CS417 Module 2 Part A - Math for Analysis of Algorithms, Part 1

Athens State University

January 23, 2016

Outline

Contents

1 Introduction and overview

What should one ask about an algorithm?

- Is the algorithm correct?
- How good is the algorithm?
  - time efficiency
  - space efficiency
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

Analysis of Algorithms

- How does one compare two algorithms?
- Need to predict the resources an algorithm requires
- The typical metric is algorithm running time

2 Analysis of Algorithms

Random-access machine (RAM) model

Predicting resource requirements requires a way to model computation
Random-access machine (RAM) model

- Instructions are executed one after another
- Use the basic operations of a digital computer
  - Arithmetic
    - Add, subtract, multiply, divide, remainder, bit shifts
  - Data movement
    - Load, store, copy
  - Control
    - Conditonal/unconditional branch, subroutine call/return

Random-access machine (RAM) model

- Each instruction takes a constant amount of time
- Loops and subroutines are NOT considered basic operations.
  - They are the composition of many single-step operations.
- Unconcerned about precision of data
- Limit on word size
  - If input of size n, assume integers are represented by $c * log(n)$ bits for some constant $c \geq 1$
  - Note that c is a constant implies word size cannot grow arbitrarily

Definition of time efficiency

Time efficiency measured by determining the number of repetitions of *basic operations* as a function of *input size*

$$T(n) \approx c_{op} * C(n)$$

$T(n)$ running time

c_{op} execution time for a basic operation

$C(n)$ Number of times basic operation is executed

n The size of the problem input

Best, Worst, and Average-Case Efficiency

Using the RAM model, we can count how many steps an algorithm takes on any given input instance by executing it.

But we want to know how an algorithm behaves over *all* instances

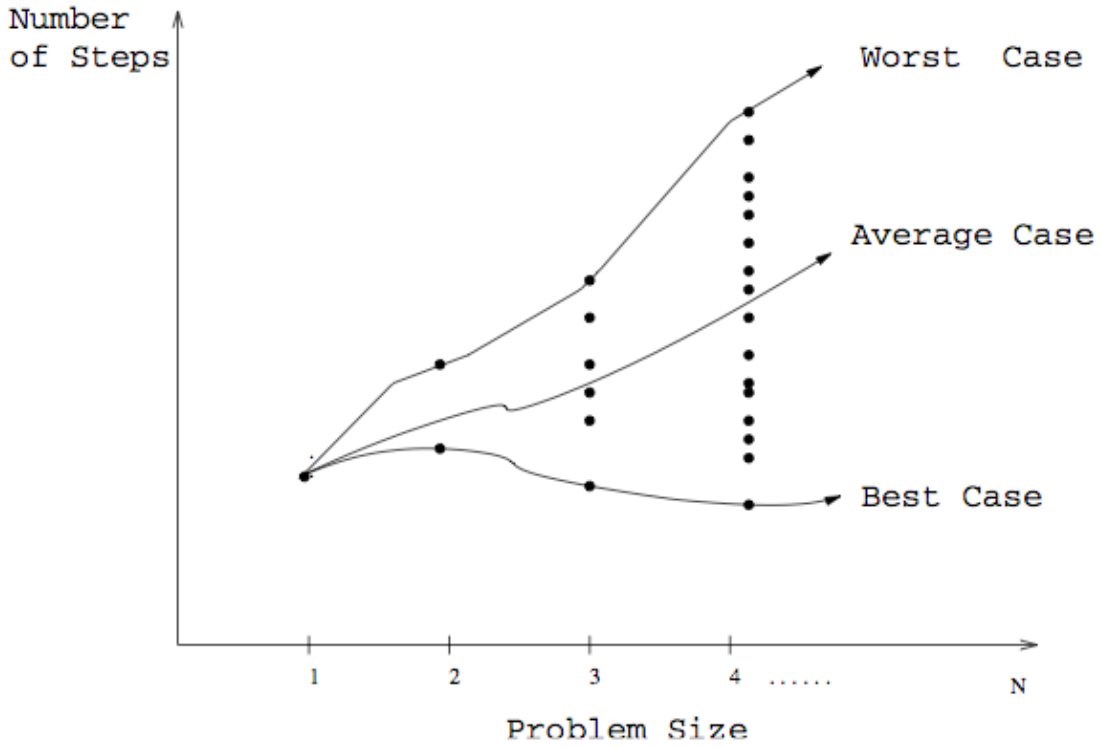
Definition of best, worst, and average-case

