WIA2005: Algorithm Design and Analysis

Lecture 1: Introduction to Algorithm Design & Analysis Fundamentals (Pt1)

Dr. Asmiza A. Sani

Semester 2, Session 2024/25

Learning Objectives

- The student will:
  - Know the concept of algorithm.
  - Understand and analyse time complexity of iterative algorithm.

What is Algorithm?

- Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

- We can also view an algorithm as a tool for solving a well-specified computational problem.

Why study algorithm?

Old roots, new opportunities.

Internet. Web search, packet routing, distributed file sharing, ...

Biology. Human genome project, protein folding, ...

Computers. Circuit layout, file system, compilers, ...

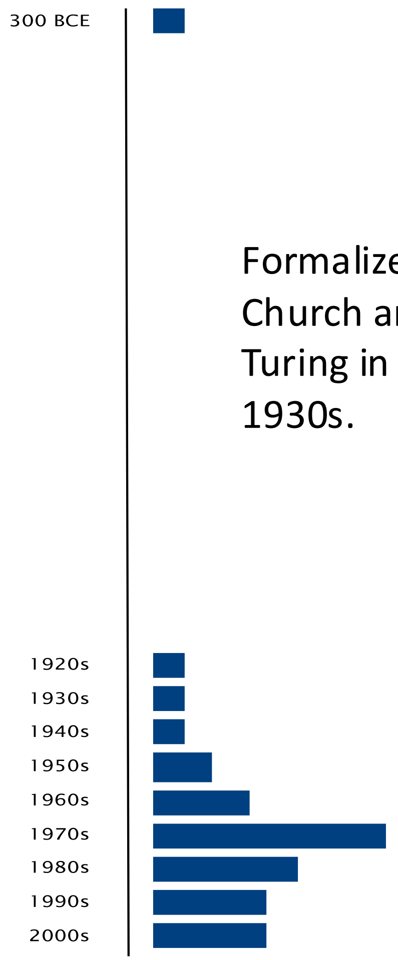
Computer graphics. Movies, video games, virtual reality, ...

Security. Cell phones, e-commerce, voting machines, ...

Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

Social networks. Recommendations, news feeds, advertisements, ...

Physics. N-body simulation, particle collision simulation, ...



Formalized by Church and Turing in 1930s.

Their impact is broad and far-reaching!

Why study algorithm?

Algorithms help us to understand scalability.

Performance often draws the line between what is feasible and what is impossible.

Algorithmic mathematics provides a language for talking about program behavior.

Performance is the currency of computing.

The lessons of program performance generalize to other computing resources.

Properties of Algorithm

- 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.

Analysing an Algorithm

- An algorithm can be analysed for different properties:
    - Time
    - Space
    - Network consumption
    - Power consumption
    - CPU registers

*For this course, we will only be focusing on **time analysis.***

How to describe how fast an algorithm is?

- Example 1: Both of the following codes gives the same output. Which one is a faster Python code?

Code 1

```
sum = 0;
for index in range(201):
  sum = sum + index
print(sum)
```

Code 2

```
sum = (200/2)*(200+1);
```

How to describe how fast an algorithm is?

- Example 2: Both of the following codes gives the same output. Which one is a faster Javascript code?

Code 1

```
function reverse(s) {
 var o = ''; for (var i = s.length - 1; i >= 0; i--)
 o += s[i];
 return o; }
```

Code 2

```
function reverse(s) {
  return s.split('').reverse().join(''); }
```

What is the formal notation to describe running times of the algorithm in the previous slide?

How to describe how fast an algorithm is?

- Normally, what we probably do is run both programs and see how long it takes to execute on a given input.
- Probably observe:
  - One code is better than the other on certain input.
  - One code is better than the other on a different machine.

Difficult to generalise about the code performance that is independent from its implementation language and platform specific execution time.

- So instead, we will formally analyse algorithm based on the running time (or space) in terms of the input size.
  - Asymptotic analysis using Asymptotic Notation

Running time complexity – The Asymptotic analysis

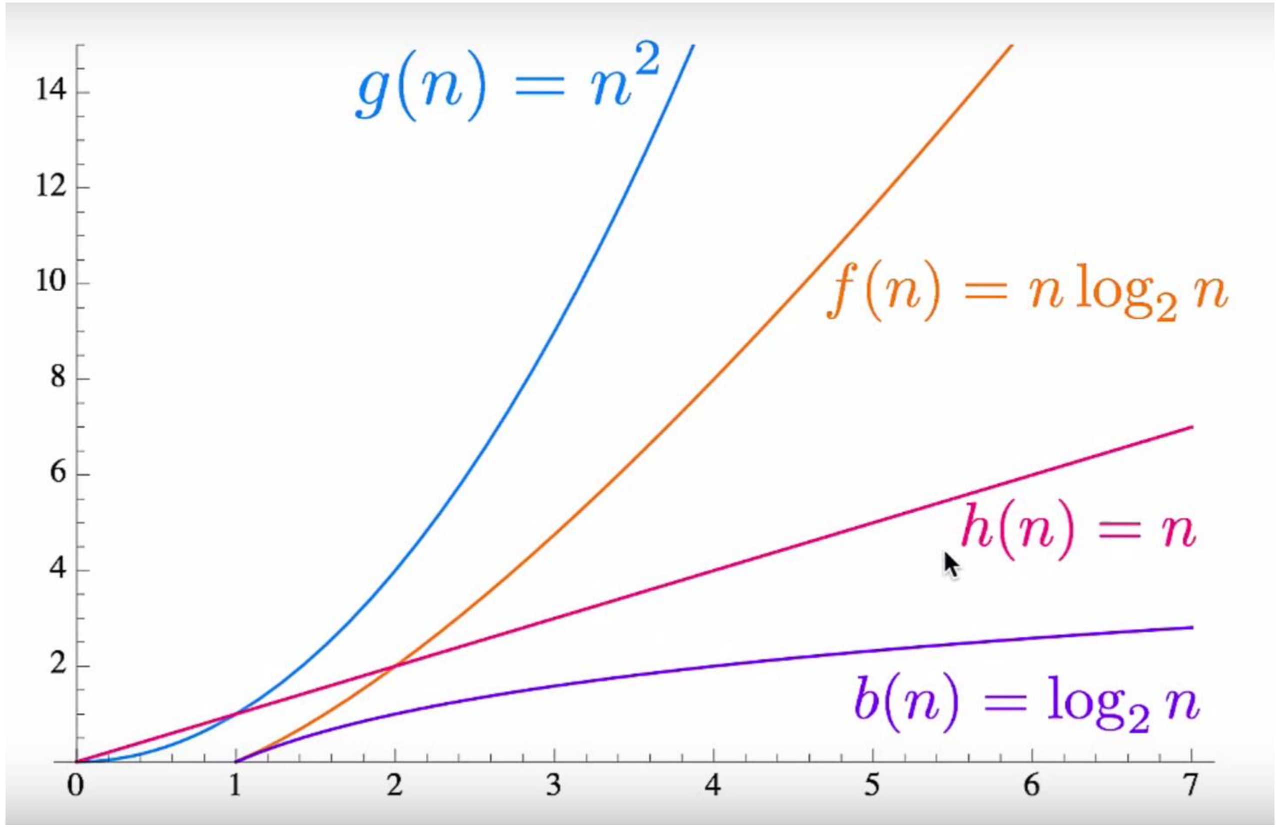
Growth of Functions

- The properties to be observed in algorithm analysis is rate of growth, or order of growth, of the running time
  - Rate of growth: The rate at which running time increases as function of input.
  - Represented as T(n).

