Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 12: Analysis of Algorithms

Some background knowledge:

- What is an Algorithm?
- Why consider Efficiency and Correctness?
- What is Computational Complexity?
- What are the factors affecting efficiency?
- What is the Motivation for studying Algorithms?

Algorithms and Programs

- Algorithm: a method or a process followed to solve a problem.
  - A recipe.
- An algorithm takes the input to a problem (function) and transforms it to the output.
  - A mapping of input to output.
- A problem can have many algorithms.

Algorithm Properties

- An algorithm possesses the following properties:
  - It must be correct.
  - It must be composed of a series of concrete steps.
  - There can be no ambiguity as to which step will be performed next.
  - It must be composed of a finite number of steps.
  - It must terminate.

- A computer program is an instance, or concrete representation, for an algorithm in some programming language.

Solution Vs Efficient Solution

- Goal (1) → ***To design an algorithm that is easy to understand, code, debug***, is the basic concern of Software Engineering.
- Goal (2) → ***To design an algorithm that makes efficient use of the computer's resources***, is the concern of data structures and algorithm analysis.
- When goal (2) is important, how do we measure an algorithm's cost?

Time and space

- To analyze an algorithm means:
  - developing a formula for predicting *how fast* an algorithm is, based on the size of the input (time complexity), and/or
  - developing a formula for predicting *how much memory* an algorithm requires, based on the size of the input (space complexity)
- Usually time is our biggest concern
  - Most algorithms require a fixed amount of space

What does “size of the input” mean?

- If we are searching an array, the “size” of the input could be the size of the array
- If we are merging two arrays, the “size” could be the sum of the two array sizes
- If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be the parameter that most influences the actual time/space required
  - It is *usually* obvious what this parameter is
  - Sometimes we need two or more parameters

How to Measure Efficiency?

Critical concerns →

- Correctness
- Time Efficiency
- Space Efficiency
- Optimality

Basic factor affecting running time is the size of input

1. Empirical comparison (run programs)
2. Asymptotic or Theoretical Algorithm Analysis

Comparing Algorithms: an Empirical Approach

- (1) Implement each candidate →
  - That could be lots of work – also error-prone
- (2) Run it →
  - Which inputs?
- (3) Time it→
  - What machines / OS?

Exact Computational Complexity

Measure of the number of steps taken in the algorithm as a function of the input size.

Given 2 algorithms A and B, solving the same problem, require 100 and 1000 units of work respectively.

How can we translate this to A and B's Time Complexity?

If we are running A and B on a Pentium III 500 MHz PC, this means that there are 500 000 000 cycles / s.

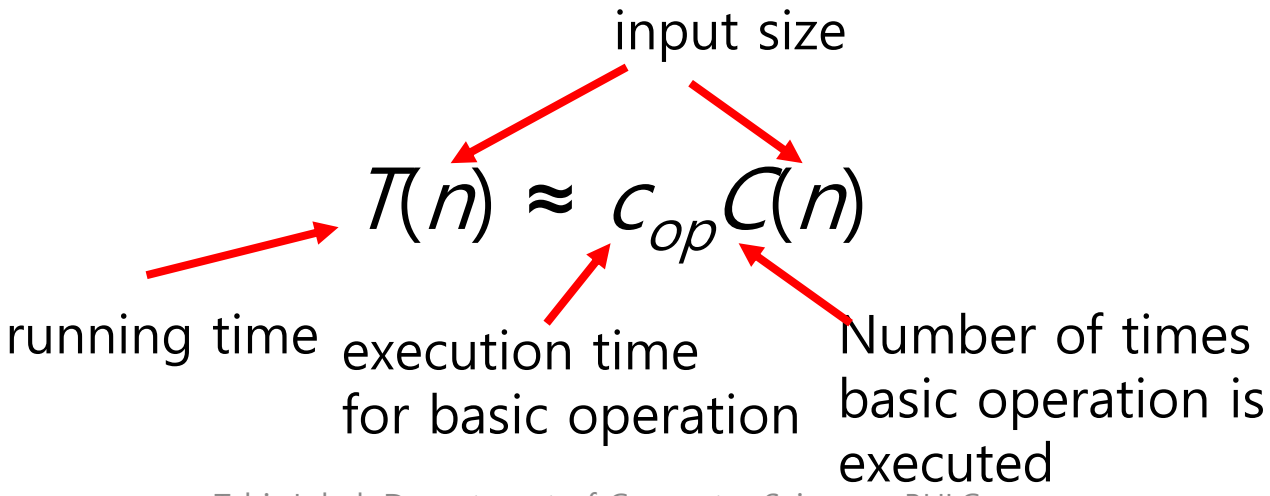
If each work unit takes 2 cycles, then each unit of work would take 1/250 000 000 seconds or 4 ns to complete.

