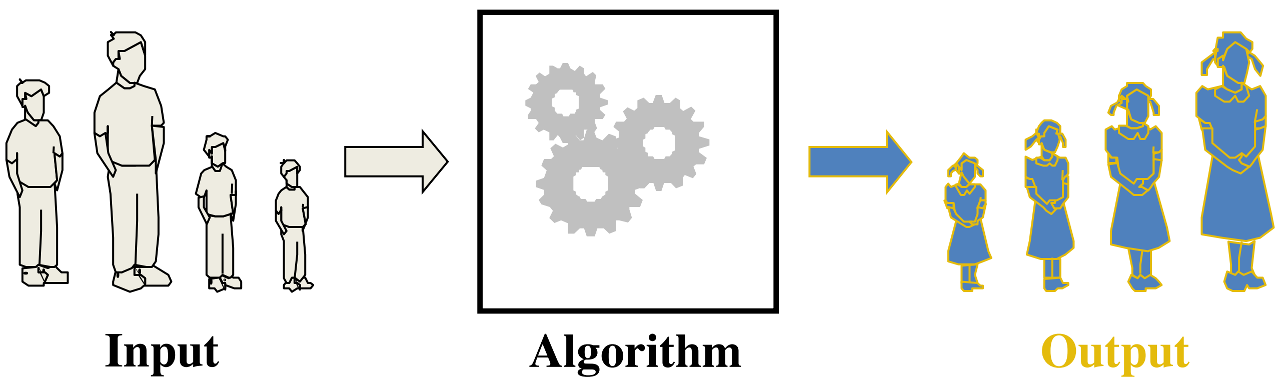
Data Structures & Algorithms

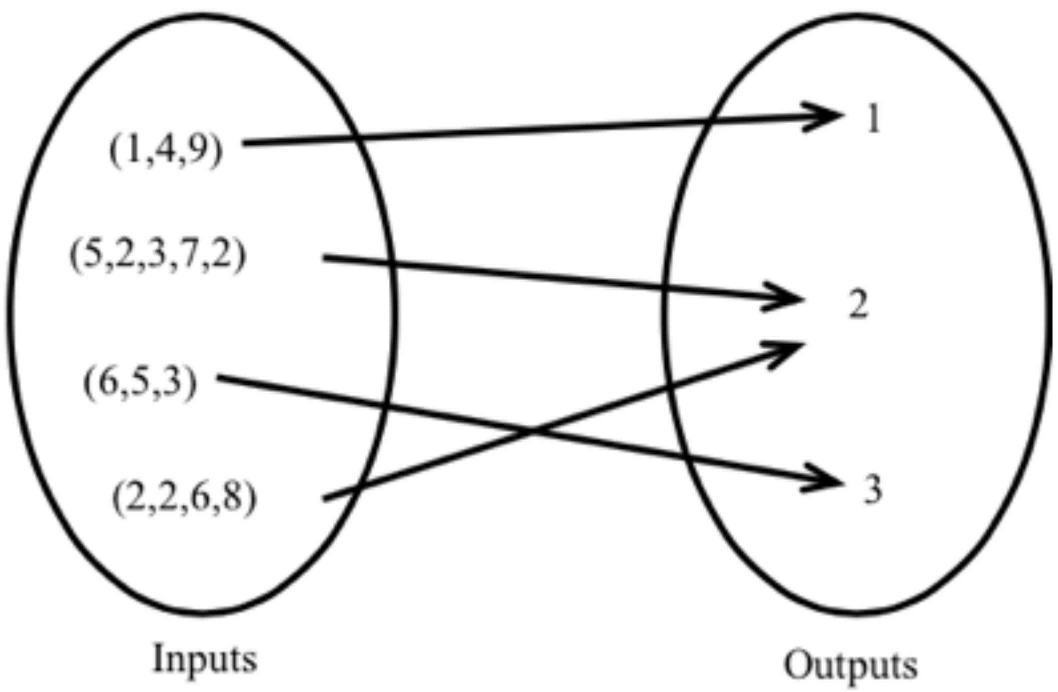
Adil M. Khan
Professor of Computer Science
Innopolis University

Algorithm Analysis

Algorithm



A more specific example: Find Minimum!



Think of a few more examples as an exercise!

Algorithm

- Another way
  - A tool to solve a well-defined computational problem
  - The statement of the problem defines the desired input/output relationship

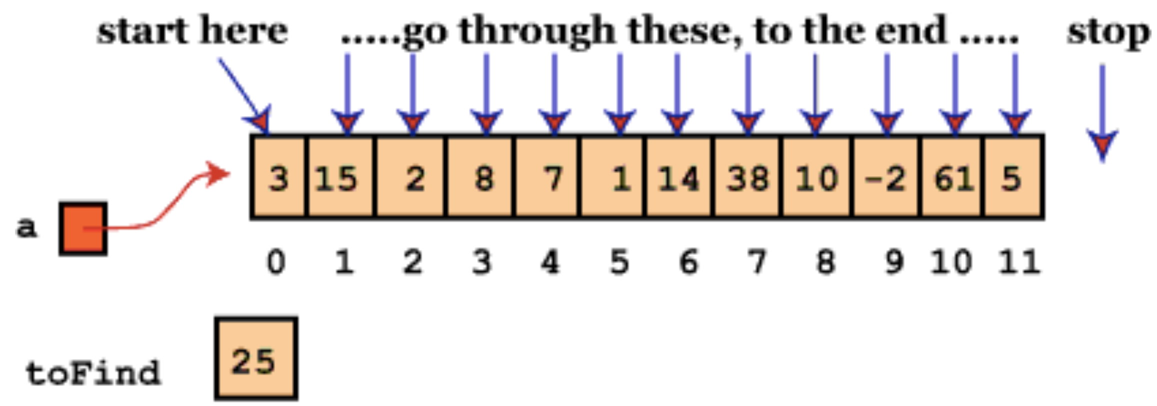
"Sorting a sequence of numbers into non-decreasing order"