Time function characteristics

- Constant
- Logarithmic
- Linear
- Quadratic
- Cubic
- Exponential

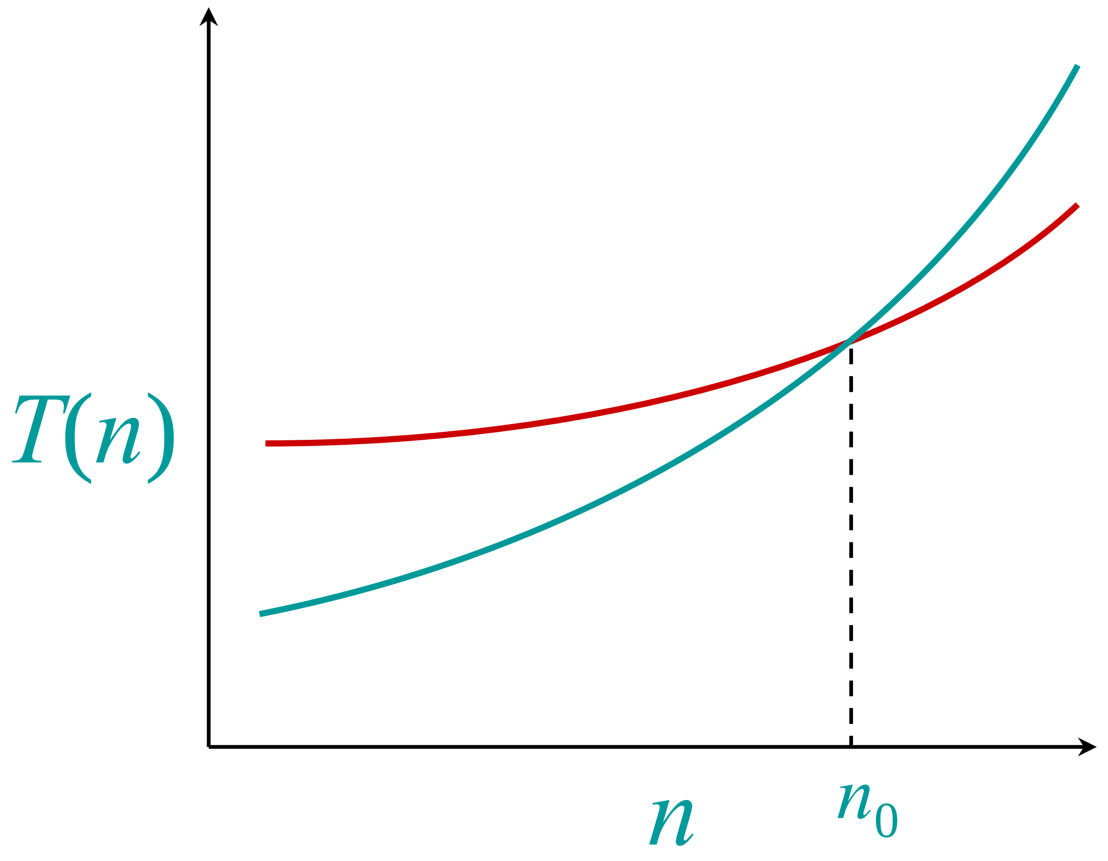
$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots .\ 2^n < 3^n < n^n$



Growth of Functions

- Let's say, we have 2 functions:
  - $f(n) = n^2$ (red)
  - $f(n) = n^3$ (green)

- Which one has the higher growth rate?



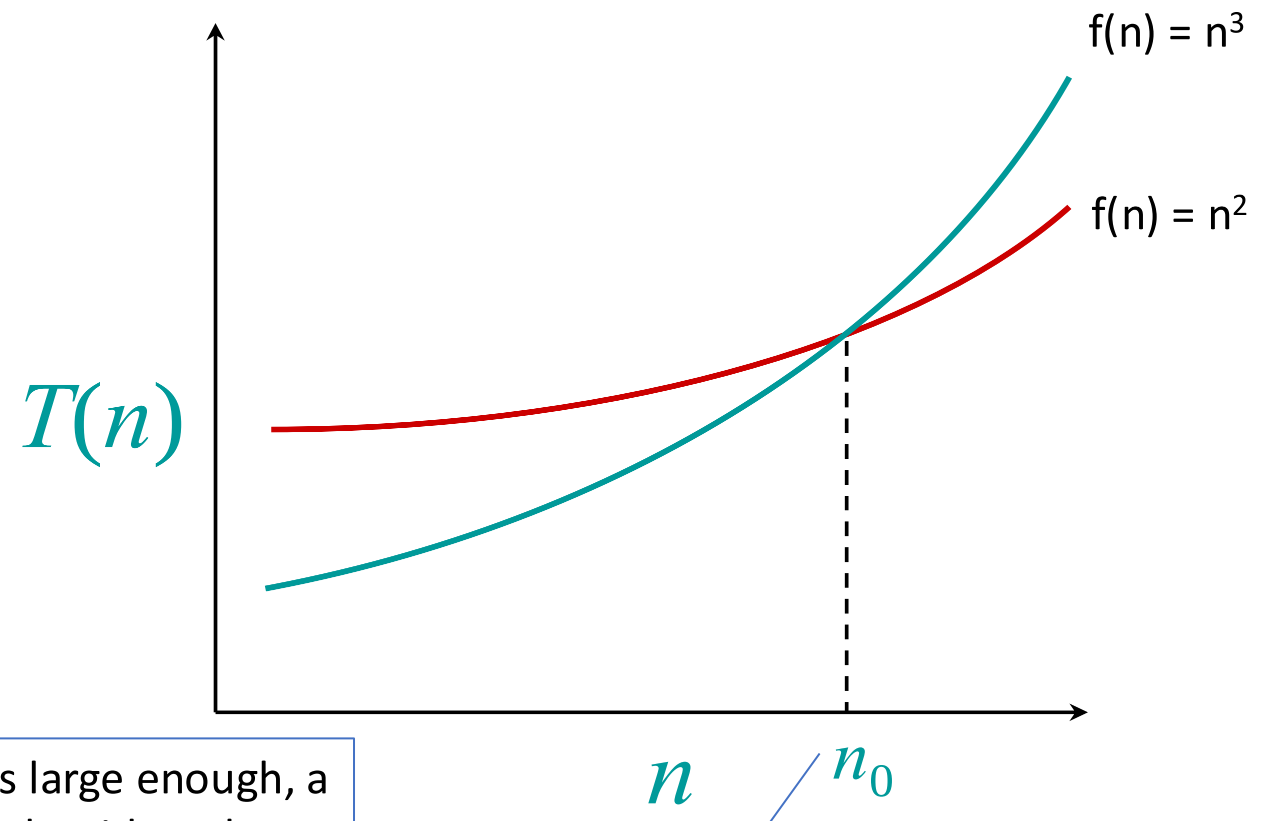
Which algorithm is better?

The higher growth rate?

OR

The lower growth rate?

Which algorithm is better?



When n gets large enough, a the (green) algorithm always have a higher T(n) than the (red) algorithm.

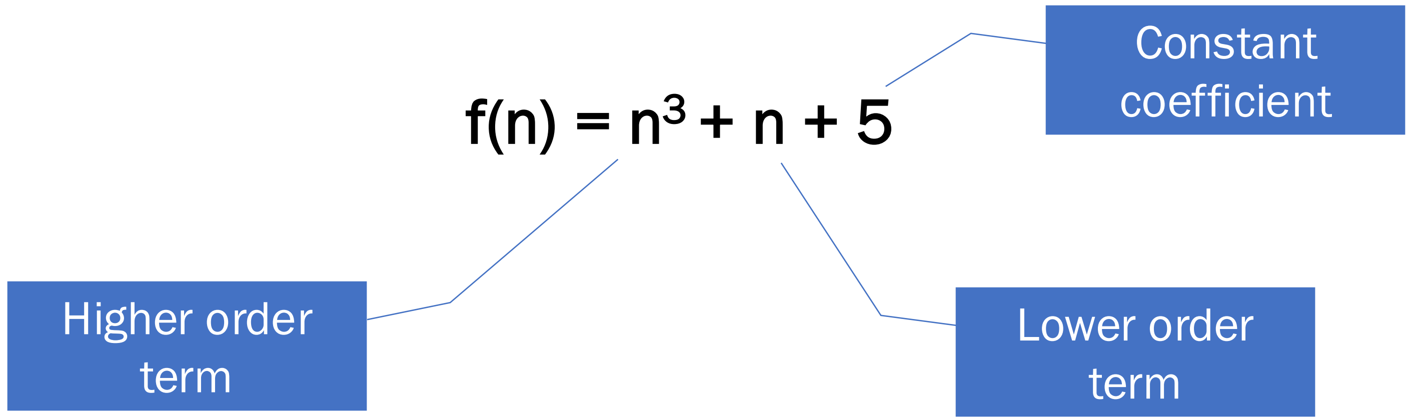
Asymptotic Notation

Asymptotic notation can be divided into 3 forms:

- Big O notation (O) – represents the worst time (upper bound)
- Big Omega notation (Ω) – represents the best time (lower bound)
- Big Theta notation (Θ) – represents the average time.

Asymptotic Notation

- When we are looking for the approximation rate of growth of function in asymptotic analysis, the lower order term and constant coefficients are dropped.



$$T(n) = \Theta(n^3)$$

[Extra] Asymptotic Analysis– Why we dropped the lower order term?

- Less interested the exact time required by algorithm but more in how the time grows as the size of input increases.
- Example, given the worst-case time of an algorithm t(n) (measured in second) is:

$$t(n) = 60n^2 + 5n + 1$$

n	$t(n) = 60n^2 + 5n + 1$	$60n^2$
10	6051	6000
100	600,501	600,000
1000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

[Extra] Asymptotic Analysis– Why we dropped the lower order term?

