Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Evaluation

Programming Assignments	40%
In-class quizzes	15%
Exams	40%
Class Participation	5%

- Course Website: https://users.cs.fiu.edu/~giri/teach/3530Fall2016.html

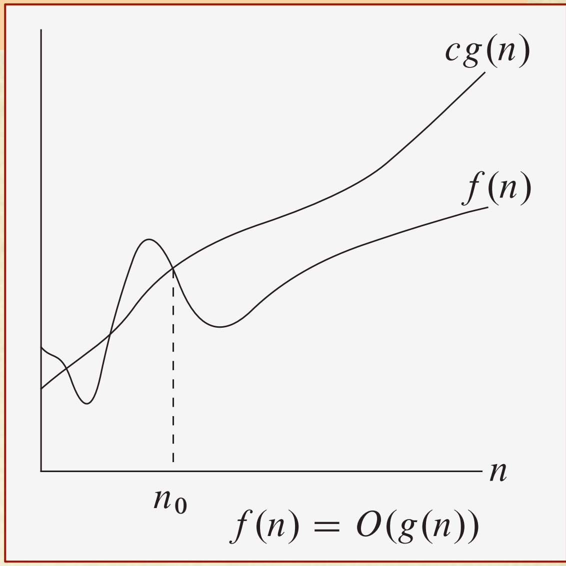
Advantages of Asymptotic Analysis & Big-Oh Notation

- Allows for rough measure of running time
- Abstracts main features of code without focusing on details of implementation or hardware or language or environment
- Tells us how time complexity scales with input size
- Allows for a quick high-level comparison of algorithms

Asymptotic Running Time

- To compute **asymptotic running time**,
  - Consider the worst-case scenario & count number of steps as a function of length of input
  - Write down an expression for the running time for the worst-case input
  - Eliminate all terms except the dominant term(s)
  - Eliminate constants where possible
  - Simplify expression where possible
  - What remains is an upper bound on the asymptotic running time using **big-Oh notation**

How to compare big-Ohs?



- We say that f(n) = O(g(n)) if
  - There exists positive integer n_0, and
  - A positive constant c, such that
  - For all $n > n_0$,

    $f(n) <= c\ g(n)$
- In other words, for all large enough values of n, there exists a constant c, such that
  - cf(n) is always bounded from above by g(n)
- Note that the condition may be violated for a finitely many small values of n

Examples

- Let $f(n) = 1000n^2$
- Let $g(n) = 0.001n^2$
- Let $h(n) = 1000n^2 + 10^6$
- What are the relationships between f(n), g(n) and h(n)?
- Can you say that:
  - $f(n) = O(g(n))$?
  - Or vice versa?

- $g(n) = O(f(n)) = O(h(n))$
  - $n_0 = 1$ and $c = 1$;
- $f(n) = O(g(n))$
  - $n_0 = 1$ and $c = 10^6$;
- $f(n) = O(h(n))$
  - $n_0 = 1$ and $c = 1$;
- $h(n) = O(f(n))$
  - $n_0 = \sqrt{1000}$ and $c = 2$;
  - $n_0 = 10$ and $c = 11$;
- $h(n) = O(g(n))$
  - $n_0 = \sqrt{1000}$ and $c = 2 \times 10^6$;

Facts about proving big-Oh

- $O(n) = O(2n)$
  - Because $n = O(n) = O(2n)$
  - **Ignore** multiplicative constants
- $n^2 + 6n - 9 = O(n^2)$
  - Try $n_0 = 3$ and $c = 2$
  - **Ignore** additive lower order terms; only dominant term left
- $n = O(n^2)$
  - Try $n_0 = 1$ and $c = 1$
  - Gives **loose upper bounds**

More facts about big-Oh

- If $f(n) = O(g(n))$, it may not be true that $g(n) = O(f(n))$
  - ASYMMETRY
  - Can you come up with examples?
- It is possible for $f(n) = O(g(n))$ as well as $g(n) = O(f(n))$
  - Can you come up with examples?
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
  - TRANSITIVITY
  - Can you come up with examples?
- Do you think: $f(n) + g(n) = O(\max(f(n), g(n)))$?
- Do you think: $\mathrm{Max}(f(n), g(n)) = O(f(n) + g(n))$?

