Introduction to Sorting
Comparison Sorts I

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, September 30, 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

Analysis of Algorithms [1/4]

Efficient [noun form: **efficiency**]

- General meaning. Using few **resources**: time, space, etc.
- Specific meaning. Fast—not using much *time*.
- Other specific meanings when qualified: **space efficient**, etc.

Idea for Measuring Efficiency

- View the tasks an algorithm performs as a series of steps. The steps we count are called **basic operations**.
- Determine the maximum number of basic operations required for input of a given size. Write this as a formula.
- Look at the most important part of the formula.

The most important part of $n \log_{10} n + 72n + 3n^2 + 936$ is $3n^2$.
We often do not care about the 3. The really important part is n^2.

Review
Analysis of Algorithms [2/4]

Our *usual* **model of computation**:

- Legal **operations**: no data access except thru provided channels.
- **Basic operations** (the operations we count):
  - A built-in operation on a fundamental type.
  - A call to a client-provided function.
- We are given a collection. Its **size** is the number of items in it.

It matters what we count!

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 $n \geq n_0$.

We will use big-*O* every single day for the rest of the semester.

Efficiency expressed using big-*O* tells us much of what we need to know to determine if an algorithm is **scalable** (works well with large problems—we say such an algorithm **scales well**).

Review
Analysis of Algorithms [3/4]

Cannot read all of input ↑

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

Probably not scalable ↓

Faster ↑

Slower ↓

*As we will see, the base of the logarithms does not matter.

I will also allow $O(n^3)$, $O(n^4)$, etc.

We are interested in the fastest category that an algorithm fits in.

When we use big-*O* with an algorithm, unless we say otherwise, we are describing **worst-case** behavior: for input of a given size, what is the *maximum* number of basic operations performed?

Analysis of Algorithms [4/4]

When determining big-*O*, we can collapse any *fixed* number of steps into a single step without altering the order.

Rule of Thumb. In nested loops with basic operations in the body of the innermost loop, if each loop is executed *n* times, or executed *i* times, where *i* goes up to *n* (plus a constant?), then the order is $O(n^t)$ where *t* is the number of nested loops.

```
for (int i = 0; i < n-4; ++i)
    for (int j = 0; j < i; ++j)
        for (int k = j; k < i+4; ++k)
            ++a[k];
```

