Introduction to Analysis of Algorithms continued
Introduction to Sorting

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, October 7, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell

Unit Overview
Recursion & Searching

Major Topics

- ✓ Introduction to Recursion
- ✓ Search Algorithms
- ✓ Recursion vs. Iteration
- ✓ Eliminating Recursion
- ✓ Recursive Search with Backtracking

DONE

Unit Overview
Algorithmic Efficiency & Sorting

Major Topics

- (part) 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

Efficiency

- General sense: efficient = **using few resources**.
  - The most important resource is time. Thus …
- Specific sense: efficient (if not qualified) = executing **quickly**.

We wish to discuss efficiency of algorithms in a way that is independent of implementation, hardware, etc.

We are most interested in:

- **Time** Used by the Algorithm
  - Expressed in terms of number of **steps**
- How the **Size of the Input** Affects Running Time
- **Worst-Case** Behavior
  - Maximum number of steps the algorithm requires for a given input size.

Our **model of computation** specifies what these mean.

Review
Introduction to Analysis of Algorithms — Model of Computation

The **model of computation** used in this class:

- The following operations will be considered a single **step**:
  - Built-in operations on fundamental types.
  - Calls to client-provided functions — including operators.
    - In particular, member (and other) functions for a template parameter type.
- When we discuss efficiency, we will always consider a function that is given a list of items. The **size** of the input will be the number of items in the list.
  - We will generally denote the size of the input by “*n*”.

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

continued

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.

Order & Big-*O* Notation — Example 1, Problem

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

```
int func1(int p[], int n) // n is length of array p
{
   int sum = 0;
   for (int i = 0; i < n; ++i)
      sum += p[i];
   return sum;
}
```

See the next slide.

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Example 1, Solution

I count 9 single-step operations in **`func1`**.

Strictly speaking, it is correct to say that **`func1`** is $O(4n+6)$. In practice, however, we always place a function into one of a few well-known categories.

ANSWER: Function **`func1`** is $O(n)$.

- This works with (for example) $k = 5$ and $n_0 = 100$.
- That is, $4n + 6 \le 5 \times n$, whenever $n \ge 100$.

What if we count "**`sum += p[i]`**" as one step? What if we count the loop as one?

- Moral: collapsing a **constant** number of steps into one step does not affect the order.
- This is why I said we can be *somewhat* imprecise about what a "step" is.

Operation	Times Executed
`int p[]`	1
`int n`	1
`int sum = 0`	1
`int i = 0`	1
`i < n`	$n + 1$
`++i`	n
`p[i]`	n
`sum += …`	n
`return sum`	1
TOTAL	$4n + 6$

Why are we so interested in the running time of an algorithm for **very large** problem sizes?

- Small problems are easy and fast.
- We expect more of faster computers. Thus, problem sizes keep getting bigger.
- As we saw with search algorithms, the advantages of a fast algorithm become more important at very large problem sizes.

Recall:

- "The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less." — Nick Trefethen

An algorithm (or function or technique …) that works well when used with increasingly large problems & large systems is said to be **scalable**.

- Or, it **scales well**.
- This class is all about things that scale well.

This definition applies in general, not only in computing.

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

Know these!

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.

Order & Big-*O* Notation — Example 2, Problem

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.

Order & Big-*O* Notation — Example 2, Solution

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, $3n^2+4n+6 \le 4n^2$. 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.

Order & Big-O Notation — Example 3, Solution

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.

Order & Big-O Notation — Rule of Thumb & Example 4

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 $O(n^2)$.

Introduction to Sorting
The Basics — What is Sorting?

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 →	**3**	**1**	**5**	**3**	**5**	**2**	→	**1**	**2**	**3**	**3**	**5**	**5**
Other data →	a	c	c	b	a	c		c	c	b	a	a	c

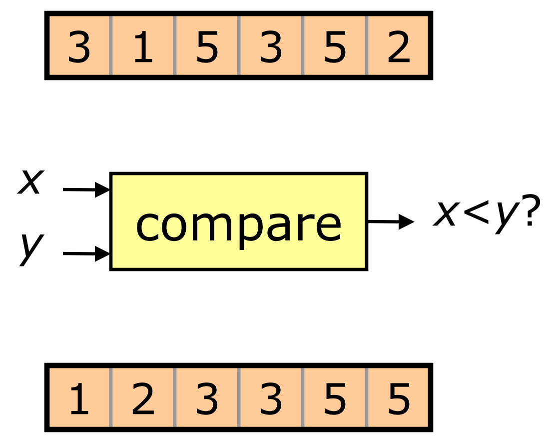
Efficient sorting is of great interest.

- 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 “<”.

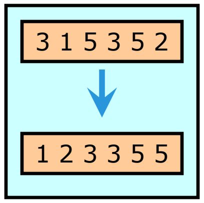


In the next few class meetings, we will analyze various **general-purpose comparison sorts**, in terms of efficiency and other desirable properties.

- 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.

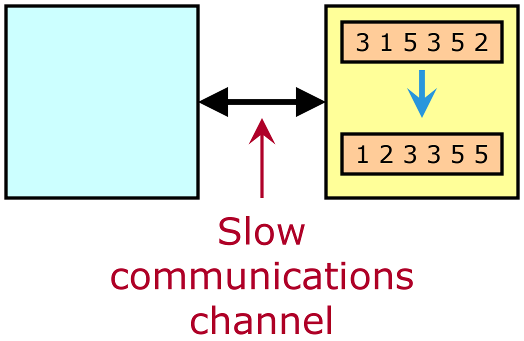
Introduction to Sorting
The Basics — Internal vs. External

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.

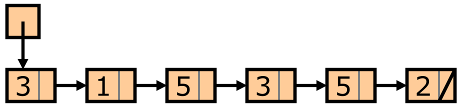


For now, as we look at sorting, we will concentrate on sorting internal data.

- 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.

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**.

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

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