Worst-case efficiency maximum number of steps taken in any instance of size n

Best-case complexity minimum number of steps taken in any instance of size n

Average-case complexity average number of steps over all instances of size n

Best, Worst, and Average-case Efficiency



Best, Worst, and Average-case Efficiency

Each of these $T(n)$ functions are functions over the size of possible problem instances.
But... these functions can be hard to work with

- They have some bumps
  - Example: binary search will run better if $n = 2^k - 1$
- Require too much detail to specify precisely
  - What's the time complexity of Microsoft Word?

The Big Oh Notation

- $f(n) \in O(g(n))$
  - $c * g(n)$ is an upper bound on $f(n)$
  - Thus, there is some constant c s.t. $f(n) \leq c * g(n)$ for $n \geq n_0$
- $f(n) \in \Omega(g(n))$
  - $c * g(n)$ is a lower bound on $f(n)$.
  - Thus, there is some constant c s.t. $f(n) \geq c * g(n)$ for all $n \geq n_0$
- $f(n) \in \Theta(g(n))$
  - $c_1 * g(n)$ is an upper bound on $f(n)$ and $c_2 * g(n)$ is a lower bound on $f(n)$ for all $n \geq n_0$

This notation characterizes functions according to their growth rates: different functions with the same growth rate are grouped into families using the same notation. Why do we say "Big-O"? The letter 'O' is used because the growth rate of a function is known to mathematicians as the *order* of the function.

From a mathematical standpoint, this notation is used as a measure of *infinite asymptotics*. From a practical computer science standpoint, this means that the as n, the problem size, goes toward infinity, the functions $T(n)$ will go towards the corresponding $O(n)$. As a quick math aside, this notation is also used to estimate error in fields such as real analysis and numerical analysis. In that case, the notation is a measure of *infinitesimal asymptotics*

The Big Oh Notation

Note the use of the constant c in the definition... it is a constant multiple as $n \to \infty$

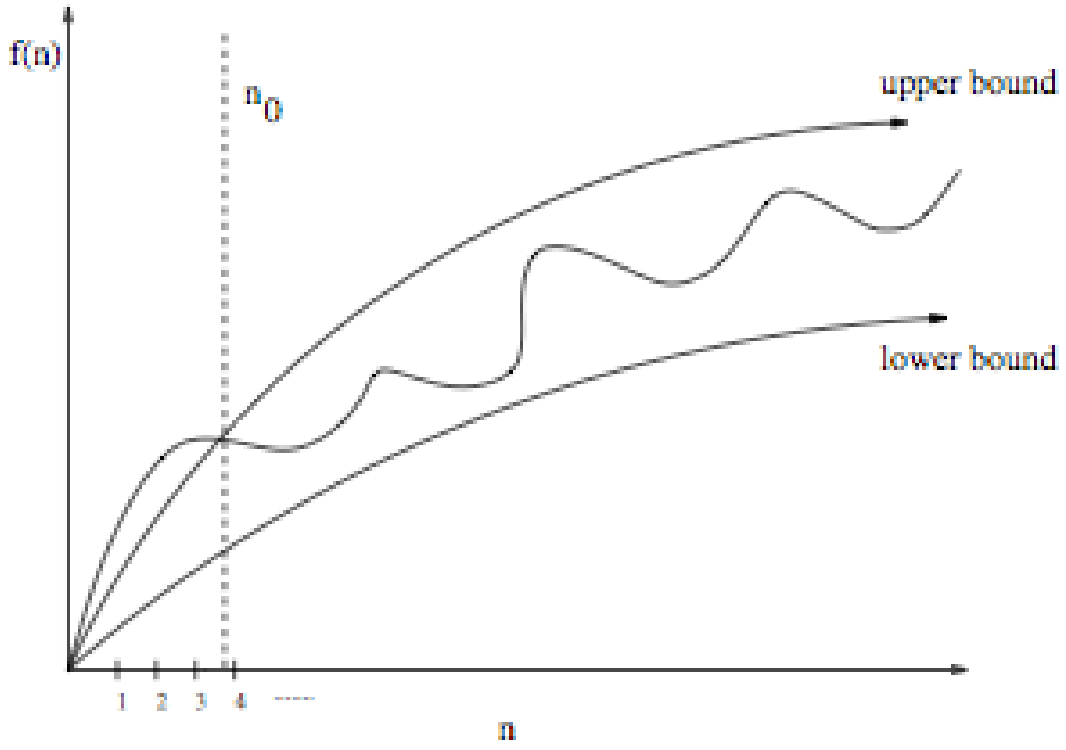
Think about these questions:

