

Asymptotic Notation

Divide and Conquer

Comparison Sorts II

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, September 30, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2005–2020 Glenn G. Chappell

Some material contributed by Chris Hartman

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
 - Comparison Sorts III
 - Non-Comparison Sorts
 - Sorting in the C++ STL

Review

Efficiency

- General meaning. Using few **resources**: time, space, etc.
- Specific meaning. Fast (not using much *time*).
- For other kinds of efficiency, we qualify: **space efficiency**, etc.
- Unless we say otherwise, we are talking about the **worst case**: maximum resource usage—usually for a given input size.

Our **model of computation** includes:

- Legal **operations**: what we are allowed to do.
- **Basic operations**: the operations we count.
- How we measure the **size** of the input.

Scalable: works well with large problems. (Or, it **scales well**.)

Review

Analysis of Algorithms [2/2]

	Using Big-O	In Words
	$O(1)$	Constant time
Cannot read all of input ↑	$O(\log n)$	Logarithmic time
	$O(n)$	Linear time
	$O(n \log n)$	Log-linear time
..... Probably not scalable ↓	$O(n^2)$	Quadratic time
	$O(c^n)$, for some $c > 1$	Exponential time

Faster ↑

Slower ↓

Useful Rules

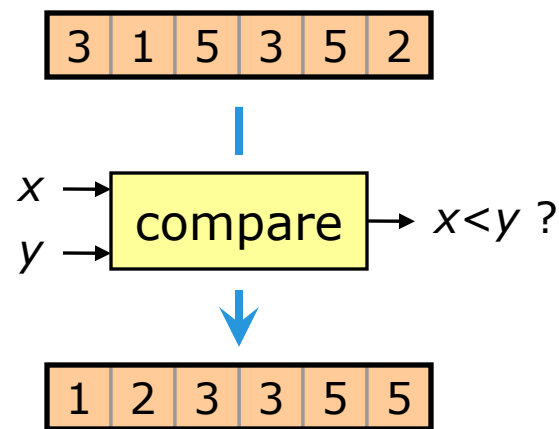
- When determining big- O , we can collapse any constant number of steps into a single step.
- Rule of Thumb.** For nested “real” loops (we do not count, for example, a loop executed only a fixed number of times) order is $O(n^t)$, where t is the number of nested loops.

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.

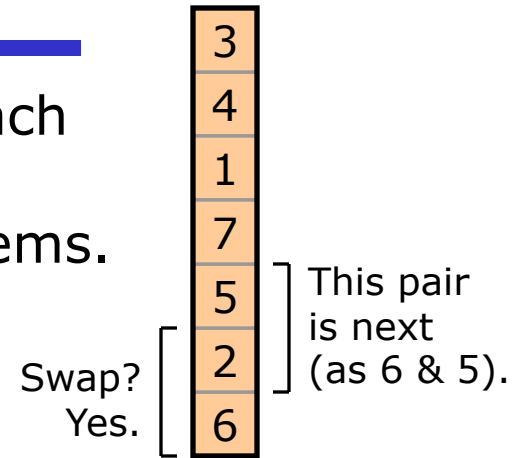
Sorting Algorithms Covered

- 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

Bubble Sort proceeds in a number of **passes**, each of which “bubbles” a large item to the top by doing compare/swap on pairs of consecutive items.

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 type 1 (all items close to their final spots). 😊
 - $O(n^2)$ for type 2 (few items out of order). ☹️

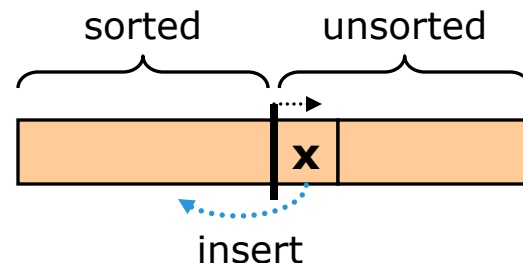


Note

- Too slow. Do not use in practice.

[See bubble_sort.cpp.](#)

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. 😊

Notes

[See insertion_sort.cpp.](#)

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

Review

Comparison Sorts I — Insertion Sort [2/2]

`std::move` (<utility>) takes one argument, which it casts to an Rvalue. Use it to force move construction/assignment.

```
a = b;           // Does a copy
a = move(b);     // Does a move
```

`std::move` does not move anything!
It casts to an Rvalue, which makes
its argument *movable*.