Asymptotic Algorithmic Analysis

- A technique used to characterize the execution behavior of algorithms in a manner *independent* of a particular platform, compiler, or language.
- Abstract away the minor variations and describe the performance of algorithms in a more theoretical, processor independent fashion.
- A method to compare speed of algorithms against one another.

Theoretical analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*
- *Basic operation*: the operation that contributes most towards the running time of the algorithm



Analyzing Running Time

T(n), or the running time of a particular algorithm on input of size n, is taken to be the number of times the instructions in the algorithm are executed. Pseudo code algorithm illustrates the calculation of the mean (average) of a set of n numbers:

	Statement	Number of times executed
1. n = read input from user	1	1
2. sum = 0	2	1
3. i = 0	3	1
4. while i < n	4	n+1
5. number = read input from user	5	n
6. sum = sum + number	6	n
7. i = i + 1	7	n
8. mean = sum / n	8	1

The computing time for this algorithm in terms on input size n is:

T(n) = 4n + 5.

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$

FIGURE **2.1** The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10} n + 1000$.

n	$f(n)$	n^2		$100n$		$\log_{10} n$		1000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1000	90.82
10	2,101	100	4.76	1,000	47.6	1	0.05	1000	47.62
100	21002	10,000	47.6	10,000	47.6	2	0.991	1000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1000	0.00

Growth Rates of the Time Complexities of Algorithms with respect to increasing problem sizes.

On a machine running at 1 GHz (assuming 1 cycle per work unit).

For small values of n:

n	*f(n) = lg n*	*f(n) = n*	*f(n) = nlgn*	*f(n) = n^2*	*f(n) = 2^n*	*f(n) = n!*
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.36 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	$8.4 * 10^{15}$ years
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	

What conclusions can we draw?

Which algorithms remain useful as problem size increases?

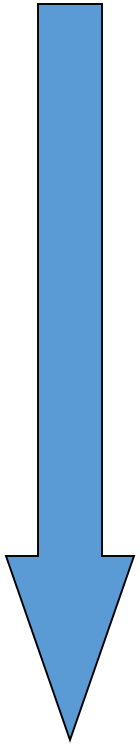
Growth Rates of the Time Complexities of Algorithms with respect to increasing problem sizes.

For larger values of n:

n	*f(n) = lg n*	*f(n) = n*	*f(n) = nlgn*	*f(n) = n^2*	*f(n) = 2^n*	*f(n) = n!*
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4 * 10^{13} years	
1000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

What conclusions can we draw?

Which algorithms remain useful as problem size increases?



Tractable → $\log n < n < n \log n < n^2$

Exponential → 2^n

Factorial → $n!$

Intractable!

How many foos?

```
for (j = 1; j <= N; ++j) {
        for (k = 1; k <= M; ++k) {
                foo( );
        }
}
```

$$\sum_{j=1}^{N} \sum_{k=1}^{M} 1 = NM$$

How many foos?

```
for (j = 1; j <= N; ++j) {
        for (k = 1; k <= j; ++k) {
                foo( );
        }
}
```

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

How many foos?

```
for (j = 0; j < N; ++j) {
        for (k = 0; k < j; ++k) {
                foo( );
        }
}
for (j = 0; j < N; ++j) {
        for (k = 0; k < M; ++k) {
                foo( );
        }
}
```

N(N + 1)/2

NM

How many foos?

```
void foo(int N) {
        if(N <= 2)
                return;
        foo(N / 2);
}
```

$$T(0) = T(1) = T(2) = 1$$
$$T(n) = 1 + T(n/2) \text{ if } n > 2$$

$$\begin{aligned} T(n) &= 1 + (1 + T(n/4)) \\ &= 2 + T(n/4) \\ &= 2 + (1 + T(n/8)) \\ &= 3 + T(n/8) \\ &= 3 + (1 + T(n/16)) \\ &= 4 + T(n/16) \\ &\dots \\ &\approx \log_2 n \end{aligned}$$

