



INTRO TO ALGORITHM ANALYSIS

COP 3502

Algorithm Analysis

- We have looked at a few number of algorithms in class so far
 - But we haven't looked at how to judge the efficiency or speed of an algorithm,
 - which is one of the goals of this class.
- We will use order notation to approximate 2 things about algorithms:
 - How much time they take
 - How much memory (space) they use.



Algorithm Analysis

- The first thing to realize is that it will be nearly **impossible** to exactly figure out how much time an algorithm will take on a particular computer.
 - Each algorithm instruction gets translated into smaller machine instructions
 - Each of which take various amounts of time to execute on different computers.
 - Also, we want to judge the algorithms independent of their specific implementation
 - An algorithm's run time can be language independent
- Therefore, rather than figuring out an algorithm's exact running time
 - **We will only want an approximation.**



Algorithm Analysis

- The type of approximation we will be looking for is a **Big-O** approximation
 - A type of order notation
 - Used to describe the limiting behavior of a function, when the argument approaches a large value.
 - In simpler terms a Big-O approximation is:
 - An Upper bound on the growth rate of a function.
 - Lower bound, and upper&lower bounds, and more involved proofs will be discussed in CS2.



Big-O

- Assume:
 - Each statement and each comparison in C takes some constant time.
- Time and space complexity will be a function of:
 - The input size (usually referred to as n)
- Since we are going for an ***approximation***,
 - we will make the following two simplifications in counting the # of steps an algorithm takes:
 - 1) Eliminate any term whose contribution to the total ceases to be significant as n becomes large
 - 2) Eliminate constant factors.



Big-O

- Thus, if we count the # of steps an algorithm takes is $4n^2 + 3n - 5$, then we will:
 - 1) Ignore $3n-5$ because that accounts for a small number of steps as n gets large (waaay less than n^2)
 - 2) Eliminate the constant factor of 4 in front of the n^2 term.
- In doing so, we conclude that the algorithm takes $O(n^2)$ steps.



Big-O

- Only consider the most significant *term*
 - *So for* : $4n^2 + 3n - 5$, we only look at $4n^2$
 - Then, we get rid of the constant 4^*
 - And we get $O(n^2)$



Big-O

- Why can we do this?
 - Because as n gets very large, the most significant term far outweighs the less significant terms and the constants.

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

■ Formal Definition

- $f(n)$ is $O[g(n)]$ if there exists positive integers c and N , such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.

➤ Think about the 2 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 $f(n)$ is $O(g(n))$

➤ So then what we were actually saying is...

➤ $f(n)$ is big-O of $g(n)$, if there is a c (c is a constant)

➤ Such that $f(n)$ is not larger than $c \cdot g(n)$ for sufficiently large values of n (greater than N)

- Let's see if we can determine c and N .



Big-O

■ Formal Definition

- $F(n)$ is $O[g(n)]$ if there exists positive integers c and N , such that $f(n) \leq c * g(n)$ for all $n \geq N$.
 - Does there exist some c that would make the following statement true?
 - $f(n) \leq c * g(n)$
 - OR for our example: $4n^2 + 3n + 10 \leq c * n^2$
 - If there does exist this c , then $f(n)$ is $O(g(n))$



Big-O

■ Formal Definition

- $f(n)$ is $O[g(n)]$ if there exists positive integers c and N , such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.

- Does there exist some c that would make the following statement true?

- $4n^2 + 3n + 10 \leq c \cdot n^2$

- Clearly $c = 4$ will not work:

- $4n^2 + 3n + 10 \leq 4n^2$

- Will $c = 5$ work?

- $4n^2 + 3n + 10 \leq 5n^2$

- Let's plug in different values of n to check...



Big-O

Formal Definition

- F(n) is $O[g(n)]$ if there exists positive integers c and N, such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.
 - Does $c = 5$, make the following statement true?
 - $4n^2 + 3n + 10 \leq 5n^2$??
 - Let's plug in different values of n to check...

