WIA2005: Algorithm Design and Analysis

Semester 2, Session 2016/17

Lecture 1: Introduction to Algorithm Design & Analysis Fundamentals

Learning Objectives

- Know what is the role of Algorithm in Computing.
- Know what is growth of function.
- Understand and analyse time complexity of iterative algorithm.
- Understand and analyse time complexity of recursive algorithm.

What is Algorithm?

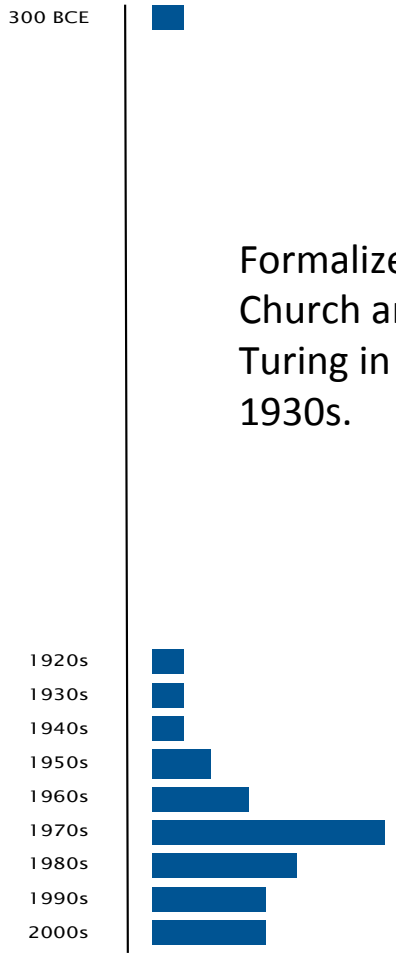
- Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***.

- 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, ...



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.

Example iterative algorithm

- For example, we might need to sort a sequence of numbers into non-decreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.
- Here is how we formally define the ***sorting problem***:

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

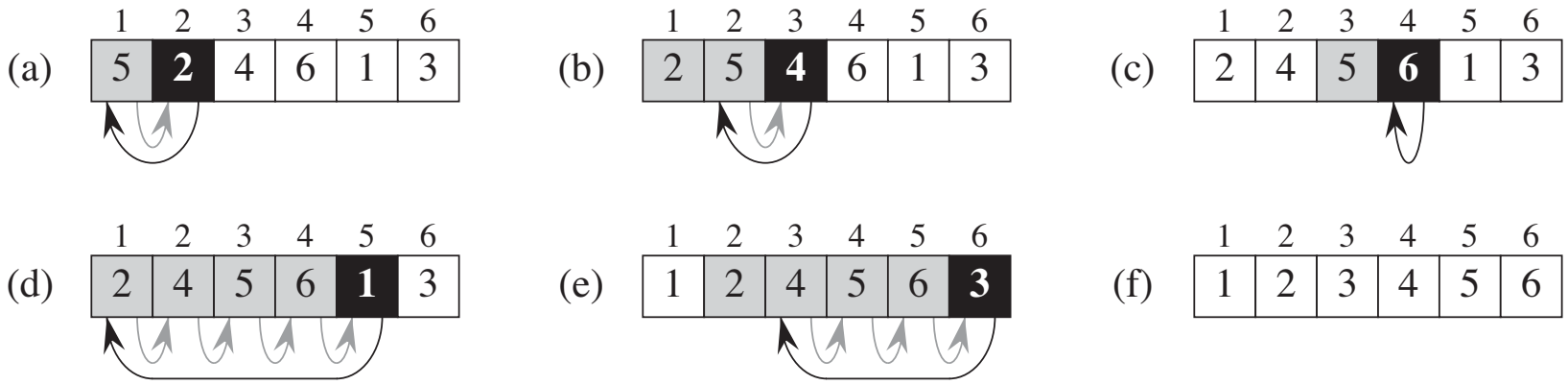
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

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

Insertion Sort Pseudocode

```
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
```

Operation of Insertion Sort



Pseudocode Convention

1. Indentation indicates block structure.
2. The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.
3. The symbol "**//**" indicates that the remainder of the line is a comment.
4. A multiple assignment of the form I = j = e assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment j = e followed by the assignment I = j.
5. Variables (such as i, j, and *key*) are local to the given procedure. Global variables is not used without explicit indication.
6. Accessing array elements by specifying the array name followed by the index in square brackets.