Big-O and friends

- Informal definitions:
  - Given a complexity function $f(n)$,
  - $O(f(n))$ is the set of complexity functions that are *upper bounds* on $f(n)$
  - $\Omega(f(n))$ is the set of complexity functions that are *lower bounds* on $f(n)$
  - $\Theta(f(n))$ is the set of complexity functions that, given the correct constants, correctly describes f(n)
- Example: If $f(n) = 17x^3 + 4x - 12$, then
  - $O(f(n))$ contains x^4, x^5, 2^x, etc.
  - $\Omega(f(n))$ contains 1, x, x^2, $\log x$, $x \log x$, etc.
  - $\Theta(f(n))$ contains x^3

Formal definition of Big-O

- A function $f(n)$ is $O(g(n))$ if there exist positive constants c and N such that, for all $n \geq N$,
  $0 \leq f(n) \leq cg(n)$
- That is, if n is big enough (larger than N—we don't care about small problems), then $cg(n)$ will be bigger than $f(n)$
- Example: $5x^2 + 6$ is $O(n^3)$ because
  $0 \leq 5n^2 + 6 \leq 2n^3$ whenever $n \geq 3$ ($c = 2$, $N = 3$)
  - We could just as well use $c = 1$, $N = 6$, or $c = 50$, $N = 50$
  - Of course, $5x^2 + 6$ is also $O(n^4)$ (loose bound), $O(2^n)$ (loose bound), or simply $O(n^2)$

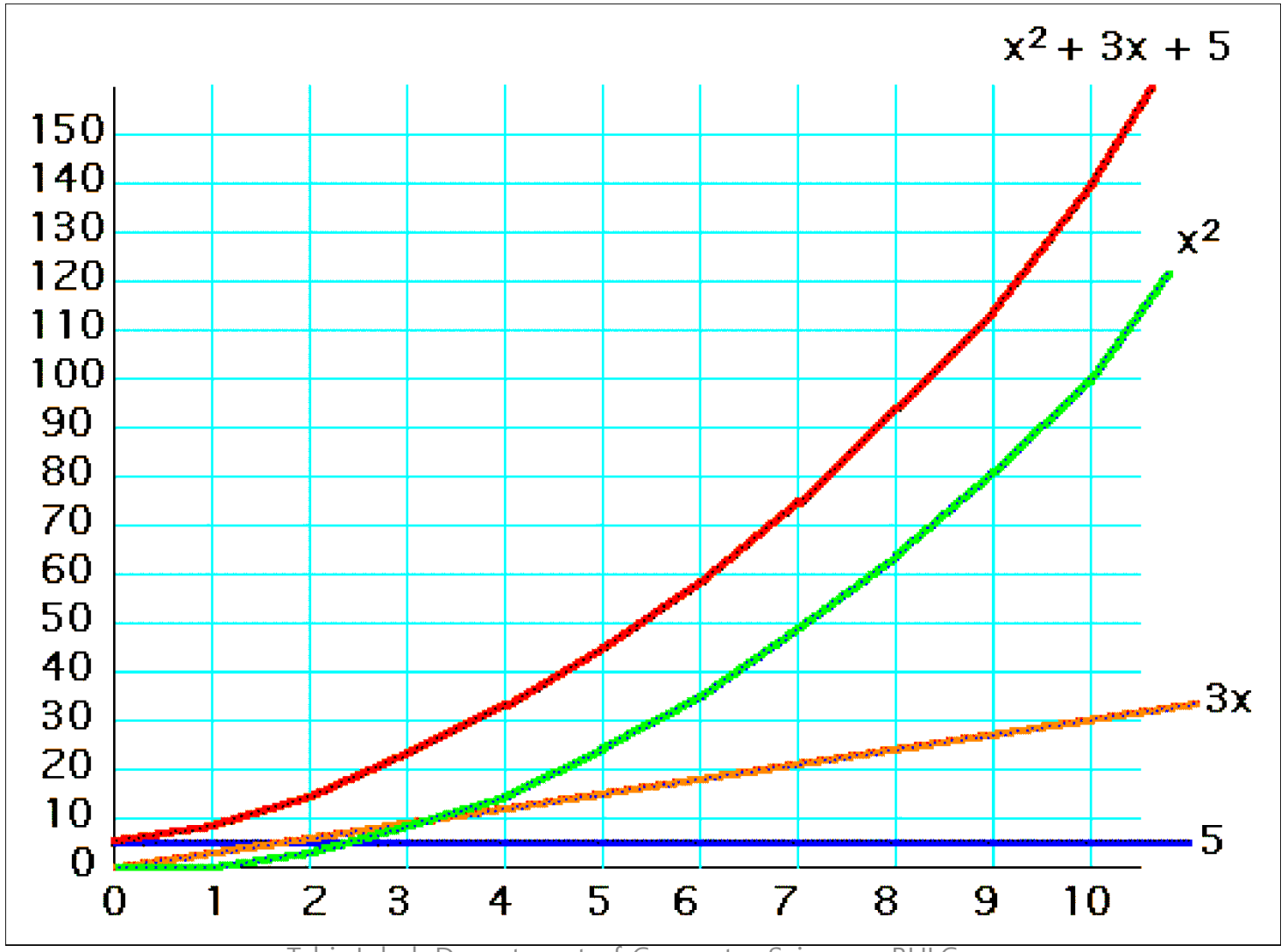
Can we justify Big O notation?

