Algorithm Analysis

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Order Analysis

- Judging the Efficiency/Speed of an Algorithm
  - Thus far, we've looked at a few different algorithms:
    - Max # of 1's
    - Linear Search vs Binary Search
    - Sorted List Matching Problem
    - and others
  - But we haven't really examined them, in detail, regarding their efficiency or speed
  - **This is one of the main goals of this class!**

Order Analysis

- Judging the Efficiency/Speed of an Algorithm
  - We will use Order Notation to approximate two things about algorithms:
    1) How much time they take
    2) How much memory (space) they use
  - Note:
    - It is nearly impossible to figure out the exact amount of time an algorithm will take
    - Each algorithm gets translated into smaller and smaller machine instructions
    - Each of these instructions take various amounts of time to execute on different computers

Order Analysis

- Judging the Efficiency/Speed of an Algorithm
  - Note:
    - Also, we want to judge algorithms independent of their implementation
    - Thus, rather than figure out an algorithm's exact running time
      - We only want an approximation (Big-O approximation)
    - Assumptions: we assume that each statement and each comparison in C takes some constant amount of time
    - Also, most algorithms have some type of input
      - With sorting, for example, the size of the input (typically referred to as n) is the number of numbers to be sorted
      - Time and space used by an algorithm function of the input

Big-O Notation

- What is Big O?
  - Sounds like a rapper.?.
    - If it were only that simple!
  - Big O comes from Big-O Notation
    - In C.S., we want to know how efficient an algorithm is…how "fast" it is
    - More specifically…we want to know **how the performance of an algorithm responds to changes in problem size**

Big-O Notation

- What is Big O?
  - The goal is to provide a *qualitative* insight on the # of operations for a problem size of *n* elements.
  - And this total # of operations can be described with a mathematical expression in terms of *n*.
    - This expression is known as Big-O
  - The Big-O notation is a way of measuring the order of magnitude of a mathematical expression.
  - O(n) means “of the order of n”

Big-O Notation

- Consider the expression:
  - $f(n) = 4n^2 + 3n + 10$
  - How fast is this "growing"?
    - There are three terms:
      - the $4n^2$, the $3n$, and the 10
    - As n gets bigger, which term makes it get larger fastest?
      - Let's look at some values of n and see what happens?

n	$4n^2$	3n	10
1	4	3	10
10	400	30	10
100	40,000	300	10
1000	4,000,000	3,000	10
10,000	400,000,000	30,000	10
100,000	40,000,000,000	300,000	10
1,000,000	4,000,000,000,000	3,000,000	10

Big-O Notation

- Consider the expression:
  - $f(n) = 4n^2 + 3n + 10$
  - How fast is this "growing"?
    - Which term makes it get larger fastest?
      - As n gets larger and larger, the $4n^2$ term DOMINATES the resulting answer
      - f(1,000,000) = 4,000,003,000,010
  - The idea of behind Big-O is to <u>reduce the expression</u> so that it <u>captures the **qualitative**</u> <u>behavior</u> in the <u>**simplest terms.**</u>

Big-O Notation

- Consider the expression: $f(n) = 4n^2 + 3n + 10$
  - How fast is this "growing"?
    - Look at VERY large values of n
      - eliminate any term <u>whose contribution</u> to the total <u>ceases to be significant as n get larger and larger</u>
      - of course, this <u>also includes constants</u>, as they little to no effect with larger values of n
        - Including constant factors (coefficients)
      - So we ignore the constant 10
      - And we can also ignore the 3n
      - Finally, we can eliminate the constant factor, 4, in front of n^2
    - We can approximate <u>the order of this function</u>, f(n), <u>**as n^2**</u>
    - We can say, $O(4n^2 + 3n + 10) = O(n^2)$
      - In conclusion, we say that f(n) takes $O(n^2)$ steps to execute

Big-O Notation

- Some basic examples:
  - What is the Big-O of the following functions:
    - $f(n) = 4n^2 + 3n + 10$
      - Answer: $O(n^2)$
    - $f(n) = 76,756,234n^2 + 427,913n + 7$
      - Answer: $O(n^2)$
    - $f(n) = 74n^8 - 62n^5 - 71562n^3 + 3n^2 – 5$
      - Answer: $O(n^8)$
    - $f(n) = 42n^4*(12n^6 - 73n^2 + 11)$
      - Answer: $O(n^{10})$
    - $f(n) = 75n*logn – 415$
      - Answer: $O(n*logn)$

Big-O Notation

- Consider the expression: $f(n) = 4n^2 + 3n + 10$
  - How fast is this "growing"?
    - We can say, $O(4n^2 + 3n + 10) = O(n^2)$
    - Till now, we have one function:
      - $f(n) = 4n^2 + 3n + 10$
    - Let us make a second function, g(n)
      - It's just a letter right? We could have called it r(n) or x(n)
        - Don't get scared about this
    - Now, let g(n) equal n^2
      - $g(n) = n^2$
    - So now we have two functions: f(n) and g(n)
      - We said (above) that $O(4n^2 + 3n + 10) = O(n^2)$
    - Similarly, we can say that the order of f(n) is O[g(n)].

