Comparison Sorts I
More on Big-*O*

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, October 9, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

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 [1/4]

Efficiency

- General: using few resources (time, space, bandwidth, etc.).
- Specific: fast (time).
  - Also can be qualified, e.g., space efficiency.

Analyzing Efficiency

- Measure running time in **steps**.
- Determine how the **size** **of the input** affects running time.
- **Worst case**: max steps for given input size.

Our **model of computation** specifies:

- A **step** is one of the following:
  - A built-in operation on a fundamental type.
  - A call to a client-provided function.
- When we are given a list as input, its **size** is the number of items in it.

Scalable: works well with large problems. Also "**scales well**".

Introduction to Analysis of Algorithms [2/4]

Algorithm *A* is *order* $f(n)$ [written $O(f(n))$] if

- There exist constants k and n_0 such that
- *A* requires **no more than** $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

Notes

- Big-*O* is important!
  - However, we do not use the above definition, if we can avoid it.
- Big-*O* refers to **worst-case** behavior, unless otherwise stated.

Review
Introduction to Analysis of Algorithms [3/4]

Here are the efficiency categories we will be using.

Know these!

Using Big-*O*	In Words
$O(1)$	Constant time
$O(\log_b n)$, for some $b > 1$	Logarithmic time
$O(n)$	Linear time
$O(n \log_b n)$, for some $b > 1$	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

Notes

- The **fastest** category an algorithm fits in, is the one we want.
- I will also allow $O(n^3)$, $O(n^4)$, etc.

Review
Introduction to Analysis of Algorithms [4/4]

Rules of Thumb

- When determining big-*O*, we can collapse any *constant* number of steps into a single step.
- For nested loops:
  - If each is either executed *n* times, or executed *i* times, where *i* goes up to *n* (plus some constant?).
  - Then the order is $O(n^t)$ where *t* is the number of loops.

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

$O(n^3)$

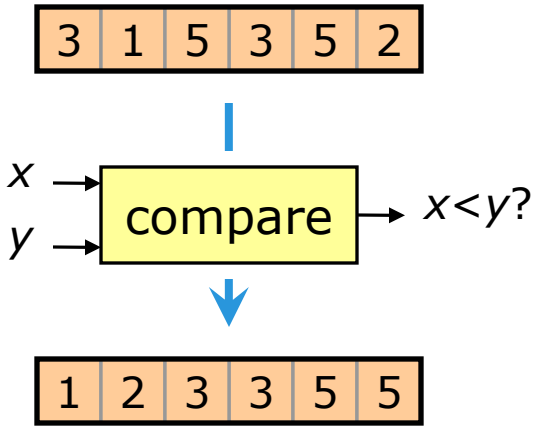
```
for (int i = 0; i < n; ++i)
   for (int j = 0; j < i; ++j)
      for (int k = 0; k < 5; ++k)
         ++arr[j][k];
```

$O(n^2)$

To **sort** a collection of data is to place it in order.

- The part of the data used to sort is the **key**.

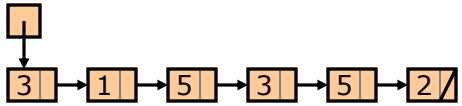
We are interested primarily in **comparison sorts**: sorting algorithms that get their information only by comparing items in pairs.



A **general-purpose comparison sort** places no restrictions on the size of the list to be sorted, or the values in it (other than that they all have the same type).

We analyze a general-purpose comparison sort using five criteria:

- (Time) Efficiency
  - What is the (worst-case!) order of the algorithm?
  - Is the algorithm much faster on average (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
  - Can the algorithm sort in-place?
    - **In-place** = no large additional storage space required.
  - How much additional storage (variables, buffers, etc.) is required?
- Stability
  - Is the algorithm stable?
    - **Stable** = never changes order of equivalent items.
- Performance on Nearly Sorted Data
  - Is the algorithm faster when its input is already sorted or nearly sorted?
    - **Nearly sorted** = (1) All items close to proper places, OR (2) few items out of order.

"Large", "close", "few": criterion is whether the number is at most a **fixed constant**.

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

Bubble Sort — Description

One of the simplest sorting algorithms is also one of the worst: **Bubble Sort**.

