Comparison Sorts III

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, March 4, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

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

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

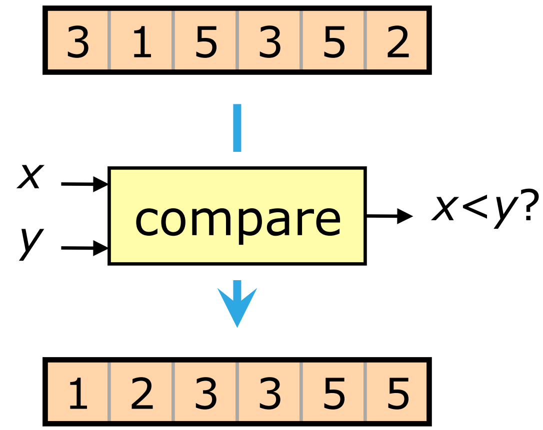
Introduction to Sorting — Basics, Analyzing

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

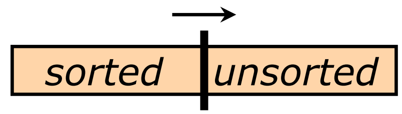
There is no *known* sorting algorithm that has all the properties we would like one to have.

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

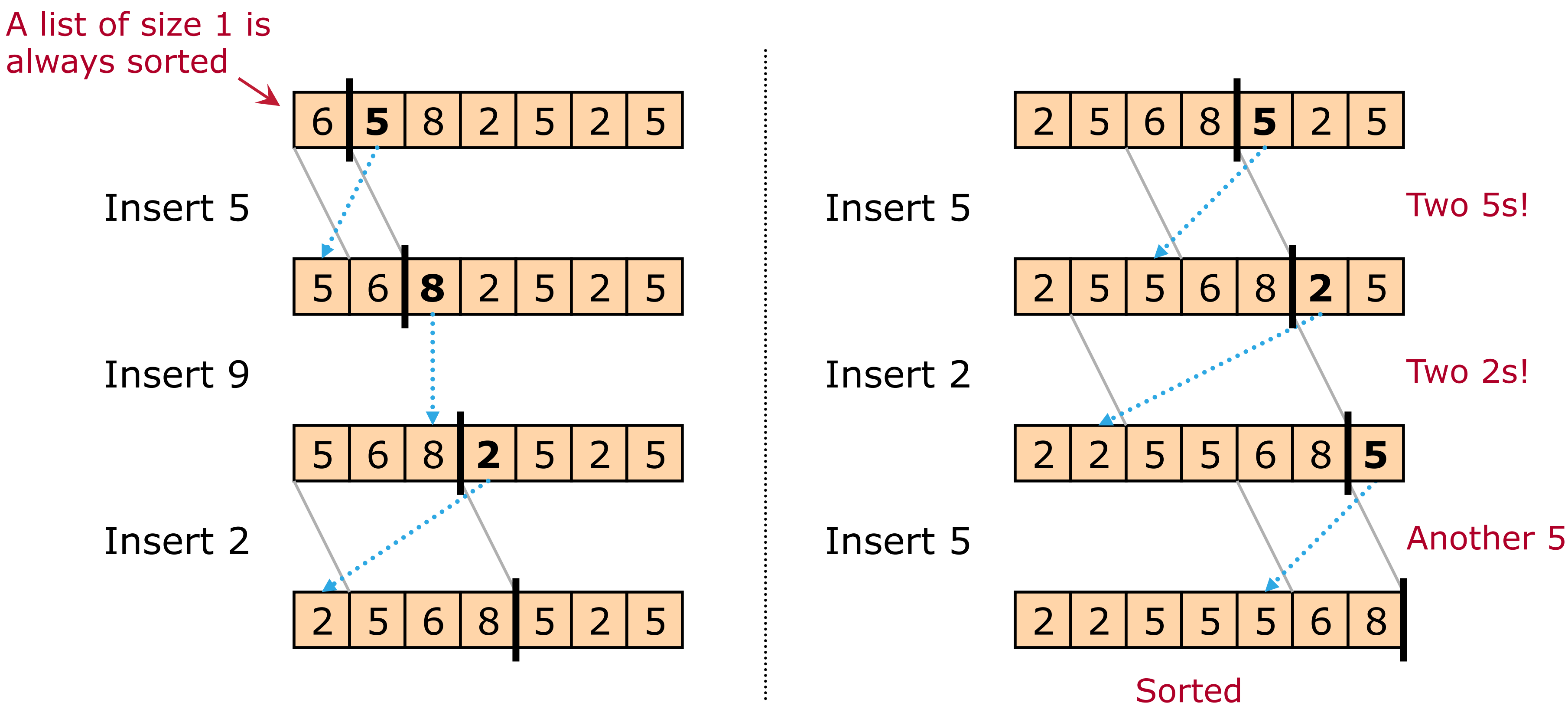
- Quadratic-Time [$O(n^2)$] Algorithms
  - ✓ Bubble Sort
  - ✓ Insertion Sort
  - Quicksort
  - Treesort (later in semester)
- Log-Linear-Time [$O(n \log n)$] Algorithms
  - ✓ Merge Sort
  - Heap Sort (mostly later in semester)
  - Introsort (not in the text)
- Special Purpose — Not Comparison Sorts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