Pseudocode Convention

7. A particular attribute is accessed using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name.
8. Parameters are passed to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure.
9. A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller.
10. The boolean operators “and” and “or” are ***short circuiting***.
11. The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called.

ANALYSIS OF ALGORITHM

Analysing Algorithm

- ***Analysing*** an algorithm has come to mean predicting the resources that the algorithm requires.

- Ultimate goal, by analysing several candidate algorithms for a problem, we can identify a most efficient one (the **running time**)
  - in terms of time and space.

Running time analysis

- Determine the running time increases related to the size of the input.

- Input = n values
  - no assumptions can be made for n (n is not always small)

Analysis of Insertion Sort

- The running time depends on the input: an already sorted sequence is easier to sort.

- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

Machine independent time

- What is insertion sort's worst-case time running on a specific machine?
  - It depends on the speed of the computer:
    - relative speed (on the same machine)
    - absolute speed (on different machines)

Therefore - Ignore machine-dependent constants, and look at growth of **T(n)** as **n -> ∞**

"Asymptotic Analysis"

Analysis of running time using cost

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)..

- 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:

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

Kinds of analysis

- Worst-case: (usually) – longest time
  - T(n) = maximum time of algorithm on any input of size n.
- Average-case: (sometimes) – average time
  - T(n) = expected time of algorithm over all inputs of size n. Need assumption of statistical distribution of inputs.
- Best-case: (bogus) – least time
  - Cheat with a slow algorithm that works fast on some input.

Insertion Sort – Best Case

- Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given.
- For example, in INSERTION-SORT, the best case occurs if the array is already sorted.
- For each j = 2, 3, …, n, we then find that A[i] ≤ *key* in line 5 when i has its initial value of j-1.
- Thus, $t_j = 1$ for j = 2, 3, …, n, and the best-case running time is

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

Insertion Sort – Worst Case

- If the array is in reverse sorted order - that is, in decreasing order – produce the worst case results.
- We must compare each element A[j] with each element in the entire sorted subarray A[1, …, j – 1], and so $t_j = j$ for $j = 2, 3, \ldots, n$.
- We find that in the worst case, the running time of INSERTION-SORT is

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

- We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i
  - it is thus a ***quadratic function*** of n.

Insertion Sort – Average Case

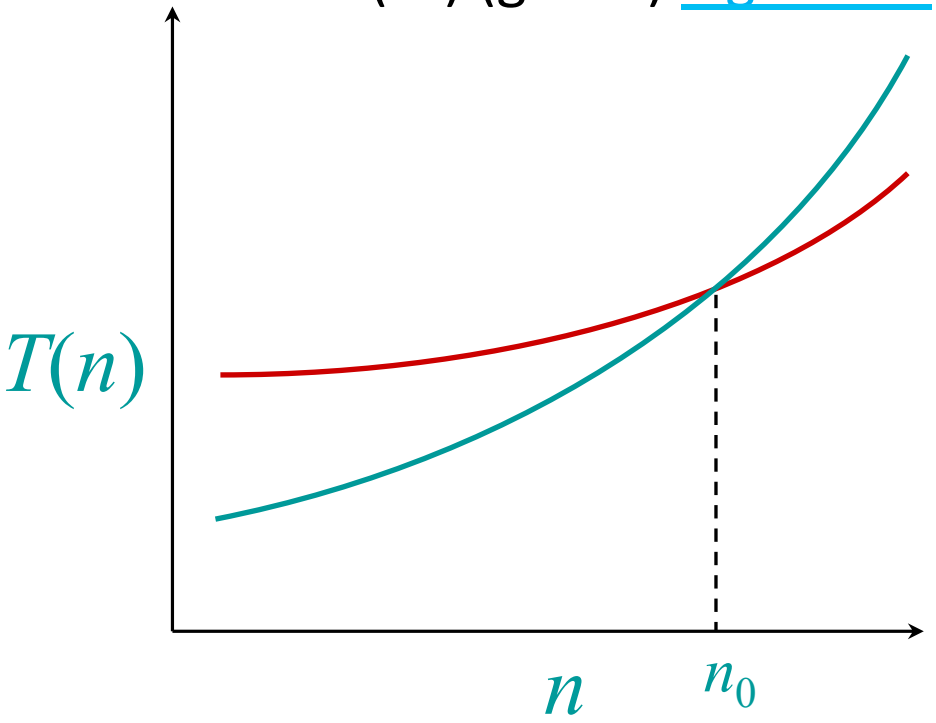
- The “average case” is often roughly as bad as the worst case.
- Suppose that we randomly choose n numbers and apply insertion sort.

