The Limits of Sorting
Comparison Sorts III

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, October 4, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Algorithmic Efficiency & Sorting

Topics

- ✓ Analysis of Algorithms
- ✓ Introduction to Sorting
- ✓ Comparison Sorts I
- ✓ Asymptotic Notation
- ✓ Divide and Conquer
- ✓ Comparison Sorts II
- The Limits of Sorting
- Comparison Sorts III
- Non-Comparison Sorts
- Sorting in the C++ STL

Review

Review
Analysis of Algorithms

Cannot read all of input ↑

Probably not scalable ↓

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
$O(c^n)$, for some $c > 1$	Exponential time

Faster

Slower

Useful Rules

- **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 Θ. This works when adding up any *fixed*, *finite* number of terms.

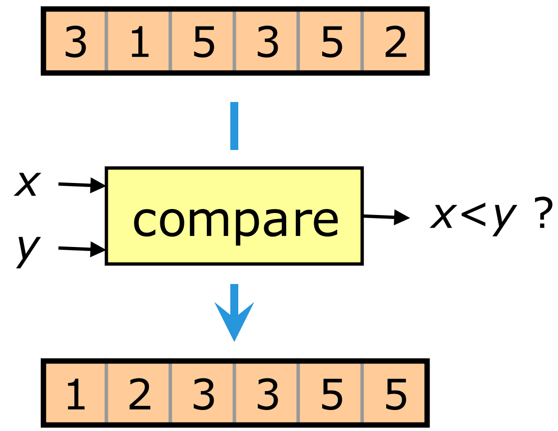
Review
Introduction to Sorting — Basics, Analyzing

Sort: Place a list in order.

Key: The part of the item we sort by.

Comparison sort: Sorting algorithm that only gets information about item by comparing them in pairs.

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



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 reverses the relative order of equivalent items.

1. All items close to proper places,
   OR
2. few items out of order.

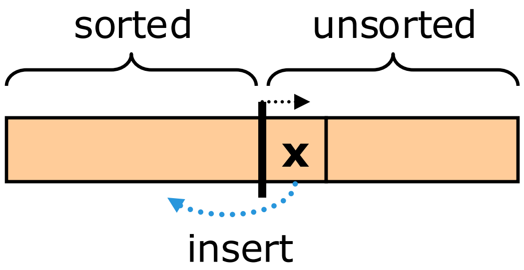
Review
Introduction to Sorting — Overview of Algorithms

Sorting Algorithms Covered

- Quadratic-Time [$O(n^2)$] Comparison Sorts
  - ✓ Bubble Sort
  - ✓ Insertion Sort
  - 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

Comparison Sorts I — Insertion Sort

Insertion Sort repeatedly does this:



Analysis

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

See `insertion_sort.cpp`*.*

Notes

- Too slow for most use cases.
- Fast in special cases: *nearly sorted data* and *small lists*.
- Thus, often used as part of other algorithms.

Review
Asymptotic Notation

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

Θ is very useful!

Ω not as much.

	1	n	$n \log n$	n^2	$5n^2$	$n^2 \log n$	n^3	n^4
$O(n^2)$	**YES**	**YES**	**YES**	**YES**	**YES**	no	no	no
$\Omega(n^2)$	no	no	no	**YES**	**YES**	**YES**	**YES**	**YES**
$\Theta(n^2)$	no	no	no	**YES**	**YES**	no	no	no

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.

Review
Divide and Conquer [1/2]

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 $T(n) = a\,T(n/b) + f(n)$; $a \geq 1$, $b > 1$, $f(n)$ 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 "Θ" above with "O".