Review
Comparison Sorts I — Insertion Sort: Illustration

Items to left of bold bar are sorted.



Bold item = item to be inserted into sorted section.



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.

Review
Comparison Sorts I — Insertion Sort: Comments

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.

Review
The Limits of Sorting [1/2]

Sorting is determining the ordering of a list. Many orderings are possible.

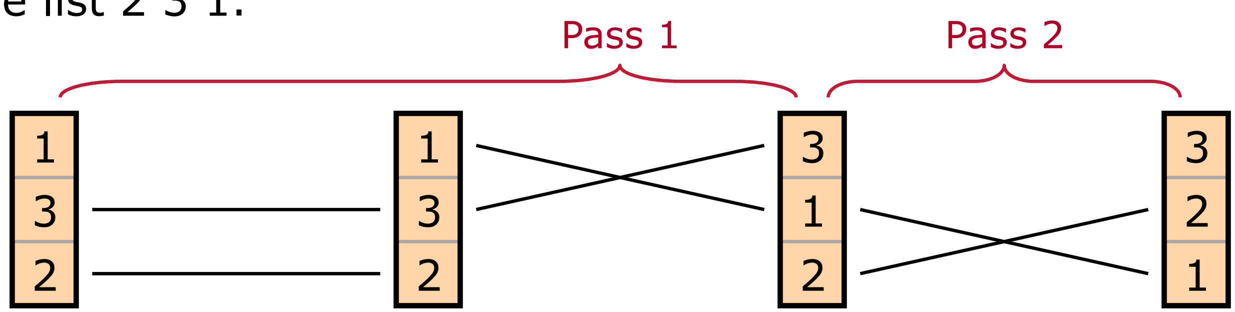
Each time we do a comparison, we find the relative order of two items.
   Say x < y; we can throw out all orderings in which y comes before x.

We cannot stop until only one possible ordering is left.

Example

- Bubble Sort the list 2 3 1.

Bubble Sort



Comparisons

$3 \overset{?}{<} 2$ No | $1 \overset{?}{<} 3$ Yes | $1 \overset{?}{<} 2$ Yes

Alternate View

Possible orderings:	Possible orderings:	Possible orderings:	Possible orderings:
123 132	123 ~~132~~	123 ~~132~~	$\boxed{123}$ ~~132~~
213 231	213 231	213 ~~231~~	~~213~~ ~~231~~
312 321	~~312~~ ~~321~~	~~312~~ ~~321~~	~~312~~ ~~321~~

Review
The Limits of Sorting [2/2]

I have said that good sorting algorithms are $O(n \log n)$.

We showed that a general-purpose comparison sort cannot do better than this.