- We cover it because it is easy to understand and analyze.
- But we never use it.

Bubble Sort proceeds in a number of "passes".

- In each pass, we go through the data, considering each consecutive pair of items.
- We compare. If the pair is out of order, we swap.
- The larger items rise to the top like bubbles.
  - I assume we are sorting in ascending order.
- After the first pass, the last item is the largest.
- Thus, 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):

- During each pass, keep track of whether we have done any swaps during that pass.
- If not, then we were done when the pass began. Quit.

Bubble Sort — Write It

TO DO

- Examine an implementation of Bubble Sort.
- Analyze it.
  - *See the next slide.*

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

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 ☺/☹

- (1) We can write Bubble Sort to be $O(n)$ if no item is far out of place. ☺
- (2) Bubble Sort is $O(n^2)$ even if only one item is far out of place. ☹

Lots of smileys here. However, these are more important.

Bubble Sort — Comments

Bubble Sort is very slow.

- It is never used, except:
  - In CS classes, as a simple example.
  - By people who do not understand sorting algorithms very well.

Insertion Sort — Description

Bubble Sort essentially constructs a sorted sequence in (backwards) order:

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

Suppose we "flip" this idea.

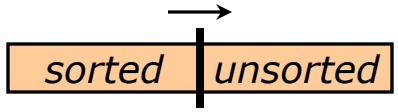
- Instead of looking through the positions and determining what item belongs in each, look through the given **items**, determine in which **position** each belongs, and then insert it in that position.

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

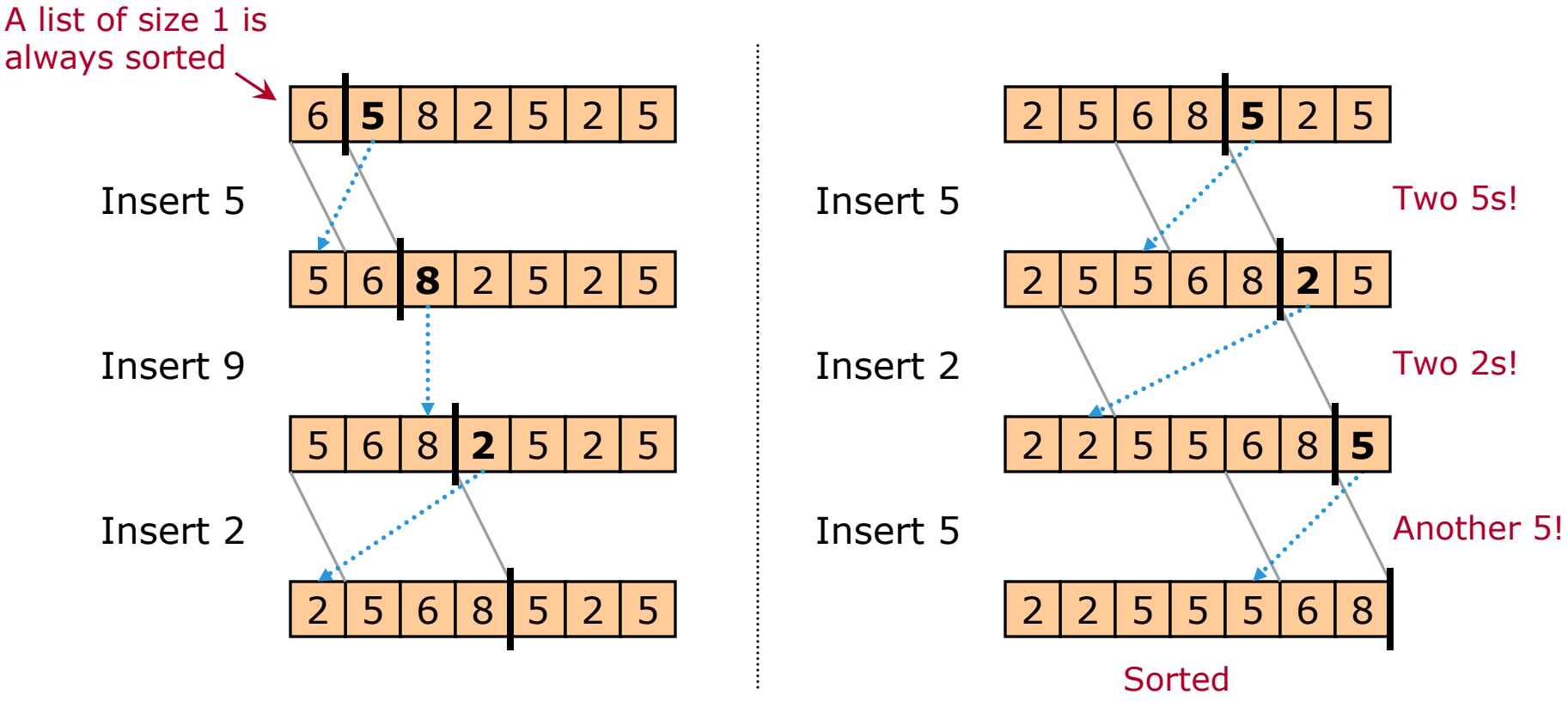
- Iterate through the items in the sequence.
- For each, insert it in the proper place among the preceding items.
- Thus, when we are processing item k, we have items 0 .. k–1 already in sorted order.