- Now, if we look at t(n) measures in minutes:

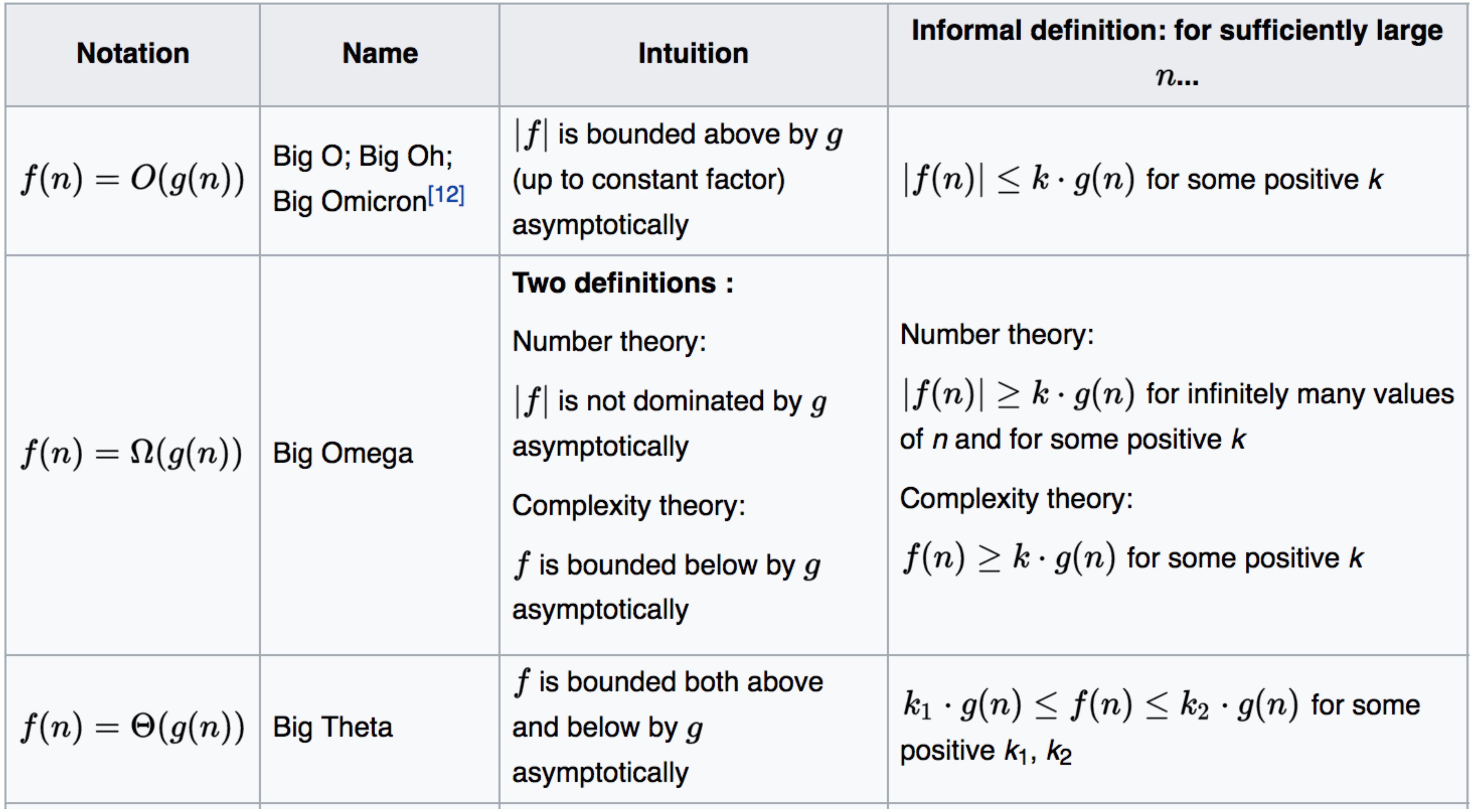
$$T(n) = n^2 + (5/60)n + 1/60$$

- Changing of units does not affect how the time grows as the size of input increases
  - only the unit in which time is measured for input of size n.
- Look at higher order term and ignore the lower order term and constant coefficients.

- Therefore, we can say:

$$T(n) = O(n^2)$$

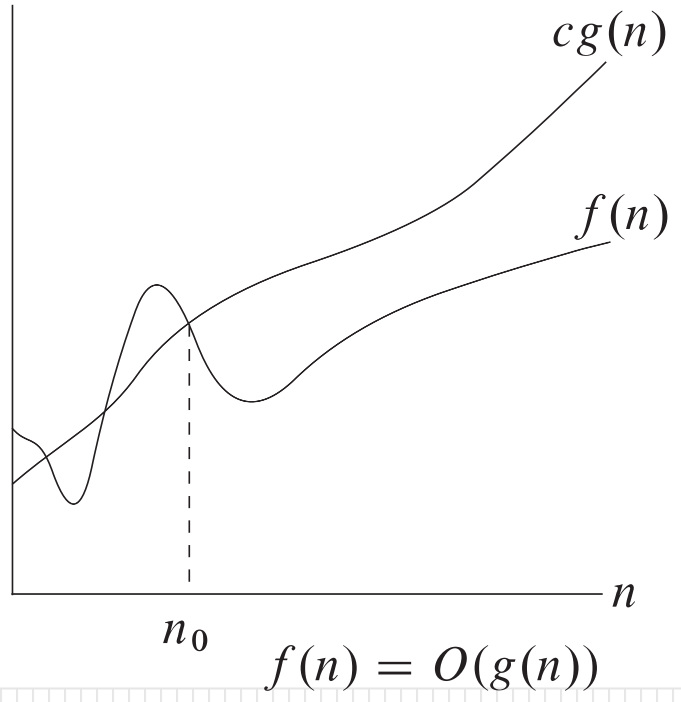
Big O, Big Omega and Big Theta



Notation	Name	Intuition	Informal definition: for sufficiently large n...
$f(n) = O(g(n))$	Big O; Big Oh; Big Omicron[12]	$\|f\|$ is bounded above by g (up to constant factor) asymptotically	$\|f(n)\| \leq k \cdot g(n)$ for some positive *k*
$f(n) = \Omega(g(n))$	Big Omega	**Two definitions :** Number theory: $\|f\|$ is not dominated by g asymptotically Complexity theory: f is bounded below by g asymptotically	Number theory: $\|f(n)\| \geq k \cdot g(n)$ for infinitely many values of *n* and for some positive *k* Complexity theory: $f(n) \geq k \cdot g(n)$ for some positive *k*
$f(n) = \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ for some positive k_1, k_2

Big O-notation

- O-notation gives an upper bound for a function to within a constant factor. We write f(n) = O(g(n)) if there are positive constants n_0 and c such that at and to the right of n_0, the value of f(n) always lies on or below cg(n).