In particular, a general purpose comparison sort must do $\Omega(n \log n)$ comparisons, in the worst case.

- The proof is based on the idea on the previous slide.
- A list of n items has $n!$ possible orderings.
- In order to reduce this down to just one ordering, at least $\log_2(n!)$ comparisons, are required, in the worst case.
- And $\log_2(n!)$ is $\Theta(n \log n)$.

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: Introduction

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

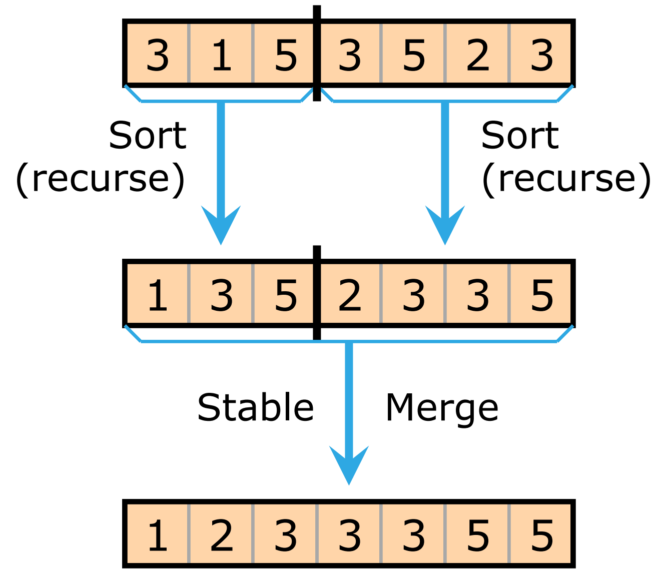
The **Stable Merge** operation is linear-time, resulting in a log-linear-time sort.

- By the Master Theorem.

Stable Merge can be done in-place for a Linked List.

In general (in particular, for an array) efficient Stable Merge generally uses a buffer.

- Linear additional space.



Review
Comparison Sorts II — Merge Sort: Analysis

See **`sorting.cpp`**, *on the web page.*

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.
- Operations needed. General: copy. Linked List: NONE (compare).

Space Usage 😐/☺/☹
- Recursive Merge Sort uses stack space: recursion depth ≈ $\log_2 n$.
  - 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.

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.

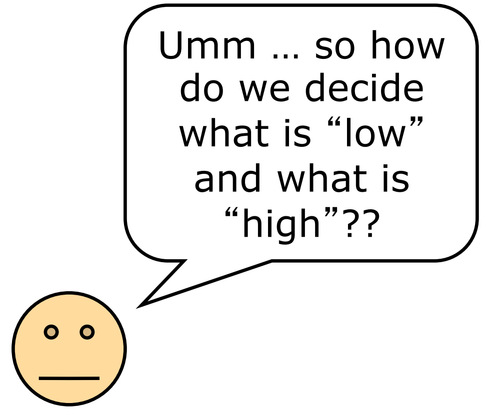
- Thus, while the overall structure is the same, different versions of Merge Sort can differ greatly in lower-level details.
- Merge Sort is *almost* two different algorithms.

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

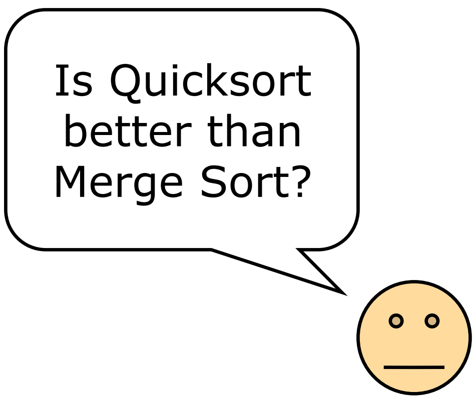
Quicksort — Introduction [1/4]

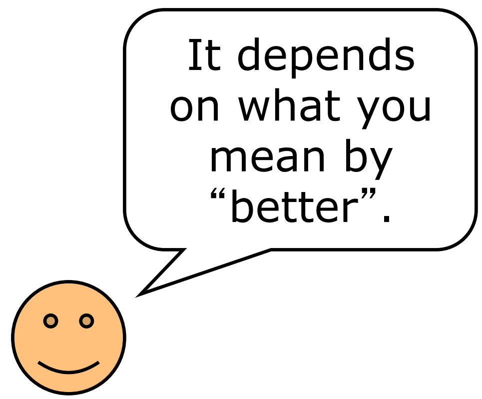
Idea: Instead of simply splitting a list in half in the middle, let’s try to be intelligent about it.