How long does it take to determine where in sub-array A[1, …, j – 1] to insert element A[j] ?

- On average, half the elements in A[1, …, j – 1] are less than A[j], and half the elements are greater.
  - Therefore, we check half of the sub array A[1, …, j – 1] , and so t_j is about j/2.
  - The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Asymptotic performance

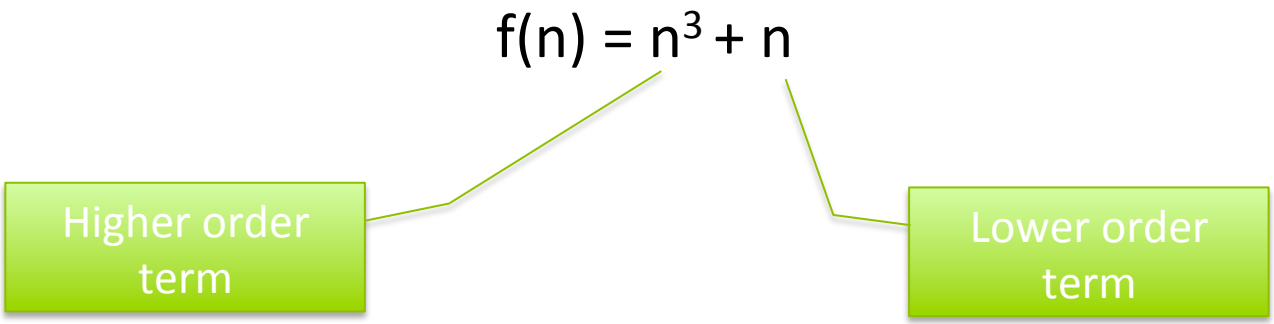
- When n gets large enough, a $O(n^2)$ (red) algorithm always beats a $O(n^3)$ (green) algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