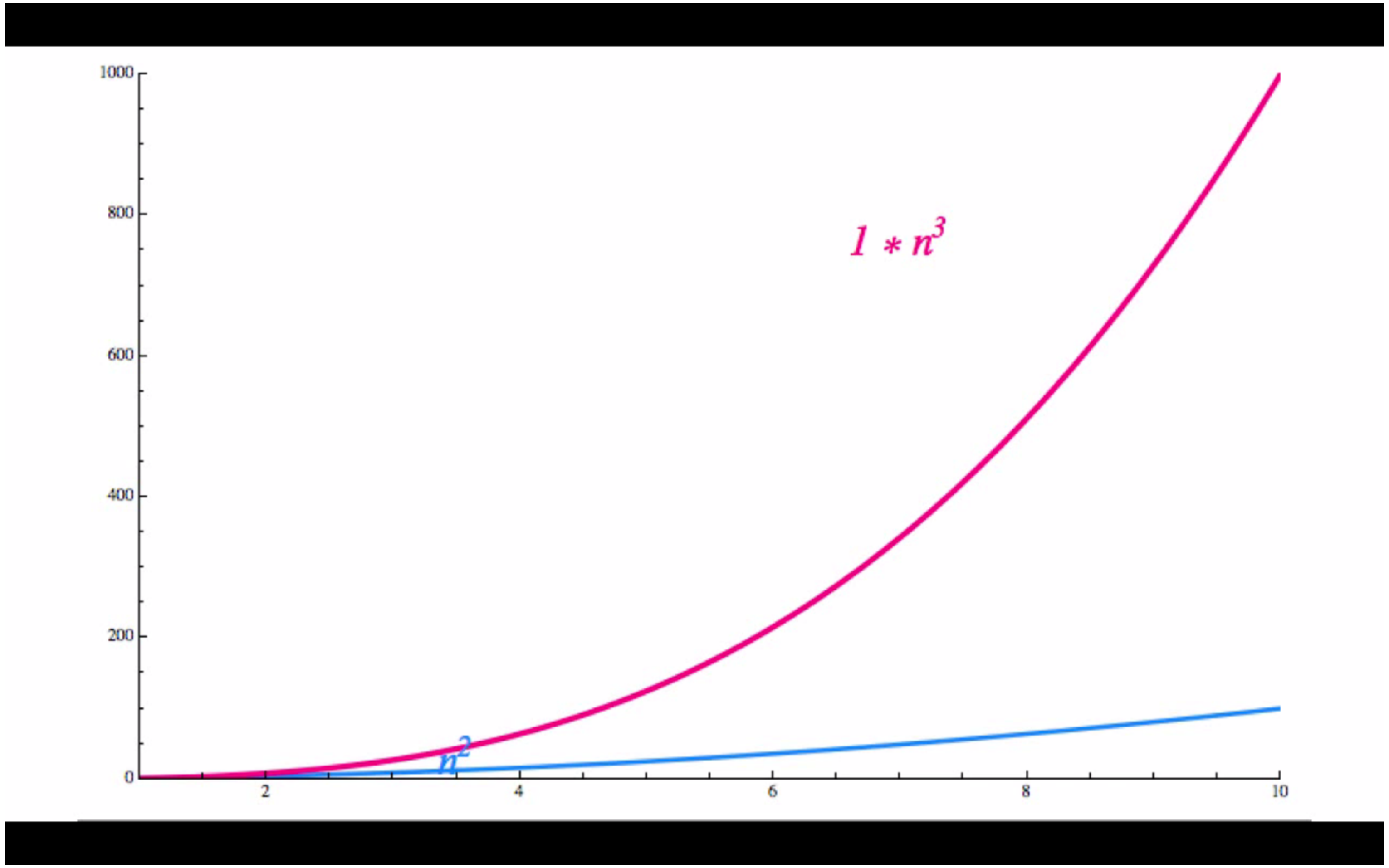


Example Big-O notation

Let say,

- $f(n) = n^2$
- $g(n) = 1 * n^3$

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



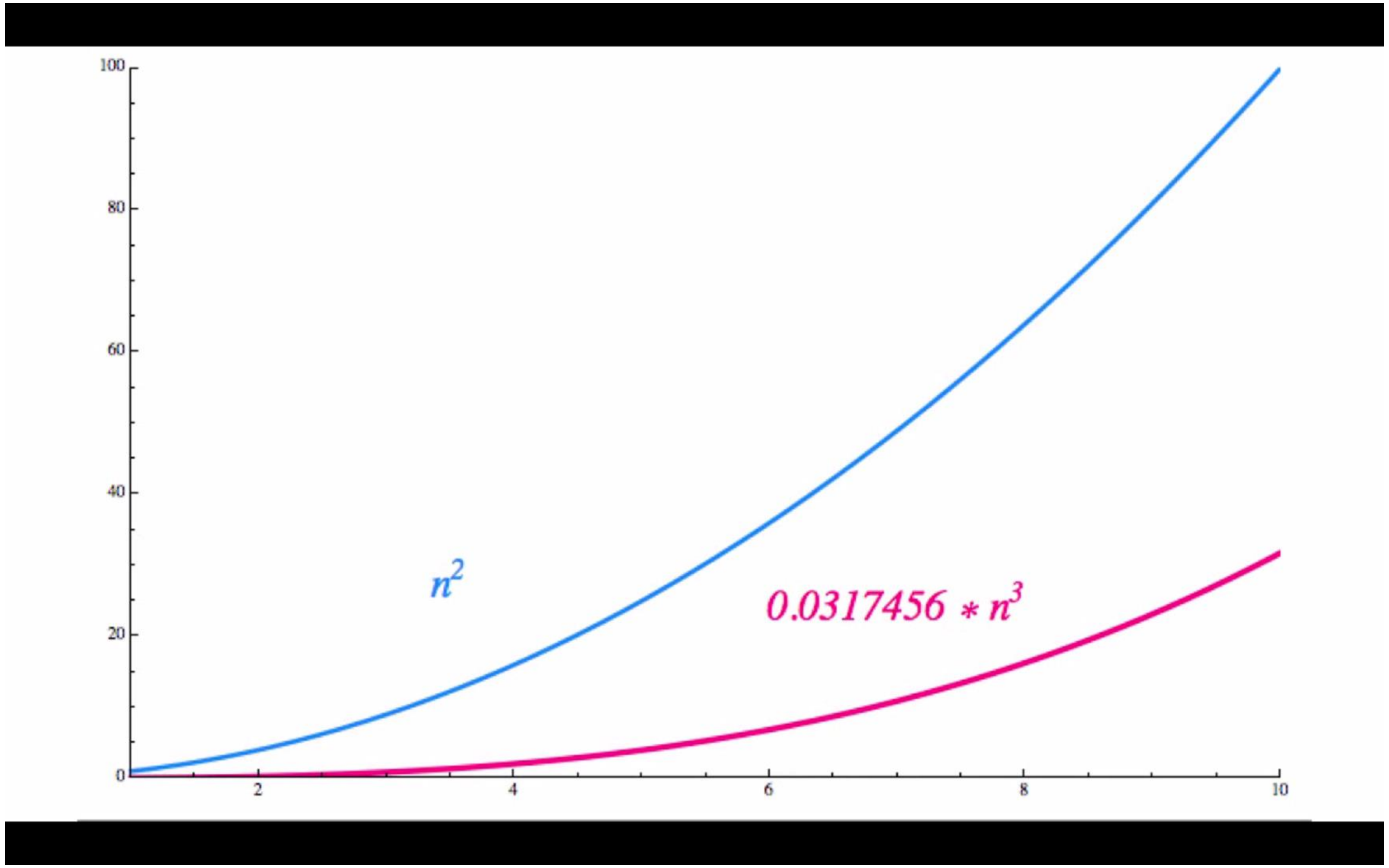
Example Big-O notation

Let say,

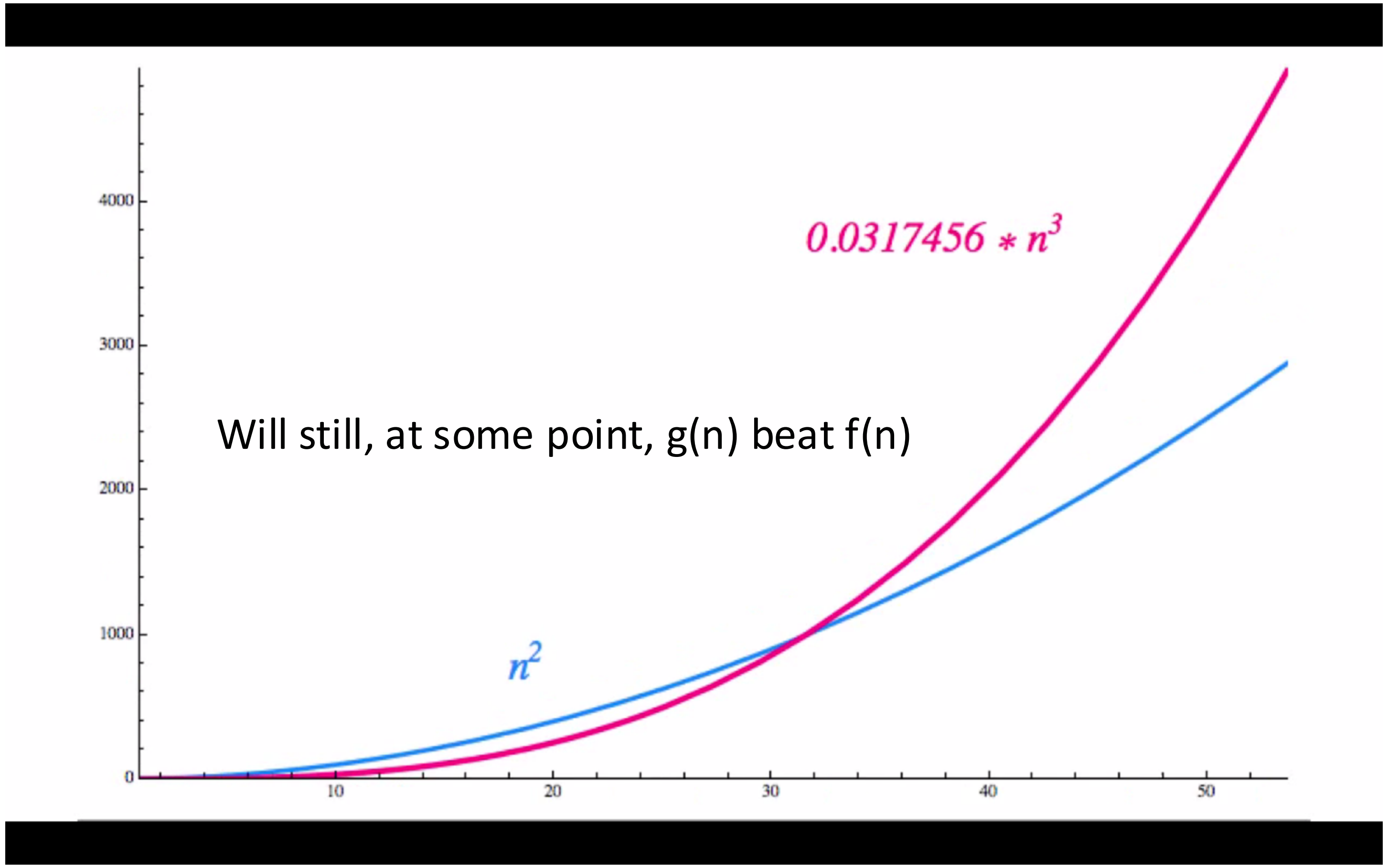
- $f(n)= n^2$
- $g(n) = 1 * n^3$

When c is less than 1,

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



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



Will still, at some point, g(n) beat f(n)

Therefore..