- Split the list into the low-valued items and the high-valued items, and then recursively sort each bunch.
- Now no Merge is necessary.
  - So maybe we can be faster than Merge Sort?



This idea leads to an algorithm called **Quicksort**.



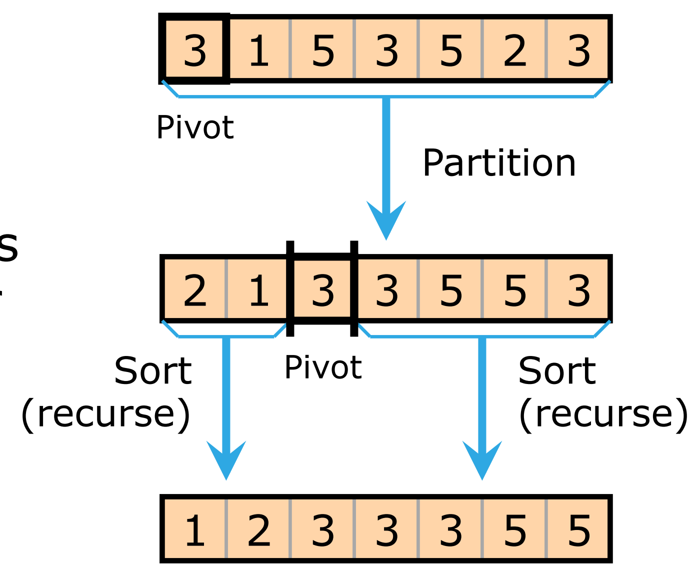


Comparison Sorts III
Quicksort — Introduction [2/4]

Let's be more precise about this algorithmic idea.

We use another variation of the 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.



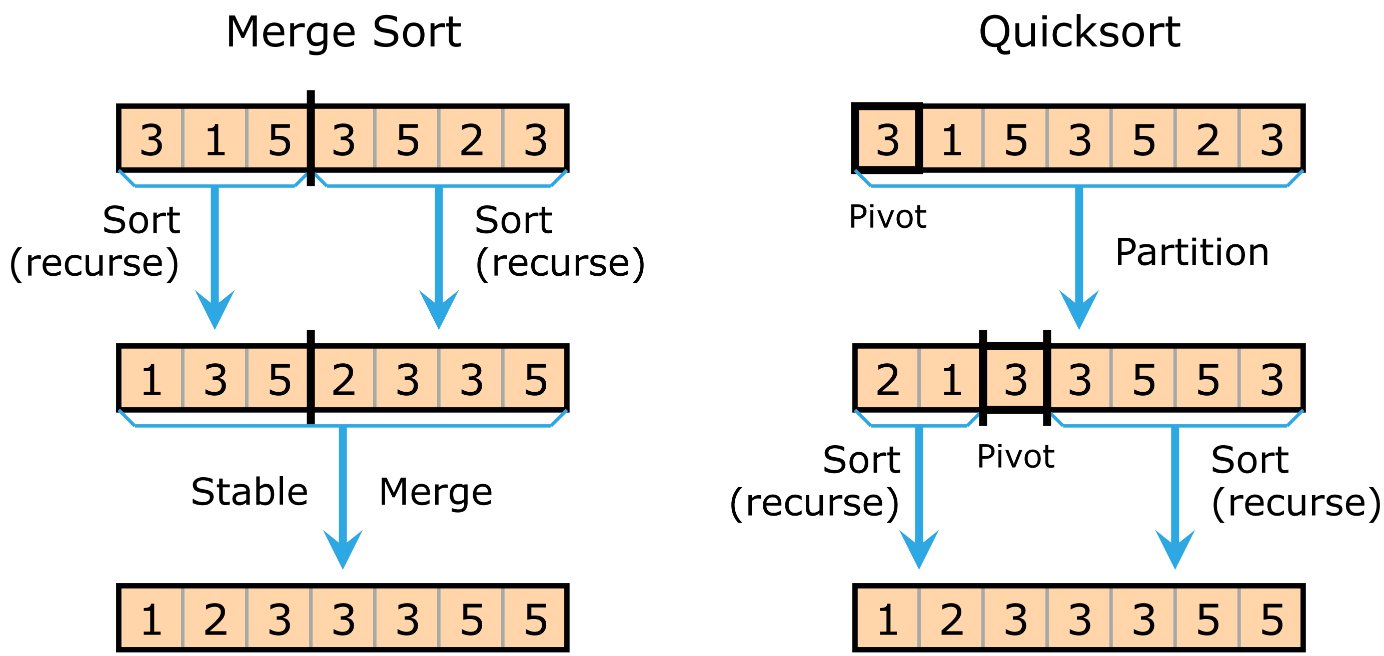
This algorithm is called **Quicksort**.