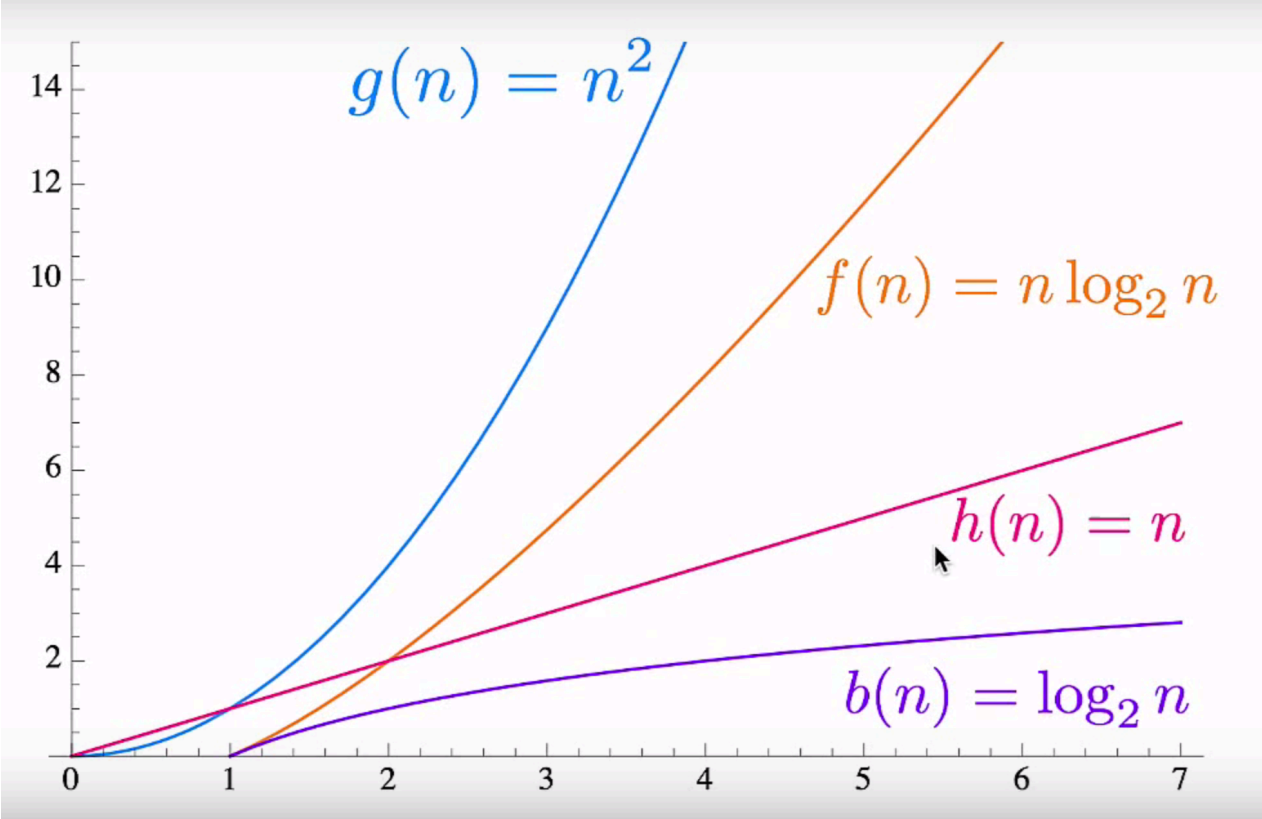
Growth of Functions

- 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.
- Lower order terms: When given an approximation of rate of growth of a function, we tend to drop the lower order terms as they are less significant to higher order terms.



Asymptotic Analysis

- Main function characteristics



Comparison of running time

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n\lg n$	189481	8.64×10^6	4.18×10^8	8.69×10^9	2.28×10^{11}	2.54×10^{12}	