# Comparison Sorts III continued

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, October 9, 2023

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

Facts
1. Algorithm that reads all of its inputs must be Ω (n)
2. A general purpose comparison sort must be Ω (n log n)

Alg X is Ω(n)

↓

Its worst case running time is at least kn
O Ω Θ

# 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

## 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 $g(n) \le k \times f(n)$ …
- $\Omega(f(n))$ if $g(n) \ge k \times f(n)$ …
- $\Theta(f(n))$ if both are true—possibly with different values of $k$.

$\Theta$ is very useful!

$\Omega$ 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 "$\Theta$" above with "$O$".

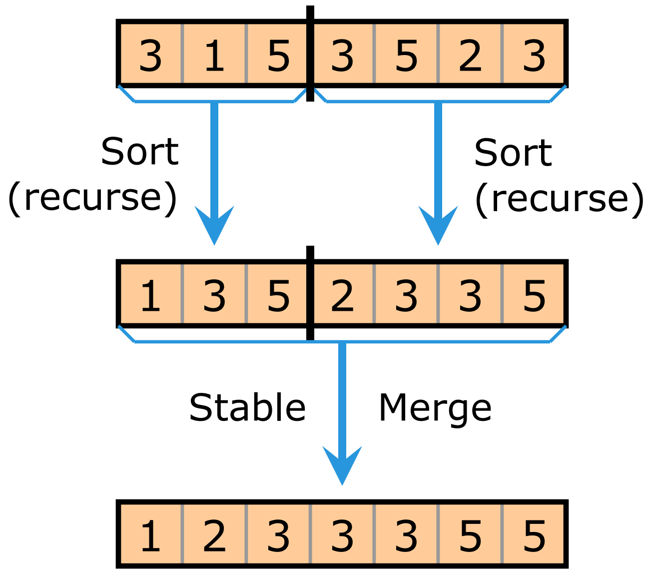
**Try It!**

Algorithm *B* is given a list as input. It uses a Decrease and Conquer strategy. It splits its input in half (or nearly so), and makes a recursive call on one of the parts. It also does other work requiring linear time.

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

*See In-Class Worksheet 2: 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

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

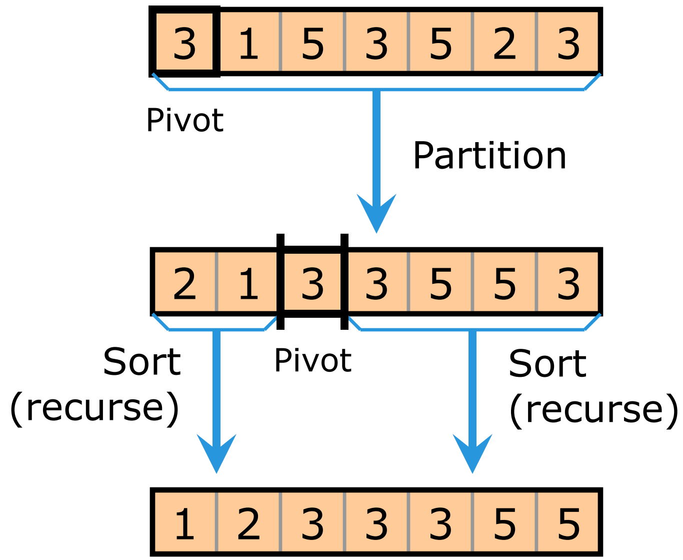
Reasoning:

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

# Comparison Sorts III — Quicksort [1/2]

**Quicksort**: choose **pivot**, **partition**, recursively sort sublists.

For the moment, we choose the first item in the list as our pivot.

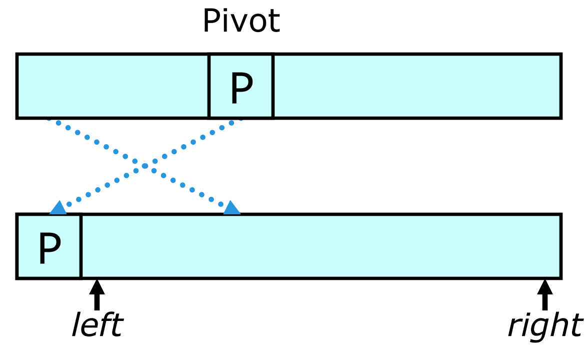



*See* `quicksort1.cpp`.