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	$4(36) + 3(6) + 10 = 190$	$6(36) = 216$

For $c = 5$, if $n \geq 5$,
 $f(n) \leq c \cdot g(n)$
 $4n^2 + 3n + 10 \leq c \cdot n^2$

✓ Therefore, $4n^2 + 3n + 10$
is $O(n^2)$ $O(g(n))$



Big-O

■ Formal Definition

- $F(n)$ is $O[g(n)]$ if there exists positive integers c and N , such that $f(n) \leq c * g(n)$ for all $n \geq N$.
- In Summary,
- $O[g(n)]$ tells us that $c * g(n)$ is an upper bound on $f(n)$.
 - $c * g(n)$ is an upper bound on the running time of the algorithm,
 - where the number of operations is, at worst, proportional to $g(n)$ for large values of n .



Big-O

- Some basic examples:
 - What is the Big-O of the following functions:
 - 1) $f(n) = 4n^2 + 3n + 10$
 - $O(n^2)$
 - 2) $f(n) = 76,756n^2 + 427,913,100n, + 700$
 - $O(n^2)$
 - 3) $754n^8 - 62n^5 - 71562n^3 + 3n^2 - 5$
 - $O(n^8)$
 - 4) $f(n) = 42n^4*(12n^6 - 73n^2 + 11)$
 - $O(n^{10})$
 - 5) $f(n) = 75n*\log n - 415$
 - $O(n*\log n)$



Big-O Notation

- Quick Example of Analyzing Code:
 - (This is to demonstrate how to use Big-O, we'll do more of this next time.)

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

1 operation

How many times
does this loop run?

$n/2$

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

1 operation

How many times
does this loop run?

n^2

- So we get:
 - 1 operation * $n/2$ iterations AND
 - 1 operation * n^2 operations
- Since the loops aren't nested we can just add to get: $n^2 + n/2$ operations
- What is this Big-O? **$O(n^2)$**



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 only takes one operation
 - **DOES** mean that the amount of work doesn't change as n gets bigger
 - “constant work”
 - An example would be inserting an element into the front of a linked list
 - No matter how big the list is, it's a constant number of operations.



Big-O Notation

■ O(n) or Order n: Linear time

- Does **NOT** mean that it takes n operations
 - it may take $7n + 5$ operations
- **DOES** mean that the amount of actual work is proportional to the input size n
- Example, if the input size doubles, the running time also doubles
- “work grows at a linear rate”
- An example, inserting an element at the END of a linked list,
 - We have to traverse to the end of the linked list which requires us to move an iterator approximately n times and then do a constant number of operations once we get there.



Big-O Notation

- $O(n \log n)$
 - Only slightly worse than $O(n)$ time
 - $O(n \log n)$ will be much less than $O(n^2)$
 - This is the running time for the better sorting algorithms we will go over later in the semester.
- $O(\log n)$ or “Order $\log n$ ”: Logarithmic time
 - Any algorithm that halves the data remaining to be processed on each iteration of a loop will be an $O(\log n)$ algorithm.
 - For example, binary search



Big O Notation

- $O(n^2)$ or “Order n^2 ”: Quadratic time
 - for (i = 0; i < n; i++)
 - for (j = 0; j < n; j++) This would be $O(n^2)$
 - a constant number of operations



Big O Notation

- $O(2^n)$ or “Order 2^n ”: Exponential time
 - Input size bigger than 40 or 50 become unmanageable, more theoretical than practical interest.
- $O(n!)$: worse than exponential!
 - Input sizes bigger than 10 will take a long time.



Average Case and Worst Case

- When we are talking about the running time of an algorithm,
 - you'll notice that depending on the input – a program may run more quickly or slowly.
- For example, Insertion sort
 - (which we haven't gone over yet...)
 - will run much for quickly for an already sorted list of numbers
 - than if we give it a list of numbers in descending order.



Average Case and Worst Case

- So, when we analyze the running times of algorithms
 - we must acknowledge the fact that these running times may vary based on the actual type of input to the algorithm, not just the size
 - In our analysis we are typically concerned with 2 things:
 - 1) What is the worst possible running time an algorithm can achieve, given any inputAND
 - 2) What is the average, or expected running time of an algorithm, averaged over all possible inputs.