Big-O Notation

Brace yourself!

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - Think about the two functions we just had:
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
      - We agreed that $O(4n^2 + 3n + 10) = O(n^2)$
      - Which means we agreed that the order of **<u>f(n) is O(g(n)</u>**
    - <u>That's all this definition says!!!</u>
    - f(n) is big-O of g(n), if there is a c
      - (c is a constant)
    - such that f(n) is not larger than c*g(n) for sufficiently large values of n (greater than N)

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - Think about the two functions we just had:
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - f is big-O of g, if there is a c such that f is not larger than c*g for sufficiently large values of n (greater than N)
      - So given the two functions above, does there exist some constant, c, that would make the following statement true?
      - $f(n) <= c*g(n)$
      - $4n^2 + 3n + 10 <= c*n^2$
      - If there does exist this c, then f(n) is O(g(n))
    - Let's go see if we can come up with the constant, c

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
    - Clearly, c cannot be 4 or less
      - Cause even if it was 4, we would have:
        - $4n^2 + 3n + 10 <= 4n^2$
        - This is NEVER true for any positive value of n!
      - So c must be greater than 4
    - Let us try with c being equal to 5
      - $4n^2 + 3n + 10 <= 5n^2$

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?

n	**$4n^2 + 3n + 10$**	**$5n^2$**
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] if there exists positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?

n	**$4n^2 + 3n + 10$**	**$5n^2$**
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?

n	**$4n^2 + 3n + 10$**	**$5n^2$**
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**
4	4(16) + 3(4) + 10 = **86**	5(16) = **80**

But now let's try larger values of n.

      - For n = 1 through 4, this statement is NOT true

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**
4	4(16) + 3(4) + 10 = **86**	5(16) = **80**
5	4(25) + 3(5) + 10 = **125**	5(25) = **125**

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**
4	4(16) + 3(4) + 10 = **86**	5(16) = **80**
5	4(25) + 3(5) + 10 = **125**	5(25) = **125**
6	4(36) + 3(6) + 10 = **172**	5(36) = **180**

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
      - $4n^2 + 3n + 10 <= 5n^2$
      - For what values of n, if ANY at all, is this true?
      - So when n = 5, the statement finally becomes true
      - And when n > 5, it remains true!
    - So our constant, 5, works for all n >= 5.

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - $f(n) = 4n^2 + 3n + 10$, and $g(n) = n^2$
    - Find the c such that $4n^2 + 3n + 10 <= c*n^2$
    - So our constant, 5, works for all n >= 5.
    - Therefore, **<u>f(n) is O(g(n))</u>** per our definition!
    - Why?
    - Because there exists positive integers, c and N,
      - Just so happens in this case that c = 5 and N = 5
    - such that f(n) <= c*g(n).

Who actually got that ?

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - What can we take from this?
      - That Big-O is hard as #$%q@$^&!!!
    - No, but seriously…
    - What we can gather is that:
    - c*g(n) is an <u>**upper bound**</u> on the value of f(n).
      - It represents the <u>worst possible scenario of running time</u>.
    - The number of operations is, at worst, proportional to g(n) for all <u>large values</u> of n.

Big-O Notation

- Summing up the basic properties for determining the order of a function:
  - If you’ve got multiple functions added together, the fastest growing one determines the order
  - Multiplicative constants don’t affect the order
  - If you’ve got multiple functions multiplied together, the overall order is their individual orders multiplied together

Big-O Notation

- Some basic examples:
  - What is the Big-O of the following functions:
    - $f(n) = 4n^2 + 3n + 10$
      - Answer: $O(n^2)$
    - $f(n) = 76{,}756{,}234n^2 + 427{,}913n + 7$
      - Answer: $O(n^2)$
    - $f(n) = 74n^8 - 62n^5 - 71562n^3 + 3n^2 – 5$
      - Answer: $O(n^8)$
    - $f(n) = 42n^4*(12n^6 - 73n^2 + 11)$
      - Answer: $O(n^{10})$
    - $f(n) = 75n*\log n – 415$
      - Answer: $O(n*\log n)$

