

Introduction to Analysis of Algorithms, cont'd

Introduction to Sorting

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, February 29, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Algorithmic Efficiency & Sorting

Major Topics

- 1/2✓ • Introduction to Analysis of Algorithms
 - Introduction to Sorting
 - Sorting Algorithms
 - More on Big- O
 - The Limits of Sorting
 - Divide-and-Conquer
 - Practical Sorting

Review

Introduction to Analysis of Algorithms — Basics [1/2]

An **efficient** algorithm is one that **uses few resources**.

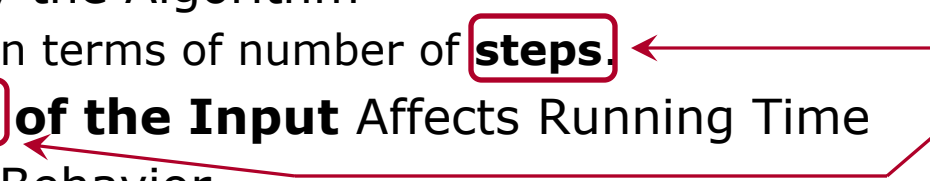
- By far the most important resource is **time**.
- Thus, when we say an algorithm is **efficient** (assuming we do not qualify this further), we mean that it can be executed **quickly**.

We wish to discuss efficiency of algorithms in a way that is independent of their implementations.

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 ever requires for a given input size.

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



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

Our **Model of computation**:

- The following operations will be considered a single **step**:
 - Built-in operations on fundamental types (arithmetic, assignment, comparison, logical, bitwise, pointer, array look-up, etc.).
 - In a template, single operations (operators and function calls) on template-parameter types.
- From now on, 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.
 - The “list” could be an array, a range specified using iterators, etc.
 - We will generally denote the size of the input by “ n ”.

When determining big- O , we can collapse any constant number of steps into a single step.

Review

Introduction to Analysis of Algorithms — Big-O [1/3]

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.

THIS IS IMPORTANT!

Review

Introduction to Analysis of Algorithms — Big-O [2/3]

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.

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



Know
these!

Review

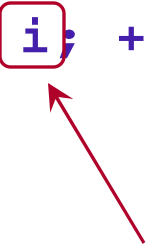
Introduction to Analysis of Algorithms — Big-O [3/3]

Example 3

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!



Review

Introduction to Analysis of Algorithms — Arith. Series

A (finite) **arithmetic series** is a sum in which consecutive pairs of terms all differ by the same amount. For example:

- $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8.$
- $50 + 60 + 70 + 80 + 90 + 100 + 110.$

To find the sum of an arithmetic series:

- Add the first and last terms.
- Multiply by the number of terms.
 - If that is not obvious: $(last - first)/difference + 1.$
- Divide by two.

For example:

- $8 + 11 + 14 + 17 + 20 + 23 + 26 + 29 + 32$
 $= (8+32) \times 9 \div 2$
 $= 180.$
- $1 + 2 + 3 + 4 + \dots + (k-1) + k$
 $= (k+1) \times k \div 2$
 $= (k^2+k)/2.$

Now we can solve Example 3.

Introduction to Analysis of Algorithms, cont'd

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
 $2 + 6 + 10 + \dots + 4(n-1)+2$.
- We can sum this now:
 $[2+4(n-1)+2] \times n \div 2 = 2n^2$.
- The total number of steps for the function as a whole is $2n^2 + 2n + 6$.
- So the function is $O(n^2)$: quadratic time.

Introduction to Analysis of Algorithms, cont'd

Order & Big-O Notation — Rule of Thumb [1/2]