$O(n^3)$

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i-5; ++j)
        for (int k = 0; k < 5; ++k)
            ++arr[j+k];
```

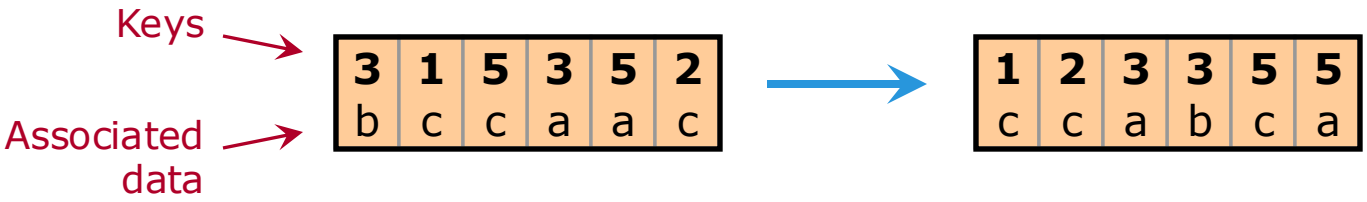
$O(n^2)$, or quadratic time

Introduction to Sorting

Introduction to Sorting
Basics [1/2]

To **sort** a collection of data means to rearrange its items in order.

Often the items we sort are themselves collections of data. The part we sort by is the **key**.



Efficient sorting is of great interest.

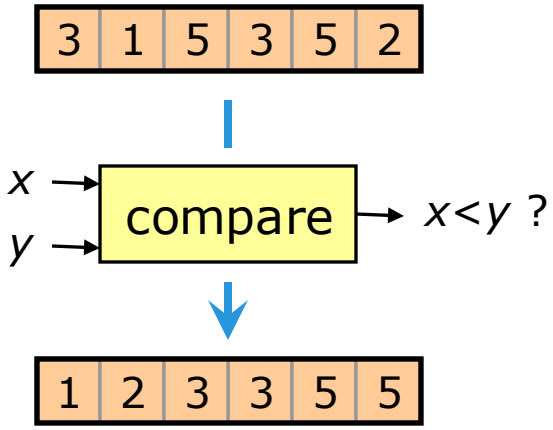
- Sorting is a common operation.
- Sorting code that is written with little thought/knowledge is often much less efficient than code using a good algorithm.
- No single known sorting algorithm has all the properties we want.
- Some algorithms (like Binary Search) require sorted data. The efficiency of sorting affects the desirability of such algorithms.

Introduction to Sorting
Basics [2/2]

A **sort** (it is a noun here) is an algorithm that does sorting.

A **comparison sort** is a sort that only gets information about the data items in its input using a *comparison function*.

- A **comparison function** is a function that takes two data items and indicates which comes first. (Think "<".)
- When we study comparison sorts, we modify our model of computation: there are fewer legal operations.



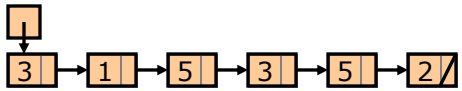
In the next few class meetings, we will analyze various **general-purpose comparison sorts**.

By **general purpose** I mean that we place no restrictions on the size of the list to be sorted, or the values in it. *This is my own definition; it is not standard terminology.*

Introduction to Sorting
Analyzing a Sort [1/2]

We analyze a general-purpose comparison sort using five factors.

- (Time) Efficiency
  - What is the (worst-case!) order of the algorithm?
  - What about **average case**—over all possible inputs of a given size?
- Requirements on Data
  - Does the algorithm require random-access data?
  - Does it work with Linked Lists?



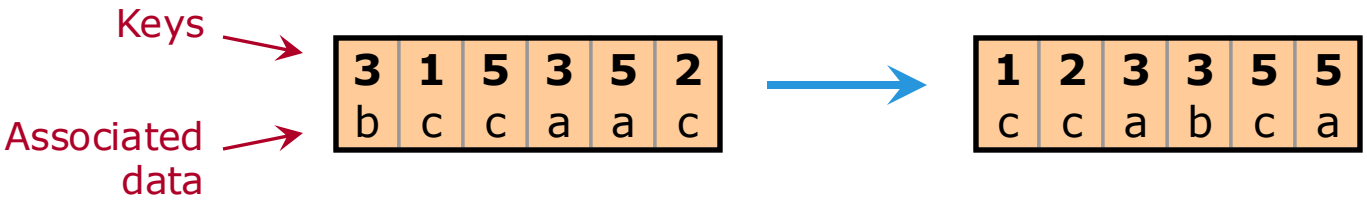
- Space Efficiency
  - Is the algorithm **in-place** (no large additional space required)?
  - How much additional space (variables, buffers, etc.) is required?
- Stability
  - Is the algorithm **stable** (never reverses the relative order of equivalent items)?
- Performance on Nearly Sorted Data
  - **Nearly sorted**. Type 1: all items close to proper spots. Type 2: few items out of order.

"No large", "close", "few": at most a fixed constant, no matter how large the input is.

Analyzing a Sort [2/2]

Again, we say a sort is **stable** if it never reverses the relative order of equivalent items.