The second line of code above is often faster. However, when we do it, we are making an implicit promise: we will not use the current value of `b` again.

```
cout << b;       // BAD!
```

```
b = c;
cout << b;       // Okay
```

There is another `std::move`, in
<algorithm>, taking 3 arguments.
It is the move version of `std::copy`.

Asymptotic Notation

Asymptotic Notation

Big-O More Generally — Introduction

Recall our definition of big- O :

Algorithm A is **order** $f(n)$ [written $O(f(n))$] if there exist constants k and n_0 such that algorithm A performs no more than $k \times f(n)$ basic operations when given input of size $n \geq n_0$.

The fundamental idea here actually has little to do with algorithms. Rather, this is a method for talking about *how quickly a function grows*—a *mathematical* function, that is.

We have applied this idea to the (mathematical) function that tells the maximum number of steps an algorithm takes for input of a given size.

But we could apply it to other things, too.

Asymptotic Notation

Big-O More Generally — Definition

Suppose we have nonnegative real-valued functions f and g on the nonnegative integers. That is, for each nonnegative integer n , $f(n)$ and $g(n)$ are nonnegative real numbers.

We say $g(n)$ is $O(f(n))$ if there exist constants k and n_0 such that $g(n) \leq k \times f(n)$, whenever $n \geq n_0$.

Big-O is an example of **asymptotic notation**: it is about what happens when a number (often n) gets arbitrarily large.

	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

Our earlier definition of big-O is a special case: let $g(n)$ be the maximum number of basic operations required to execute algorithm A for input of size n .

Asymptotic Notation

Big-O More Generally — Applications

We can now use big- O for other concepts—for example, space efficiency.

We have defined **in-place** to be the same as $O(1)$ additional space (**additional** = beyond the space required by its input).

So *in-place* means *constant* additional space.

Bubble Sort and Insertion Sort use $O(1)$, that is, constant, additional space.

So does Binary Search, if the recursion is eliminated. Otherwise, it uses *logarithmic* additional space for the recursion.

Our next sorting algorithm can use more than this.

Asymptotic Notation

Omega

Another kind of asymptotic notation: Ω (Omega).

We say $g(n)$ is $\Omega(f(n))$ if
there exist constants k and n_0 such that
 $g(n) \geq k \times f(n)$, whenever $n \geq n_0$.

The definition of big- O
has " \leq " here.

	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

If we say an *algorithm* is $\Omega(f(n))$, then we mean that, for input of size n , the algorithm's *worst-case* number of basic operations is at least $k \times f(n)$, for some number k , when n is large enough.

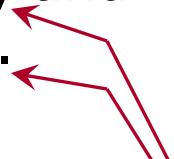
Its best-case may be smaller.

Asymptotic Notation

Theta

One last kind of asymptotic notation: Θ (Theta).

We say $g(n)$ is $\Theta(f(n))$ if
 $g(n)$ is $O(f(n))$, and
 $g(n)$ is $\Omega(f(n))$.



The values of k used above may be different.

For example, a function would be $\Theta(n^2)$ if it always lies between (say) $3n^2$ and $7n^2$, whenever n is large enough.

	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

Asymptotic Notation

Summary

Three ways to say how fast a (mathematical) function grows.
 $g(n)$ is:

- $O(f(n))$ if $g(n) \leq k \times f(n) \dots$
- $\Omega(f(n))$ if $g(n) \geq k \times f(n) \dots$
- $\Theta(f(n))$ if both are true—possibly with different values of k .

Θ is very useful!
 Ω not as much,
but we *will* use it.

	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

Useful: Let $g(n)$ be the maximum number of basic operations performed by some algorithm when given input of size n .

Or: let $g(n)$ be the maximum amount of additional space some algorithm uses when given input of size n .

Divide and Conquer

Divide and Conquer

Algorithmic Strategies

An **algorithmic strategy** is a general method for putting together an algorithm.

Example: split the input into parts and handle each part with a recursive call. This idea, called **Divide and Conquer**, is used by a number of fast algorithms.

A similar idea is used by Binary Search, which splits its input into parts, but only makes a recursive call on *one* of the parts. We call this **Decrease and Conquer**.

See CS 411 for more about these and other algorithmic strategies.

Questions

- How do we analyze the efficiency of algorithms that use Divide and Conquer or Decrease and Conquer?
- Can we use Divide and Conquer to build an improved sorting algorithm? One faster than $\Theta(n^2)$? (We have not seen any, yet.)