- From the example, regardless of what is the value of c, g(n) will always beat f(n) at some point (n_0).

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

Formal justification

$f(n) \leq cg(n)$; $n \geq n_0$, $n \geq 1$ and $c > 0$

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

Example: Let's say, $f(n) = 3n + 2$ and $g(n) = n^2$. Is $f(n) = O(g(n))$?

$f(n) \leq cg(n)$; $n \geq 1$ and $c > 0$

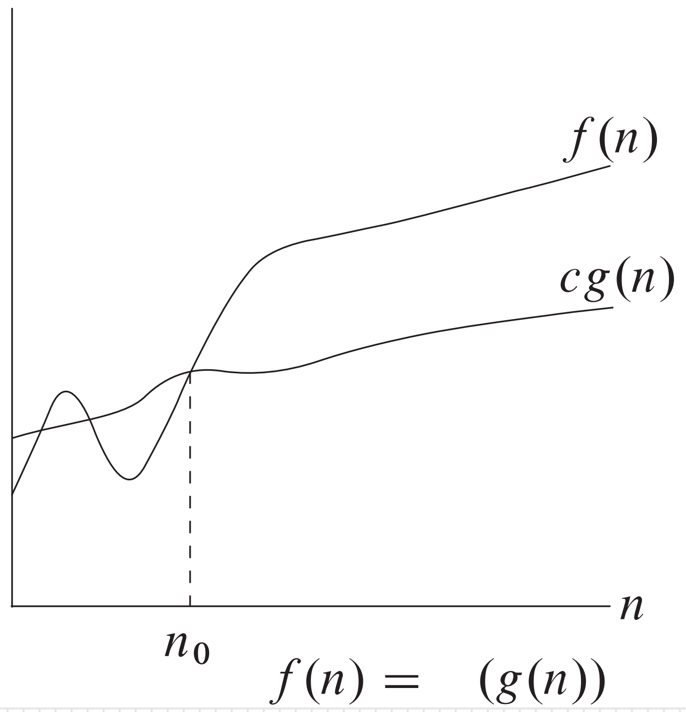
$3n + 2 \leq c.n^2$; $c = 1$ and $n \geq 4$

$= 3n + 2 \leq n^2$ -- proved

Therefore, $f(n) = O(g(n))$

Big Ω-notation

- Ω-notation gives a lower bound for a function to within a constant factor. We write f(n) = Ω(g(n)) if there are positive constants n_0 and c such that at and to the right of n_0, the value of f(n) always lies on or above cg(n).

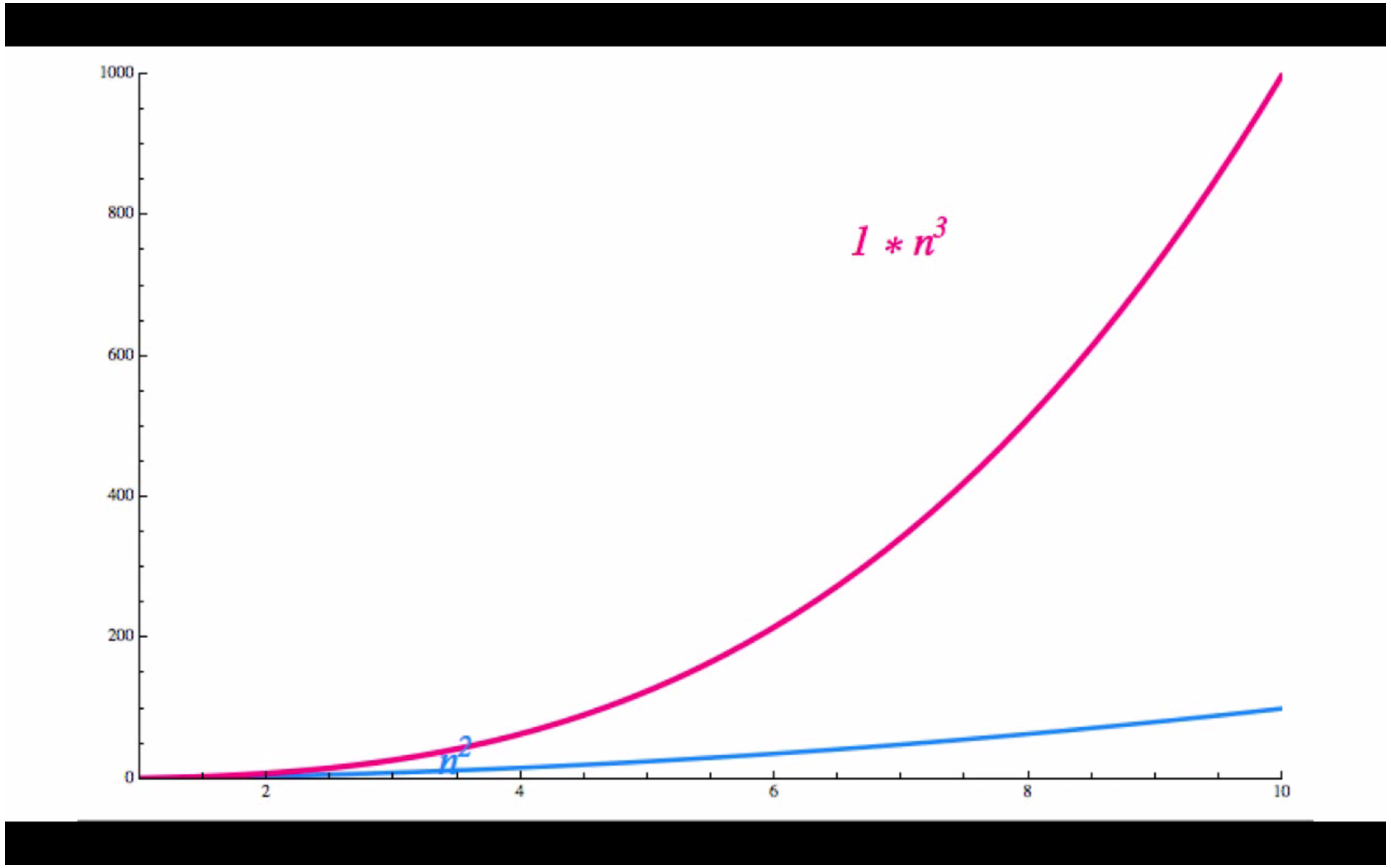


Example Big Ω-notation

Let say,

- f(n)= n^2
- g(n) = 1 * n^3

g(n) = Ω (f(n))



Formal justification

$f(n) \geq cg(n)$; $n \geq n_0$, $n \geq 1$ and $c > 0$

$f(n) = \Omega\ (g(n))$

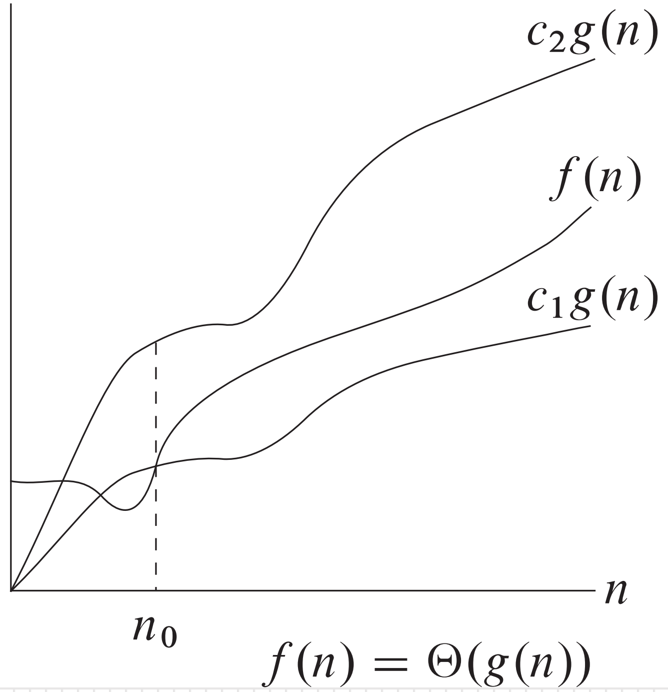
Example: Let's say, $f(n) = 3n + 2$ and $g(n) = n^2$. Is $g(n) = \Omega\ (f(n))$?