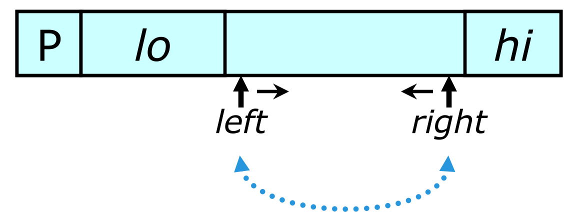
# Review
# Comparison Sorts III — Quicksort [2/2]

## Hoare's Partition Algorithm

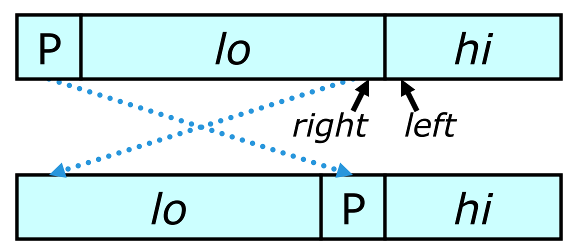
- First, get the pivot out of the way: swap it with the first list item.
- Set iterator *left* to point to the first item past the pivot. Set iterator *right* to point to the last list item.




- 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

continued

# Better Quicksort — Problem

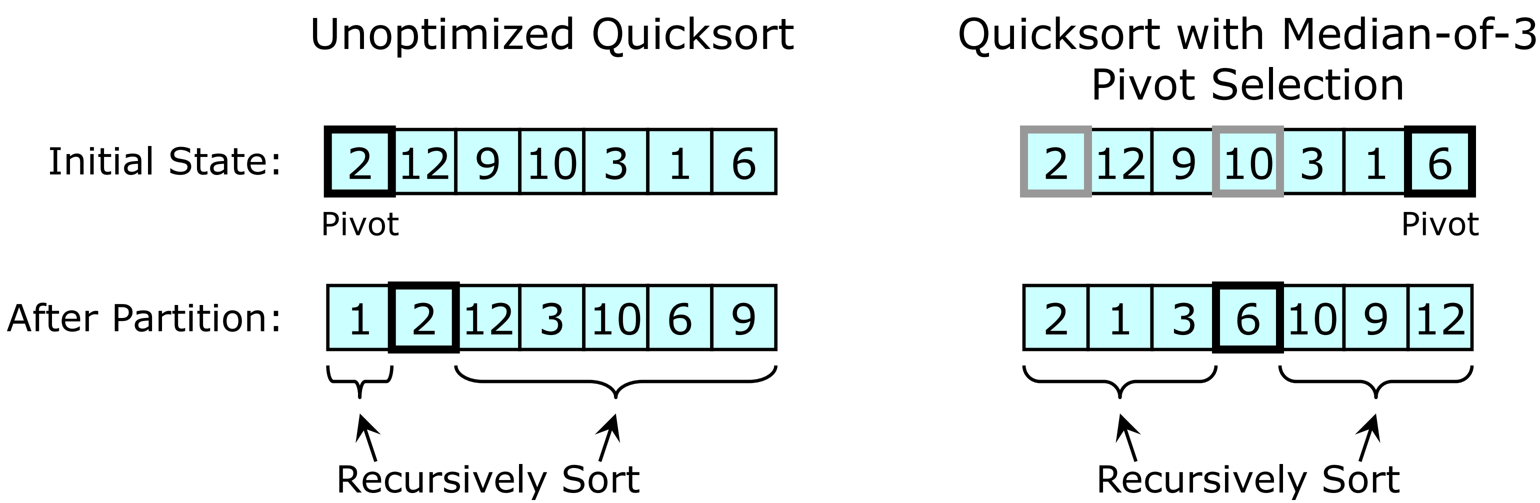
Quicksort has a serious problem.

- Try applying the Master Theorem. It does not work, because *what???*

# Better Quicksort — Optimization 1: Improved Pivot Selection [1/2]

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: still $\Theta(n^2)$.