2.20×10^{14}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Big O, Big Omega and Big Theta

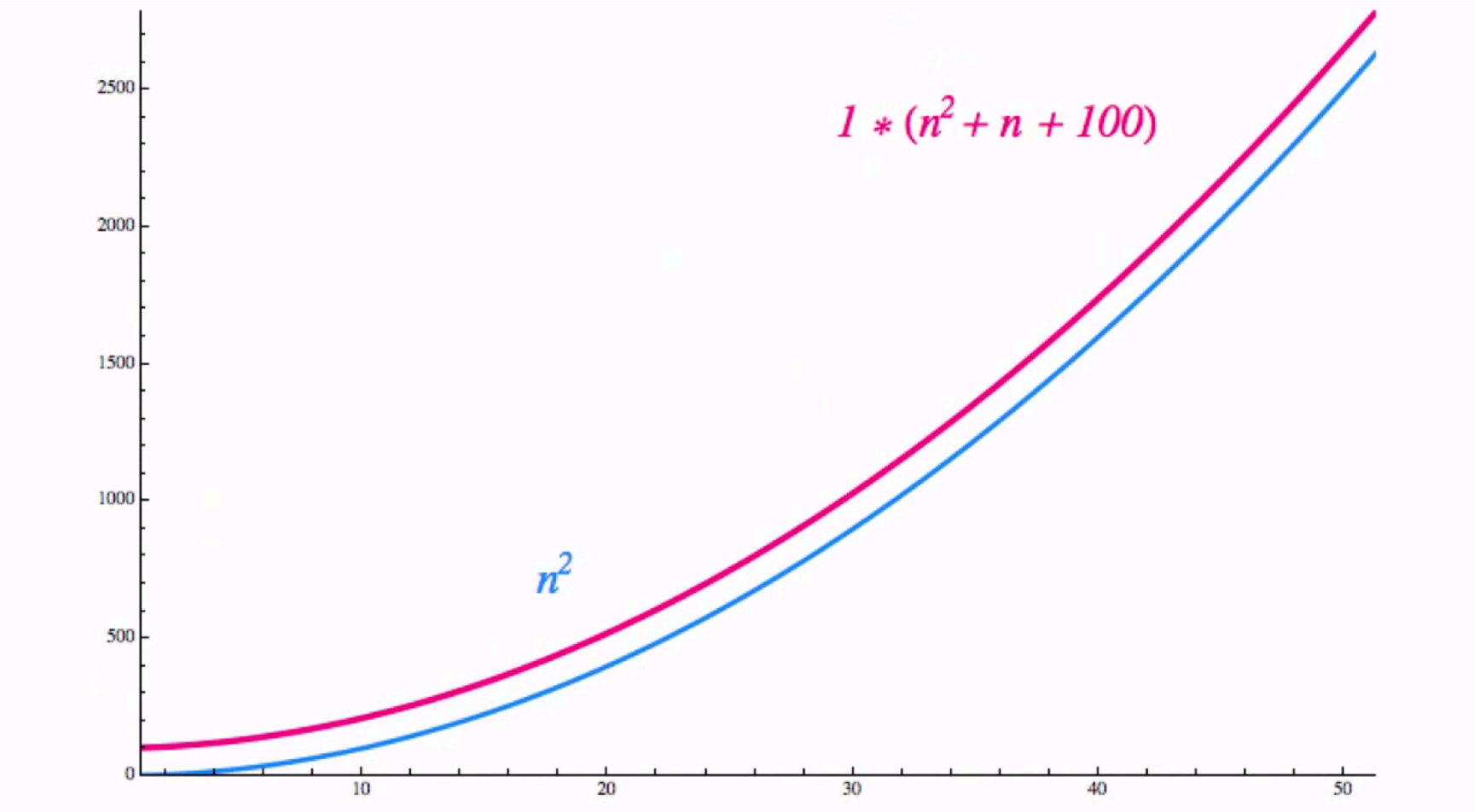
Notation	Name	Intuition	Informal definition: for sufficiently large n...
$f(n) = O(g(n))$	Big O; Big Oh; Big Omicron[12]	$\|f\|$ is bounded above by g (up to constant factor) asymptotically	$\|f(n)\| \leq k \cdot g(n)$ for some positive *k*
$f(n) = \Omega(g(n))$	Big Omega	**Two definitions :** Number 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

