- If the recursion depth is around log n, then it uses $O(n \log n)$ steps.
  - Count both sub-lists as recursive calls. Ignore the tail-recursion trick.
- 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 ($k \log n$, for some k), switch to Heap Sort for the current sublist being sorted.
  - Musser suggested a threshold of $2 \log_2 n$.

The resulting algorithm is called **Introsort**.

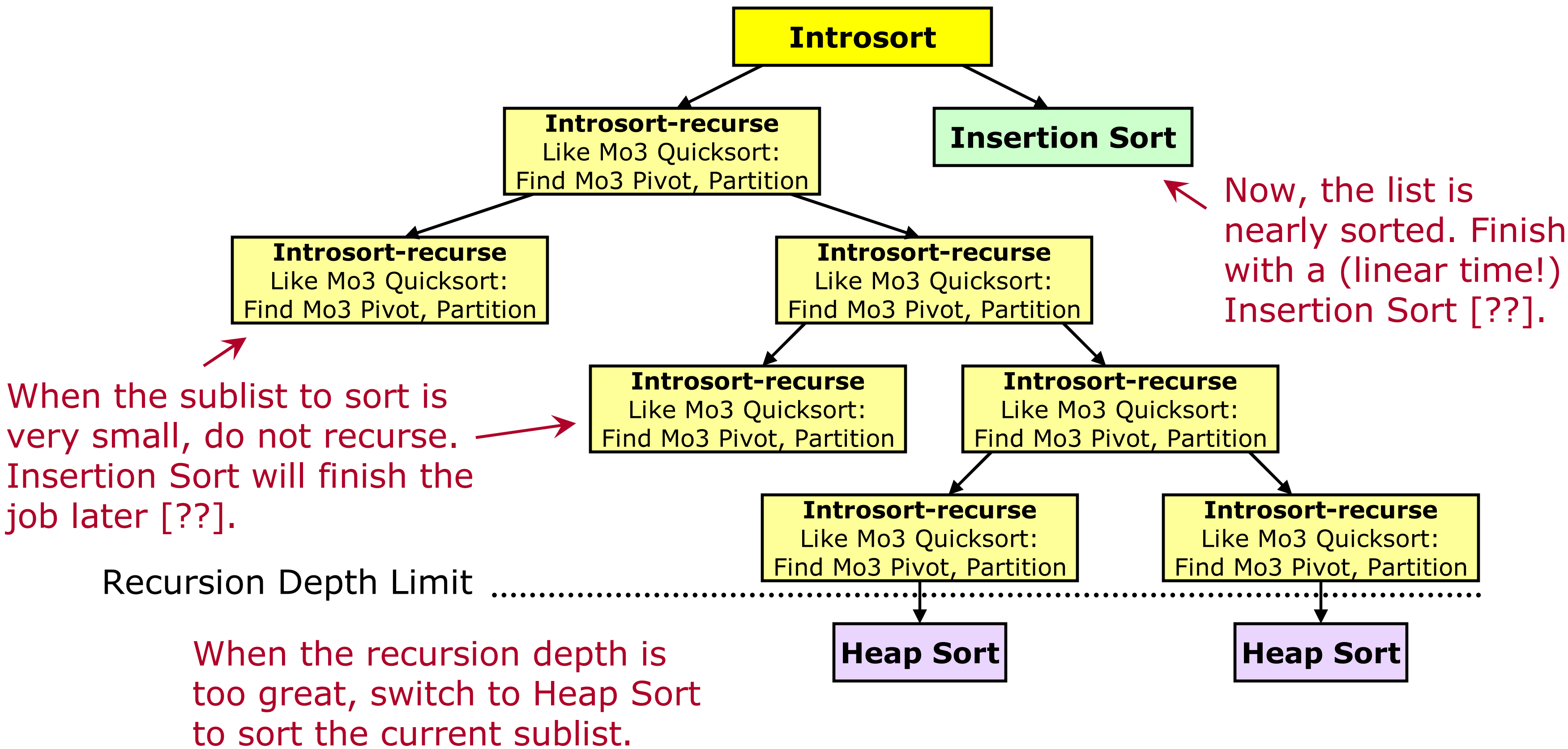
Musser's 1997 paper discusses the speed-ups we have covered:

- Use the median-of-3 rule for pivot selection.
- Stop the recursion prematurely, and finish with Insertion Sort.
  - Maybe. This can adversely affect cache performance.
- However, it is no longer necessary to handle the larger and smaller recursive calls differently, since the recursion-depth limit already makes sure that excessive recursive calls are not made.