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

# Comparison Sorts III
# Better Quicksort — Optimization 1: Improved Pivot Selection [2/2]

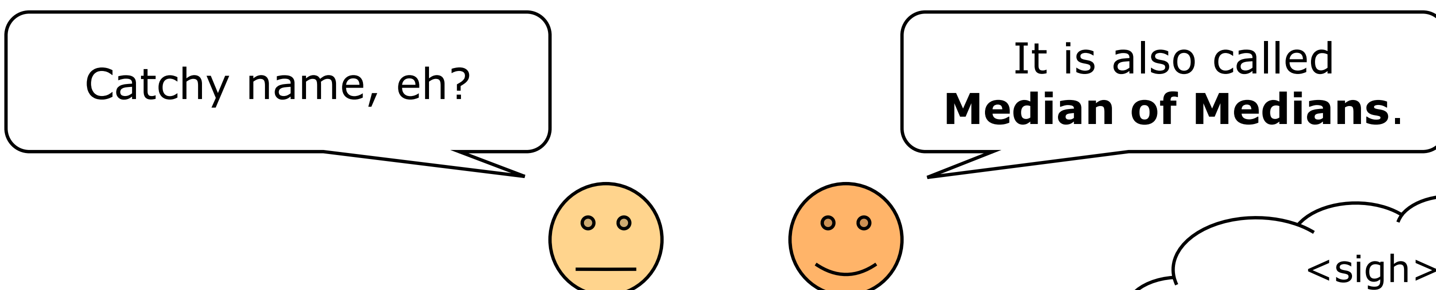
Ideally, our pivot is the *median* of the list.

**Median**: value that goes in the middle, when the list is sorted.

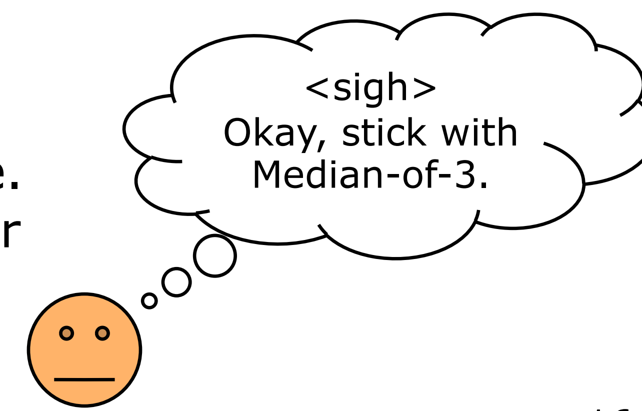
- If it were, then Partition would create lists of (nearly) equal size, and we could apply the Master Theorem, which would tell us:
- If we do $O(n)$ extra work at each step, then we get an $O(n \log n)$ algorithm (same computation as for Merge Sort).

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

- Yes! Use **BFPRT** (the **B**lum-**F**loyd-**P**ratt-**R**ivest-**T**arjan Algorithm).




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



# Better Quicksort — Optimization 2: Tail-Recursion Elimination

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?)

# Better Quicksort — Optimization 3: Finishing with Insertion Sort

A *possible* speed-up: finish with Insertion Sort

- Quicksort, but without quite going to the bottom of the 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 $\Theta(n^2)$.

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.

# 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.

*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.

These are *general* principles. They apply to many issues, not just those involving Quicksort.

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

*???*

Requirements on Data

*???*

Space Usage

*???*

Stability

*???*

Performance on Nearly Sorted Data

*???*

# Comparison Sorts III
# Introsort — Introspection

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. It allows us to eliminate the awful worst-case behavior of Quicksort.

# Comparison Sorts III
## Introsort — Heap Sort Preview

Here is a preview of a sort we will cover later in the semester.

We will study a data structure called a *Binary Heap*, which allows for fast find and removal of the item with the greatest key.

This leads to a comparison sort called **Heap Sort**. Procedure:

- Create a Binary Heap containing the dataset to be sorted.
- Repeatedly remove the item with the greatest key. Store these items in a list in reverse order: greatest at the end, etc.
- When complete, the list is a sorted version of the original dataset.

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

- Heap Sort is log-linear time.
- Heap Sort is in-place.
- Heap Sort requires random-access data.

And Heap Sort forms part of a fast Quicksort variant called *Introsort*.

# Comparison Sorts III
# Introsort — Description

Quicksort's problem is due to its recursion depth. 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** [**intro**spective **sort**].