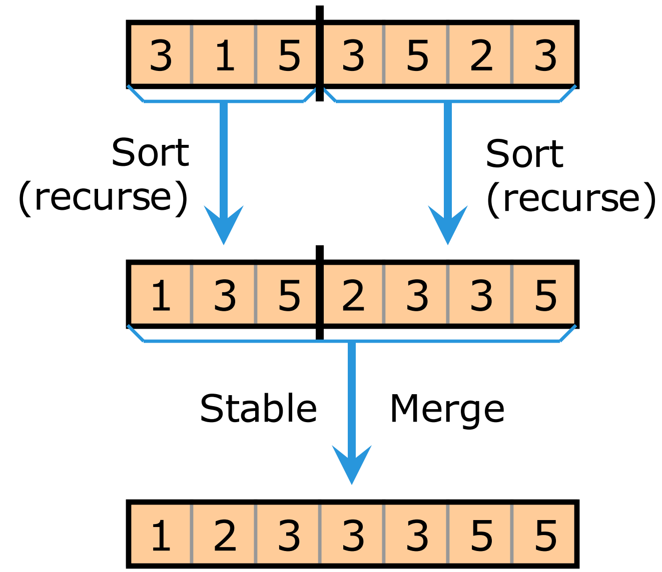
Try It!

Algorithm *A* is given a list as input. It uses a Divide and Conquer strategy. It splits its input in half (or nearly so), and handles each part with a recursive call. It also does other work requiring constant time.

Use the Master Theorem to determine the order of Algorithm *A*.

See In-Class Worksheet 1: The Master Theorem.

Merge Sort: recursively sort top & bottom halves of list, **merge**.



Analysis

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

See `merge_sort.cpp`.

Notes

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

The Limits of Sorting

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

- Slow: $\Theta(n^2)$—e.g., Bubble Sort, Insertion Sort.
- Fast: $\Theta(n \log n)$—e.g., Merge Sort.

Can we sort even faster than that?

No, we cannot—not with a general-purpose comparison sort.

Fact. A general-purpose comparison sort that lies in any time-efficiency category faster than $\Theta(n \log n)$ 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 $\Omega(n \log n)$.

Here is what Ω is good for: statements that say, "You cannot do better than this."

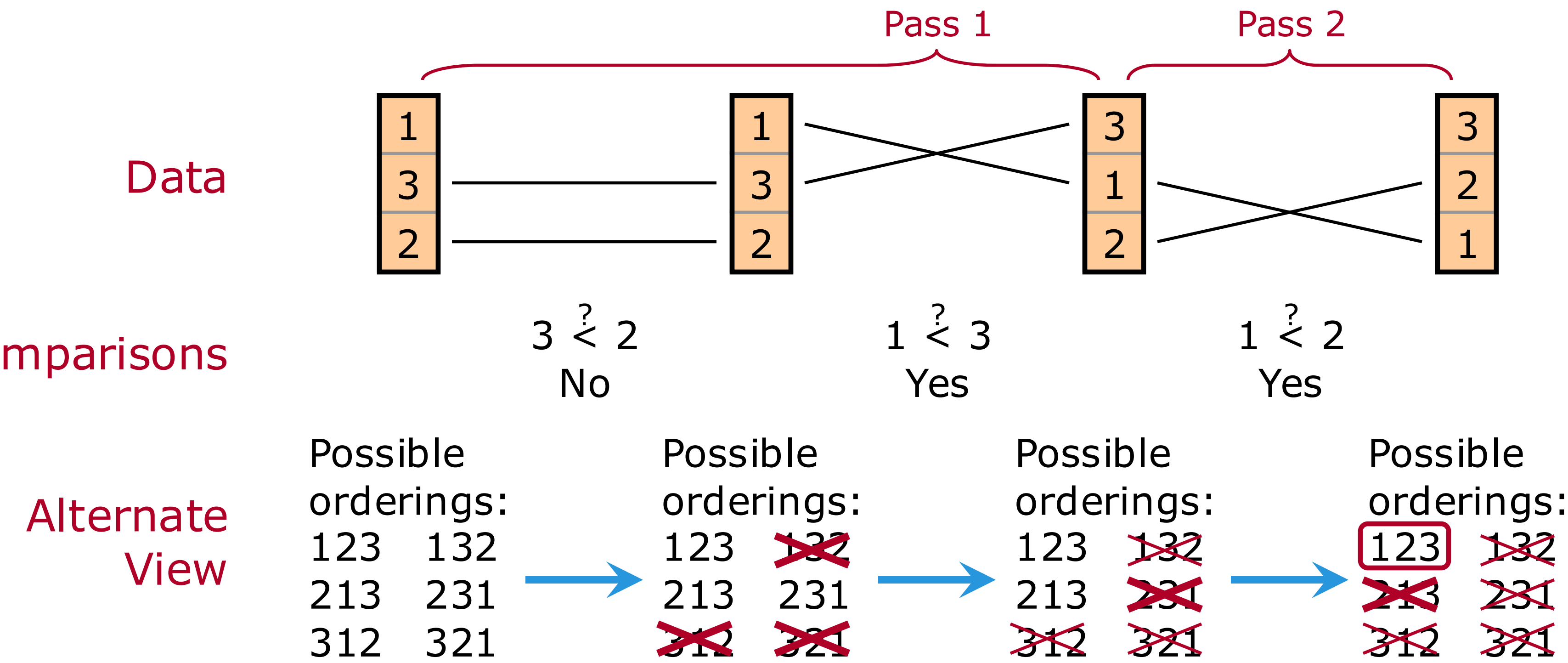
The Limits of Sorting
Proof — Background