Example 1

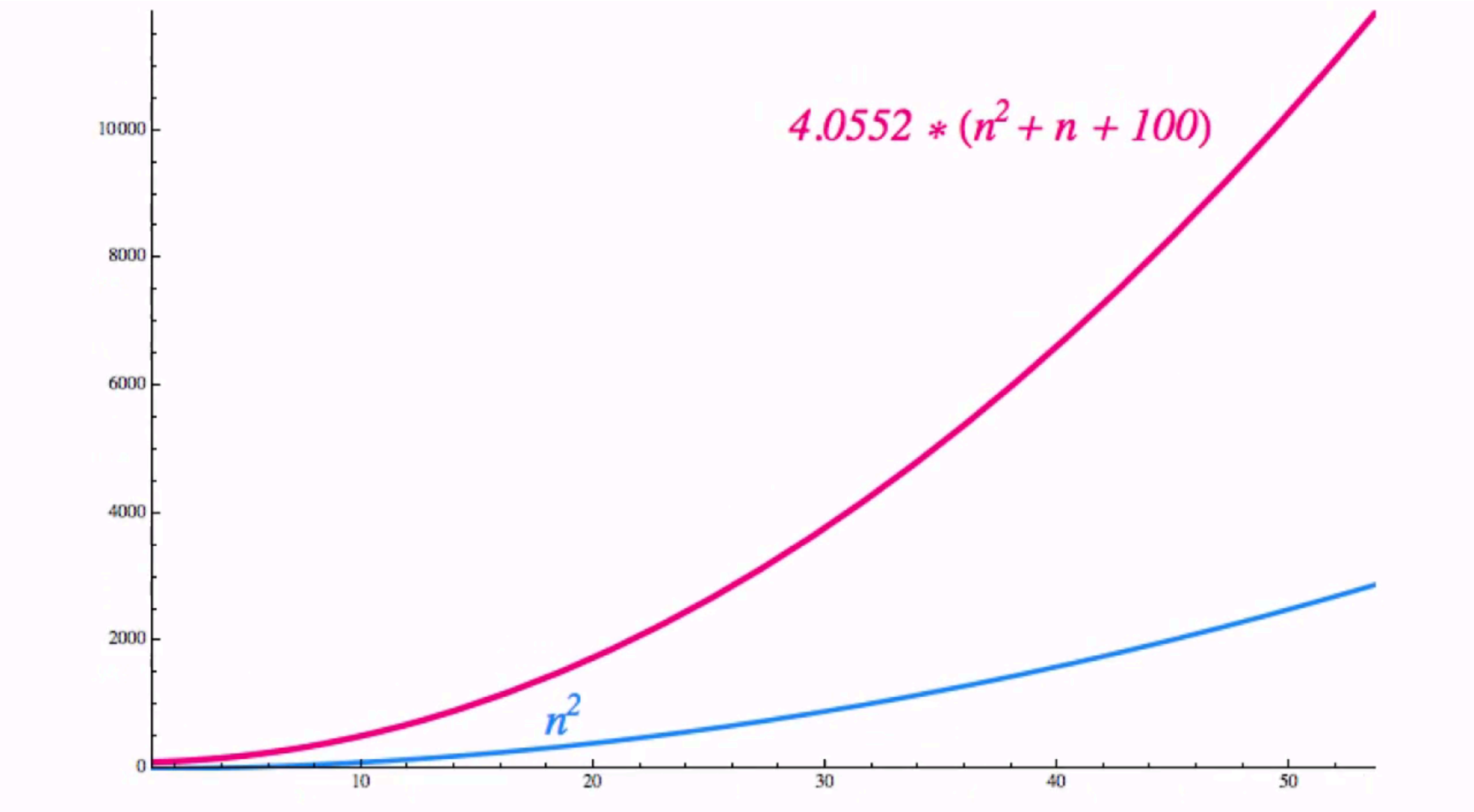
Let say,

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

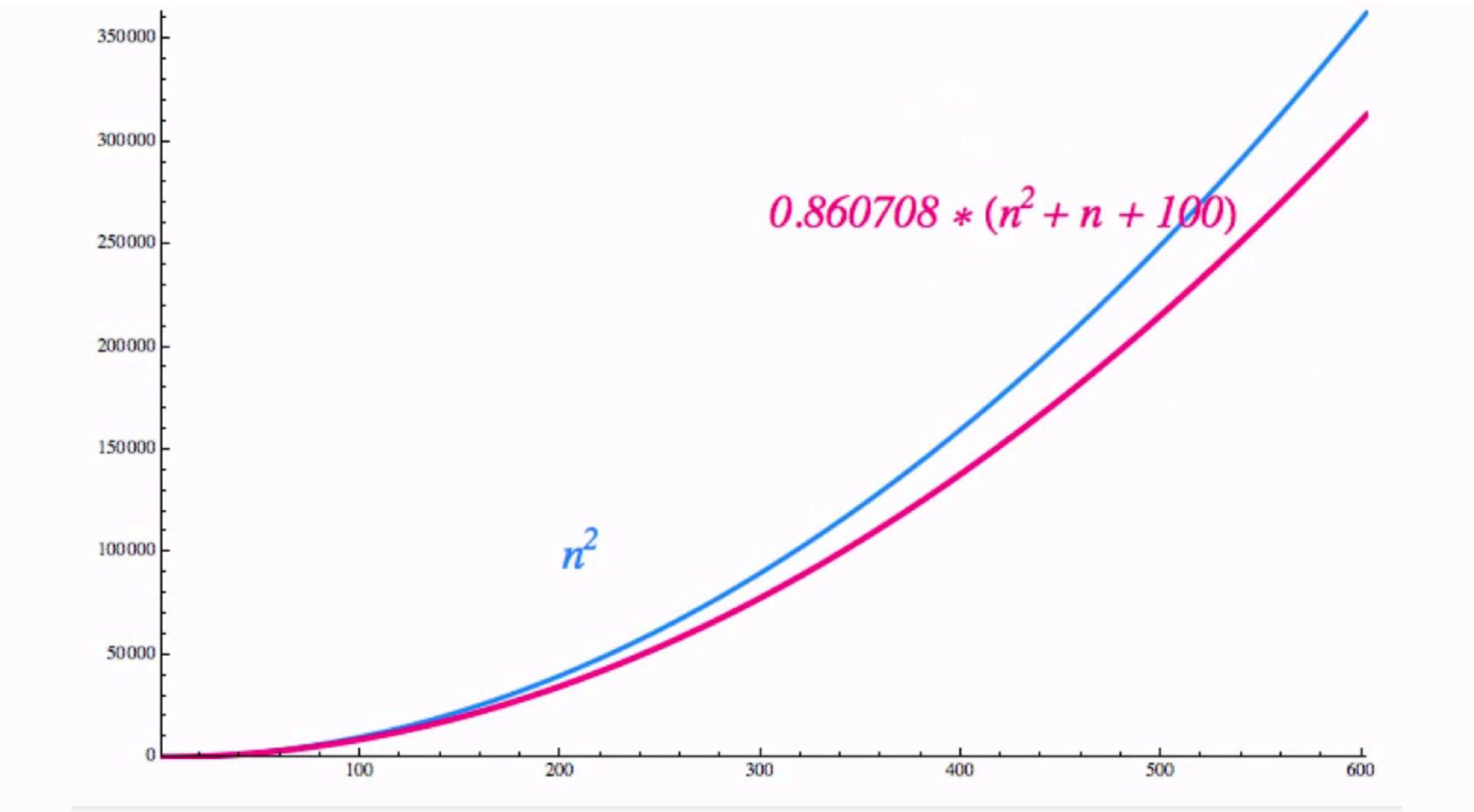
(1) f(n) is O(g(n))



