Algorithm Analysis

Kumkum Saxena

Algorithm

- An algorithm is a procedure having well defined steps for solving a particular problem.
- Algorithm is finite set of logic or instructions, written in order for accomplish the certain predefined task.
- It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

- The major categories of algorithms are given below:
- **Sort:** Algorithm developed for sorting the items in certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

- The performance of algorithm is measured on the basis of following properties:
- **Time complexity:** It is a way of representing the amount of time needed by a program to run to the completion.
- **Space complexity:** It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

- Each algorithm must have:
- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions:** The condition(s) on output.

- **Example:** Design an algorithm to multiply the two numbers x and y and display the result in z.
- Step 1 START
- Step 2 declare three integers x, y & z
- Step 3 define values of x & y
- Step 4 multiply values of x & y
- Step 5 store the output of step 4 in z
- Step 6 print z
- Step 7 STOP

- Alternatively the algorithm can be written as
- Step 1 START MULTIPLY
- Step 2 get values of x & y
- Step 3 z$\leftarrow$ x * y
- Step 4 display z
- Step 5 STOP

Characteristics of an Algorithm

- An algorithm must follow the mentioned below characteristics:
- **Input:** An algorithm must have 0 or well defined inputs.
- **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
- **Feasibility:** An algorithm must be terminated after the finite number of steps.
- **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

Why performance analysis?

- There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc.
- Why to worry about performance? The answer to this is simple, we can have all the above things only if we have performance.
- So performance is like currency through which we can buy all the above things.

Given two algorithms for a task, how do we find out which one is better?

- One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.
- There are many problems with this approach for analysis of algorithms.
  - It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
  - It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis

- It is the big idea that handles above issues in analyzing algorithms.
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

- For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic).
- To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer.
- For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.
- The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?

- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.
- For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine.
- So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.
- Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value.
- It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation.
- So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

- We can have three cases to analyze an algorithm:
  - Worst Case(upper bound)-In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.
  - Average Case-In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

- Best Case(lower bound)-In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.

- Most of the times, we do worst case analysis to analyze algorithms.
- In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- The Best Case analysis is not required. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Asymptotic Analysis

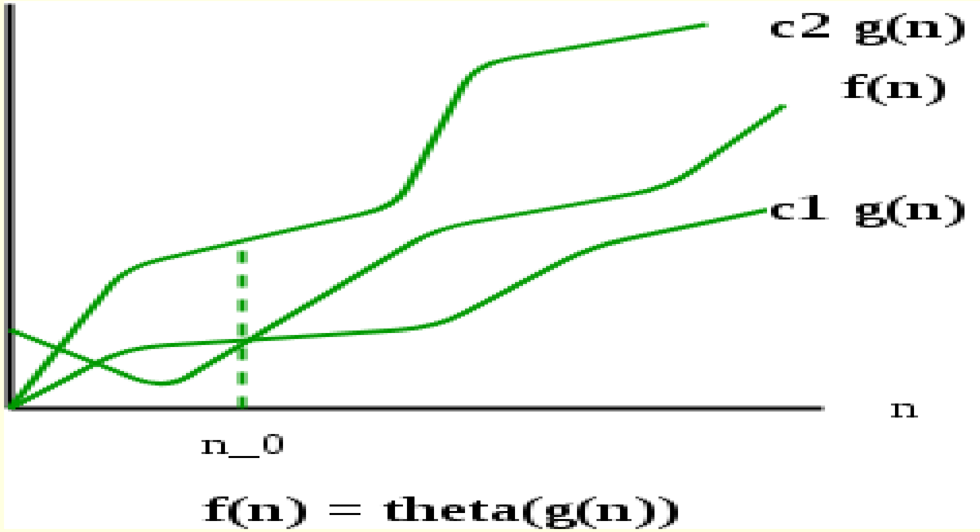
- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

Asymptotic notations

- **1) Θ Notation(Average case):** The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
- A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.
- For example, consider the following expression. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$
- Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than $\Theta n^2)$ irrespective of the constants involved.