How to prove "not" big-Oh

- $n^2 \neq O(2n)$
  - Prove by contradiction
  - Assume it is true.
  - Then, by definition, there exists positive n_0 and c such that
    - $n^2 \leq 2cn$, for all $n > n_0$
    - $n \leq 2c$, for all $n > n_0$
  - But, this is impossible if we choose $n > \max\{2c, n_0\}$
  - Hence, the contradiction!
  - Thus, our assumption has to be incorrect.

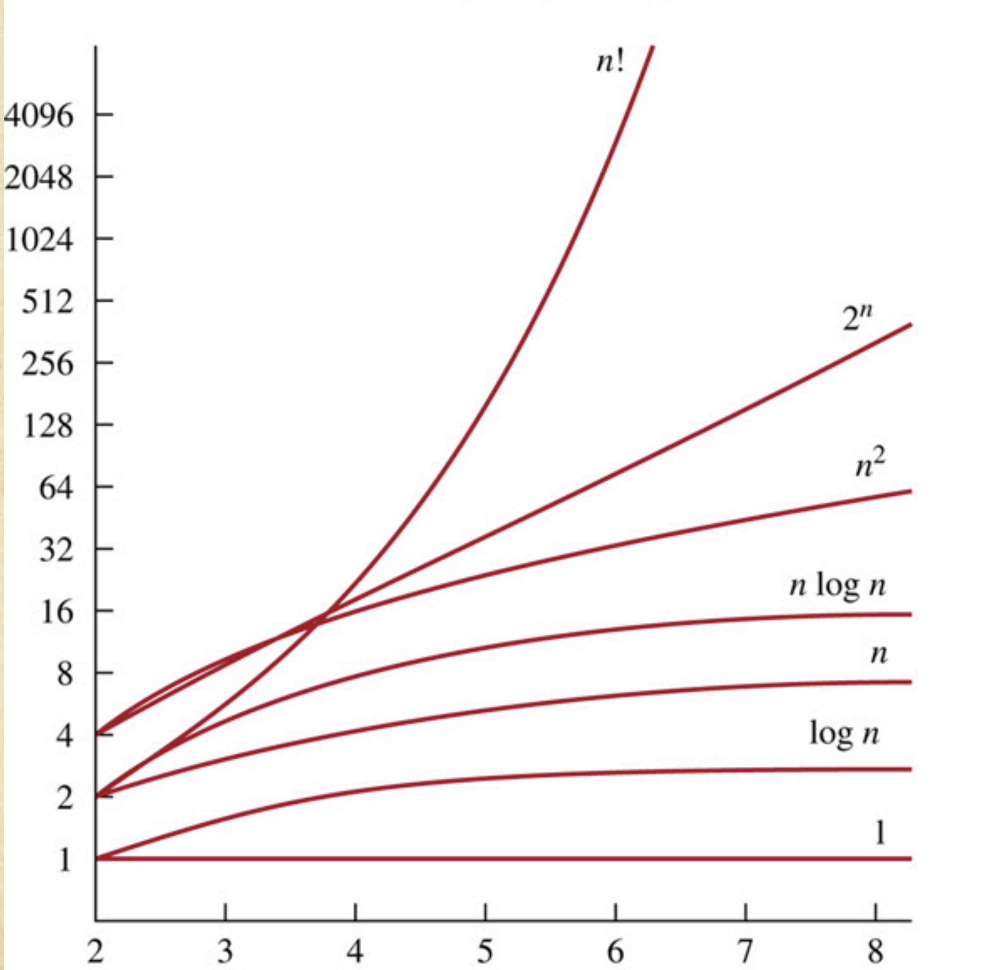
Other useful relationships

- We say that $f(n) = O(g(n))$ if
  - There exists positive n_0 and c, such that
  - For all $n \geq n_0$, $f(n) \leq c\, g(n)$
- We say that $f(n) = \Omega(g(n))$ if
  - There exists positive n_0 and c, such that
  - For all $n \geq n_0$, $f(n) \geq c\, g(n)$
- We say that $f(n) = \Theta(g(n))$ if
  - $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- We say that $f(n) = o(g(n))$ if
  - $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$