(2) f(n) is O(g(n))



(3) f(n) is Ω(g(n))


350000
300000
250000
200000
150000
100000
50000
0
100
200
300
400
500
600
$0.860708 * (n^2 + n + 100)$
n^2

(4) Therefore..

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

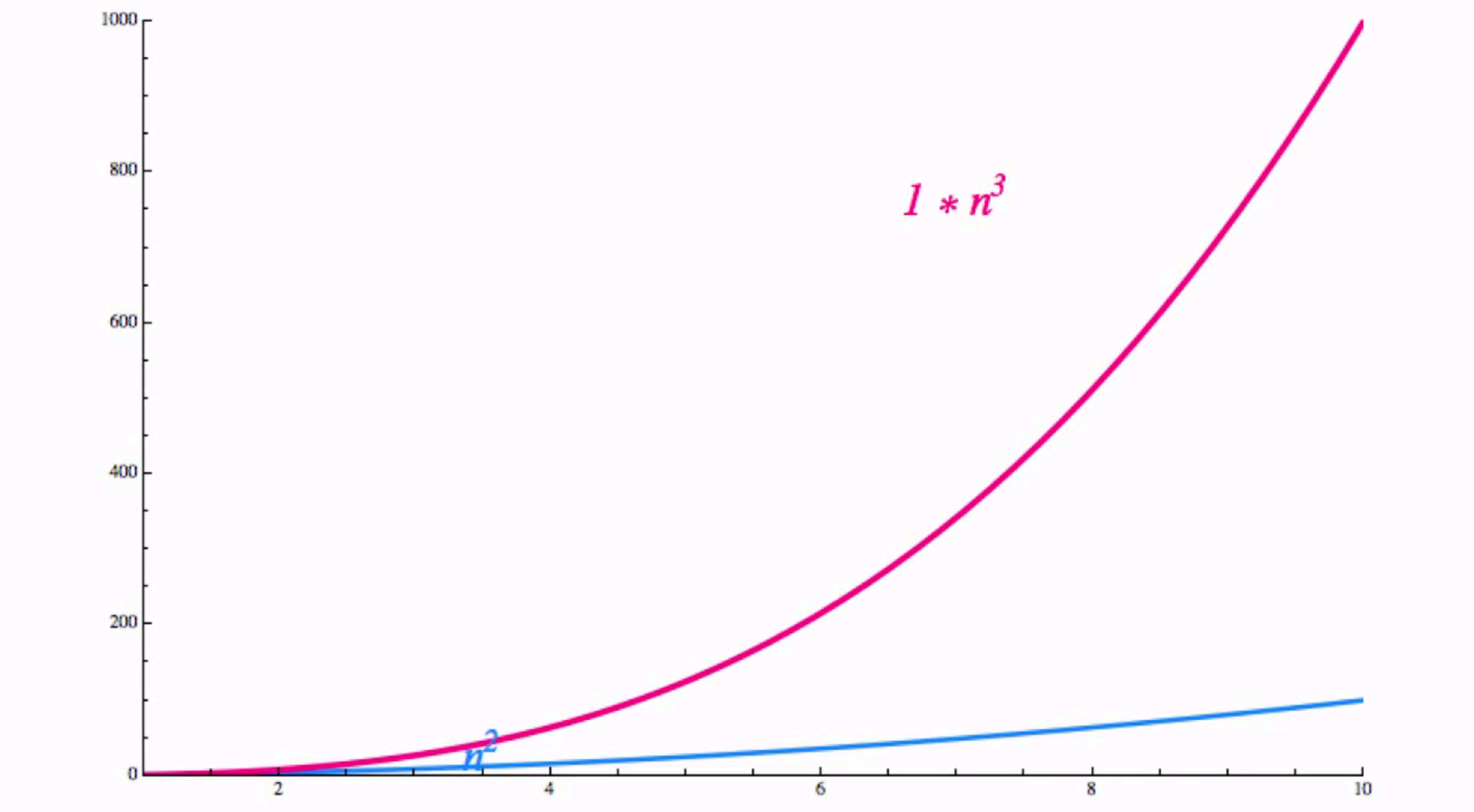
f(n) is Θ(g(n))