Comparison Sorts III
Introsort — Diagram

Here is an illustration of how Introsort works.

- In practice, the recursion will be deeper than this.
- The Insertion-Sort call *might* not be done, due to its effect on cache hits.



Comparison Sorts III
Introsort — Analysis

Efficiency ☺☺

- Introsort is $O(n \log n)$.
- Introsort also has an average-case time of $O(n \log n)$ [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 (or simulated recursion).
  - Additional space: $O(\log n)$ — 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.

Comparison Sorts III
Introsort — Comments

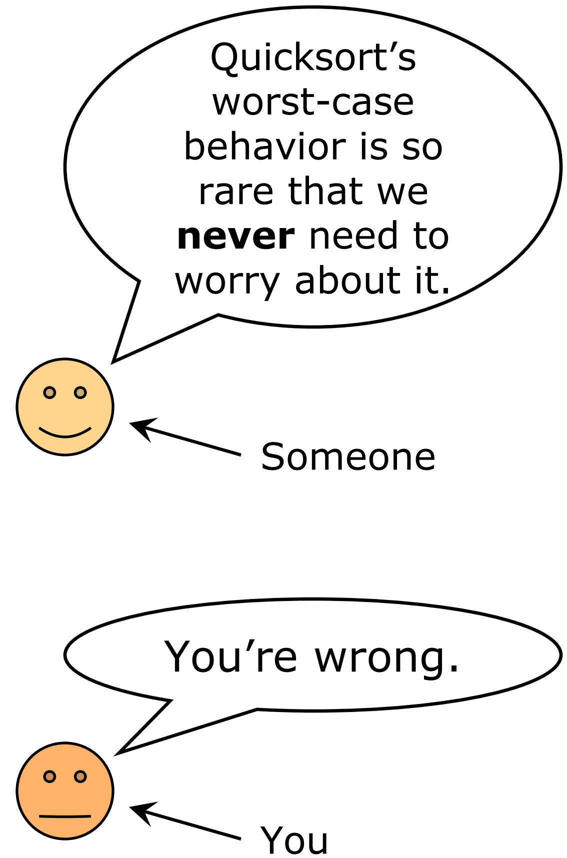
In general, when you want speed (i.e., all the time), Introsort is usually the algorithm to use.

Some Exceptions

- When you need a stable sort.
- When sorting non-random-access data.
- When you expect the data to be nearly sorted.
- When memory is limited.
  - On some embedded systems, maybe?
  - Sorting a list that does not fit into memory.
- When data are accessed over a slow connection.
  - Sorting data accessed over a network?
- When the problem you want to solve is not exactly comparison sorting.
- When your sort may be used in multiple applications.

More about some of these later in the semester.

If someone tells you that Quicksort's worst-case behavior is so rare that we **never** need to worry about it, tell them they're wrong.



Comparison Sorts III
When is it Best?

Algorithm	**When This Algorithm is the *Best* One**
Bubble Sort	Never
Insertion Sort	▪For small lists ▪When you are guaranteed nearly sorted data
Merge Sort	▪When stability is needed ▪For special data types, especially Linked Lists
Heap Sort	In certain special situations: ▪When a list is operated on during the sorting process ▪When you only care about the ordering of part of a list ▪Etc. (more about this later in the semester)
Quicksort	Never
Introsort	Most of the time (if you do not care about stability, data accessed via slow connections, sequential-access data, …)

Now, what if (say) Quicksort is written for you, but nothing else is?
Should you write your own? Maybe. It depends on the situation. **Think!**