- For a given function g(n), we denote Θ(g(n)) is following set of functions.

```
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
              that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}
```



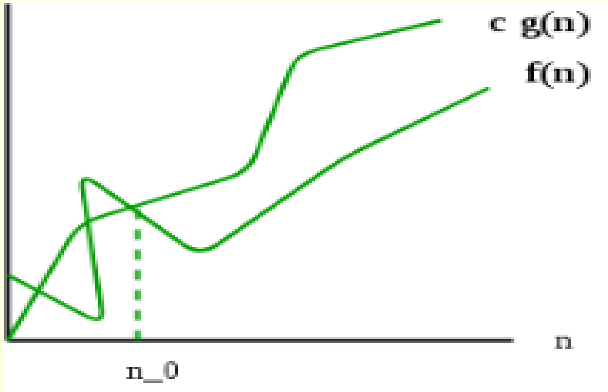
- The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0).
- The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

- **2) Big O Notation(Worst Case):** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
- For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case.
- We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
-

  If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
  - The worst case time complexity of Insertion Sort is Θ(n^2).
  - The best case time complexity of Insertion Sort is Θ(n).

- The Big O notation is useful when we only have upper bound on time complexity of an algorithm.

```
O(g(n)) = { f(n): there exist positive constants c and
                  n0 such that 0 <= f(n) <= c*g(n) for
                  all n >= n0}
```

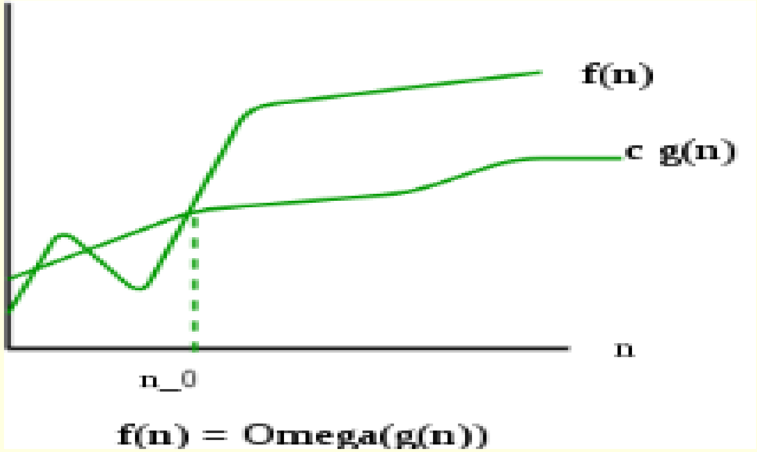


f(n) = O(g(n))

- **3) Ω Notation(Best Case):** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previously, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

- For a given function g(n), we denote by Ω(g(n)) the set of functions.

```
Ω (g(n)) = {f(n): there exist positive constants c and
                  n0 such that 0 <= c*g(n) <= f(n) for
                  all n >= n0}.
```



Some Functions

1. **O(1):** Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.
2. **O(n):** Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.
3. **O(n^c)**: Time complexity of nested loops is equal to the number of times the innermost statement is executed.
4. **O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.
5. **O(LogLogn)** Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

Big-O Notation

- What is Big O?
  - 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**
    - 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

More Algorithm Analysis

- Examples of Analyzing Code:
  - We now go over many examples of code fragments
  - Each of these functions will be analyzed for their runtime in terms of the variable n
  - Utilizing the idea of Big-O,
    - determine the Big-O running time of each

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

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, <u>we want to judge algorithms independent of their implementation</u>
    - Thus, rather than figure out an algorithm's exact running time
      - **<u>We only want an approximation</u>** (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
      - <u>Time and space used by an algorithm function of the input</u>

Big-O Notation

- What is Big O?
  - 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** 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 whose contribution to the total ceases to be significant as n get larger and larger
      - of course, this also includes constants, 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 the order of this function, f(n), **as n^2**
    - 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: $O(n^{10})$
    - $f(n) = 75n*\log n – 415$
      - Answer: $O(n*\log n)$