Sorting determines the order of a list. Many orderings are possible.

A comparison finds the relative order of two items. Say x < y; now we know that any ordering with y before x is not the answer.

We cannot stop until only one possible ordering is left.

Example. Bubble Sort the list 2 3 1.



The Limits of Sorting
Proof — Outline [1/2]

We prove that the worst-case number of comparisons performed by a general-purpose comparison sort must be $\Omega(n \log n)$.

As on the previous slide:

- 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.

How many comparisons are required?

- With each comparison, we cannot guarantee that more than half of the orderings will be thrown out. (Remember: worst case.)
- How many times must we cut *n*! in half, to get 1?
- Answer: $\log_2(n!)$.

Continued on next slide …

The Limits of Sorting
Proof — Outline [2/2]

We know that the worst-case number of comparisons performed by a general-purpose comparison sort cannot be less than $\log_2(n!)$.

Now we use **Stirling's Approximation**: $n! \approx \frac{n^n}{e^n}\sqrt{2\pi n}$.

See `stirling.py`.

Take $\log_2$ of both sides:

$$\log_2(n!) \approx n\log_2 n - n\log_2 e + \frac{1}{2}\log_2(2\pi) + \frac{1}{2}\log_2 n,$$

which is $\Theta(n \log n)$.

So $\log_2(n!)$ is $\Theta(n \log n)$.

The worst case number of comparisons done by a general-purpose comparison sort must be *at least* that big. Thus: $\Omega(n \log n)$.

The Limits of Sorting
Another View

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

Another way to say this involves a different model of computation:

- Legal operations:
  - Any operation that does not depend on the values of input data items.
  - A comparison of two data items.
- Basic operation: Comparison of two data items.
- Size: Number of items in given list.

In this model of computation, comparison sorting is the only kind of sorting that can be done.

A restatement of what was proven:

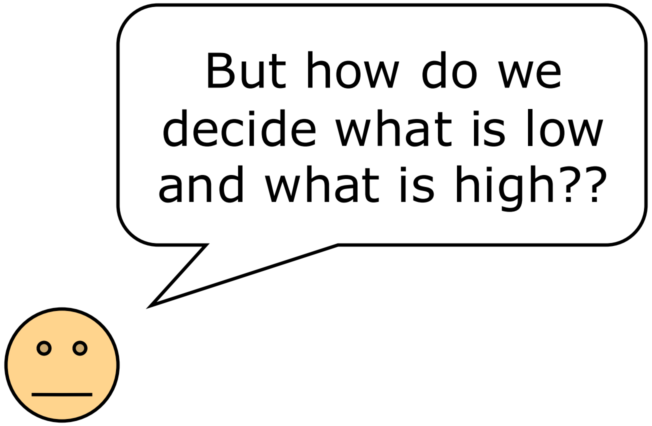
In the above model of computation, every general-purpose comparison sort is $\Omega(n \log n)$ time.