- How much faster will an $O(n^2)$ algorithm run on a computer that is twice as fast?
- How much longer does this algorithm take if the input size doubles?

The implication in this case is that as problem size increases, the coefficients in the definitions become irrelevant. It doesn't matter if two algorithms have performance of $T(n) = n^3$ or $T(n) = 100000 * n^3$, both algorithms are $O(n^3)$ and are considered equivalent from a performance standpoint.

So for our two thought questions, one needs to consider how changes in hardware capabilities and problem size are accounted for in these schemes.

The Big Oh Notation



The Big Oh Notation

Example 1 ($O(n)$). $3n^2 - 100n + 6 = O(n^2)$

- Why? $c = 3$ and $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 \neq O(n)$

- Why? for any value of c, $cn < 3n^2$ when $n > c$

Example 2 (What about $\Omega(n)$ and $\Theta(n)$?). $3n^2 - 100n + 6 = \Omega(n^2)$ Why?
$3n^2 - 100n + 6 \neq \Omega(n^3)$ Why?
$3n^2 - 100n + 6 = \Omega(n)$ Why?

Look at the how the examples work. In each case, one seeks an example of either c or n_0 (or both), such that conditions of the definition of specific type of order function is satisfied.

So, how to address the issues about $\Omega(n)$ and $\Theta(n)$? Look for the values of c and n_0 satisfy the respective definitions.

The Big Oh Notation

Your turn: Is $2^{n+1} = \theta(2^n)$
Why or why not?

The hint in this case is to apply some algebra to the expression 2^{n+1}. In particular, what identities, if any, apply to exponents expressed as a sum?

The Big Oh Notation: what does it all mean?

This notation gives us a way to compare functions that ignores constant factors and small input sizes:

$O(g(n))$ class of functions $f(n)$ that grows *no faster* than $g(n)$

$\Theta(g(n))$ class of functions $f(n)$ that grows *at the same rate* as $g(n)$

$\Omega(g(n))$ class of functions $f(n)$ that grows *at least as fast* as $g(n)$

The Big Oh Notation: some useful identities

- Addition
  - $O(f(n)) + O(g(n) \rightarrow O(max(f(n), g(n)))$
  - $\Omega(f(n)) + \Omega(g(n)) \rightarrow \Omega(max(f(n), g(n)))$
  - $\Theta(f(n)) + \Theta(g(n)) \rightarrow \Theta(max(f(n), g(n)))$

The Big Oh Notation: some useful identities

- Multiplication
  - $O(c * f(n)) \rightarrow O(f(n))$
  - $O(f(n)) * O(g(n)) \rightarrow O(f(n) * g(n))$

These identities make it easier for us to analyze algorithm performance. We can make an estimate of the $T(n)$ and then see if we can apply any of these identities to decompose that expression.

The Big Oh Notation: good news!

Constant functions $O(1)$

- Algorithms not dependent upon the parameter n

Logarithmic functions $O(logn)$

- Binary search and some graph algorithms

Linear functions $O(n)$

- Looking at each item once, such as linear search

Superlinear functions $O(n * lg(n))$

- Sorting algorithms such as Quicksort and Mergesort

The Big Oh Notation: good news!

Quadratic functions $O(n^2)$

- Looking at all pairs of items, insertion or selection sort

Cubic functions $O(n^3)$

- Finding triples of items in n-element universe

Exponential functions $O(c^n)$ for $c > 1$

- Enumerating all subsets of n items

Factorial functions $O(n!)$

- Permutations or orderings of n items

The result is that many useful algorithms fit into a small set of order functions. The downside is that the order of these algorithms fall into categories that are not polite in their behavior. Consider, for example, how quickly functions that are exponential or factorial in nature will grow.