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];
```

- This has order $O(n^3)$.

Introduction to Analysis of Algorithms, cont'd

Order & Big-O Notation — Rule of Thumb [2/2]

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

What is Sorting? [1/2]

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

- “Sort” here is computing jargon. Normal people use “sort” to mean placing things in categories.



Efficient sorting is of great interest.

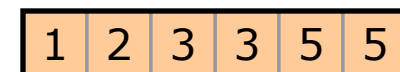
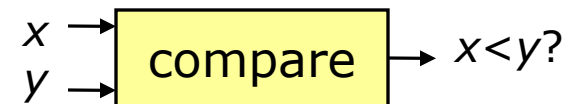
- Sorting is a very common operation.
- Sorting code that is written with little thought is often **much** less efficient than code using a good sorting algorithm.
- Some algorithms (like Binary Search) require sorted data. The efficiency of sorting affects the desirability of such algorithms.

Introduction to Sorting

What is Sorting? [2/2]

We will deal primarily with algorithms that solve the **General Sorting Problem**.

- In this problem, we are given:
 - A sequence.
 - Items are all of the same type.
 - There are no other restrictions on the number of items in the sequence or their values.
 - A comparison function.
 - Given two sequence items, determine which should come first.
 - Using this function is the only way we can make such a determination.
- We return:
 - A sorted sequence with the same items as the original sequence.



In the next few class meetings, we will be analyzing solutions to the General Sorting Problem, in terms of efficiency and other desirable properties.

Introduction to Sorting

A Little About Linked Lists [1/2]

In addition to sorting data in (smart) arrays, we are also interested in sorting **Linked Lists**.

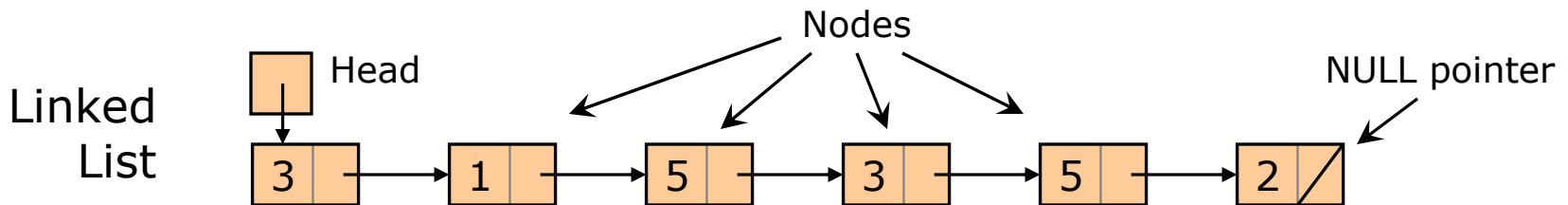
We discuss Linked Lists in detail later in the semester. For now:

- Like an array, a Linked List is a structure for storing a sequence of items.

Array

3	1	5	3	5	2
---	---	---	---	---	---

- A Linked List is composed of **nodes**. Each has a single data item and a pointer to the next node.



- These pointers are the **only** way to find the next data item. Thus, unlike an array, we cannot quickly skip to (say) the 100th item in a Linked List. Nor can we quickly find the previous item.
- A Linked List is a one-way sequential-access data structure. Thus, its natural iterator is a **forward iterator**, which has only the ++ operator.

Introduction to Sorting

A Little About Linked Lists [2/2]

Why not always use (smart) arrays?

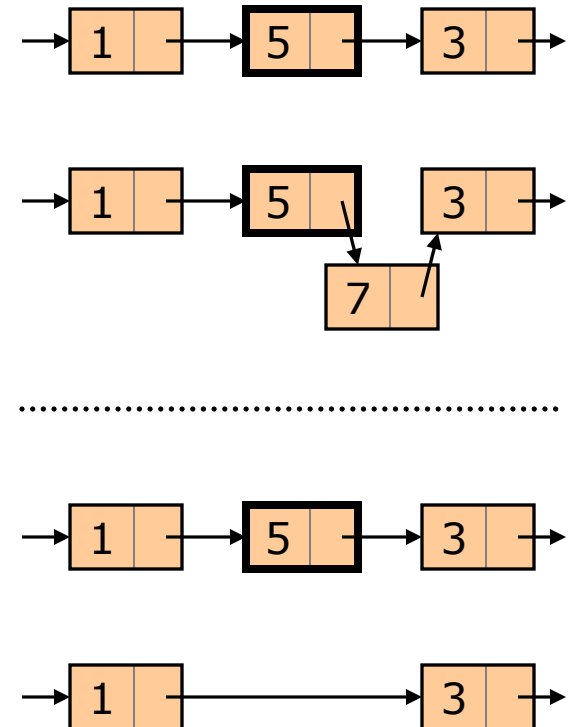
- One important reason: we can often insert and remove much faster with a Linked List.

Inserting

- Inserting an item at a given position in an array is $O(n)$.
- Inserting an item at a given position (think “iterator”) in a Linked List is $O(1)$.
- Example: insert a “7” after the bold node.

Removing

- Removing the item at a given position from an array is $O(n)$.
- Removing the item at a given position from a Linked List is $O(1)$.
 - We need an iterator to the *previous* item.
- Example: Remove the item in the bold node.



Introduction to Sorting, cont'd

Analyzing Sorting Algorithms

We will analyze sorting algorithms according to five criteria:

- Efficiency
 - What is the (worst-case) order of the algorithm?
 - Is the algorithm much faster on average (over all inputs of a given size)?
 - Requirements on Data
 - Does the algorithm require random-access data?
 - Does it work with Linked Lists?
 - Space Usage
 - Can the algorithm sort in-place?
 - Here, **in-place** = no extra buffers holding a large number of data items.
 - How much additional storage (variables, buffers, etc.) is used?
 - 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.
- Items in the list to be sorted**, as opposed to pointers, etc.
- ↓
- “Large”, “close”, “few”:
criterion is whether
some number is at most
a **fixed constant**.

Introduction to Sorting, cont'd

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. Generally, these fall into two categories: $O(n^2)$ and $O(n \log n)$.

- Quadratic [$O(n^2)$] Algorithms
 - Bubble Sort
 - Selection Sort
 - Insertion Sort
 - Quicksort
 - Treesort (later in semester)
- Log-Linear [$O(n \log n)$] Algorithms
 - Merge Sort
 - Heap Sort (mostly later in semester)
 - Introsort (not in text)
- Special-Purpose Algorithm
 - Radix Sort

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