Recall the example used to illustrate what **keys** are.



The algorithm used to sort the above list is *not* stable, since the items at right, which are equivalent, had their relative order reversed.

Remember: two items are equivalent if they have equivalent keys; any associated data will be ignored when comparing items.

Introduction to Sorting
Overview of Algorithms

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 [$O(n^2)$] 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

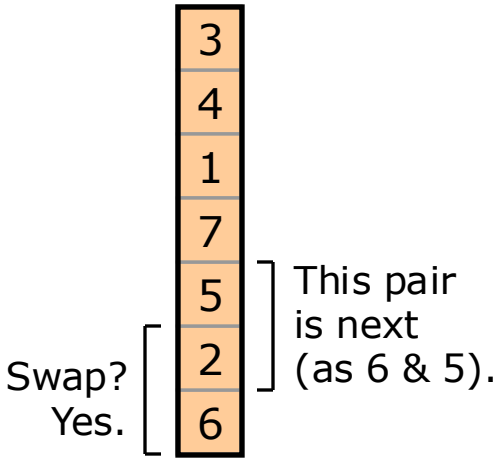
It may seem odd that an algorithm called "Quicksort" is in the slow category. But this is not a mistake! *More about this in a few days.*

Comparison Sorts I

Bubble Sort — Description

We begin with a very simple sort: **Bubble Sort**.

- Bubble sort is easy to understand and analyze. But we do not use it for anything practical.



Bubble Sort proceeds in a number of **passes**.

- In each pass, we compare consecutive pairs of items. An out-of-order pair is swapped.
- Think of a vertical list, bottom to top. Large items rise like bubbles.
- After the first pass, the last item is the largest.
- So later passes need not go through all the data.

We can improve Bubble Sort's performance on some nearly sorted data—specifically, Type 1 (all items close to proper place):

- In each pass, track whether we have done swaps during that pass.
- If not, then the data were sorted when the pass began. Quit.

Comparison Sorts I
Bubble Sort — CODE

TO DO

- Examine an implementation of Bubble Sort.
- Analyze it.
  - *Coming up.*

See `bubble_sort.cpp`.

Bubble Sort — Analysis

(Time) Efficiency ☹

- Bubble Sort is $O(n^2)$.
- Bubble Sort also has an average-case time of $O(n^2)$. ☹

Requirements on Data ☺

- Bubble Sort does not require random-access data.
- It works on Linked Lists.

Space Efficiency ☺

- Bubble Sort can be done in-place.

Stability ☺

- Bubble Sort is stable.

Performance on Nearly Sorted Data ☺/☹

- Type 1 (all close). An optimized Bubble Sort is $O(n)$ for all items close to their proper spots. ☺
- Type 2 (few wrong). Bubble Sort can be $O(n^2)$ if only one item is out of order. ☹

There are several smileys here, but **these** are more important.

Bubble Sort — In Practice

Bubble Sort is virtually never used in practice. Its primary purpose is to be an example of an easy-to-understand sorting algorithm.

All the other sorts we cover will all be at least a *little bit* practical—and some will be *very* practical.

Insertion Sort — Description

We can think of Bubble Sort as constructing a sorted sequence in backwards order:

- Find the greatest item (by "bubbling"), then the next greatest, etc.