- Big O notation is a *huge* simplification; can we justify it?
  - It only makes sense for *large* problem sizes
  - **For sufficiently large problem sizes, the highest-order term swamps all the rest!**

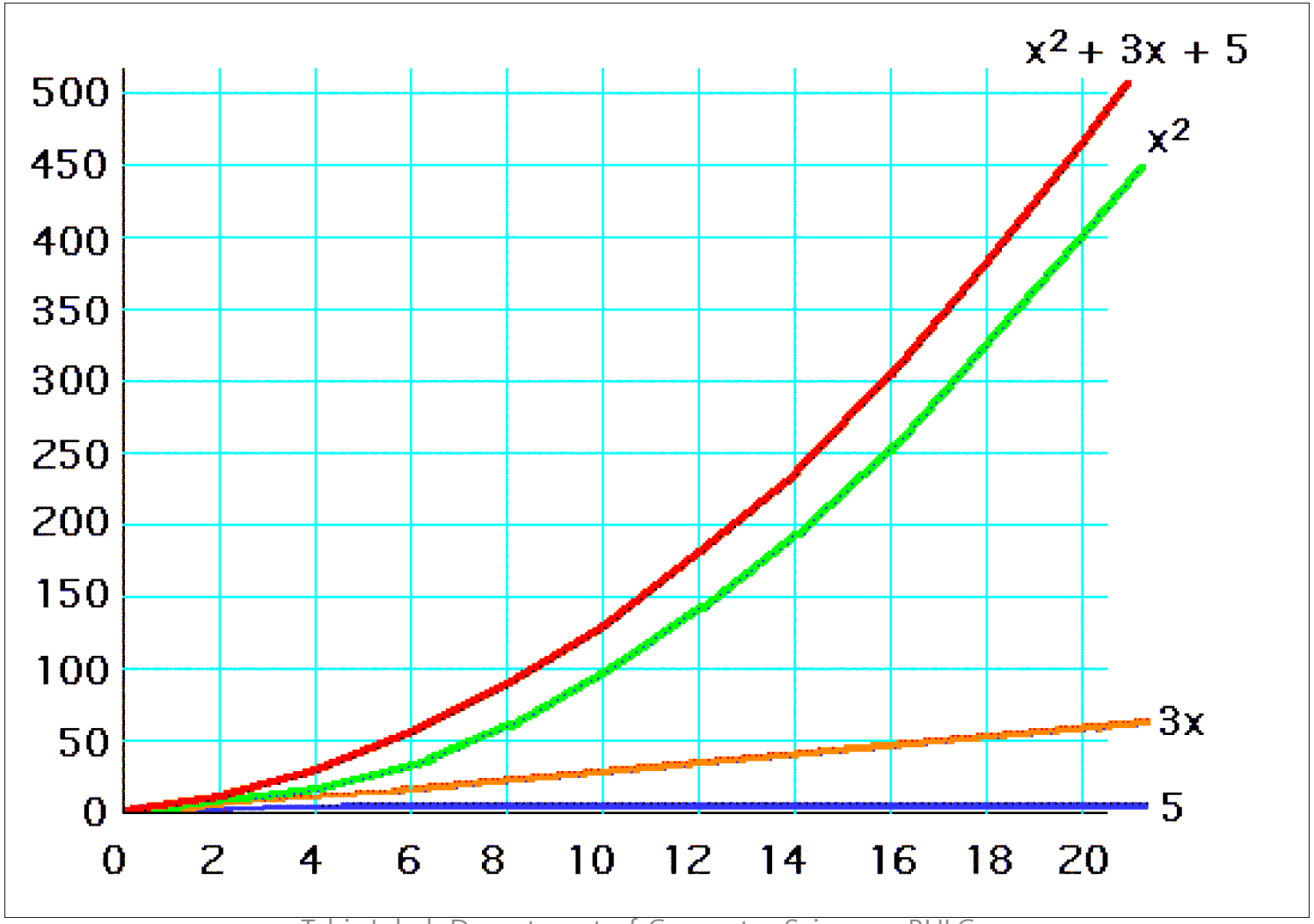
- Consider $R = x^2 + 3x + 5$ as x varies:

$x = 0$	$x^2 = 0$	$3x = 0$	$5 = 5$	$R = 5$
$x = 10$	$x^2 = 100$	$3x = 30$	$5 = 5$	$R = 135$
$x = 100$	$x^2 = 10000$	$3x = 300$	$5 = 5$	$R = 10,305$
$x = 1000$	$x^2 = 1000000$	$3x = 3000$	$5 = 5$	$R = 1,003,005$
$x = 10,000$	$x^2 = 10^8$	$3x = 3*10^4$	$5 = 5$	$R = 100,030,005$
$x = 100,000$	$x^2 = 10^{10}$	$3x = 3*10^5$	$5 = 5$	$R = 10,000,300,005$

$y = x^2 + 3x + 5$, for x=1..10



$y = x^2 + 3x + 5$, for x=1..20



Formal definition of Big-Ω

- A function $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and N such that, for all $n \geq N$, $0 \leq cg(n) \leq f(n)$
- That is, if n is big enough (larger than N—we don't care about small problems), then $cg(n)$ will be *smaller* than $f(n)$
- Example: $5x^2 + 6$ is $\Omega(n)$ because $0 \leq 20n \leq 5n^2 + 6$ whenever $n \geq 4$ ($c=20$, $N=4$)
  - We could just as well use $c = 50$, $N = 50$
  - Of course, $5x^2 + 6$ is also $O(\log n)$, $O(\sqrt{n})$, and even $O(n^2)$