$g(n) \geq cf(n)$; $n \geq 1$ and $c > 0$
$n^2 \ \geq c.3n + 2$; $c = 1$ and $n \geq 4$
$= n^2 \geq 3n + 2$ -- proved

Therefore, $g(n) = \Omega\ (f(n))$

Big Θ-notation

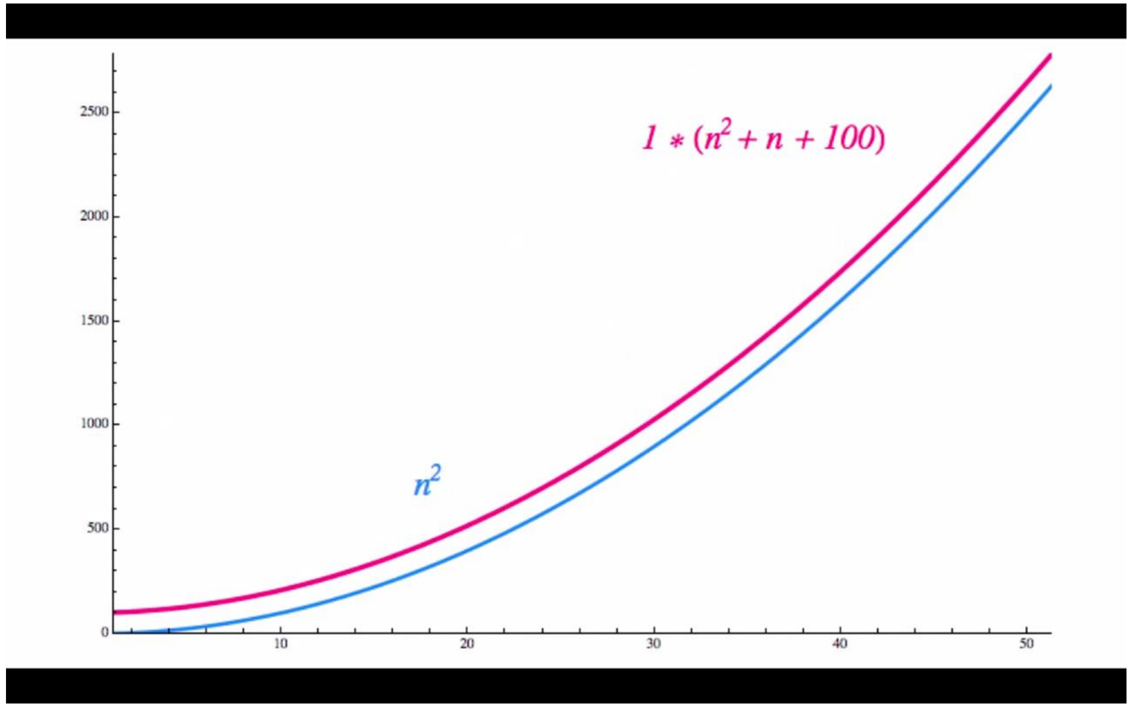
- Θ-notation bounds a function to within constant factors. We write f(n) = Θ(g(n)) if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of f(n) always lies between $c_1g(n)$and $c_2g(n)$inclusive.

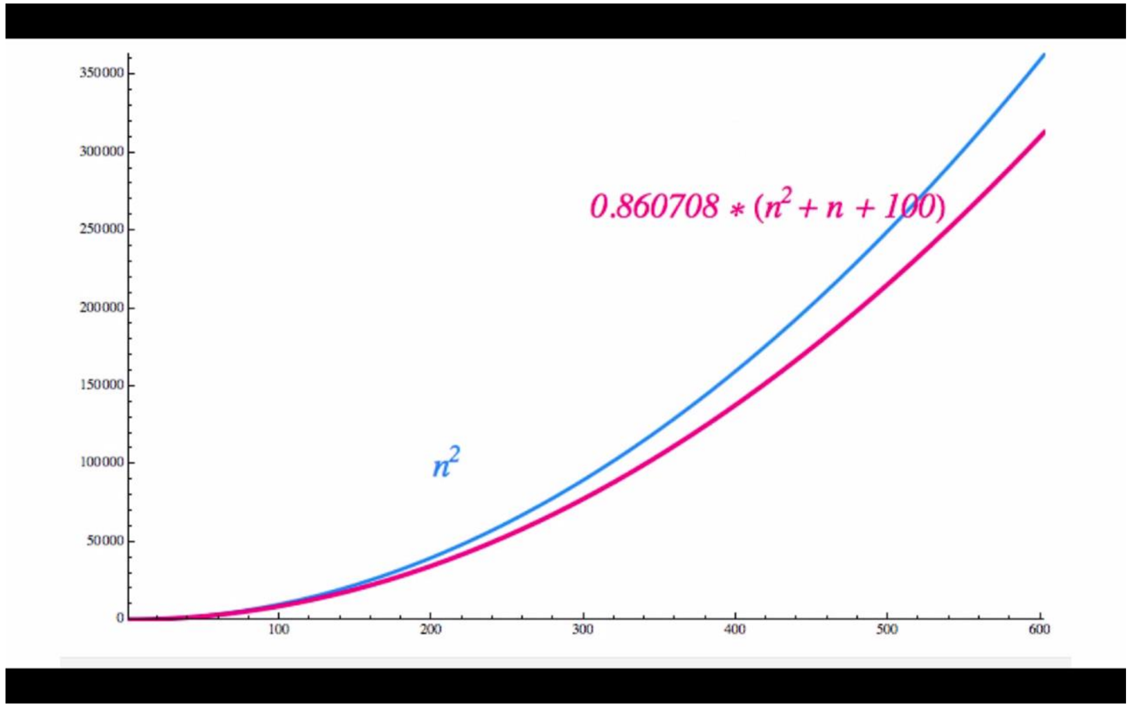


Example Big-Θ

Let say,

- $f(n) = n^2$
- $g(n) = n^2 + n + 100$





Therefore..

- f(n) and g(n) have the asymptotic growth rate equivalence:

$$f(n) = \Theta(g(n))$$

Formal justification

$c_1 g(n) \leq f(n) \leq c_2 g(n)$; $n \geq n_0$, $n \geq 1$ and $c_{1,}\ c_2 > 0$

$f(n) = \Theta\ (g(n))$

Example: Let's say, $f(n) = 3n + 2$ and $g(n) = n$. Is $f(n) = \Theta\ (g(n))$?

(1) $f(n) \geq c_1 g(n)$; $n \geq 1$ and $c > 0$

$3n + 2 \ \geq c_1 n$; $c_1 = 1$ and $n \geq 1$

$= 3n + 2 \geq n$ -- Proved

(2) $f(n) \leq \ c_2 g(n)$; $n \geq 1$ and $c > 0$

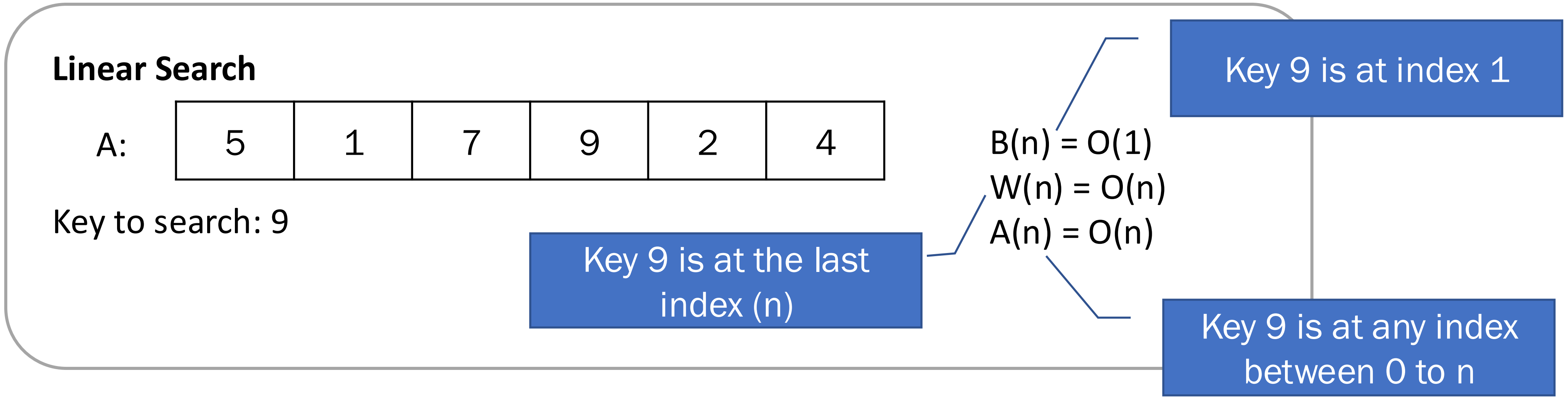
$3n + 2 \leq \ c_2 n$; $c = 4$ and $n \geq 1$

$= 3n + 2 \geq n$ -- Proved

Therefore, $f(n) = \Theta\ (g(n))$

Best, Average and Worst-Case Running Time

- Analysis of algorithm always interested in Worst Time.
- Average time is used when there are no difference between best and worst time.
- Best time are often not considered during analysis of algorithm.