- C. A. R. Hoare, 1961.

Comparison Sorts III
Quicksort — Introduction [3/4]

Compare Merge Sort & Quicksort.

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



Comparison Sorts III
Quicksort — Introduction [4/4]

Just for fun, here is Quicksort in Python, using **list comprehensions**:

```
def quicksort(xs):
    """Returns items in xs sorted by <, assuming no duplicates.

    Uses Quicksort via list comprehensions."""

    if len(xs) == 0: return []
    else:
        pivot = xs[0]
        return quicksort([x for x in xs if x < pivot]) \
             + [pivot] \
             + quicksort([x for x in xs if x > pivot])
```

Note: This is a poor implementation. It does not handle equivalent items, and it is inefficient in both time and space. But it is instructive.

Comparison Sorts III
Quicksort — Partition [1/2]

Goal: Create a sorting algorithm that is faster than Merge Sort.

Now we look at how to do the Partition.

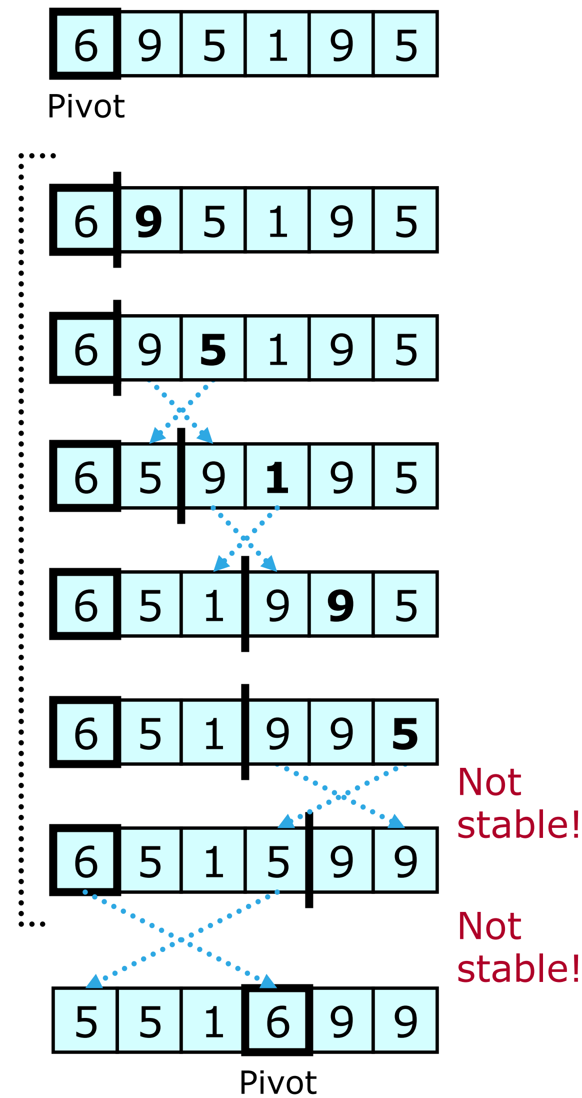
- Stable?
  - We get a stable sort if pivot choice + Partition is stable.
  - However, this makes our algorithm slower than Merge Sort.
  - The fastest known partition algorithms are not stable, and so we implement Quicksort in a non-stable manner.
- In-Place?
  - If we give up stability, then we can do a fast, in-place Partition.
  - If Partition is in-place, then the only significant memory used by the sorting function is that required for recursion. (An in-place Partition does *not* give us an in-place sort!)