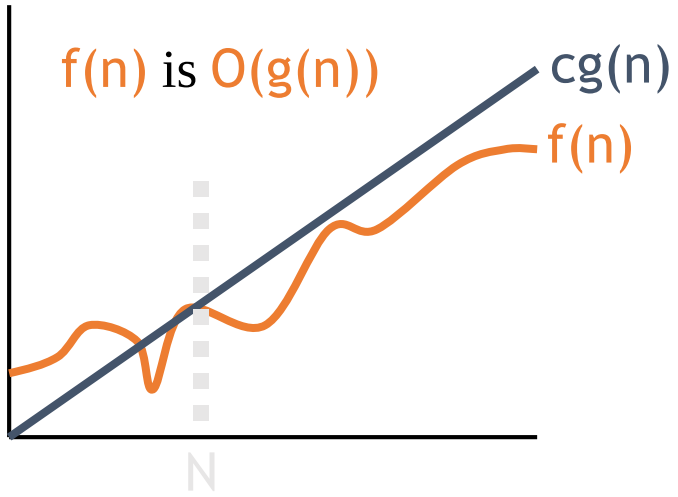
Formal definition of Big-Θ

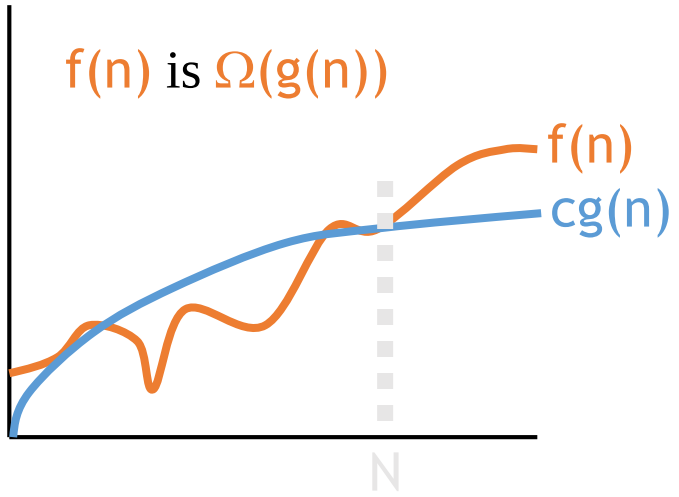
- A function $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1 ***and*** c_2 and N such that, for all $n \geq N$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- That is, if n is big enough (larger than N), then $c_1 g(n)$ will be *smaller* than $f(n)$ and $c_2 g(n)$ will be *larger* than $f(n)$
- In a sense, Θ is the "best" complexity of $f(n)$
- Example: $5x^2 + 6$ is $\Theta(n^2)$ because

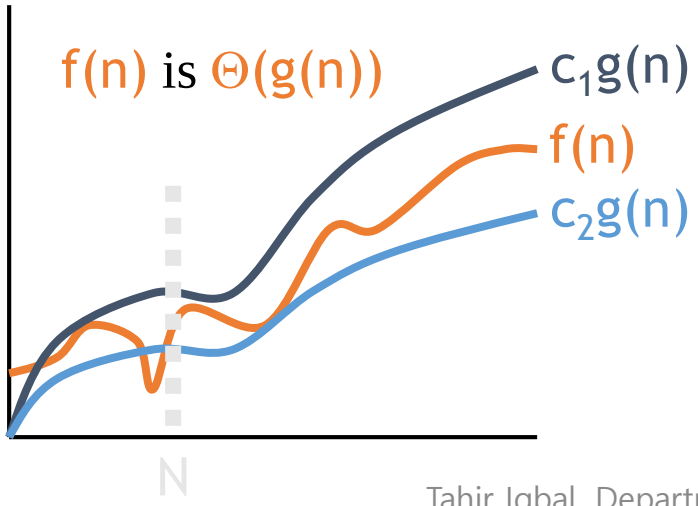
  $n^2 \leq 5n^2 + 6 \leq 6n^2$

  whenever $n \geq 5$ $\quad (c_1 = 1, c_2 = 6)$

Graphs







- Points to notice:
  - What happens near the beginning ($n \leq N$) is not important
  - $cg(n)$ always passes through 0, but $f(n)$ might not (why?)

Analyzing Some Simple Programs

General Rules:

- All basic statements (assignments, reads, writes, conditional testing, library calls) run in constant time: O(1).
- The time to execute a loop is the sum, over all times around the loop, of the time to execute all the statements in the loop, plus the time to evaluate the condition for termination. Evaluation of basic termination conditions is O(1) in each iteration of the loop.
- The complexity of an algorithm is determined by the complexity of the most frequently executed statements. If one set of statements have a running time of $O(n^3)$ and the rest are O(n), then the complexity of the algorithm is $O(n^3)$. This is a result of the Summation Rule.

Fixed-Size Loops

```
for(int suit = Card.CLUBS; suit <= Card.SPADES; suit++)
   {
      for(int value = Card.TWO; value <= Card.ACE; value++)
       {
         myCards[cardNum] = new Card(value, suit);
         cardNum++;
       }
   }
```

- Loops that perform a constant number of iteration are considered to execute in constant time. They don't depend on the size of some data set.

Loops That Work on a Data Set

- Loops like on the previous slide are fairly rare.
- Normally a loop operates on a data set which can vary is size.

```
double minimum(double[] values)
{  int n = values.length;
   double minValue = values[0];
   for(int i = 1; i < n; i++)
      if(values[i] < minValue)
         minValue = values[i];
   return minValue;
}
```

- The number of executions of the loop depends on the length of the array, values. The actual number of executions is (length - 1).
- The run time is O(N).

Loops variable length

Compute the big-Oh running time of the following C++ code segment:

```
for (i = 2; i < n; i++) {
sum += i;
}
```

The number of iterations of a for loop is equal to the top index of the loop minus the bottom index, plus one more instruction to account for the final conditional test.

In this case, we have n - 2 + 1 = n - 1. The assignment in the loop is executed n - 2 times. So, we have (n - 1) + (n - 2) = (2n - 3) instructions executed = O(n).