Divide and Conquer

The Master Theorem — Background

Say we are analyzing a recursive algorithm.

- Its worst-case number of operations, for input of size n , is $T(n)$.
- We want to know how fast $T(n)$ grows.

Suppose our algorithm uses Divide/Decrease and Conquer:

- The number of recursive calls it makes is a .
- The size of the dataset passed to each recursive call is n/\underline{b} (or a nearby integer, if n/b is not an integer).
- Whatever other work the algorithm does requires $f(n)$ operations.

This gives us a recurrence relation:

- $T(n) = a T(n/b) + f(n)$.
 - “ n/b ” can be a nearby integer.

If the algorithm splits its input into parts of size (about) n/b , then b is the number of parts, and those parts must be (nearly) equal-sized.

Given such a recurrence, we can often determine the order of $T(n)$ using the **Master Theorem**.

Divide and Conquer

The Master Theorem — Statement

The **Master Theorem**

Suppose $T(n) = a T(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is $\Theta(n^d)$.

- “ n/b ” can be a nearby integer.

Compare a to b^d .

- 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 ”.

↑
Recall: $\log_b a$ is the power we would need to raise b to, in order to get a . So $b^k = a$.

Divide and Conquer

The Master Theorem — Using It

A typical application of the Master Theorem proceed as follows.

We are analyzing an algorithm that takes input of size n . It splits its input into nearly equal-sized parts, and it makes recursive calls, each call handling one of the parts.

Find b , a , d .

- b is the number of nearly equal-sized parts.
- a is the number of recursive calls.
- $f(n)$ is the amount of other work done in the body of the algorithm.
- Write $f(n)$ as $\Theta(n^d)$ or $O(n^d)$.

Other work means anything the algorithm does other than making recursive calls.

Compare a to b^d , and apply the appropriate case: 1, 2, or 3.

Sequential Search is $\Theta(n)$ (use the Rule of Thumb).

Analyze Binary Search using the Master Theorem:

- Find b, a, d .
 - Binary Search splits its input into 2 nearly-equal-sized parts.
 - $b = 2$.
 - Binary Search makes 1 recursive call.
 - $a = 1$.
 - In addition, Binary Search does two comparisons and finds the middle of a random-access dataset: constant time.
 - $f(n)$ is $\Theta(1)$. 1 is n^0 . So $d = 0$.
- Which Case?
 - Compare a ($= 1$) with b^d ($= 2^0 = 1$). $a = b^d \rightarrow$ Case 2.
- Conclusion
 - By Case 2 of the Master Theorem, $T(n)$ is $\Theta(n^d \log n)$.
 - That is, $T(n)$ is $\Theta(n^0 \log n)$.
 - Simplify. Binary Search is $\Theta(\log n)$: logarithmic time.

Divide and Conquer

Logarithmic Time

Divide/Decrease and Conquer are common ways to get algorithms that are $\Theta(\log n)$ or $\Theta(n \log n)$.

We said that the base of the logarithm does not matter. Why?

- Suppose (for example) that an algorithm takes $5 \log_2 n$ steps.
- This algorithm is $\Theta(\log_2 n)$.
- Is it also $\Theta(\log_{10} n)$? Yes!
- $5 \log_2 n = 5(\log_2 10 \times \log_{10} n) = (5 \log_2 10) \times \log_{10} n$.

This is just a number.
It is about 16.6.

Fact. If b and c are greater than 1, then $\Theta(\log_b n)$ and $\Theta(\log_c n)$ are the same thing—and similarly for $O(\log_b n)$ and $O(\log_c n)$.

So we generally leave off the base and simply say $\Theta(\log n)$, $O(\log n)$, $\Theta(n \log n)$, etc.

Comparison Sorts II

Comparison Sorts II

Merge Sort — Introduction

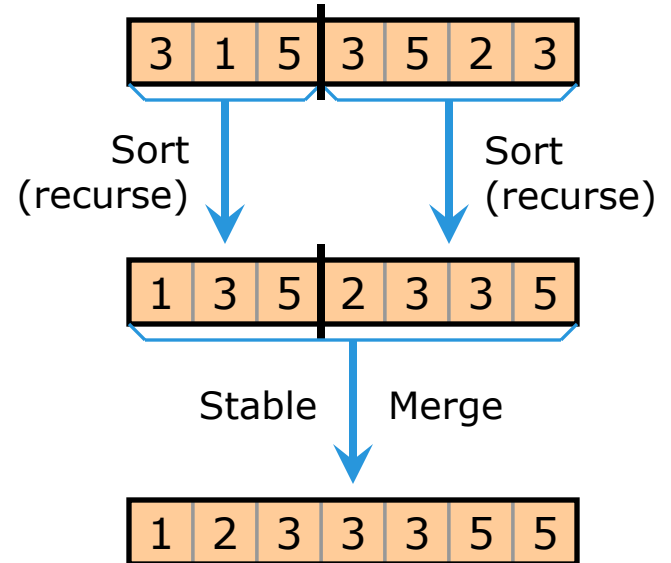
We can use Divide and Conquer to build a better sort.