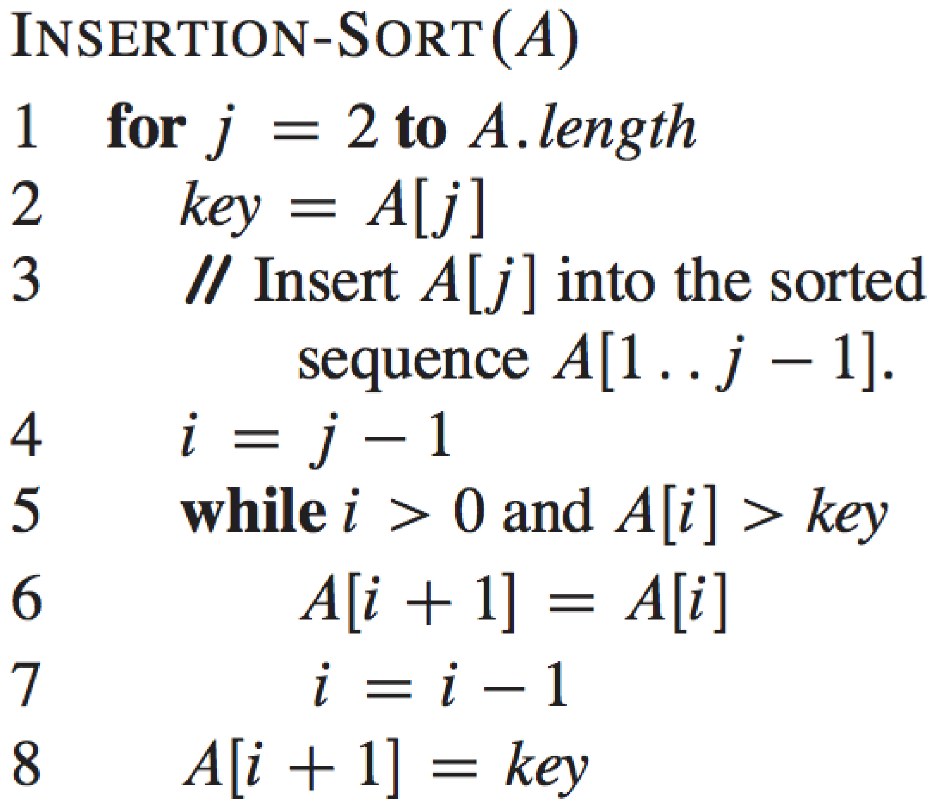
Representing Algorithm

Up until now you should have understood what is Asymptotic Analysis.
From this point on, we will be looking at algorithm and see what is the running time complexity for each of them.
Before that, let's look at how algorithm is written.

Pseudocode

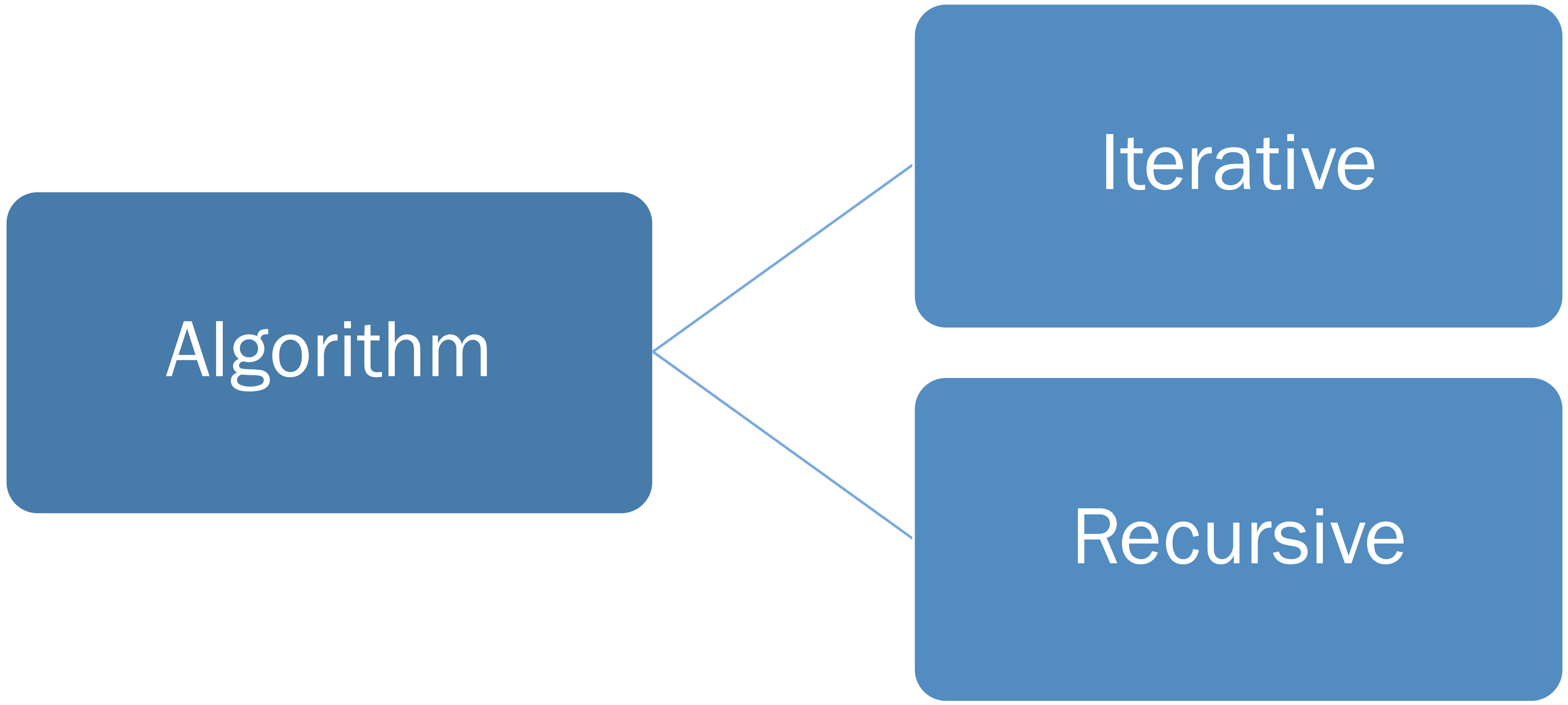
- Pseudocode is an informal representation to describe the algorithm.
- It uses conventional structure (intended for reading by human) which is almost similar to programming language, but omits any machine essentials, for example:
    - Variable declarations
    - System specific codes
- The purpose of pseudocode is to:
    - Make comprehension of algorithm easier (human readable).
    - Describe an implementation-independent algorithm.

Example of Pseudocode



```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted
            sequence $A[1..j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Types of Algorithm



Analysing Iterative algorithm

Iterative Algorithm

- Iterative algorithm uses looping statements such as “for”, “while”, and “do-while” to repeat the same steps.

CODE 1

```
for I in range (0,10,1)
    print(“hello”)
```

CODE 2

```
i = 1
while ( i < 10) :
    i+=1
```

Analysis of Iterative Algorithm

Determine higher order term by calculating the time a statement is executed:

- operation
- comparison
- loop
- pointer reference
- function calls

Analysis of Iterative Algorithm

- Problems that can be solve with iterative, can be solve using recursion as well, and vice versa.
- But the time complexity analysis is different.

Example Problem

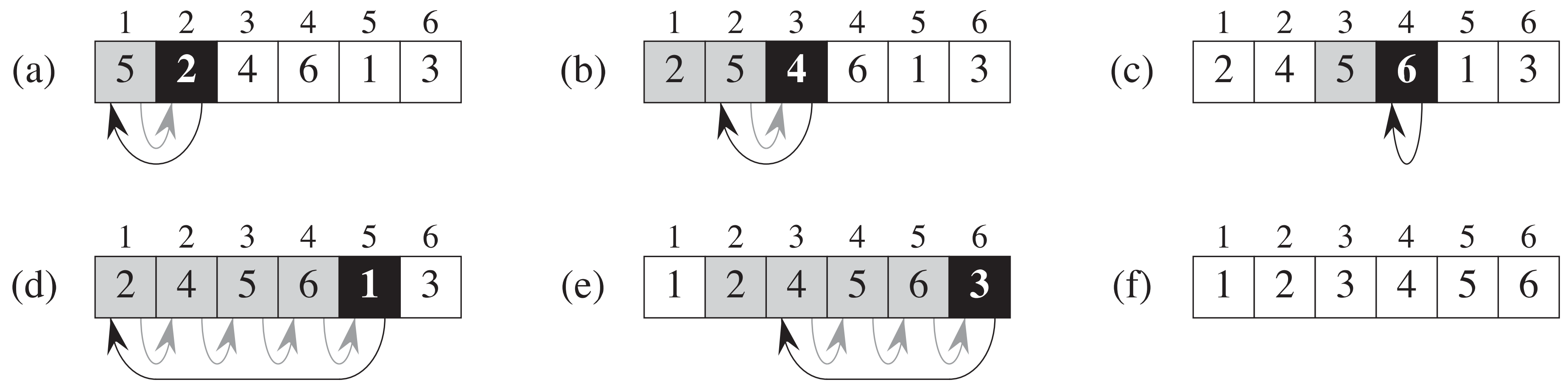
- For example, we might need to sort a sequence of numbers into non-decreasing order.
- Here is how we formally define the sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that a'_1 a'_2 a'_n.

Input: 5 2 4 6 1 3
Output: 1 2 3 4 5 6

Operation of Insertion Sort



(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

Insertion Sort