On the upside, consider the relationship between linear and superlinear order functions. The logarithmic term in superlinear order functions means that we have an algorithm that is trying its best to be linear in nature. This means we get a much better performance than we get with quadratic, cubic, or larger exponential functions. This is a good thing!

n	lg n	n	n lg n	n^2	2^n	n!
10	0.003 us	0.01 us	0.033 us	0.1 us	1 us	3.363 ms
20	0.004 us	0.02 us	0.086 us	0.4 us	1ms	77.1y
30	0.005 us	0.03 us	0.147 us	0.9 us	1s	8.4x10^15y
40	0.005 us	0.04 us	0.213 us	1.6 us	18.3m	
50	0.006 us	0.05 us	0.282 us	2.5 us	13d	
100	0.007 us	0.10 us	0.644 us	10 us	4x10^13y	
1x10^3	0.010 us	1.00 us	9.966 us	1 ms		
1x10^4	0.013 us	10 us	130 us	100 ms		
1x10^5	0.017 us	0.10 ms	1.67 ms	10s		
1x10^6	0.020 us	1 ms	19.93 ms	16.7m		
1x10^7	0.023 us	0.01 s	0.23 s	1.16d		
1x10^8	0.027 us	0.10 s	2.66 s	115.7d		
1x10^9	0.030 us	1 s	29.9 s	31.7y		

The Big Oh Notation: practical observations

Note what the table tells us about working with $O(n)$

- All such algorithms take roughly the same time for $n = 10$
- Any algorithm that is $O(n!)$ becomes useless for $n \geq 20$
- Algorithms that are $O(2^n)$ become impractical for $n > 40$
- Algorithms that are $O(n^2)$ remain usable up to about $n = 10000$ but quickly deteriorate with larger inputs
- Linear ($O(n)$) and log-linear ($O(n * lg(n))$) algorithms remain practical for large input sizes
- Linear is best, but constant is even better.

3 Analysis of Algorithms: Using Big Oh

General Plan for Analyzing Non-recursive Algorithms

- Determine what algorithm parameter defines *input size* n
- Identify the algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size n
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum as much as possible using the tools you were taught in Discrete Math

Let's revisit the Insertion sort algorithm

Algorithm 1: Insertion Sort

```
Input: A set s of items of length n
Output: The set s in sorted order
for i ← 1 ... n do
    j ← i;
    while j > 0 and s (j) < s (j-1) do
        swap the j-th and j-1st elements in s;
        j ← j- 1;
```

Analyzing Insertion Sort

Let $T(n)$ be the running time of the algorithm and let t_j be the number of times that the *while* loop test is executed for a value j. Don't forget that when a loop runs, the test is executed one more time than the loop body.

The running time of the algorithm will be

$$\begin{aligned} T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ + c_6 \sum_{j=2}^{n} t_j - 1 \\ + c_7 \sum_{j=2}^{n} t_j - 1 \\ + c_8(n-1) \end{aligned}$$

where c_i is the time required to execute statement i

Analyzing insertion sort: best case

The array is already sorted.

What happens when we try to execute the *while* loop?

All t_j are 1.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_1 + c_2 + c_4 + c_5 + c_8) \end{aligned}$$

In this case, insertion sort is $O(n)$.

Analyzing insertion sort: worst case

The array is reverse sorted order.

- At position j, have to compare with the $j-1$ elements to the left.
- Get one additional test after $j-1$ tests

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$$

$$\sum_{j=2}^{n} t_j - 1 = \sum_{j=2}^{n} j - 1$$

Analyzing insertion sort: worst case

- Note that we have *arithmetic series* in both cases in the previous slides

$$\begin{aligned} \sum_{j=2}^{n} j &= (\sum_{j=1}^{n} j) - 1 \\ &= \frac{n(n+1)}{2} - 1 \end{aligned}$$