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 <u>make a second function</u>, **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, <u>let g(n) equal n^2</u>
      - **$g(n) = n^2$**
    - So now we have <u>two functions</u>: **f(n)** and **g(n)**
      - We said (above) that <u>$O(4n^2 + 3n + 10) = O(n^2)$</u>
    - Similarly, we can say that the **order of f(n) is O[g(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 **$g(n) = n^2$**
      - 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, <u>**does there exist**</u> some <u>**constant**</u>, <u>**c**</u>, that would make the following statement true?
      - f(n) <= c*g(n)
      - **$4n^2$ + 3n + 10 <= c*n^2**
      - <u>If there does exist this c</u>, <u>**then f(n) is O(g(n))**</u>
    - 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**
    - **<u>Find the c</u>** such that **$4n^2 + 3n + 10 <= c*n^2$**
    - Clearly, c cannot be 4 or less
      - Cause <u>even if it was 4</u>, we would have:
        - **$4n^2 + 3n + 10 <= 4n^2$**
        - This is <u>NEVER true for any positive value of n</u>!
      - So **<u>c must be greater than 4</u>**
    - Let us <u>try with c being equal to 5</u>
      - **$4n^2 + 3n + 10 <= 5n^2$**

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 **$g(n) = n^2$**
    - **<u>Find the c</u>** 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)] <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**
    - **<u>Find the c</u>** 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 **$g(n) = n^2$**
    - **<u>Find the c</u>** 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 **$g(n) = n^2$**
    - **<u>Find the c</u>** 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 **$g(n) = n^2$**
    - **<u>Find the c</u>** 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 **$g(n) = n^2$**
    - **<u>Find the c</u>** 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**
    - **<u>Find the c</u>** 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 <u>when n = 5</u>, the <u>statement finally becomes **true**</u>
      - And **<u>when n > 5, it remains true</u>**!
    - 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**
    - **<u>Find the c</u>** 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 <u>there exists positive integers, c and N</u>,
      - Just so happens in this case that c = 5 and N = 5
    - **<u>such that f(n) <= c*g(n)</u>**.

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 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:
  1) If you’ve got multiple functions added together, the fastest growing one determines the order
  2) Multiplicative constants don’t affect the order
  3) If you’ve got multiple functions multiplied together, the overall order is their individual orders multiplied together

Big-O Notation

- Quick Example of Analyzing Code:
  - 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 **<u>for loop</u>**, 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 $O(n^2)$

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:
    - If the input size doubles, the running time also doubles
  - 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

- **<u>O(n^2)</u>** or "Order n^2": **<u>Quadratic time</u>**
  - If input size doubles, running time increases by a factor of 4
- **<u>O(n^3)</u>** or "Order n^3": **<u>Cubic time</u>**
  - If input size doubles, running time increases by a factor of 8
- **<u>O(n^k)</u>**: <u>Other polynomial time</u>
  - 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 – Practical 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 for an input size of 500, the algorithm would run in 100 milliseconds!
      - 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 – Practical 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 – Practical 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 – Practical 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!**
    - $T(16) = c*\log_2 16 = 10$ 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

More Algorithm Analysis

Kumkum Saxena

More Algorithm Analysis

- Example 2:
  - Determine the Big O running time of the following code fragment:

```
int func1(int n) {
        int i, j, x = 0;
        for (i = 1; i <= n; i++) {
                for (j = 1; j <= n; j++) {
                        x++;
                }
        }
        return x;
}
```

More Algorithm Analysis

- Example 2:
  - So look at what's going on in the code:
    - We care about the total number of REPETITIVE operations
    - We have two loops
      - AND they are NESTED loops
    - The outer loop runs n times
      - From i = 1 up through n
      - How many operations are performed at each iteration?
        - Answer is coming…
    - The inner loop runs n times
      - From j = 1 up through n
      - And only one operation (x++) is performed at each iteration

More Algorithm Analysis

- Example 2:
  - So look at what's going on in the code:
    - Let's look at a couple of iterations of the OUTER loop:
      - When i = 1, what happens?
        - The inner loop runs n times
        - Resulting in n operations from the inner loop
      - Then, i gets incremented and it becomes equal to 2
      - When i = 2, what happens?
        - Again, the inner loop runs n times
        - Again resulting in n operations from the inner loop
    - We notice the following:
      - For EACH iteration of the OUTER loop,
      - The INNER loop runs n times
        - Resulting in n operations

More Algorithm Analysis

- Example 2:
  - So look at what's going on in the code:
    - And how many times does the outer loop run?
      - n times
    - So the outer loop runs n times
    - And for each of those n times, the inner loop also runs n times
      - Resulting in n operations
    - So we have n operations per iteration of OUTER loop
    - And outer loop runs n times
    - Finally, we have n*n as the number of operations
    - We approximate the running time as $O(n^2)$