Big-O Notation

- Quick Example of Analyzing Code:
  - This is just to show you how we use Big-O
    - we‘ll do more of these (a lot more) on Friday
  - Use big-O notation to analyze the time complexity of the following fragment of C code:

```
for (k=1; k<=n/2; k++) {
        sum = sum + 5;
}

for (j = 1; j <= n*n; j++) {
        delta = delta + 1;
}
```

Big-O Notation

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - We care about the total number of REPETITIVE operations.
      - Remember, we said we care about the running time for LARGE values of n
      - So in a **for loop**, with n as part of the comparison value determining when to stop

        ```
        for (k=1; k<=n/2; k++)
        ```

      - Whatever is INSIDE that loop will be executed a LOT of times
      - So we examine the code within this loop and see how many operations we find
        - When we say operations, we're referring to mathematical operations such as +, -, *, /, etc.

Big-O Notation

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - We have 2 loops,

```
for (k=1; k<=n/2; k++) {
      sum = sum + 5;
}

for (j = 1; j <= n*n; j++) {
      delta = delta + 1;
}
```

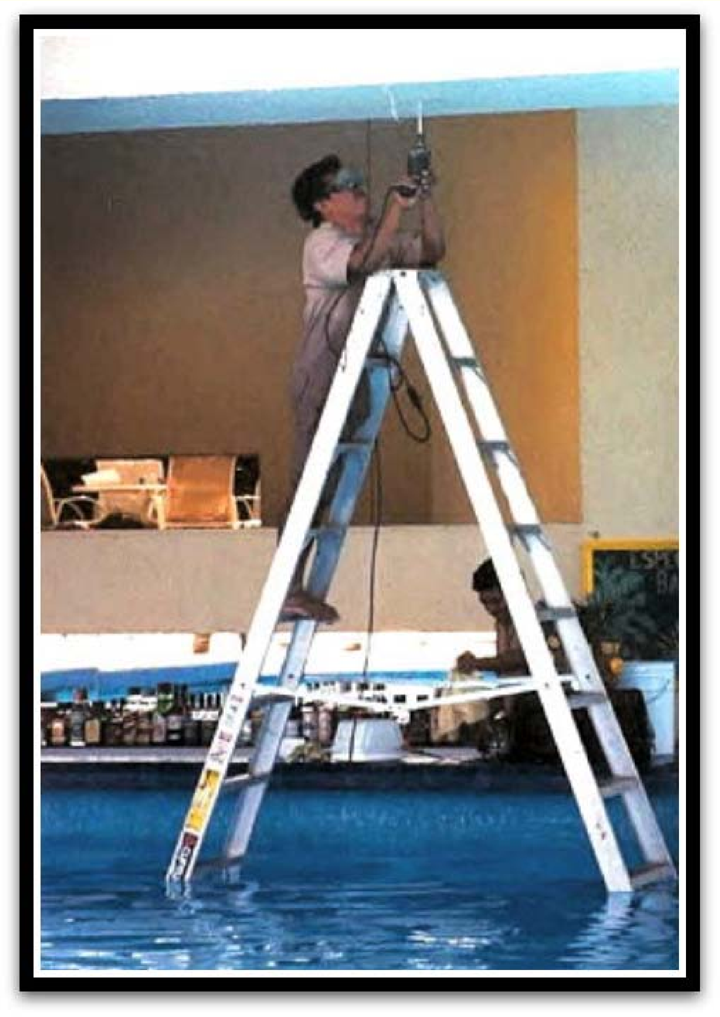
Big-O Notation

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - We have 2 loops,
      - The first loop runs n/2 times
      - Each iteration of the <u>first loop</u> results in <u>one operation</u>
        - The + operation in: `sum = sum + 5;`
      - So there are n/2 operations in the first loop
      - The second loop runs n^2 times
      - Each iteration of the <u>second loop</u> results in <u>one operation</u>
        - The + operation in: `delta = delta + 1;`
      - So there are n^2 operations in the second loop.