```
INSERTION-SORT(A)
1 for $j = 2$ to $A.length$
2     $key = A[j]$
3     // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4     $i = j - 1$
5     while $i > 0$ and $A[i] > key$
6         $A[i+1] = A[i]$
7         $i = i - 1$
8     $A[i+1] = key$
```

Analysis of Insertion Sort

- Before we apply the asymptotic analysis, we need to look at the algorithm.
- Intuitively, we can say that the running time depends on the input - an already sorted sequence is easier to sort.
  - Therefore, we want to look for the upper bounds.
- We also need to parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Input = n values
  - no assumptions can be made for n, but n is not always small.

Running time complexity of Insertion Sort

```
INSERTION-SORT(A)                                   cost    times
1  for $j = 2$ to $A.length$                         $c_1$   $n$
2      $key = A[j]$                                  $c_2$   $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1..j-1]$.                      $0$     $n-1$
4      $i = j - 1$                                   $c_4$   $n-1$
5      while $i > 0$ and $A[i] > key$                $c_5$   $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                           $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
7          $i = i - 1$                               $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
8      $A[i+1] = key$                                $c_8$   $n-1$
```

To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Best running time

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

T(n) = Ω(n)

Worst running time

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

T(n) = O(n^2)

More example on analysis of iterative algorithm

Example 1

- Consider the following iterative program:

```
def function1 ():
   for i in range (1,n,1)
      print(“Hello world”)
```

Example 1

- Consider the following iterative program:

```
def function1 ():
    for i in range (1,n,1)
        print(“Hello world”)
```

The number of loop “Hello word” is going to be printed is stated in the condition (n times)

Complexity = O(n)

Example 2

- Consider the following iterative program:

```
def function2():
    for i in range (1,n,1)
        for j in range(1,n,1) {
            print(“Hello world”)
```

Example 2

- Consider the following iterative program:

```
def function2():
    for i in range (1,n,1)
        for j in range(1,n,1) {
            print("Hello world")
```

Outer 1: n times

Inner loop: n times

Complexity = O(n^2)

Example 3

- Consider the following iterative program:

```
def function3():
for i in range (n/2,n,1)
    for j in range(n/2,n,1) {
        for k in range(1,n,k*2) {
            print(“Hello world”)
```

Example 3

- Consider the following iterative program:

```
def function3():
for i in range (n/2,n,1)
   for j in range(n/2,n,1) {
      for k in range(1,n,k*2) {
         print("Hello world")
```

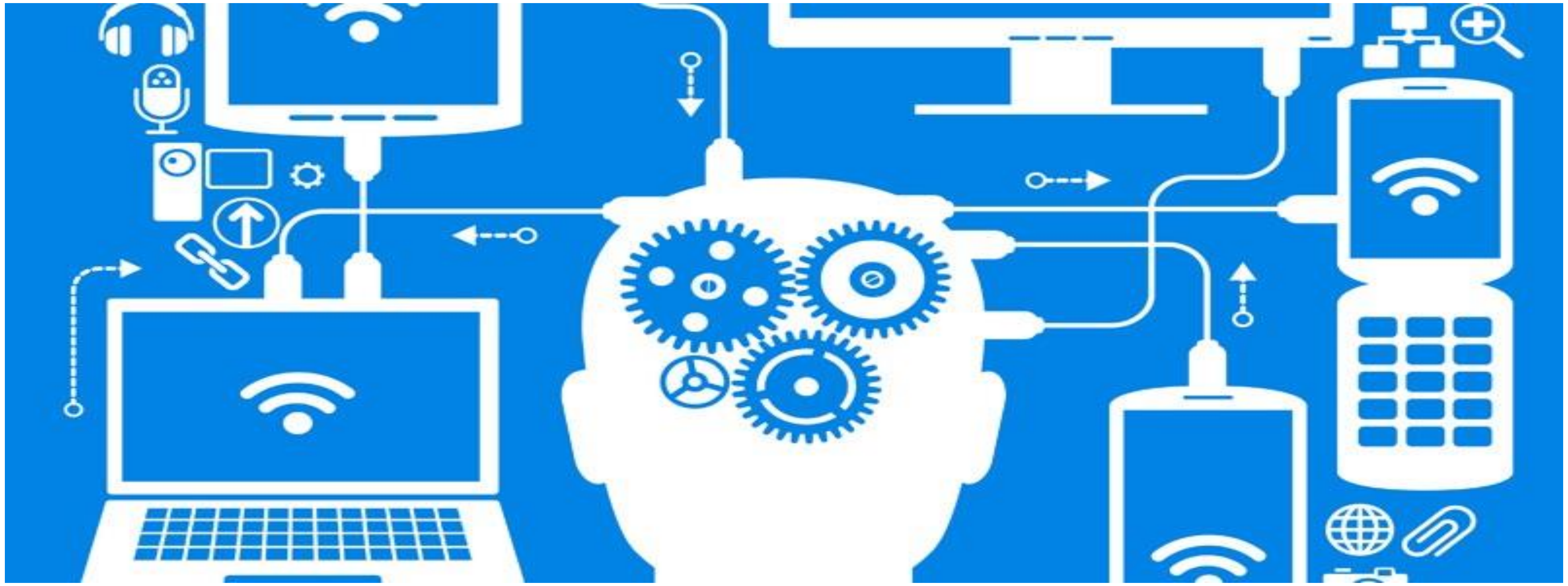
Loop 1: n/2 times

Loop 2: n/2 times

Loop 3: $\log_2 n$

Complexity = $O(n^2\log_2 n)$

In the next lecture..



Lecture 2: Introduction to Algorithm Design & Analysis (Pt2)

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. 2009. Introduction to Algorithms, 3rd edition. MIT Press.
- Robert Sedgewick and Kevin Wayne. 2011. Algorithm. 5th Edition. Addison-Wesley.
- Wikipedia
- Big O asymptotic notation, big theta and big omega –
  - https://www.youtube.com/watch?v=8Y6gqjIxAlc
  - https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation
- Time complexity analysis of iterative program
  - https://www.youtube.com/watch?v=FEnwM-iDb2g&index=2&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0

Self-practice

Example 4

```
def A ():
        for I in range (n)
            print (“Hello”)
```

Example 5

```
def A ():
        for i in range(n)
            for j in range (n)
                print ("Hello")
```

Example 6

```
def A ():
    i = 1, s = 1
    while (s <= n):
        i++
        s = s + i
        print (“Hello”)
```

Example 7

```
def A ():
        for i in range (1,n,1)
            for j in range (1, j<=i,1)
              for k in range (1, 100, 1)
                   print(“Hello”)
```

Example 8

```
def A ():
    for i in range (1, n, 1)
        for j in range (1, i², 1)
            for k in range (1, n/2, 1)
                print (“Hello”)
```

Example 9

```
def A ():
    for i in range (1, i<=n, i*2)
        print(“Hello”)
```

Example 10

```
def A ():
        for i in range (n/2, i<=n,1)  ————  $\frac{n}{2}$
            for j in range (1, j<=n/2, 1)  ————  $\frac{n}{2}$
              for k in range (1,k<=n, k * 2)  ——  $\log_2 n$
                   print("Hello")
```

Since no dependency among 3 loops

$$\frac{n}{2} * \frac{n}{2} * \log_2 n$$

$$= O(n^2 \log_2 n)$$

Example 11

```
def A ():
        for i in range (n/2, i<=n, i)
            for j in range (1, j<=n, 2*j)
              for k in range (1, k<=n,k * 2)
                 print ("Hello")
```

Example 12

```
def A ():
    for i in range (1,  i<=n, 1)
        for j in range(1, j<=n, j+i)
        print ("Hello")
```

Example 13

```
def A ():
        n= $2^{2^k}$
        for i in range (1,  i<=n, 1)
             j=2
             while (j<=n):
                   j=j*j;
                   print (“Hello”)
```