Comparison Sorts III
Quicksort — Partition [2/2]

An In-Place Partition Algorithm

- Make sure the pivot lies in the first position (swap if not).
- Create list of items < the pivot (the "left list") at the start of the list as a whole.
  - Start: the left list holds only the pivot.
  - Iterate through rest of the list.
  - If an item is less than the pivot, swap it with the item just past the end of the left list, and move the left-list end mark one to the right.
- Lastly, swap the pivot with the last item in the left list.
- Note the pivot's new position.

Note: This is one common in-place partition algorithm. At least one other such algorithm is also common.



Comparison Sorts III
Quicksort — Write It

TO DO

- Examine code for Quicksort, with the in-place Partition being a separate function.

Comparison Sorts III
Better Quicksort — Problem

Quicksort has a problem.

- Try applying the Master Theorem. It doesn't 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 $O(n^2)$. ☹
- 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.

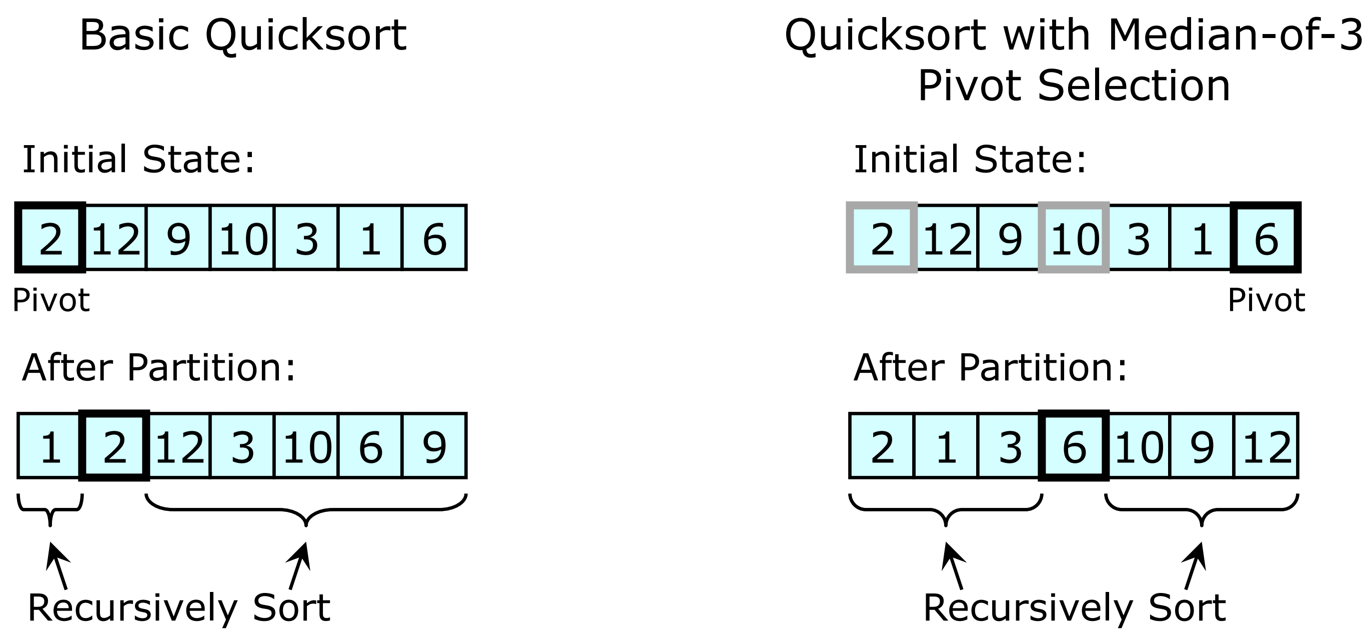
Quicksort is *usually* very fast; thus, people want to use it.

