Introduction to Analysis (cont.)
Comparison Sorts I

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, February 27, 2012

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

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**".

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 $n \geq n_0$.

We are usually not interested in the exact values of *k* and n_0. Thus:

- We don't worry much about whether some algorithm is (say) five times faster than another.
- We ignore small problem sizes.

Big-*O* is important!

- We will probably use it *every day* for the rest of the semester (the concept, not the above definition).

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Worst Case & Average Case

When we use big-*O*, unless we say otherwise, we are always referring to the **worst-case** behavior of an algorithm.

- For input of a given size, what is the **maximum** number of steps the algorithm requires?

We can also do average-case analysis. However, we need to say so. We also need to indicate what kind of average we mean. For example:

- We can determine the average number of steps required over all inputs of a given size.
- We can determine the average number of steps required over repeated applications of the same algorithm.

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Efficiency Categories

An $O(1)$ algorithm is **constant time**.
- The running time of such an algorithm is essentially independent of the input.
- Such algorithms are rare, since they cannot even read all of their input.

An $O(\log_b n)$ [for some b] algorithm is **logarithmic time**.
- Again, such algorithms cannot read all of their input.
- As we will see, we do not care what b is.

An $O(n)$ algorithm is **linear time**.
- Such algorithms are not rare.
- This is as fast as an algorithm can be and still read all of its input.

An $O(n \log_b n)$ [for some b] algorithm is **log-linear time**.
- This is about as slow as an algorithm can be and still be truly useful (scalable).

An $O(n^2)$ algorithm is **quadratic time**.
- These are usually too slow for anything but very small data sets.

An $O(b^n)$ [for some b] algorithm is **exponential time**.
- These algorithms are *much* too slow to be useful.

Faster

Slower

Notes
- Gaps between these categories are *not* bridged by compiler optimization.
- We are interested in the **fastest category** above that an algorithm fits in.
  - Every $O(1)$ algorithm is also $O(n^2)$ and $O(237^n + 184)$; but "$O(1)$" interests us most.
- **I will also allow $O(n^3)$, $O(n^4)$, etc.** However, we will not see these much.

Determine the order of the following, and express it with "big-*O*":

```
int func2(int p[], int n) // n is length of array p
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            sum += p[j];
    return sum;
}
```

See the next slide.

In Example 2:

- There is a loop within a loop. The body of the inside (*j*) loop looks like this:

```
for (int j = 0; j < n; ++j)
    sum += p[j];
```

- A single execution of this inside loop requires $3n+2$ steps.
  - If we treat "`sum += p[j];`" as a single step.
- However, the loop itself is executed *n* times by the outside (*i*) loop. Thus a total of $n \times (3n+2) = 3n^2+2n$ steps are required.
- The rest of the function takes $2n+6$ steps, for a total of $(3n^2+2n) + (2n+6) = 3n^2+4n+6$.
- Again, strictly speaking, it would be correct to say that `func2` is $O(3n^2+4n+6)$, but that is not how we do things.
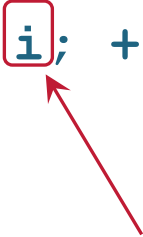- Instead, we note that, for large *n*, $3n^2+4n+6 \leq 4n^2$. Thus, `func2` is $O(n^2)$: quadratic time.

Determine the order of the following, and express it using "big-*O*":

```
int func3(int p[], int n) // n is length of array p
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < i; ++j)
            sum += p[j];
    return sum;
}
```

Notice!

See the next slide.

In Example 3:

- The number of steps taken by the j loop is $4i+2$.

- So the total number of steps used by the j loop as i goes from 0 to $n-1$ is

  $2 + 6 + 10 + \ldots + [4(n-1)+2]$.

- Computing the sum, we obtain $[2+4(n-1)+2] \times n \div 2 = 2n^2$.

- The total number of steps for the function as a whole is

  $2n^2 + 2n + 6$.

- Thus the function is $O(n^2)$: quadratic time.

When computing the number of steps used by nested loops:

- For nested loops, each of which is either
  - executed *n* times, or
  - executed *i* times, where *i* goes up to *n*.
    - Or up to *n* plus some constant.
- The order is $O(n^t)$ where *t* is the number of loops.

Example 4

```
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];
```

- By the above rule of thumb, this has order $O(n^3)$.

Example 5

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

Notice!

- The *k* loop uses a **constant** number of operations.
- By the Rule of Thumb, this has order $O(n^2)$.