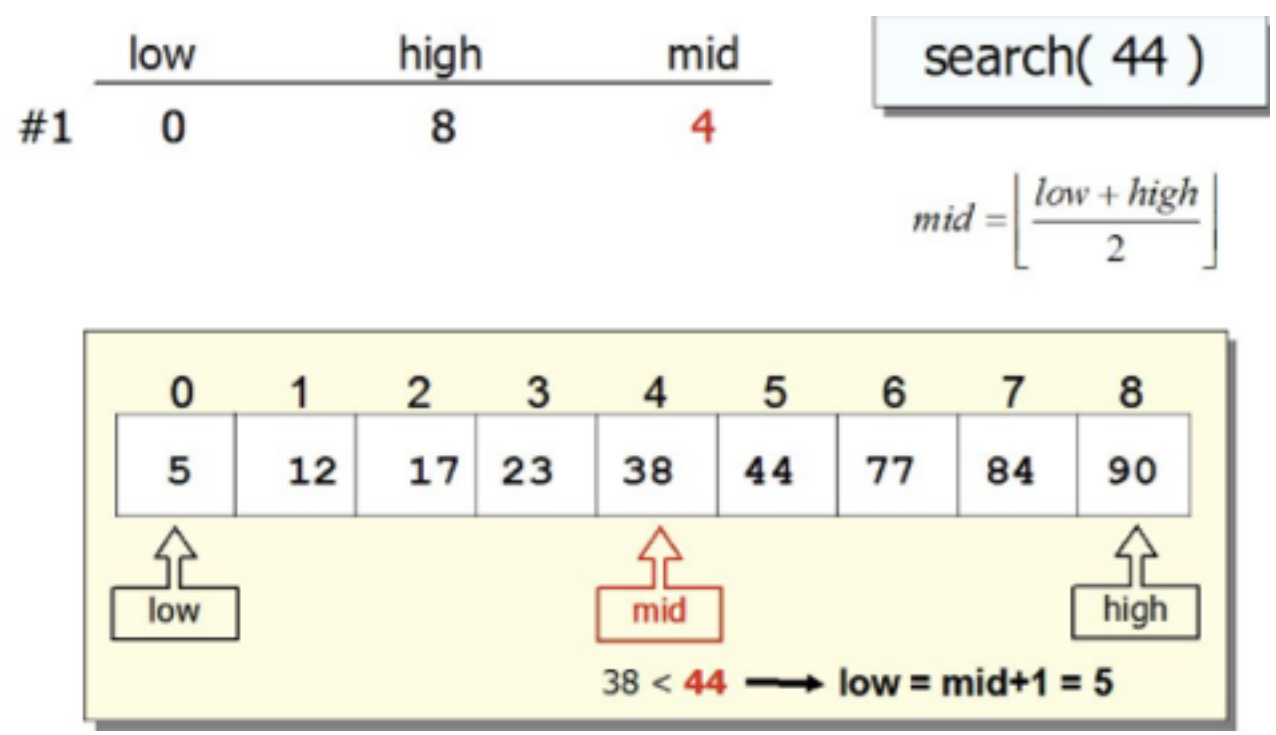
Two Characteristics of Algorithmic Problems

- They have practical applications
- They have many candidate solutions

Why Analyze Algorithms?

- Allows us to:
  - Compare the merits of two alternative approaches to a problem we need to solve
  - Determine whether a proposed solution will meet required resource constraints before we invest money and time coding





Performed before coding!

Analyzing Algorithms

- How do we analyze algorithms?

  Complexity Analysis: predicting the resources that an algorithm requires!

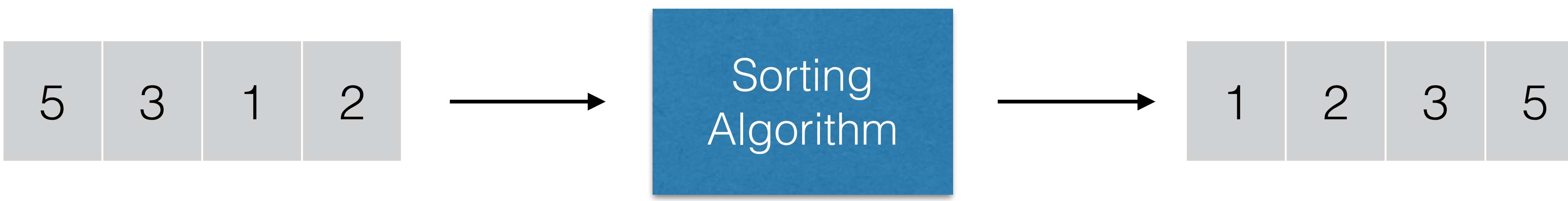
- ***Time Complexity:*** amount of time that an algorithm takes to run to completion

- ***Space Complexity:*** amount of memory that an algorithm needs to run to completion

Time Complexity

How to Measure Time Complexity?

