

Introduction to Analysis of Algorithms

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, February 27, 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

Next we begin a unit on Algorithmic Efficiency and Sorting.

- Topics will include:
 - Introduction to Analysis of Algorithms
 - Introduction to Sorting
 - Sorting Algorithms
 - Lots of them!
 - More on Big- O
 - The Limits of Sorting
 - Divide-and-Conquer
 - Practical Sorting
- We will (mostly) follow the text.
 - Efficiency and sorting are in Chapter 9.

After the Midterm Exam, we will look at Data Abstraction.

- Chapter 3.

Introduction to Analysis of Algorithms

Basics — What is “Efficiency”?

Now we begin looking at the first Big Topic of the course:

- Analysis of Algorithms

What do we mean by an “efficient” algorithm?

- We mean an algorithm 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**.

How do we determine whether an algorithm is efficient?

- Implement it, and run the result on some computer?
- But computers get faster all the time.
- And there are differences in compilers.

Is there some way to measure efficiency that does not depend on the current state of technology?

Introduction to Analysis of Algorithms

Basics — Measuring Efficiency

Is there some way to measure efficiency that does not depend on the current state of technology?

- Yes!

The Idea

- We divide the tasks an algorithm performs into “steps”.
- We determine how many steps are required for input of a given size. Write this as a formula, based on the size of the input.
- Look at the most important part of the formula.
 - For example, the most important part of “ $6n \log n + 1720n + 3n^2 + 14325$ ” is “ n^2 ”.

Next we look at this in more detail.

Introduction to Analysis of Algorithms

Basics — Standard Meaning of “Efficiency”

“Efficiency” is a vague term.

Generally, when we talk about the efficiency of an algorithm, we are interested in:

- **Time** Used by the Algorithm
 - Expressed in terms of number of steps.
 - Sometimes we will also talk about “space efficiency”, etc.
- How the **Size of the Input** Affects Running Time
 - Larger input typically means slower running time. How much slower?
- **Worst-Case** Behavior
 - What is the maximum number of steps the algorithm ever requires for a given input size?
 - Occasionally we also analyze average-case behavior. But then we will need to be clear: average over what?

To make the above ideas precise, we need to say what is meant by a **step**, and how we measure the **size** of the input.

- These two are part of our **model of computation**.

Introduction to Analysis of Algorithms

Basics — Model of Computation

Our **model of computation** will use the following definitions.

- 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 ”.

Notes

- As we will see, we can afford to be *somewhat* imprecise about what constitutes a single “step”.
- In a formal mathematical analysis of the properties and limits of computation, both of the above definitions would need to change.

Introduction to Analysis of Algorithms

Order & Big-O Notation — Definition

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!

Introduction to Analysis of Algorithms

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;
}
```

Introduction to Analysis of Algorithms

Order & Big-O Notation — Example 1, Solution

I count 9 single-step operations in this function (numbers in brackets are the number of times the operation is performed):

- `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: $4n+6$ steps.

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 $k = 5$ and $n_0 = 100$.
- That is, $4n + 6 \leq 5 \times n$, whenever $n \geq 100$.

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.

Introduction to Analysis of Algorithms

Order & Big-O Notation — Why?

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

- We expect more of faster computers.
- Thus, problem sizes keep getting bigger.

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 large problems & large systems is said to be **scalable**.

- Or “it scales well”.
- This class is all about things that scale well.

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.



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.

Introduction to Analysis of Algorithms

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;
}
```

Introduction to Analysis of Algorithms

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];
```

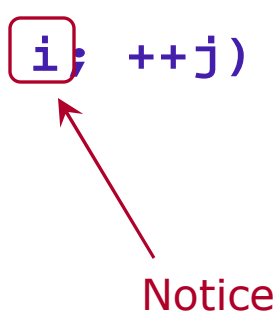
- A single execution of this inside loop requires $3n+2$ steps.
 - If we treat "`sum += p[j];`" as a single step.
- However, the loop itself is executed n times by the outside (i) loop. Thus a total of $n \times (3n+2) = 3n^2+2n$ steps are required.
- The rest of the function takes $2n+6$ steps, for 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 , $3n^2+4n+6 \leq 4n^2$. Thus, `func2` is $O(n^2)$: quadratic time.

Introduction to Analysis of Algorithms

Order & Big-O Notation — Example 3, Problem

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!

Introduction to Analysis of Algorithms

Summing Arithmetic Series [1/3]

A **series** is a sequence of numbers that are added together.

- Each number in a series is called a **term**.

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

- $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$.
- $50 + 60 + 70 + 80 + 90 + 100 + 110$.
- $8 + 11 + 14 + 17 + 20 + 23 + 26 + 29 + 32$.
- $15 + 10 + 5 + 0 + (-5) + (-10) + (-15) + (-20) + (-25)$.
- $7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7$.

Not an arithmetic series:

- $1 + 10 + 100 + 1000 + 10000$.

How can we compute the sum of an arithmetic series efficiently?

Introduction to Analysis of Algorithms

Summing Arithmetic Series [2/3]

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:

- $8 + 11 + 14 + 17 + 20 + 23 + 26 + 29 + 32$
 $= (8+32) \times 9 \div 2$
 $= 180.$
- $7 + 8 + 9 + 10 + \dots + 32045 + 32046$
 $= (7+32046) \times 32040 \div 2$
 $= 513,489,060.$

Introduction to Analysis of Algorithms

Summing Arithmetic Series [3/3]

This technique works with variables, too:

- $1 + 2 + 3 + 4 + \dots + (k-1) + k$
= $(k+1) \times k \div 2$
= $(k^2+k)/2$.

Now we can solve Example 3.

Next time ...