More Algorithm Analysis

- Example 3:
  - Determine the Big O running time of the following code fragment:

```
int func3(int n) {
        int i, x = 0;
        for (i = 1; i <= n; i++)
                x++;
        for (i = 1; i<=n; i++)
                x++;
        return x;
}
```

More Algorithm Analysis

- Example 3:
  - So look at what's going on in the code:
    - We care about the total number of REPETITIVE operations
    - We have two loops
      - They are NOT nested loops
    - The first loop runs n times
      - From i = 1 up through n
      - only one operation (x++) is performed at each iteration
    - How many times does the second loop run?
      - Notice that i is indeed reset to 1 at the beginning of the loop
      - Thus, the second loop runs n times, from i = 1 up through n
      - And only one operation (x++) is performed at each iteration

More Algorithm Analysis

- Example 3:
  - So look at what's going on in the code:
    - Again, the loops are NOT nested
    - So they execute sequentially (one after the other)
  - Therefore:
    - Our total runtime is on the order of n+n
    - Which of course equals 2n
  - Now, in Big O notation
    - We approximate the running time as O(n)

More Algorithm Analysis

- Example 4:
  - Determine the Big O running time of the following code fragment:

```
int func4(int n) {
        while (n > 0) {
                printf(“%d”, n%2);
                n = n/2;
        }
}
```

More Algorithm Analysis

- Example 4:
  - So look at what's going on in the code:
    - We have one while loop
      - You can't just look at this loop and say it iterates n times or n/2 times
      - Rather, it continues to execute as long as n is greater than 0
      - The question is: **<u>how many iterations will that be?</u>**
    - Within the while loop
      - The last line of code divides the input, n, by 2
      - So n is halved at each iteration of the while loop
    - If you remember, we said this ends up running in log n time
    - Now let's look at how this works

More Algorithm Analysis

- Example 4:
  - So look at what's going on in the code:
    - For the ease of the analysis, we make a new variable
      - originalN:
        - originalN refers to the value originally stored in the input, n
        - So if n started at 100, originalN will be equal to 100
    - The first time through the loop
      - n gets set to originalN/2
        - If the original n was 100, after one iteration n would be 100/2
    - The second time through the loop
      - n gets set to originalN/4
    - The third time through the loop
      - n gets set to originalN/8

Notice:
After **three** iterations, n gets set to originalN/2^{3}

More Algorithm Analysis

- Example 4:
  - So look at what's going on in the code:
    - In general, after k iterations
      - n gets set to originalN/2^k
    - The algorithm ends when originalN/2^k = 1, approximately
    - We now solve for k
    - Why?
      - <u>Because we want to find the **total # of iterations**</u>
    - Multiplying both sides by 2^k, we get originalN = 2^k
    - Now, using the definition of logs, we solve for k
      - k = log originalN
    - So we approximate the running time as O(log n)

More Algorithm Analysis

- Example 5:
  - Determine the Big O running time of the following code fragment:

```
int func5(int** array, int n)  {
        int i = 0, j = 0;
        while (i < n) {
                while (j < n && array[i][j] == 1)
                        j++;
                i++;
        }
        return j;
}
```

More Algorithm Analysis

- Example 5:
  - So look at what's going on in the code:
    - At first glance, we see two NESTED loops
    - This can often indicate an $O(n^2)$ algorithm
      - But we need to look closer to confirm
    - Focus on what's going on with i and j

```
int func5(int** array, int n)  {
        int i = 0, j = 0;
        while (i < n) {
                while (j < n && array[i][j] == 1)
                        j++;
                i++;
        }
```

More Algorithm Analysis

- Example 5:
  - So look at what's going on in the code:
    - Focus on what's going on with i and j
      - i and j clearly increase (from the j++ and i++)
      - BUT, they never decrease
      - AND, neither ever gets reset to 0

```
int func5(int** array, int n)  {
       int i = 0, j = 0;
       while (i < n) {
              while (j < n && array[i][j] == 1)
                     j++;
              i++;
       }
```

More Algorithm Analysis

- Example 5:
  - So look at what's going on in the code:
    - And the OUTER while loop ends once i gets to n
    - So, what does this mean?
      - The statement i++ can never run more than n times
      - And the statement j++ can never run more than n times

```
int func5(int** array, int n)  {
        int i = 0, j = 0;
        while (i < n) {
                while (j < n && array[i][j] == 1)
                        j++;
                i++;
        }
```