- As mentioned earlier, an algorithm can be considered as a black box that transforms input objects into output objects



- Consider the amount of ***time* (T)** consumed as a ***function of the input size (n) — T(n)***

Input Size (n)

- The n could be
  - The number of items in a container
  - The length of a string or file
  - The number of digits (or bits) in an integer
  - The degree of a polynomial

How to Measure Time Complexity?

- Even for inputs of the same size, the time consumed can be very different

  Example: an algorithm that finds the first prime number in an array by scanning it left to right

  What can happen?

Think of a few more examples as an exercise!

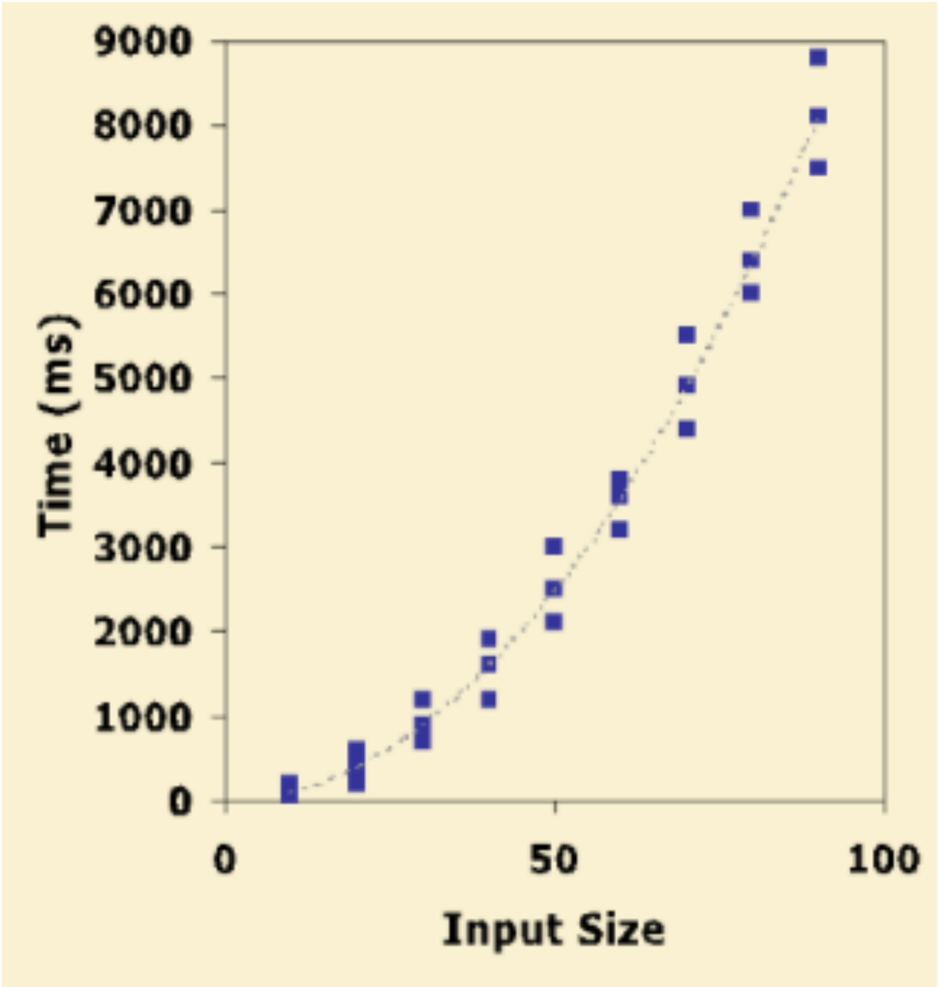
How to Measure Time Complexity?

- Analyze running time for the
  - ***best case:*** usually useless
  - ***average case:*** very difficult to determine
  - ***worst case:*** a safer choice

Why is the worst case a safer choice?

How to Measure Worst-Case Time Complexity?

- ***Experimental Evaluation***
  - Write a program implementing the algorithm
  - Run it with inputs of varying size and composition
  - Measure the actual running time
  - Plot the results



What is wrong with this approach?

How to Measure Worst-Case Time Complexity?

- ***Theoretical Approach***
  - Pseudocode description of the algorithm instead of an implementation
  - Characterize running time as a function of the input size, ***n***
  - Allows us to evaluate the running time of an algorithm independent of the hardware/software environment

Pseudocode

- A high-level description of an algorithm
- More structured than English prose
- Less detailed than a program

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
Input: array ***A*** of ***n*** integers
Output: maximum element of ***A***

currentMax $\leftarrow$ ***A***[0]
for ***i*** $\leftarrow$ 1 **to** ***n*** $-$ 1 **do**
if ***A***[*i*] > ***currentMax*** **then**
currentMax $\leftarrow$ ***A***[***i***]
return ***currentMax***

Measuring Time Complexity

- Consider this statement in your algorithm

  x = x + 1;

- What we want to measure is

  - ***Execution time:*** The time a single execution of this statement would take

  - ***Frequency count:*** The number of times it is executed

Measuring Time Complexity

- Total time taken is ***approximately*** the product of execution time and the frequency count

- However, execution time is tied to the underlying machine and the compiler, so we neglect it and only concentrate on the frequency count

- Frequency count will vary based on the size of the data set used, that is, ***input to the algorithm***

Measuring Time Complexity

- Example 1

```
x = x + 1
```

- Example 2

```
for i = 1 to n
    x = x + 1
```

- Example 2

```
for i = 1 to n
    for j = 1 to n
        x = x + 1
```

Determine the frequency count for the assignment statement in these three examples!