Note: if the for loop terminating condition is i <= n, rather than i < n, then the number of times the conditional test is performed is: ((top_index + 1) – bottom_index) + 1)

Nested Loops

```
void bubbleSort(double[] data)
{  int n = data.length;
   for(int i = n - 1; i > 0; i--)
     for(int j = 0; j < i; j++)
        if(data[j] > data[j+1])
        {   double temp = data[j];
            data[j] = data[j + 1];
            data[j + 1] = temp;
        }
}
```

- Number of executions?

Example

Consider the sorting algorithm shown below. Find the number of instructions executed and the complexity of this algorithm.

```
1)      for (i = 1; i < n; i++) {
2)              SmallPos = i;
3)              Smallest = Array[SmallPos];
4)              for (j = i+1; j <= n; j++)
5)                      if (Array[j] < Smallest) {
6)                              SmallPos = j;
7)                              Smallest  = Array[SmallPos]
                }
8)              Array[SmallPos] = Array[i];
9)             Array[i] = Smallest;
        }
```

The total computing time is:

$$
\begin{aligned}
T(n) &= (n) + 4(n-1) + n(n+1)/2 - 1 + 3[n(n-1)/2] \\
&= n + 4n - 4 + (n^2 + n)/2 - 1 + (3n^2 - 3n)/2 \\
&= 5n - 5 + (4n^2 - 2n)/2 \\
&= 5n - 5 + 2n^2 - n \\
&= 2n^2 + 4n - 5 \\
&= O(n^2)
\end{aligned}
$$

Example

Consider the sorting algorithm shown below. Find the number of instructions executed and the complexity of this algorithm.

```
1)      for (i = 0; i < n-1; i++) {
2)              SmallPos = i;
3)              Smallest = Array[SmallPos];
4)              for (j = i+1; j <= n-1; j++)
5)                      if (Array[j] < Smallest) {
6)                              SmallPos = j;
7)                              Smallest  = Array[SmallPos]
                }
8)              Array[SmallPos] = Array[i];
9)             Array[i] = Smallest;
        }
```

The total computing time is:

$$
\begin{aligned}
T(n) &= (n) + 4(n-1) + n(n+1)/2 - 1 + 3[n(n-1) / 2] \\
&= n + 4n - 4 + (n^2 + n)/2 - 1 + (3n^2 - 3n) / 2 \\
&= 5n - 5 + (4n^2 - 2n) / 2 \\
&= 5n - 5 + 2n^2 - n \\
&= 2n^2 + 4n - 5 \\
&= O(n^2)
\end{aligned}
$$

Example

Consider the sorting algorithm shown below. Find the number of instructions executed and the complexity of this algorithm.

```
1)      for (i = 1; i < n; i++) {
2)              sPos = i;
3)              min = Array[sPos];
4)              for (j = i+1; j <= n; j++)
5)                      if (Array[j] < min) {
6)                              sPos = j;
7)                              min  = Array[sPos]
                }
8)              Array[sPos] = Array[i];
9)             Array[i] = min;
        }
10)     for (i = n ; i >= 1; i --)
11)         print (Array[ i ]);
```

Example

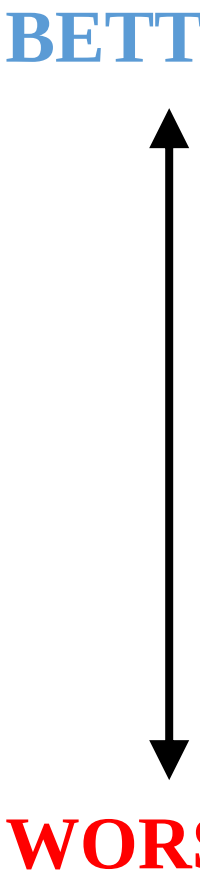
The following program segment initializes a two-dimensional array A (which has n rows and n columns) to be an n x n identity matrix – that is, a matrix with 1's on the diagonal and 0's everywhere else. More formally, if A is an n x n identity matrix, then:

A x M = M x A = M, for any n x n matrix M.

What is the complexity of this C++ code?

```
1)      cin >> n;          // Same as: n = GetInteger();
2)      for (i = 1; i < n; i ++)
3)              for (j = 1; j < n; j ++)
4)                      A[i][j] = 0;
5)      for (i = 1; i < n; i ++)
6)              A[i][i] = 1;
```

Common time complexities



- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time