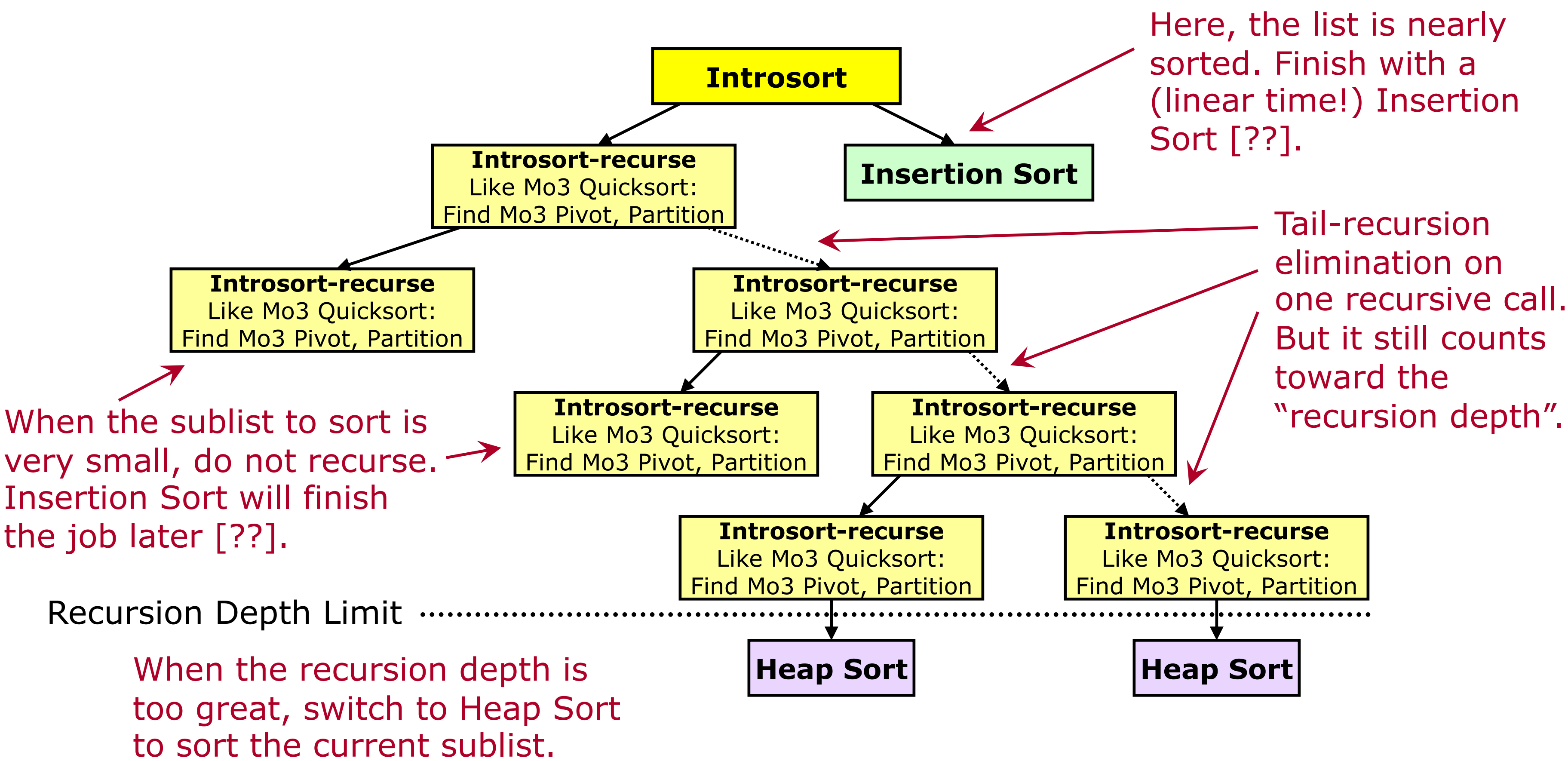
Musser's 1997 paper recommends the optimizations we covered:

- Median-of-3 pivot selection.
- Tail-recursion elimination on one 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.



# Comparison Sorts III
# Introsort — Analysis

Efficiency

*???*

Requirements on Data

*???*

Space Usage

*???*

Stability

*???*

Performance on Nearly Sorted Data

*???*

## Comparison Sorts III
## Introsort — Notes

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.