Measuring Time Complexity

- Example 1
  - There is no loop
  - frequency count is ***1***
- Example 2
  - inside a for-loop
  - frequency count is ***n*** — statement is execute ***n*** times
- Example 3
  - Nested loops — loop within a loop
  - frequency count is ***n^2***

Primitive Operations

- x = x + 1 is an example of primitive operations
- Some other examples include

Primitive Operations Examples	
Evaluating and expression	x^2+e^y
Indexing into an array	A[5]
Calling a method	mySort(A,n)
Returning from a method	return(cnt)

Measuring Time Complexity

- To measure the time complexity
  - We count the total number of primitive operations for an algorithm as a function of the input size ***T(n)***

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	1
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	
15	
16	
17	

Measuring Time Complexity

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	n-1
15	n-1
16	n-1
17	1

$$T(n) = 5n + 5$$

Growth Rate

- Changing hardware/software environment
  - Affects ***T(n)*** by a constant factor
  - Does not alter the growth rate of ***T(n)***
- Thus, we focus on the big-picture which is the growth rate of an algorithm
- PrintArray algorithm (Example) has a linear growth rate, that is, it grows proportionally with ***n***

Growth Rate

- Remember, growth rate is not affected by
  - constant factors
  - lower-order terms
- Examples:
  - $10^2 n + 10^5$ is a linear function
  - $10^2 n^2 + 10^5 n$ is a quadratic function

Growth Rate

- Why is it not affected by the constant factors and the lower order terms?
- ***6n vs. 3n*** — getting a computer twice as fast makes the former same as the latter
- ***2n vs. 2n + 8*** — difference becomes insignificant when n becomes larger and larger
- x^3 **vs.** kx^2 — the former will always eventually overtake the latter no matter how big your make ***k***

Measuring Time Complexity

- So for our example: $T(n) = 5n + 5$
- But we just learned that constant terms don't matter
- Thus $T(n) = n.$

Big-Oh Notation

- Or asymptotic analysis
- The big-oh notation is widely used to characterize running times and space bounds of an algorithm
- The big-oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth

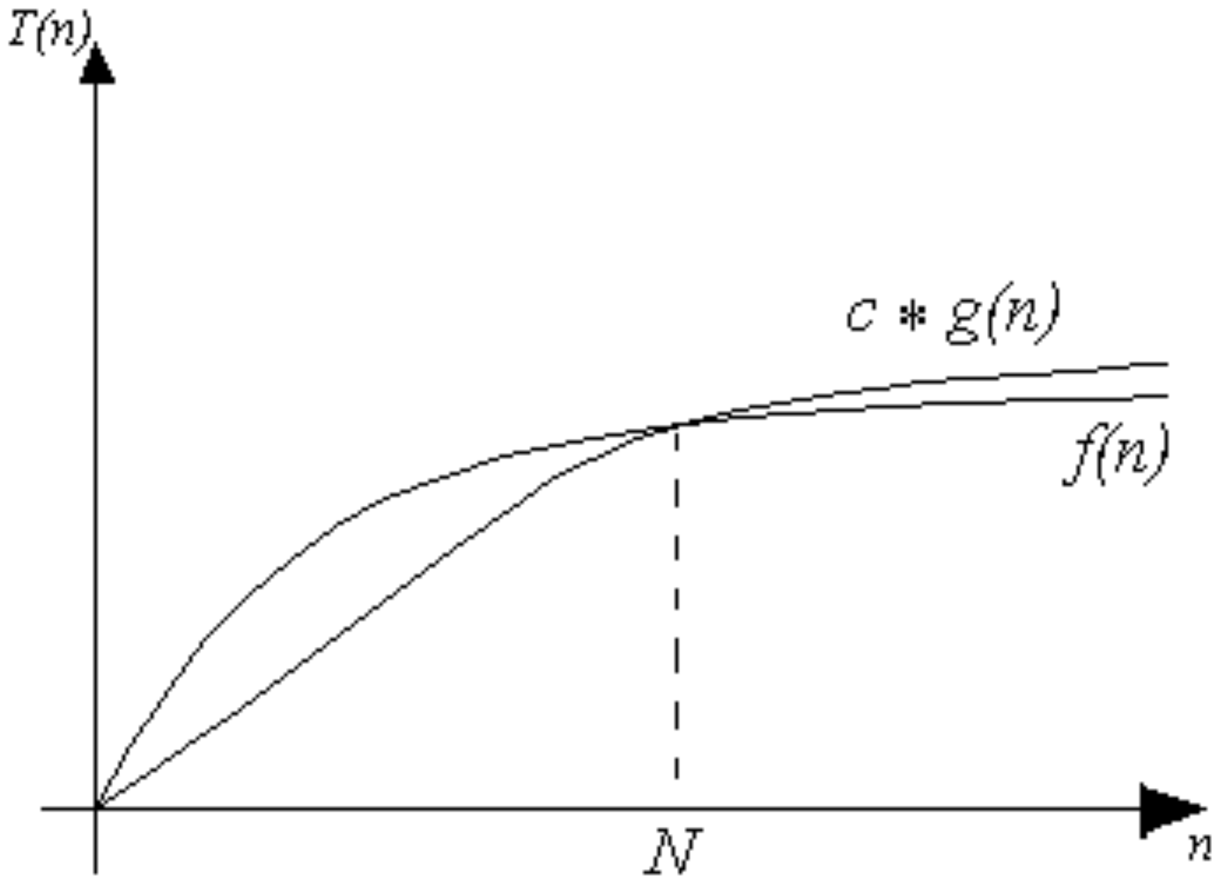
Big-Oh Notation

- Given functions ***f(n)*** and ***g(n)***, we say that ***f(n)*** is in ***O(g(n))*** if there exist two constants ***c*** and ***k*** such that

$$f(n) \leq cg(n) \;\; for \; all \; n \geq k$$

Big-Oh Notation

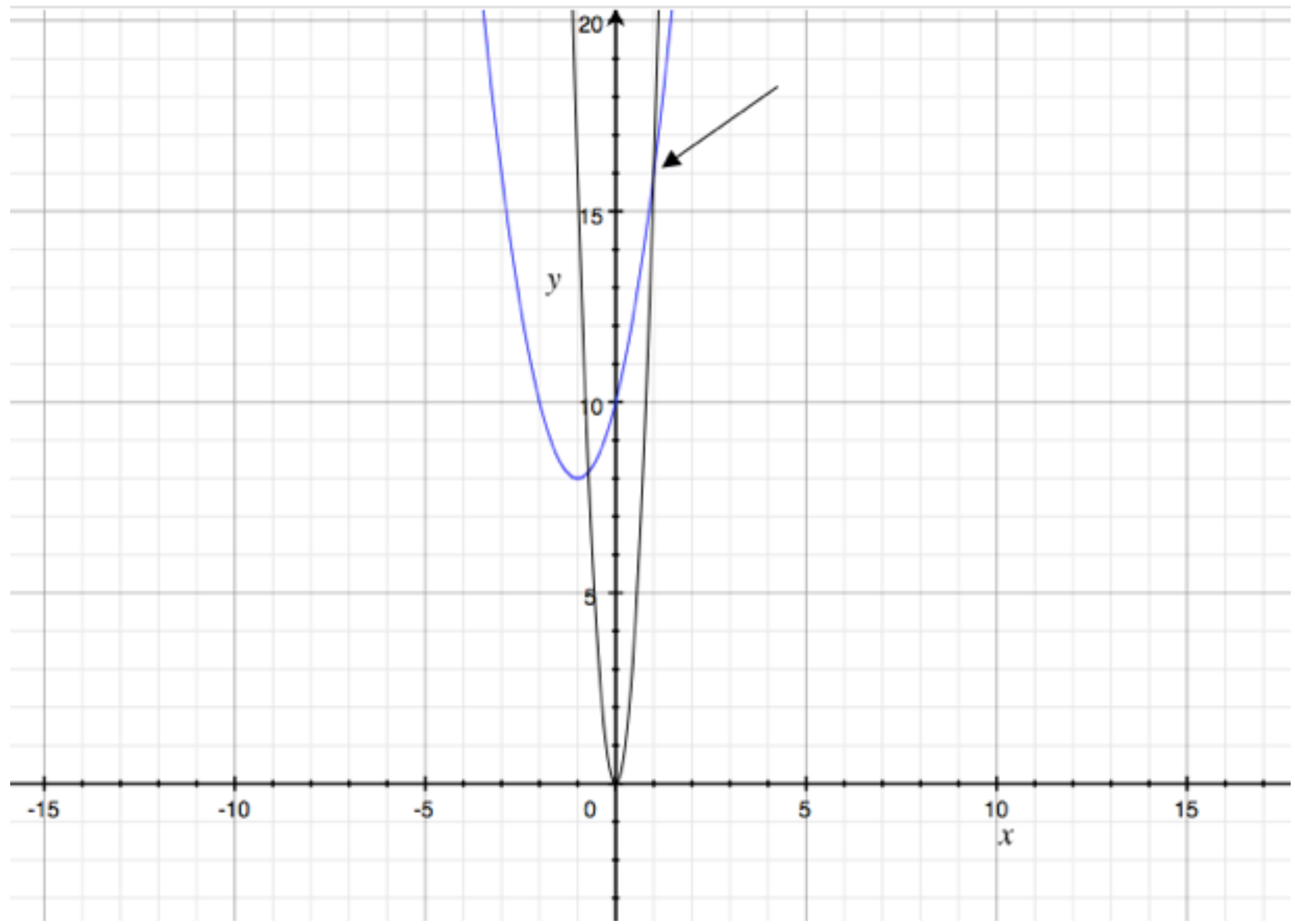
- In other words, ***f(n)*** is bounded above by a constant times of ***g(n)***