Comparison Sorts I
Insertion Sort — Illustration

Items to left of bold bar are sorted.



Bold item = item to be inserted into sorted section.



Insertion Sort — Issues

When we insert an item into the list of already sorted items, three operations are needed:

- **Find** (the proper location).
- **Remove** (the old copy).
- **Insert** (into the proper location).

If we have **random-access** data (an array?), then

- Find *could* be fast [Binary Search: $O(\log n)$].
  - In practice, we do not actually use Binary Search, as we will see.
- Remove + Insert is slow-ish [move items up: $O(n)$].
- Find + Remove + Insert is $O(n)$.

If we have a **Linked List**, then

- Find is slow-ish [Sequential Search: $O(n)$].
- Remove is fast [Linked-List removal: $O(1)$].
- Insert is fast [Linked-List insertion: $O(1)$].
- Find + Remove + Insert is $O(n)$.

In both cases, Find + Remove + Insert is $O(n)$.

Comparison Sorts I
Insertion Sort — Write It

TO DO

- Implement Insertion Sort.
- Analyze, as before.
  - *See the next slide.*

Done. See **`insertion_sort.cpp`**, *on the web page.*

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.

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.)

Implementation Issue

- For random-access data, Insertion Sort *could* use Binary Search in the "find" step. However, this is not the best method for nearly sorted data. Since that is when Insertion Sort is actually useful, we generally do the "find" step of Insertion Sort using a backwards Sequential Search.

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
  - *A* requires **no more than** $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

The fundamental idea here actually has very 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 us the maximum number of steps an algorithm takes for input of a given size.
- But we could apply it to other things, too.

More on Big-*O*
Big-*O* More Generally — Definition

Say we have nonnegative real-valued functions f and g on the nonnegative integers.

- So, 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 $n \geq n_0$.

We get our previous definition of big-*O* if we let $g(n)$ be the maximum number of steps required to execute algorithm A for input of size n.

More on Big-*O*
Big-*O* More Generally — Applications

We can now use big-*O* for more general 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.
  - Thus: *Constant* additional space.
- So Bubble Sort and Insertion Sort use $O(1)$, that is, constant, additional space.
  - Our next sorting algorithm will use more than this.

More on Big-*O*
Related Concepts

Related to big-*O* are Ω (omega) and Θ (theta).

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 $n \geq n_0$.

Our big-*O* definition has "≤" here.

If we say an *algorithm* is $\Omega(f(n))$, then we mean that its *worst-case* number of steps is at least $k \times f(n)$, for some k. (Its best-case may be smaller.)

We say $g(n)$ is $\Theta(f(n))$ if

- $g(n)$ is $O(f(n))$, and
- $g(n)$ is $\Omega(f(n))$.

The "*k*" values used above may be different.

- For example, a function would be $\Theta(n^2)$ if it always lies between (say) $3n^2$ and $5n^2$, for sufficiently large n.

More on Big-*O*
Summary

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.