Comparison Sorts III

Comparison Sorts III
Quicksort — Introduction [1/3]

Idea

- Instead of simply splitting a list in half in the middle, try to be intelligent about it.
- Split the list into the low-valued items and the high-valued items; then recursively sort each bunch.
- Now no Merge is necessary.

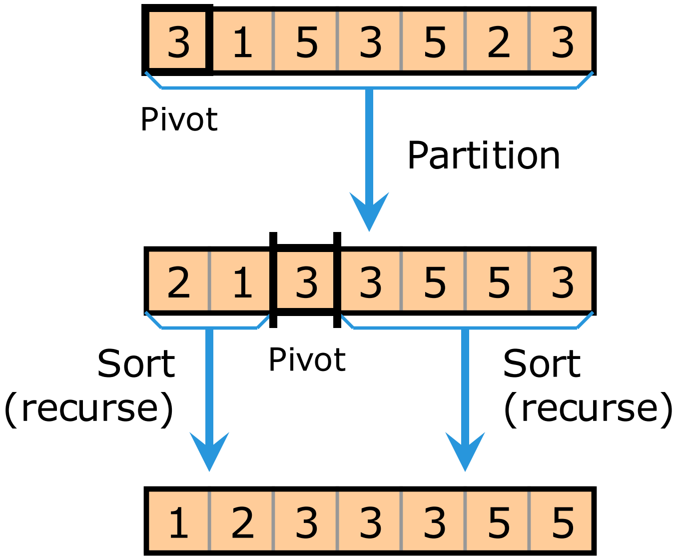


Quicksort — Introduction [2/3]

Let's be more precise about this algorithmic idea.

We use another Divide-and-Conquer technique:

- Pick an item in the list.
  - This first item will do—*for now*.
  - The chosen item is called the **pivot**.
- Rearrange the list so that the items before the pivot are all less than or equivalent to the pivot, and the items after the pivot are all greater than or equivalent to the pivot.
  - This operation is called **Partition**.
    It can be done in linear time.
- Recursively sort the sub-lists: items before pivot, items after pivot.

