CS484 Module 2 Part B - Math for Analysis of Algorithms, Part 2

Athens State University

January 07, 2014

Outline

Contents

1 Summation

Algorithm analysis: fun with sums

- Computing the time taken in a loop means we have to sum together the cost of all of the operations in the loop
- Have to use all of nice and cute summation identities we hope you were taught in discrete math

Sums and Loops

- So...
$$\sum_{k=0}^{n-1}(n) = n \approx O(n)$$
- And
$$\sum_{k=1}^{n}(k) = \frac{n(n+1)}{2} \approx O(n^2)$$

This explains why you are taught in Data Structures to count the nesting level of a set of nested loops to determine the "Big-Oh" of a program or function. It's a useful trick to remember.

And recursion

Geometric sums

$$\sum_{k=0}^{n}(a^k) = \frac{1-a^{n+1}}{1-a}$$

Binary representation $1111\ldots1_2 = \sum_{k=0}^{n}(2^k) = 2^{n+1}-1$

Useful for recursive algorithms

A few other useful summation formulas

Sum of squares

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Factorial sum

$$\sum_{k=0}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$

2 Logarithms

Logarithms?

Logarithm inverse exponential function

Recall that $b^x = y$ is equivalent to $x = log_b y$

And that $b^{log_b y} = y$

Logarithms: special cases

- *Binary logarithm* : $log_2(n) = lg(n)$
- *Natural logarithm*
  - $e \approx 2.71828$
  - Seen in a lot of calculus related stuff
- *Common logarithms* : logarithms with base $b = 10$
  - Deals with the way that we count
  - Not as important for our work as the other two

Recall what the "binary" stands for in a binary tree. So most of the algorithms that we regularly use will have some connection back to binary logarithms in their "Big-Oh".

Logarithms: useful properties

- $log_a(xy) = log_a(x) + log_a(y)$
- $log_a(b) = \frac{log_c(b)}{log_c(a)}$

Two important implications:

- The base of a logarithm has no real impact on the growth rate
- Logarithms cut any function down to size
  - $log_a(n^b) = b * log_a(n)$

Logarithms: binary search and trees

We spent some serious time looking at binary search in Data Structures.

- Binary Search is a $O(lg(n))$ algorithm (Why?)
- A binary tree of height 1 can have 2 leaf nodes.

What is the height h of a rooted binary tree with n leaf nodes?

- For n leaves, $n = 2^h$
- This implies that $h = log_2(n) = lg(n)$

Logarithms, multiplication, and exponentiation

- Logarithms were first developed as tool to simplify multiplication of numbers
- Important identity: $log_a(n^b) = b * log_a n$
- This gives us a cute way to compute a^b on our fancy TI-88 calculator:

$$a^b = exp(ln(a^b)) = exp(b * \ln(a)) \tag{1}$$

Logarithms, multiplication, and exponentiation

The equation for a^n on the previous slide has problems with dealing with precision of real numbers.
So, do we have to fall back to doing $n - 1$ multiplies?
Actually, no... Recall that $n = \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil$

Logarithms, multiplication, and exponentiation

OK... note what we can now do to compute a^n
If n is even, then $a^n = (a^{n/2})^2$
If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$

An upcoming homework assignment will have you use these facts to develop and analyze a recursive algorithm to quickly compute a^n. (HINT: think logarithms when you analyze this)

Logarithms, summations, and Harmonic numbers

The Harmonic numbers are a special case of an arithmetic progression:

$$H(n) = \sum_{i=1}^{\infty} \frac{1}{i} \approx ln(n)$$

This is important because seeing this series appear in analysis of an algorithm leads one to conclude that an algorithm is $O(n * lg(n))$

3 Working with recurrence relations

Recurrence relations: concept

Recurrence relations are recursive definitions of mathematical functions or sequences
For example, the recurrence relation for $f(n) = n^2$ is

$$g(0) = 0$$
$$g(n) = g(n-1) + 2n - 1$$

Recurrence relations: concept

Many sequences of numbers we use in computer science can be generated using a recurrence relation
For example, the Fibanocci sequence $< 1, 1, 2, 3, 5, 8, 13, \ldots >$ is generated by the recurrence relation:

$$
\begin{aligned}
&f(0) = 1 \\
&f(1) = 1 \\
&f(n) = f(n-1) + f(n-2)
\end{aligned}
$$

Recurrence relations: closed form

Closed form an equivalent definition without the recursion

Finding the closed form is described as "solving" the recurrence relation

Two most used methods for solving recurrence relations

- Iteration (expansion) method
- Master Theorem method

Recurrence relation: closed form: iteration method

$$
\begin{aligned}
g(n) &= g(n-1) + 2n - 1 \\
&= [g(n-2) + 2(n-1) - 1] + 2n - 1 \\
&= g(n-2) + 2(n-1) + 2n - 2 \\
&= [g(n-3) + 2(n-2) - 1] + 2(n-1) + 2n - 2 \\
\ldots \\
&= g(n-i) + 2(n-i+1) + \ldots + 2n - i \\
\ldots \\
&= g(n-n) + 2(n-n+1) + \ldots + 2n - n \\
&= 0 + 2 + 4 + \ldots + 2n - n \\
&= n^2
\end{aligned}
$$

Recurrence relations: relationship to algorithms

Suppose we use the recurrence relation for the Fibonacci sequence to build an algorithm that generates the sequence

Algorithm 1: Recursive Algorithm for generating the Fibonacci sequence

Input: An integer n
Output: The Fib-sequence up to n
if *n is 0 or 1* **then**
  return 1;
else
  return `FibRec` (n-1) + `FibRec` (n-2);

Recurrence relations: relationship to algorithms

Contrast this against an algorithm built using the closed form:

Algorithm 2: Iterative Algorithm for generating the Fibonacci sequence

---

Input: An integer n
Output: The Fib-sequence up to n
F [0] ← F [1] ← 1;
for i *from 2 to n* **do**
  F [i] ← F [i-1] + F [i-2];

Recurrence relations: relationship to algorithms

- Analysis of the recursive solution will show the time complexity of the algorithm is exactly the same as Fib-sequence recurrence relation
- Finding the closed form (using some stuff you learn in discrete math) shows this is a $\Theta(c^n)$ algorithm with $c \approx 1.5$
- However, the iterative solution is $O(n)$!

4 Key Points

Key Points

- Analysis of algorithms is one of those places where you get to use all the fancy stuff you use in discrete math
- Fun with sums
- And with logarithms
- And recursion (and recurrence relations) is your friend.