More Algorithm Analysis

- Example 5:
  - So look at what's going on in the code:
    - The MOST number of times these two statements can run (combined) is 2n times
    - So we approximate the running time as O(n)

```
int func5(int** array, int n)  {
      int i = 0, j = 0;
      while (i < n) {
            while (j < n && array[i][j] == 1)
                  j++;
            i++;
      }
```

More Algorithm Analysis

- Example 6:
  - Determine the Big O running time of the following code fragment:
    - What’s the one big difference here???

```
int func6(int** array, int n)  {
        int i = 0, j;
        while (i < n) {
                j = 0;
                while (j < n && array[i][j] == 1)
                        j++;
                i++;
        }
        return j;
}
```

More Algorithm Analysis

- Example 6:
  - So look at what's going on in the code:
    - The difference is that we RESET j to 0 a the beginning of the OUTER while loop

```
int func6(int** array, int n)  {
        int i = 0, j;
        while (i < n) {
                j = 0;
                while (j < n && array[i][j] == 1)
                        j++;
                i++;
        }
        return j;
}
```

More Algorithm Analysis

- Example 6:
  - So look at what's going on in the code:
    - The difference is that we RESET j to 0 a the beginning of the OUTER while loop
    - How does that change things?
      - Now j can iterate from 0 to n for EACH iteration of the OUTER while loop
        - For each value of i
      - This is similar to the 2nd example shown
    - So we approximate the running time as $O(n^2)$

More Algorithm Analysis

- Example 7:
  - Determine the Big O running time of the following code fragment:

```
int func7(int A[], int sizeA, int B[], int sizeB) {
        int i, j;
        for (i = 0; i < sizeA; i++)
                for (j = 0; j < sizeB; j++)
                        if (A[i] == B[j])
                                return 1;
        return 0;
}
```

More Algorithm Analysis

- Example 7:
  - So look at what's going on in the code:
    - First notice that the runtime here is NOT in terms of n
    - It will be in terms of sizeA and sizeB
    - And this is also just like Example 2
    - The outer loop runs sizeA times
    - For EACH of those times,
      - The inner loop runs sizeB times
    - So this algorithm runs sizeA*sizeB times
    - We approximate the running time as O(sizeA*sizeB)

More Algorithm Analysis

- Example 8:
  - Determine the Big O running time of the following code fragment:

```
int func8(int A[], int sizeA, int B[], int sizeB) {
      int i, j;
      for (i = 0; i < sizeA; i++) {
            if (binSearch(B, sizeB, A[i]))
                  return 1;
      }
      return 0;
}
```

More Algorithm Analysis

- Example 8:
  - So look at what's going on in the code:
    - Note: we see that we are calling the function binSearch
    - As discussed previously, a single binary search runs in O(log n) time
      - where n represents the number of items within which you are searching
  - Examining the for loop:
    - The for loop will execute sizeA times
    - For EACH iteration of this loop
      - a binary search will be run
    - We approximate the running time as O(sizeA*log(sizeB))

And More Algorithm Analysis

Kumkum Saxena

And More Algorithm Analysis

- Examples of Analyzing Code:
  - Last time we went over examples of analyzing code
    - We did this in a somewhat naïve manner
      - Just analyzed the code and tried to “trace” what was going on
  - This Lecture:
    - We will do this in a more structured fashion
    - We mentioned that summations are a tool for you to help coming up with a running time of iterative algorithms
    - Today we will look at some of those same code fragments, as well as others, and show you how to use summations to find the Big-O running time

More Algorithm Analysis

- Example 1:
  - Determine the Big O running time of the following code fragment:
    - We have two for loops
    - They are NOT nested
      - The first runs from k = 1 up to (and including) n/2
      - The second runs from j = 1 up to (and including) n^2

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

More Algorithm Analysis

- Example 1:
  - Determine the Big O running time of the following code fragment:
    - Here's how we can express the number of operations in the form of a summation:

$$\sum_{k=1}^{n/2} 1 + \sum_{j=1}^{n^2} 1$$

The constant value, 1, inside each summation refers to the one, and only, operation in each for loop.

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

Now you simply solve the summation!