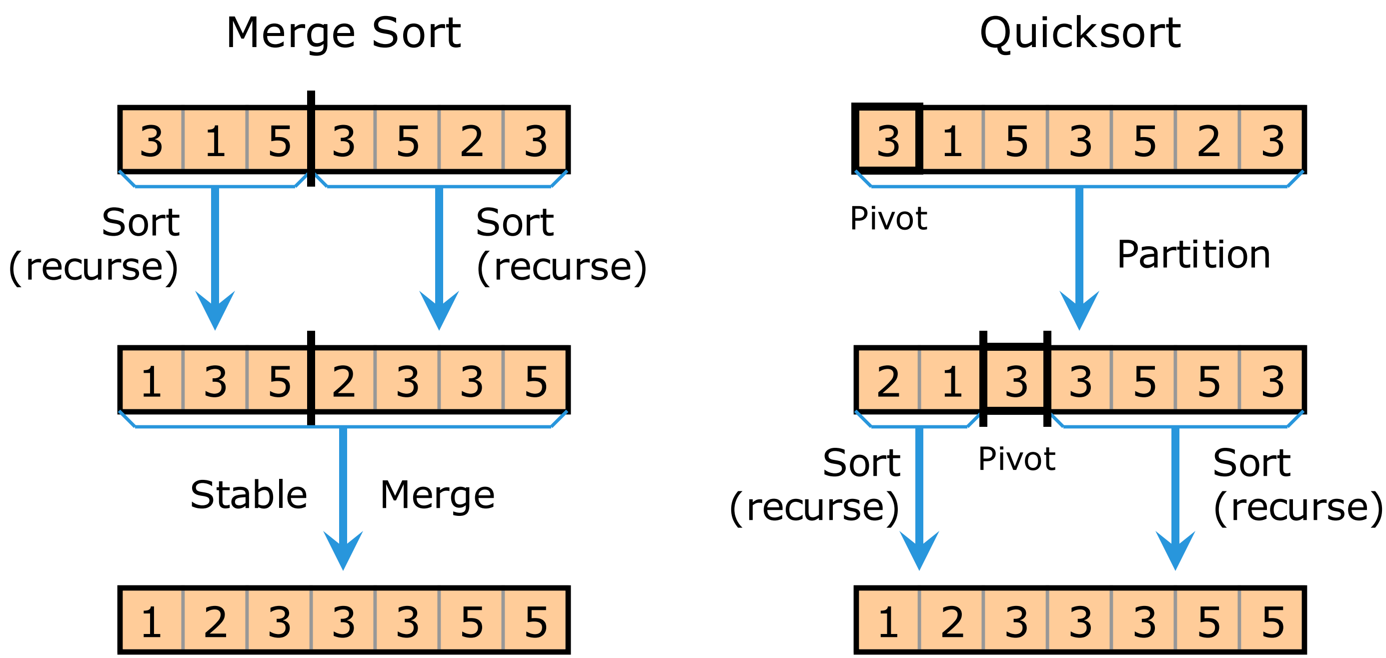


This algorithm is called **Quicksort** [C.A.R. ("Tony") Hoare, 1961].

Comparison Sorts III
Quicksort — Introduction [3/3]

Compare Merge Sort & Quicksort.

- Both use Divide-and-Conquer.
- Both have an auxiliary operation (Stable Merge, Partition) that does all modification of the data set and that takes linear time.
- Merge Sort recurses first. Quicksort recurses last.



Comparison Sorts III
Quicksort — Partition [1/2]

How do we do the Partition operation?

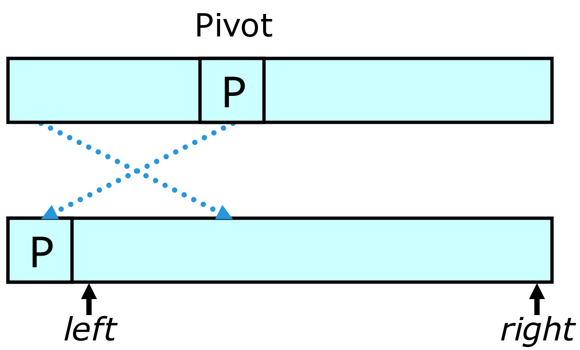
There are multiple practical partition algorithms that are used with Quicksort. Generally, these are:

- In-place.
- Linear-time.
- Not stable.

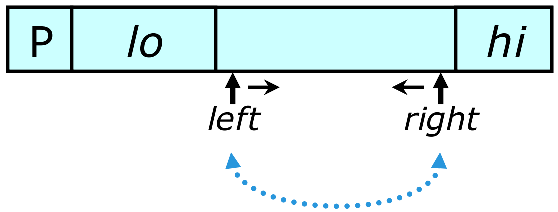
We look at the details of a common method of doing the Partition: **Hoare's Partition Algorithm**.

Hoare’s Partition Algorithm

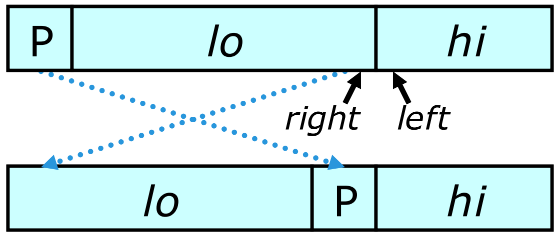
- First, get the pivot out of the way: swap it with the first list item.
- Set iterator *left* to point to the first item past the pivot. Set iterator *right* to point to the last list item.



- Move iterator *left* up, leaving only low items below it. Move iterator *right* down, leaving only high items above it.
- If both iterators get stuck—*left* points to a high item and *right* points to a low item—then swap the items and continue.



- Eventually *left* & *right* cross each other.
- Finish by swapping the pivot with the last low item.



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`*.*

Comparison Sorts III
TO BE CONTINUED …

Comparison Sorts III will be continued next time.