Here, ***N*** is representing ***k***

Big-Oh Notation

Suppose $f(n) = 2n^2+4n+10$
and $f(n) = O(g(n))$ where $g(n)=n^2$



Proof:

$f(n) = 2n^2+4n+10$

$f(n) \leq 2n^2 + 4n^2 + 10n^2$ for $n \geq 1$

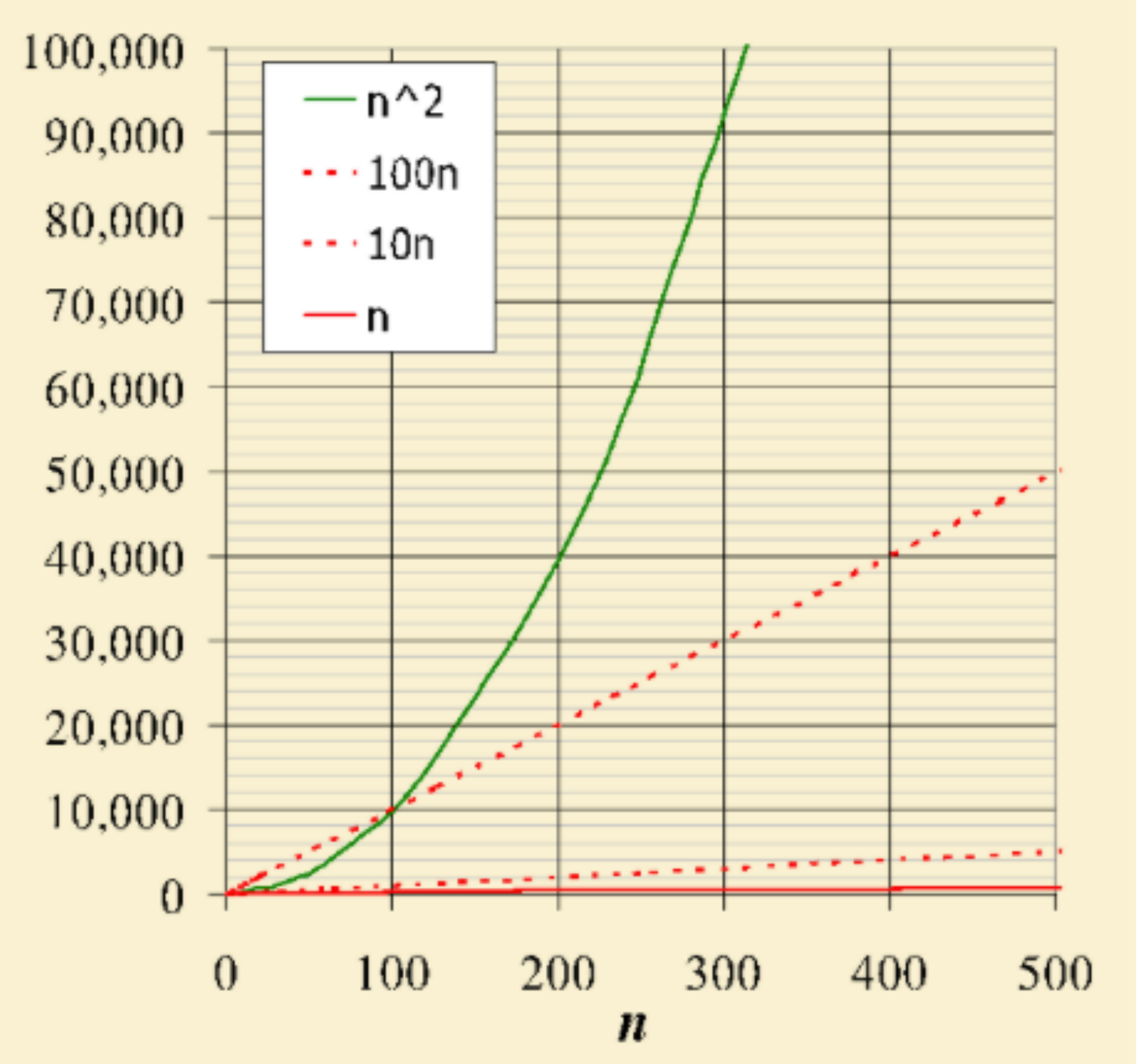
$f(n) \leq 16n^2$

$f(n) \leq 16g(n)$ Where c = 16 and $k = 1$

Big-Oh Notation

- n^2 is not $O(n)$

- $n^2 \le cn$
- $n \le c$
- The above inequality cannot be satisfied since c must be a constant



Big-Oh and Algorithm Analysis

- *f(n)* will normally represent the computing time of some algorithm
    - Time complexity ***T(n)***
- *f(n)* can also represent the amount of memory an algorithm needs to run
    - Space complexity ***S(n)***

Big-Oh and Time Complexity

- If an algorithm has a time complexity of ***O(g(n))*** it means that its execution will take no longer than a constant times of ***g(n)***
- More formally, *g(n)* is an asymptotic upper bound for *f(n)*