- Now let $k = j - 1$

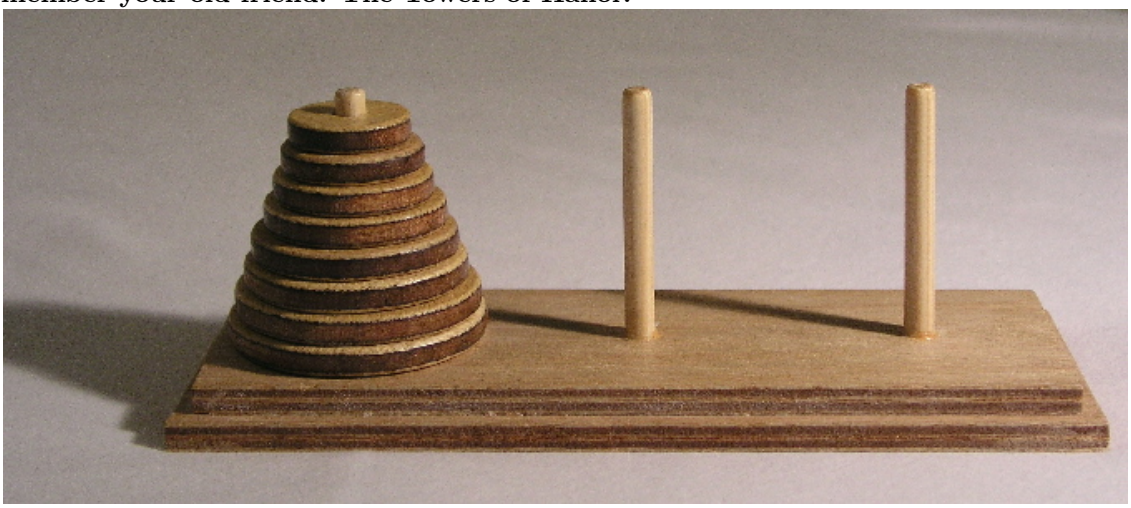
$$\begin{aligned} \sum_{j=2}^{n} (j-1) &= \sum_{k=1}^{n-1} k \\ &= \frac{n(n-1)}{2} \end{aligned}$$

- Do the algebra and you see that the worse case is $O(n^2)$

General Plan for Analyzing Recursive Algorithms

1. Determine what algorithm parameter defines *input size* n
2. Identify the algorithm's *basic operation*
3. Check if the number of times the basic operation is executed can vary on different inputs of the same size
   - If so, then one must analyze worst, average, and best cases separately
4. Define a recurrence relation with an appropriate initial condition that expresses the number of times the basic operation is executed
5. Solve the recurrence relations (or at minimum, determine the relation's Big Oh)

Remember your old friend: The Towers of Hanoi?



A recursive algorithm for the Towers of Hanoi

Algorithm 2: Recursive Towers of Hanoi

Input: A disk, source, destination, and tower
Output: All disks on the destination tower
if disk = *0* **then**
    Move disk from source to dest;
else
    `MoveTower` (disk- 1, source, spare, dest);
    Move disk from source to dest;
    `MoveTower` (disk- 1, spare, dest, source);

Analyzing the Towers of Hanoi

Let $M(n)$ be the number of moves our algorithm takes for n-disk tower.
Base case: $n = 1$ implies that $M(1) = 1$. Why?
Recurrence Relation: $M(n) = 2M(n-1) + 1$

- First move the $(n-1)$-disk tower to the spare
- Move the n-th disk
- Move the $(n-1)$-disk tower on top of the n-th disk

Analyzing the Towers of Hanoi

$$
\begin{aligned}
M(1) &= 1 \\
M(2) &= 2M(1) + 1 = 3 \\
M(3) &= 2M(2) + 1 = 7 \\
M(4) &= 2M(3) + 1 = 15 \\
M(5) &= 2M(4) + 1 = 31
\end{aligned}
$$

- OK, so we can guess that $M(n) = 2^n - 1$
- So, now what we know about Professor Rimes's "Towers of Hanoi" assignment in CS372?

4 Key Points

Key Points

- The RAM Model of Computation
- Time efficiency
  - Definition
  - Best, Worst, and Average-Case Efficiency
  - The Big Oh Notation
- Analysis of Algorithms using Big Oh
  - Non-recursive algorithms
  - Recursive algorithms