- So for each **position**, starting with the last, it finds the **item** that belongs there.

Suppose we flip this around.

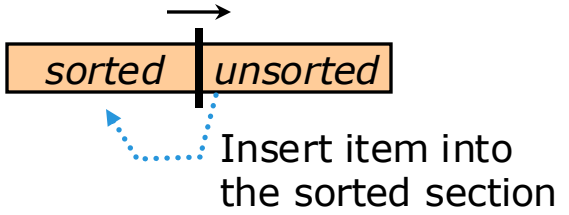
- Instead of looking through the positions and determining what item belongs in each, look through the given **items**, determine in which **position** each belongs, and then insert it in that position.

This idea leads to an algorithm called **Insertion Sort**.

- Iterate through the items in the sequence.
- For each, insert it in the proper place among the preceding items.
- Thus, when we are processing item k, we have items 0 .. k–1 already in sorted order.

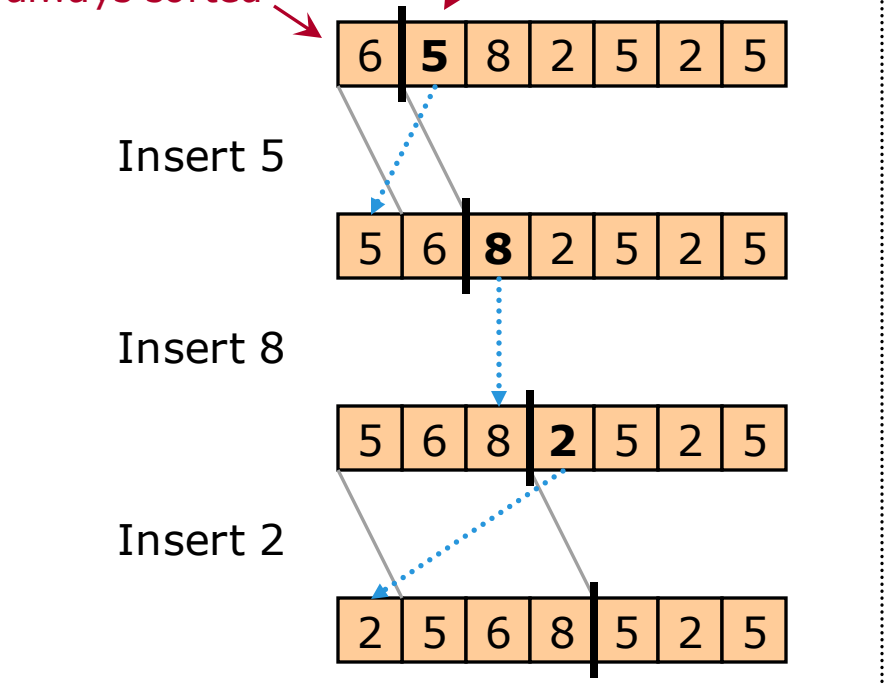
Insertion Sort — Illustration

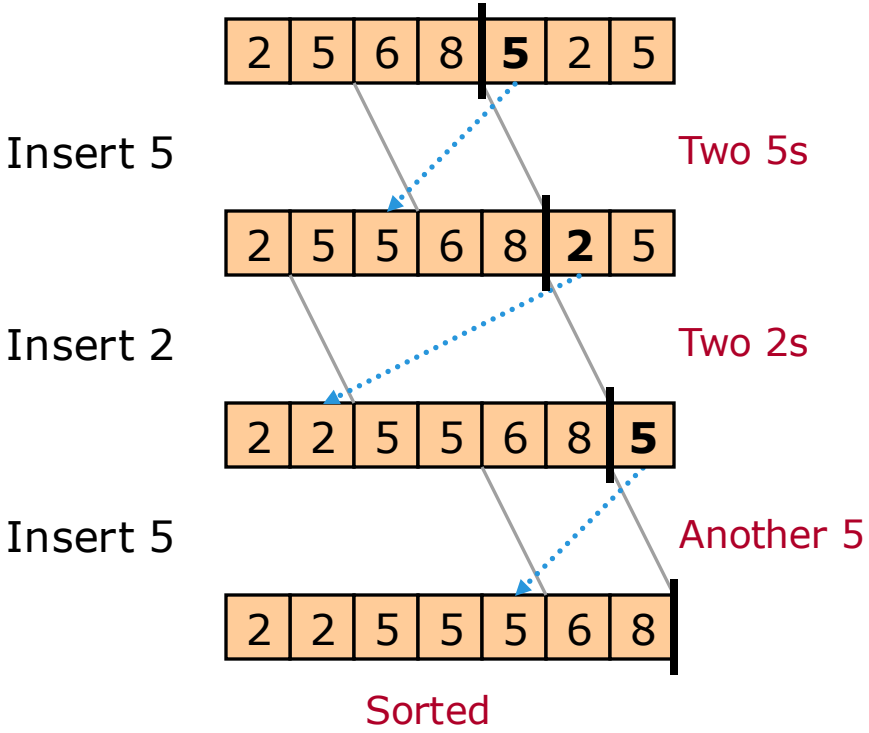
Items to the left of the bar are sorted.



Bold item: to be inserted into sorted section

A list of size 1 is always sorted





Insertion Sort — How to Search

What is the best way to find the insertion location—the spot in the sorted part of the list where an item should be inserted?

- Sequential Search?
- Binary Search?

We usually use a third option: **backward Sequential Search**—Sequential Search proceeding from back to front.

Why?

- First, Insertion Sort is most useful when the dataset is already nearly sorted. For such data, a backward Sequential Search tends to find the insertion location quickly.
- Second, using Binary Search would not make the algorithm any faster. For an array, we need to go backwards sequentially through the data anyway, since each data item after the insertion location must be moved up. And for a Linked List—or other non-random-access structure—Binary Search is not very fast anyway.

Insertion Sort — CODE

TO DO

- Implement Insertion Sort.
- Analyze, as before.
  - *Coming up.*

Done. See `insertion_sort.cpp`.

Comparison Sorts I
Insertion Sort — On std::move

`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 a non-const 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`.

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 ☺

- The usual implementation is $O(n)$ for both Type 1* (all close) and Type 2 (few wrong).

*For forward-only sequential-access data, significant extra space usage is required to allow for linear-time sorting of all nearly sorted datasets.

Insertion Sort — In Practice

Insertion Sort is too slow for most use cases.

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.

- Optimized sorting code typically does Insertion Sort on small lists.
- Some sorting methods get the data nearly sorted, and then finish with a call to Insertion Sort. *More on this when we cover Quicksort.*