Average Case and Worst Case

- In our analysis we are typically concerned with 2 things:
 - 1) What is the worst possible running time an algorithm can achieve, given any input

AND

- 2) What is the average, or expected running time of an algorithm, averaged over all possible inputs.
- As you might imagine, #2 is very useful but might be difficult to compute.
 - For #1, you usually have to figure out what input will cause the algorithm to act most inefficiently
 - For example, a descending list in insertion sort.
 - Then, simply calculate how long the algorithm would take to run based on that worst-case input.



Average Case and Worst Case

- However, if we can show that
 - The Best Case – (i.e. the fastest possible running of an algorithm on any input)
 - AND
 - The Worst Case
 - Are the same big-O bound,
 - THEN we know the average case is that big-O bound.



Average Case and Worst Case

- When computing the average case running time
 - We may assume that all inputs are random, or equally likely
 - However,
 - This may not always be the case
 - For example, if the user is given a menu of several choices, it may be the case that some choices are chosen far more frequently than others.
 - In this case,
 - assuming that each case is chosen equally may not give you an accurate average case running time.



Using Order Notation to Estimate Time

- Let's say you are told:
 - Algorithm A runs in $O(n)$ time,
 - and for an input size of 10, the algorithm runs in 2 ms.
 - Then, you can expect it to take 100ms to run on an input size of 500.
- So in general, if you know an algorithm is $O(f(n))$,
 - Assume that the exact running time is $c \cdot f(n)$, where c is some constant
 - Then given an input size and a running time, you should be able solve for c .



Practice Problems

1) Algorithm A runs in $O(n^2)$ time, and for an input size of 4, the algorithm runs in 10 ms.

- How long can you expect it to take to run on an input size of 16?

- Given $O(f(n))$, we know $\rightarrow c \cdot f(n) = \text{time in ms}$

- So we're given $f(n) = n^2$ and $n = 4$, and $\text{time} = 10\text{ms}$

- So we can solve for c : $c \cdot 4^2 = 10\text{ms}$, $c = 10/16$

- Now in the second part, $n = 16$ and we want to find the time, now we can plug in c :

- $(10/16) \cdot 16^2 = 160 \text{ ms}$



Practice Problems

- 1) Algorithm A runs in $O(\log_2 n)$ time, and for an input size of 16, the algorithm runs in 28 ms.
- How long can you expect it to take to run on an input size of 64?
 - $C \cdot \log_2(16) = 28\text{ms} \rightarrow 4c = 28\text{ms} \rightarrow c = 7$
 - If $n = 64$, let's solve for time:
 - $7 \cdot \log_2 64 = \text{time ms}$
 - $7 \cdot 6 = 42 \text{ ms}$



Reasonable vs Unreasonable Algorithms

- One thing we can use order notation for is
 - to decide whether or not an algorithm can be implemented in practice and run in a reasonable amount of time.
 - In a very general sense,
 - algorithms that run in polynomial time with respect to the input, are considered to be REASONABLE.
 - So this would include any algorithm that runs in $O(n^k)$ time, where k is some constant.
 - In most everyday problems, k is never more than 3 or so
 - While $O(n^3)$ algorithms are quite slow for larger input sizes, they will still finish in a reasonable amount of time.



Reasonable vs Unreasonable Algorithms

- However, there are mathematical functions that are “larger” than polynomials.
 - In particular, exponential functions grow much more quickly than polynomials.
 - Exponential, meaning it runs in $O(c^n)$ time, where c is some constant.
 - It is considered to be an UNREASONABLE algorithm.
 - Running such an algorithm would take too much time for any substantial value of n .
 - For example, consider computing 2^{100} .



Reasonable vs Unreasonable Algorithms

- Often times, exhaustive search algorithms are UNREASONABLE.
 - In a chess game, one way for a computer player to choose a move is to map out all possible moves by the computer and the opponent, several moves into the future.
 - Then by judging which would lead to a better board position, the computer would choose the best move.
 - Unfortunately, there are too many board positions to consider them all
 - So such an algorithm would be unreasonable.
 - (Most computer chess programs only search a few possible moves, not all of them. And only consider a few of the opponents responses.)