Time Complexity

$O(1)$	Constant (computing time)
$O(n)$	Linear (computing time)
$O(n^2)$	Quadratic (computing time)
$O(n^3)$	Cubic (computing time)
$O(2^n)$	Exponential (computing time)
$O(\log n)$	is faster than $O(n)$ for sufficiently large n
$O(n \log n)$	is faster than $O(n^2)$ for sufficiently large n

Time Complexity

n	T(n)							
	1	**logn**	**n**	**nlogn**	**n^2**	**n^3**	**2^n**	**n!**
10	1 mic-sec	3.32 mic-sec	10 mic-sec	33.2 mic-sec	100 mic-sec	1 mil-sec	1.02 mil-sec	3.63 sec
20	1 mic-sec	4.32 mic-sec	20 mic-sec	86.4 mic-sec	400 mic-sec	8 mil-sec	1.05 sec	771 cent
50	1 mic-sec	5.64 mic-sec	50 mic-sec	282 mic-sec	2.5 mil-sec	125 mil-sec	35.7 years	9 * 10^50 cent
100	1 mic-sec	6.64 mic-sec	100 mic-sec	664 mic-sec	10 mil-sec	1 sec	4 * 10^14 cent	-
1000	1 mic-sec	9.97 mic-sec	1 mil-sec	9.97 mil-sec	1 sec	16.7 min	-	-
1000000	1 mic-sec	19.9 mic-sec	1 sec	19.9 sec	11.57 days	317 cent	-	-

Time Complexity

$f1(n) = 10\ n + 25\ n^2$	$O(n^2)$
$f2(n) = 20\ n \log n + 5\ n$	$O(n \log n)$
$f3(n) = 12\ n \log n + 0.05\ n^2$	$O(n^2)$
$f4(n) = n^{1/2} + 3\ n \log n$	$O(n \log n)$

Time Complexity

Arithmetic of Big-Oh

if

$T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

then

$T_1(n) + T_2(n) = O(max(f(n), g(n))$

Time Complexity

Arithmetic of Big-Oh

if

$f(n) \leq g(n)$

then

$O(f(n) + g(n)) = O(g(n))$

Time Complexity

Arithmetic of Big-Oh

if

$T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

then

$T_1(n)\ T_2(n) = O(f(n)\ g(n))$

Space Complexity

- Determine how much space an algorithm requires by analyzing its storage requirements as a function of the input size

- ***Example:***

  - Let's say, our algorithm reads a stream of ***n*** characters

  - But always <u>stores a constant number</u> of them

  - then, its space complexity is ***O(1)***

Space Complexity

- ***Another Example:***
  - Let's say, our algorithm reads a stream of ***n*** characters
  - and <u>stores all</u> of them
  - then, its space complexity is ***O(n)***

Space Complexity

- ***Exercise:***

  - Let’s say, our algorithm reads a stream of ***n*** characters

  - and <u>stores all</u> of them, and each record results in the creation of a <u>constant number</u> of other records

  - then, its space complexity is ?

Space Complexity

- ***Another Exercise:***

  - Let’s say, our algorithm reads a stream of ***n*** characters

  - and <u>stores all</u> of them, and each record results in the creation of a number of new records — the <u>number is proportional to the size of the data</u>

  - then, its space complexity is ?

Time-Space Tradeoff

- Generally, decreasing the time complexity of an algorithm results in increasing its space complexity — and vice versa
- This is called the time-space tradeoff
- ***Example:*** Storing a sparse matrix as a two-dimensional linked list vs. a two-dimensional array

Big-Oh Notation

- Gives us the upper bound - worst case time complexity
- What if we are interested in:
  - Average case time complexity
  - Best case time complexity
- Especially when they differ significantly