More Algorithm Analysis

- Example 1:
  - Determine the Big O running time of the following code fragment:
    - Here's how we can express the number of operations in the form of a summation:

$$\sum_{k=1}^{n/2} 1 + \sum_{j=1}^{n^2} 1$$

You use the formula: $\sum_{i=1}^{n} k = k * n$

$$\sum_{k=1}^{n/2} 1 + \sum_{j=1}^{n^2} 1 = \frac{n}{2} + n^2$$

    - This is a **CLOSED FORM** solution of the summation
    - So we approximate the running time as $O(n^2)$

More Algorithm Analysis

- Example 2:
  - Determine the Big O running time of the following code fragment:
    - Here we again have two for loops
    - But this time they are nested

```
int func2(int n) {
        int i, j, x = 0;
        for (i = 1; i <= n; i++) {
                for (j = 1; j <= n; j++) {
                        x++;
                }
        }
        return x;
}
```

More Algorithm Analysis

- Example 2:
  - Determine the Big O running time of the following code fragment:
    - Here we again have two for loops
    - But this time they are nested
      - The outer loop runs from i = 1 up to (and including) n
      - The inner loop runs from j = 1 up to (and including) n
    - The sole (only) operation is a “x++” within the inner loop

More Algorithm Analysis

- Example 2:
  - Determine the Big O running time of the following code fragment:
    - We express the number of operations in the form of a summation and then we solve that summation:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

You use the formula: $\sum_{i=1}^{n} k = k * n$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n \quad = n^2$$

All we did is apply the above formula twice.

- This is a **CLOSED FORM** solution of the summation
- So we approximate the running time as $O(n^2)$

More Algorithm Analysis

- Example 3:
  - Determine the Big O running time of the following code fragment:
    - Here we again have two for loops
    - And they are nested. So is this $O(n^2)$?

```
int func3(int n) {
        sum = 0;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n * n; j++) {
                        sum++;
                }
        }
}
```

More Algorithm Analysis

- Example 3:
  - Determine the Big O running time of the following code fragment:
    - Here we again have two for loops
    - And they are nested. So is this $O(n^2)$?
      - The outer loop runs from i = 0 up to (and not including) n
      - The inner loop runs from j = 0 up to (and not including) n^2
    - The sole (only) operation is a "sum++" within the inner loop

More Algorithm Analysis

- Example 3:
  - Determine the Big O running time of the following code fragment:
    - We express the number of operations in the form of a summation and then we solve that summation:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n^2-1} 1$$

You use the formula: $\sum_{i=1}^{n} k = k * n$

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n^2-1} 1 = \sum_{i=0}^{n-1} n^2 = n^2 \sum_{i=0}^{n-1} 1 = n^3$$

All we did is apply the above formula twice.

- This is a **<u>CLOSED FORM</u>** solution of the summation
- So we approximate the running time as $O(n^3)$

More Algorithm Analysis

- Example 4:
  - Write a summation that describes the number of multiplication operations in this code fragment:
    - Here we again have two for loops
    - Pay attention to the limits (bounds) of the for loop

```
int func3(int n) {
	bigNumber = 0;
	for (i = 100; i <= 2n; i++) {
		for (j = 1; j < n * n; j++) {
			bigNumber += i*n + j*n;
		}
	}
}
```

More Algorithm Analysis

- Example 4:
  - Write a summation that describes the <u>number of multiplication operations</u> in this code fragment:
    - Here we again have two for loops
    - Pay attention to the limits (bounds) of the for loop
      - The outer loop runs from i = 100 up to (and including) 2n
      - The inner loop runs from j = 1 up to (and not including) n^2
    - Now examine the number of <u>**multiplications**</u>
      - Because this problem specifically said to “describe the number of multiplication operations, we do not care about ANY of the other operations
      - bigNumber += i*n + j*n;
      - There are TWO multiplication operations in this statement

More Algorithm Analysis

- Example 4:
  - Write a summation that describes the <u>number of multiplication operations</u> in this code fragment:
    - We express the number of multiplications in the form of a summation and then we solve that summation:

$$\sum_{i=100}^{2n} \sum_{j=1}^{n^2-1} 2$$

$$\sum_{i=100}^{2n} \sum_{j=1}^{n^2-1} 2 = \sum_{i=100}^{2n} 2(n^2-1) = 2(n^2-1)\sum_{i=100}^{2n} 1 = 2(n^2-1)(2n-99)$$

    - This is a **<u>CLOSED FORM</u>** solution of the summation
    - Shows the number of multiplications