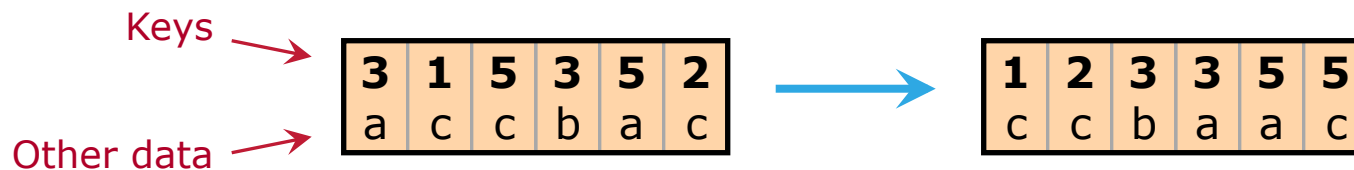To **sort** a collection of data is to place it in order.

3	1	5	3	5	2

⟶

1	2	3	3	5	5

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

Keys ⟶

Other data ⟶

3	**1**	**5**	**3**	**5**	**2**
a	c	c	b	a	c

⟶

1	**2**	**3**	**3**	**5**	**5**
c	c	b	a	a	c

Efficient sorting is of great interest.

- Sorting is a very common operation.
- Sorting code that is written with little thought/knowledge is often **much** less efficient than code using a good algorithm.
- Some algorithms (like Binary Search) require sorted data. The efficiency of sorting affects the desirability of such algorithms.
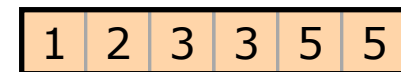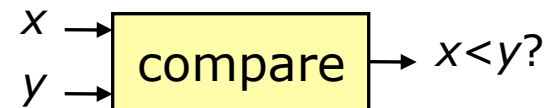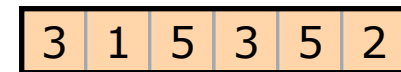
Introduction to Sorting
The Basics — Comparison Sorts

We are interested primarily in **comparison sorts**.

- A *comparison sort* is an algorithm that sorts its input, and *only* gets information about its input using a **comparison function**.

- A *comparison function* is a function that takes two data items and returns true/false to indicate which comes first.

  - Think "<".

In the next few class meetings, we will analyze various **general-purpose comparison sorts**, in terms of efficiency and other desirable properties.
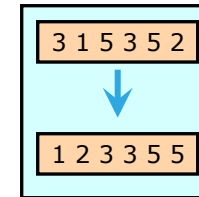
- Here, I use *general purpose* to mean that we place no restrictions on the size of the list to be sorted, or the values in it.
- The only restriction we place on the list, is that the items in it all have the same type.

| 3 | 1 | 5 | 3 | 5 | 2 |

$x \rightarrow$
| compare | $\rightarrow x<y?$
$y \rightarrow$

| 1 | 2 | 3 | 3 | 5 | 5 |

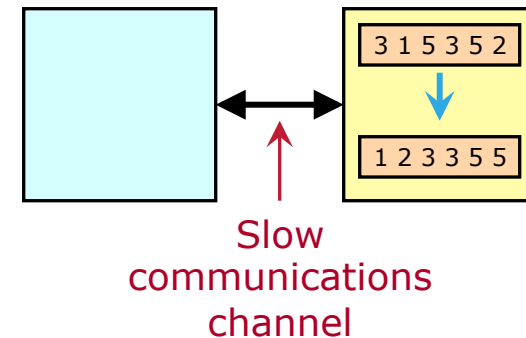# Introduction to Sorting
The Basics — Internal vs. External

Internal data lie in memory.

External data are accessed via some
external device.

- Disk, network, etc.
- Because of relatively slow access time,
  the fact that data are external can affect
  the design of algorithms.

Slow
communications
channel

For now, as we look at sorting, we will concentrate on sorting internal
data.

- All the algorithms we discuss *will work* for external sorting. However, they
  may be poor choices, due to excessive use of a slow communications
 channel.

Introduction to Sorting
Analyzing Sorting Algorithms

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) only a few items out of order.
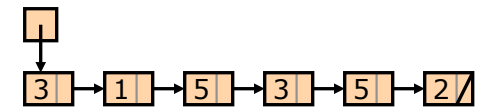
> "Large", "close", "few": criterion is whether the number is at most a **fixed constant**.

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
- Special Purpose — Not Comparison Sorts
  - Pigeonhole Sort
  - Radix Sort

It may seem odd that an algorithm called "Quicksort" is in the *slow* category. *More about this later.*

Comparison Sorts I
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.

Comparison Sorts I
Bubble Sort — Write It

TO DO

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

Comparison Sorts I
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.

Comparison Sorts I
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.