Big-Omega & Big-Theta

Big-Omega

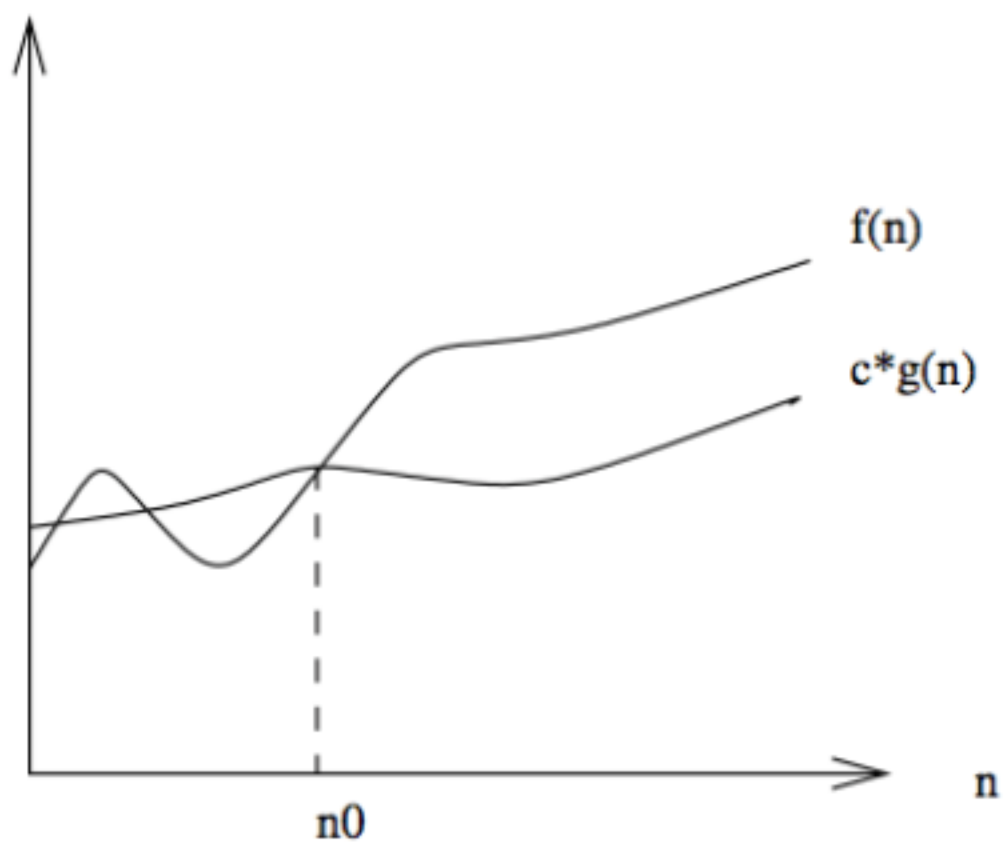
- Asymptotic lower bound

- Given functions ***f(n)*** and ***g(n)***, we say that ***f(n)*** is in ***big-omega*** of g(n) if there exist two constants ***c*** and ***k*** such that

$$f(n) \geq c g(n), \; for \; all \; n \geq k$$

Big-Omega

- In other words, ***f(n)*** is bounded below by a constant times of ***g(n)***



Here, ***n0*** is representing ***k***

- Lower bounds are useful because they say that an algorithm requires **at least** so much time

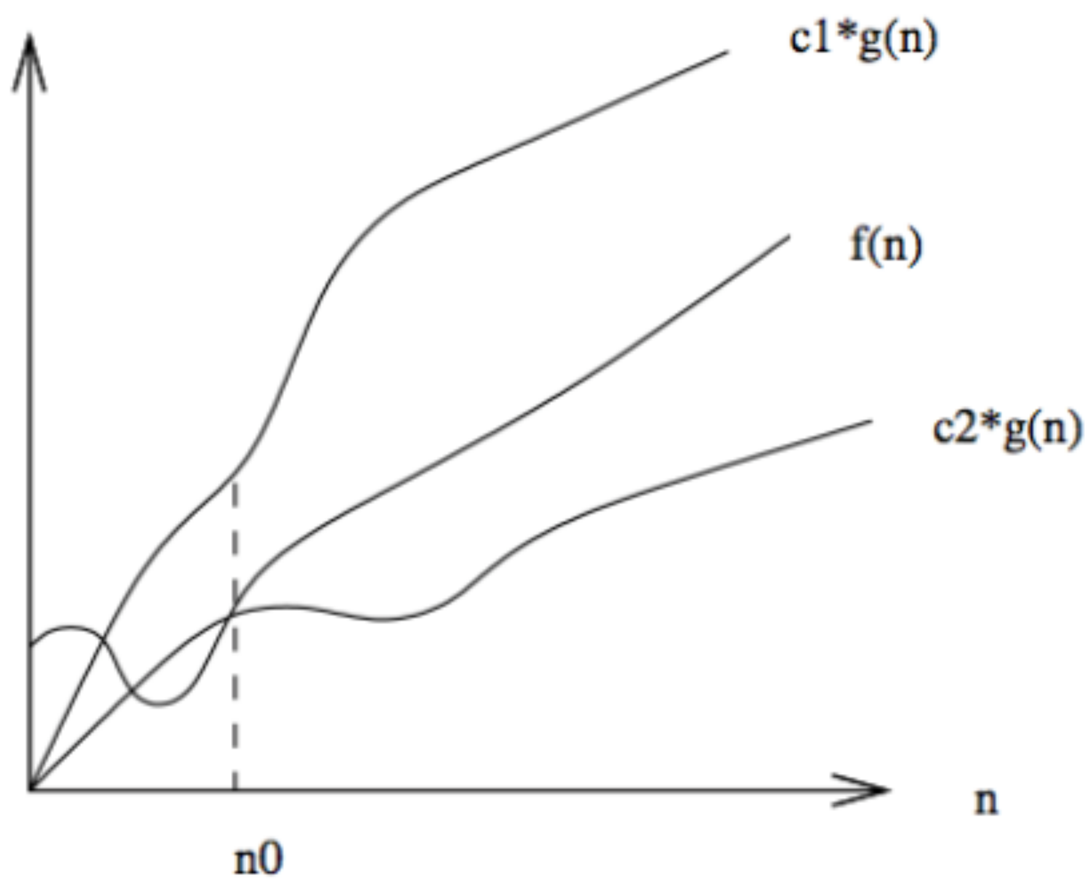
Big-Theta

- Given functions ***f(n)*** and ***g(n)***, we say that ***f(n)*** is ***big-theta*** of g(n) if there exist constants ***c1, c2*** and ***k*** such that

$$f(n) \leq c_1 g(n), \, and, \, f(n) \geq c_2 g(n) \quad for\ all\ n \geq k$$

Big-Theta

- In other words, ***f(n)*** is bounded above by **c1** times of ***g(n)*** and below by **c2** times of ***g(n)***



Here, ***n0*** is representing ***k***

Putting Them Together