We are given a list to sort.

Split the list into two parts that are the same size—or nearly so.

Sort each part with a recursive call.

Merge the parts into a single sorted list. Do this without reversing the relative order of equivalent items—that is, in a **stable** manner: **Stable Merge**.



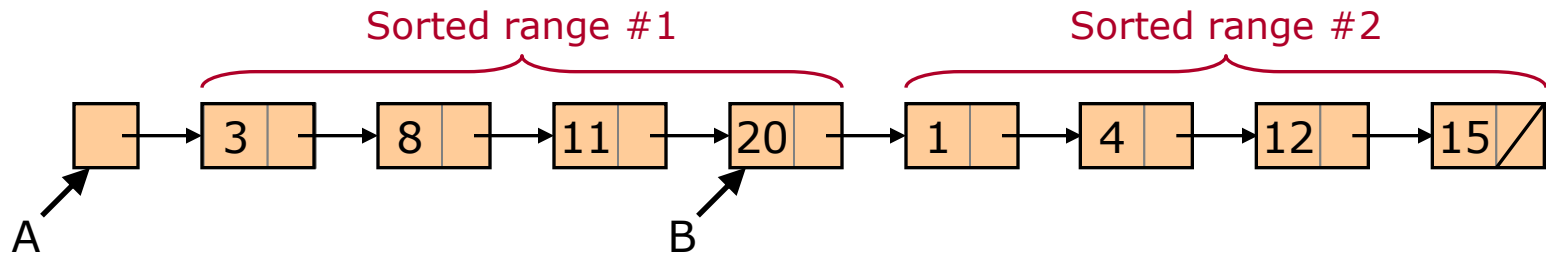
This algorithm is called **Merge Sort** [John von Neumann, 1945].

Comparison Sorts II

Merge Sort — Merging in a Linked List

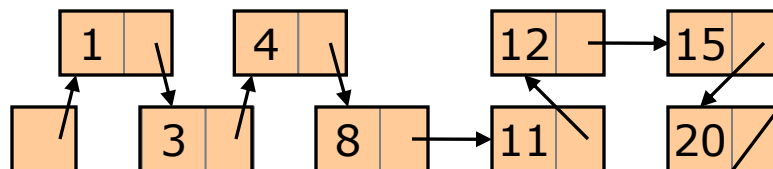
Consider how a Stable Merge would be done.

We can do an efficient Stable Merge of a Linked List in-place.



To merge two sorted ranges within a Linked List:

- Two pointers: A & B. A starts at the head, B at the end of range #1.
- Check whether the item after B's node is less than the item after A's node. If so, remove the item after B's node and re-insert it after A.
 - This uses only pointer operations. We do not move any data items.
- Advance A and/or B as appropriate, and repeat.