Summary

- To prove $f(n) = O(g(n))$
  1. Pick a value of c
  2. Find a n_0 such that
  3. $f(n) \leq c\, g(n)$ for all $n > n_0$
  4. If not, go back to step 1 and refine value of c
- To prove $f(n) \neq O(g(n))$
  - Assume that c is fixed to some positive value
  - And assume there exists a n_0 such that
  - $f(n) \leq c\, g(n)$ for all $n > n_0$
  - Now try to prove a contradiction to this claim

Find the relationships here?



1	constant
log *n*	logarithmic
n	linear
n log *n*	*n*-log-*n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

What's the flaw here?

- **WRONG CLAIM**: $n^2 = O(n)$, because
  - choose $c = n$, and $n_0 = 1$,
  - It is easy to see that $n^2 \leq c\,n$ for all $n > n_0$
- What's wrong with this?
- **FLAW**: c must be a positive constant; it cannot depend on n

Ignoring constants in big-Oh

- **WRONG CLAIM**: $e^{3n} = O(e^n)$ because constant factors can be ignored
- What's wrong with this?
- **FLAW**: A constant factor in an exponent is not the same as a constant factor in front of a term
  - It is not true that
    - $e^{3n} \le c\, e^n$
    - In fact, $e^{3n} = e^n e^{2n}$
    - Another way to look at it: $e^{3n} = (e^n)^3$
    - e^{2n} is much bigger than e^n !!!

Time Complexities

- Sequence of Statements

  statement 1;

  statement 2;

  ...

  statement k;

- *total time* = sum of times for all statements:

T(n) = time(statement 1) + time(statement 2) + ... + time(statement k)

- If each statement is "simple" (only involves *basic* operations) then the time for each statement is constant and the total time is also *constant: O(1)*.

Time Complexities … 2

- Loops
  - The running time of a loop is, at most, the running time of the statements inside the loop x the # of iterations

```
//executes n times
For i = 1 to n do
  m = m + 2;  // constant time
```

Total time T(n) = constant c x n = cn = O(n)

Time Complexities … 3

- **Nested Loops**
  - Analyze from the *inside out*. Total running time is the product of the size of the loops

```
//outer loop executes n times
For i = 1 to n do
    //inner loop executes n times
    For i = 1 to n do
            k = j + 1;  // constant time
```

Total time $T(n) = c \times n \times n = cn^2 = O(n^2)$

Challenging Cases

MaxSubseqSum(A)

Initialize maxSum to 0

N := size(A)

For i = 1 to N do

    For j = i to N do

        Initialize thisSum to 0

        for k = i to j do

            add A[k] to thisSum

        if (thisSum > maxSum) then

            update maxSum

$$\sum_{k=i}^{j} 1 = j - i + 1$$

$$\sum_{j=i}^{N} (j - i + 1) = \frac{(N-i+1)(N-i+2)}{2}$$

$$\sum_{i=1}^{N} \frac{(N-i+1)(N-i+2)}{2}$$

$$= \sum_{i=1}^{N} \frac{i^2}{2} - (N + \frac{3}{2}) \sum_{i=1}^{N} i + \frac{1}{2}(N^2 + 3N + 2) \sum_{i=1}^{N} 1$$

$$= \frac{N^3 + 3N^2 + 2N}{6} = O(N^3)$$

Challenging Case … 2

BINARYSEARCH(A, key, low, high)

If (low > high) return not found

mid = (low + high)/2

If A[mid] = key then return mid

If A[mid] > key then

    BinarySearch(A, key, low, mid-1)

Else

    BinarySearch(A, key, mid+1, high)

- On each recursive call, high-low+1 is halved
- How many times do you have to halve N before it becomes smaller than 1?
- Answer ≈ $\log_2 N$ Why?