- So we try to figure out how to make it better.
- In the decades following Quicksort's introduction in 1961, many people published suggested improvements. We will look at three of the best ones …

Comparison Sorts III
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 acceptable performance on most nearly sorted data.

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

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

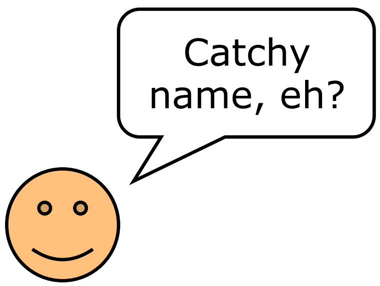
Ideally, our pivot is the **median** of the list.

The item 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 at most $O(n)$ extra work at each step, then we get an $O(n \log n)$ algorithm.

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

- Yes we can! Use the Blum-Floyd-Pratt-Rivest-Tarjan Algorithm.



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

Comparison Sorts III
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. Additional space: $O(n)$. ☹

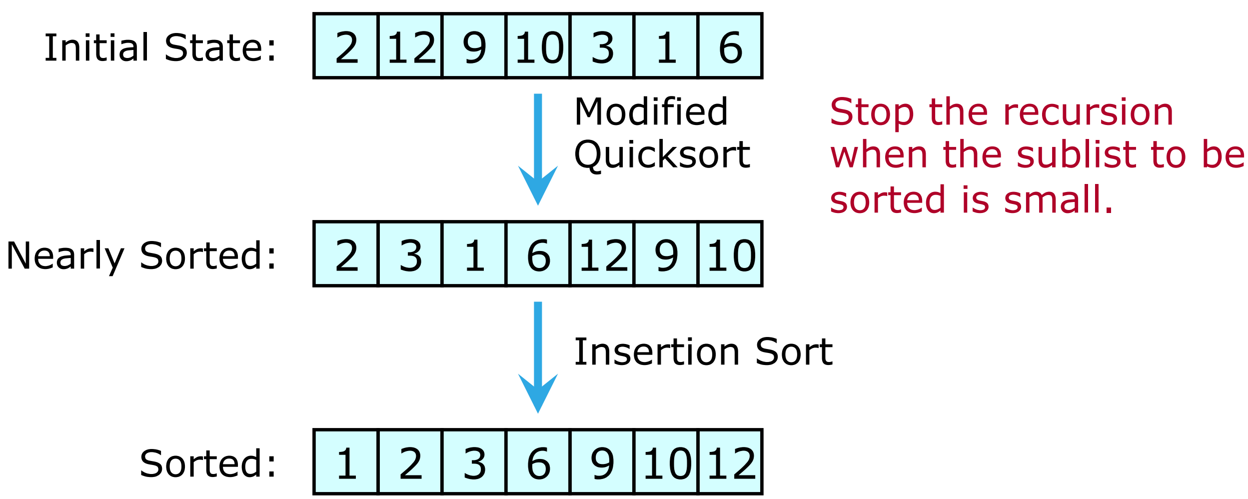
We can significantly improve this:

- 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.

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.



*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 (or just examine rewritten code) to do:
  - Median-of-3 pivot selection.
  - Tail-recursion elimination for reduced recursion depth.
  - Finishing with Insertion Sort.

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.