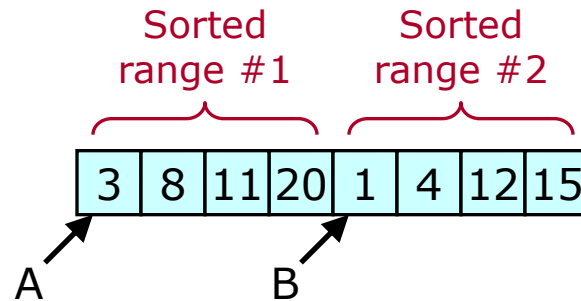
Result of
Stable Merge
operation

Comparison Sorts II

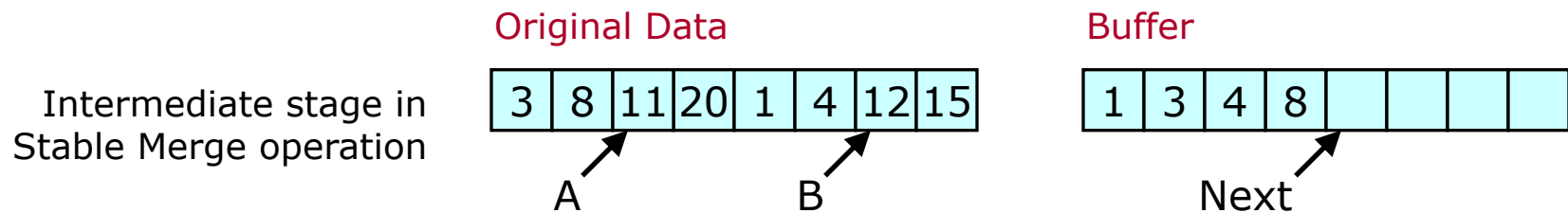
Merge Sort — General-Purpose Merge

Efficient Stable Merge in an array generally uses a separate buffer.

- This Stable Merge algorithm does not *require* an array; it works with just about any kind of data.



As before, use two pointers. Check which item comes first, and copy that to the buffer. Advance pointers as appropriate.



At the end, we *may* copy the buffer back to the original array.

Comparison Sorts II

Merge Sort — CODE

TO DO

- Implement Merge Sort.
 - Make the Stable Merge a separate function. Use the general-purpose Stable Merge algorithm.
- Analyze.
 - *Coming up.*

Done. See `merge_sort.cpp`.

Note. Our code allocates the buffer every time a Stable Merge is done. It also merges to the buffer and then copies the data back every time. There are ways to handle the Stable Merge more efficiently. However, this simple version of Merge Sort should give us a decent idea of how it works and how fast it is.

Comparison Sorts II

Merge Sort — Analysis [1/3]

We wish to analyze Merge Sort using the Master Theorem.
How much “other work” [$f(n)$] does it do?

In addition to the recursive calls, Merge Sort does:

- Base-case check: $\Theta(1)$.
- Find the middle: $\Theta(1)$ for array, $\Theta(n)$ for Linked List.
- Stable Merge: $\Theta(n)$ for both versions.

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.

Merge Sort’s other work: $\Theta(1) + \Theta(1) + \Theta(n)$ or $\Theta(1) + \Theta(n) + \Theta(n)$.
Result: $\Theta(n)$ —linear time—for both.

Comparison Sorts II

Merge Sort — Analysis [2/3]

Analyze Merge Sort using the Master Theorem

- Find b, a, d .
 - Merge Sort splits its input into 2 nearly-equal-sized parts.
 - $b = 2$.
 - Merge Sort makes 2 recursive calls.
 - $a = 2$.
 - Merge Sort's other work, from the previous slide: linear time.
 - $f(n)$ is $\Theta(n)$. n is n^1 . So $d = 1$.
- Which Case?
 - Compare a ($= 2$) with b^d ($= 2^1 = 2$). $a = b^d \rightarrow$ Case 2.
- Conclusion
 - By Case 2 of the Master Theorem, $T(n)$ is $\Theta(n^d \log n)$.
 - That is, $T(n)$ is $\Theta(n^1 \log n)$.
 - Simplify. Merge Sort is $\Theta(n \log n)$: log-linear time.

Comparison Sorts II

Merge Sort — Analysis [3/3]

(Time) Efficiency 😊

See `iterative_merge_sort.cpp`.

- Merge Sort is $\Theta(n \log n)$.
- Merge Sort also has an average-case time of $\Theta(n \log n)$.

Requirements on Data 😊

- Merge Sort does not require random-access data.
- Operations needed. General: copy. Linked List: NONE (compare).

Space Efficiency 😊/😊/😞

- Recursive Merge Sort uses stack space: recursion depth $\approx \log_2 n$.
 - An iterative version can avoid this (small) memory requirement. ←
- For a Linked List, no more is needed: $\Theta(\log n)$ additional space. 😊
 - Or $\Theta(1)$ additional space, for an iterative version. 😊
- General-purpose Merge Sort uses a buffer: $\Theta(n)$ 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 — Notes

Merge Sort has the characteristics we are most interested in:

- It runs in $\Theta(n \log n)$ time.
- It is stable.
- It works well with different kinds of data—Linked Lists, in particular.
 - Note that it may be written differently for different kinds of data.

Merge Sort is very practical and is often used.

- Merge Sort is considered to be the *fastest known* general-purpose comparison sort:
 - When a stable sort is required.
 - When sorting a Linked List.
- Merge Sort is the usual implementation for three of the seven sorting algorithms in the C++ Standard Template Library.

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