Big-O Notation

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - The first loop has n/2 operations
    - The second loop has n^2 operations
    - They are NOT nested loops.
      - One loop executes AFTER the other completely finishes
    - So <u>we simply ADD their operations</u>
    - The total number of operations would be $n/2 + n^2$
    - In Big-O terms, we can express the number of operations as $O(n^2)$

Brief Interlude: Human Stupidity



Big-O Notation

- Common orders (listed from slowest to fastest growth)

Function	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	Poly-log
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Big-O Notation

- O(1) or “Order One”: Constant time
  - does not mean that it takes only one operation
  - does mean that the work doesn’t change as n changes
  - is a notation for “constant work”
  - An example would be finding the smallest element in a sorted array
    - There’s nothing to search for here
    - The smallest element is always at the beginning of a sorted array
    - So this would take O(1) time

Big-O Notation

- O(n) or “Order n”: Linear time
  - does not mean that it takes n operations
    - maybe it takes 3*n operations, or perhaps 7*n operations
  - does mean that the work changes in a way that is proportional to n
  - Example:
    - <u>If the input size doubles, the running time also doubles</u>
  - is a notation for “work grows at a linear rate”
  - You usually can’t really do a lot better than this for most problems we deal with
    - After all, you need to at least examine all the data right?

Big-O Notation

- $O(n^2)$ or "Order n^2 ": Quadratic time
  - If input size doubles, running time increases by a factor of 4
- $O(n^3)$ or "Order n^3 ": Cubic time
  - If input size doubles, running time increases by a factor of 8
- $O(n^k)$: Other polynomial time
  - Should really try to avoid high order polynomial running times
    - However, it is considered good from a theoretical standpoint

Big-O Notation

- $O(2^n)$ or "Order 2^n ": Exponential time
  - more theoretical rather than practical interest because they cannot reasonably run on typical computers for even for moderate values of n.
  - Input sizes bigger than 40 or 50 become unmanageable
    - Even on faster computers
- O(n!): even worse than exponential!
  - Input sizes bigger than 10 will take a long time

Big-O Notation

- O(n logn):
  - Only slightly worse than O(n) time
    - And O(n logn) will be much less than $O(n^2)$
    - This is the running time for the better sorting algorithms we will go over (later)
- O(log n) or "Order log n": Logarithmic time
  - If input size doubles, running time increases ONLY by a constant amount
  - any algorithm that halves the data remaining to be processed on each iteration of a loop will be an O(log n) algorithm.

Big-O Notation – Problems

- Practical Problems that can be solved utilizing order notation:
  - Example:
    - You are told that algorithm A runs in O(n) time
    - You are also told the following:
      - For an input size of 10
      - The algorithm runs in 2 milliseconds
    - As a result, you can expect that it will take 100 milliseconds to run on an input size of 500
      - Notice the input size jumped by a multiple of 50
        - From 10 to 500
      - Therefore, given a O(n) algorithm, the running time should also jump by a multiple of 50, which it does!

Big-O Notation – Problems

- Practical Problems that can be solved utilizing order notation:
  - General process of solving these problems:
    - We know that Big-O is NOT exact
      - It's an upper bound on the actual running time
    - So when we say that an algorithm runs in O(f(n)) time,
    - Assume the EXACT running time is c*f(n)
      - where c is some constant
    - Using this assumption,
      - we can use the information in the problem to solve for c
      - Then we can use this c to answer the question being asked
    - Examples will clarify…

Big-O Notation – Problems

- Practical Problems that can be solved utilizing order notation:
  - Example 1: Algorithm A runs in $O(n^2)$ time
    - For an input size of 4, the running time is 10 milliseconds
    - How long will it take to run on an input size of 16?
    - Let $T(n) = c*n^2$
      - T(n) refers to the running time (of algorithm A) on input size n
      - Now, plug in the given data, and **find the value for c!**
    - $T(4) = c*4^2 = 10$ milliseconds
      - Therefore, c = 10/16 milliseconds
    - Now, answer the question by using c and solving T(16)
    - $T(16) = c*16^2 = (10/16)*16^2 = 160$ milliseconds

Big-O Notation – Problems

- Practical Problems that can be solved utilizing order notation:
  - Example 2: Algorithm A runs in $O(\log_2 n)$ time
    - For an input size of 16, the running time is 28 milliseconds
    - How long will it take to run on an input size of 64?
    - Let $T(n) = c*\log_2 n$
      - Now, plug in the given data, and **find the value for c!**
    - $T(16) = c*\log_2 16 = 28$ milliseconds
      - $c*4 = 28$ milliseconds
      - Therefore, $c = 7$ milliseconds
    - Now, answer the question by using c and solving T(64)
    - $T(64) = c*\log_2 64 = 7*\log_2 64 = 7*6 = 42$ milliseconds

Base Conversions

WASN'T

THAT

MARVELOUS!

Daily Demotivator

Algorithm Analysis

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I