Example 2

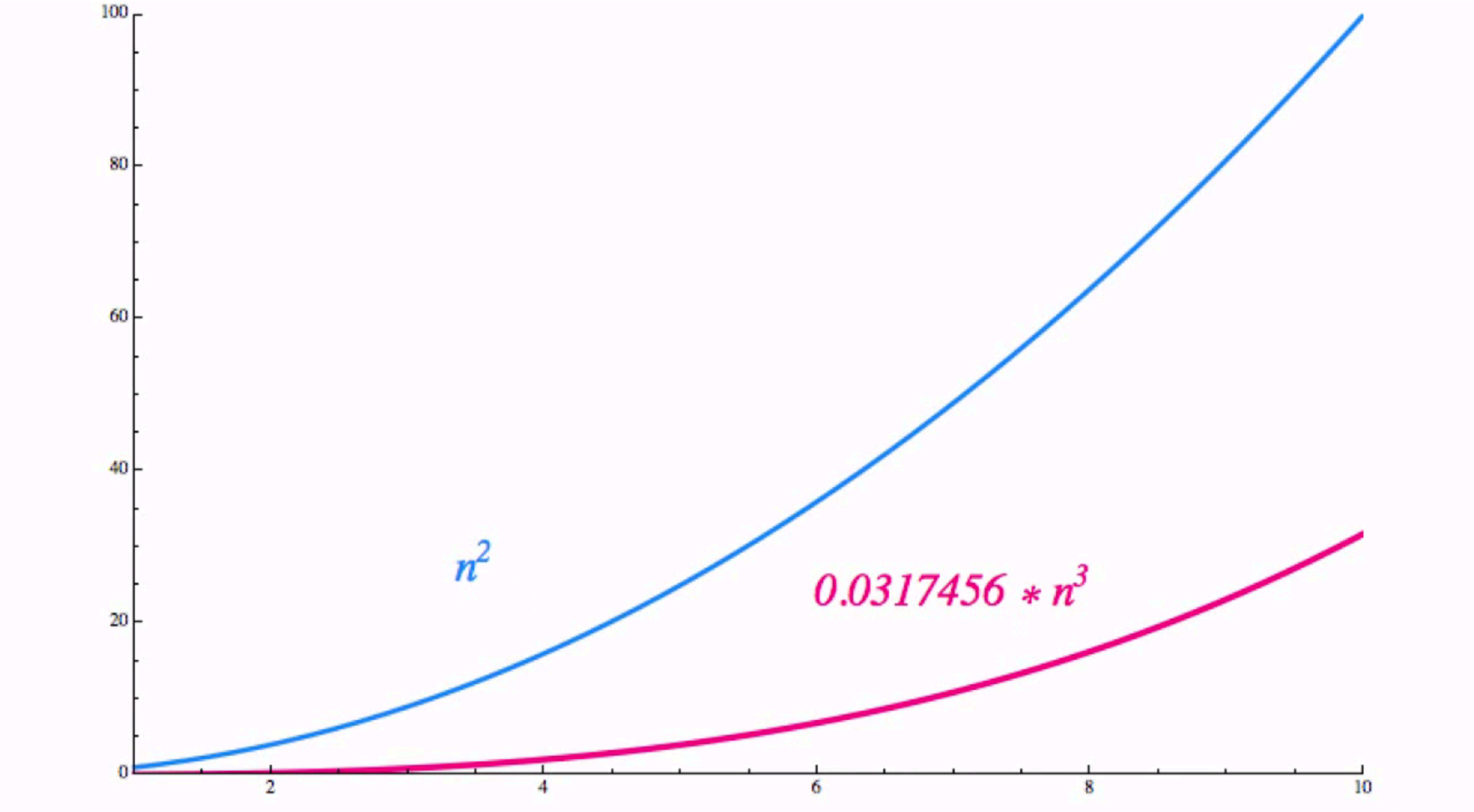
Let say,

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

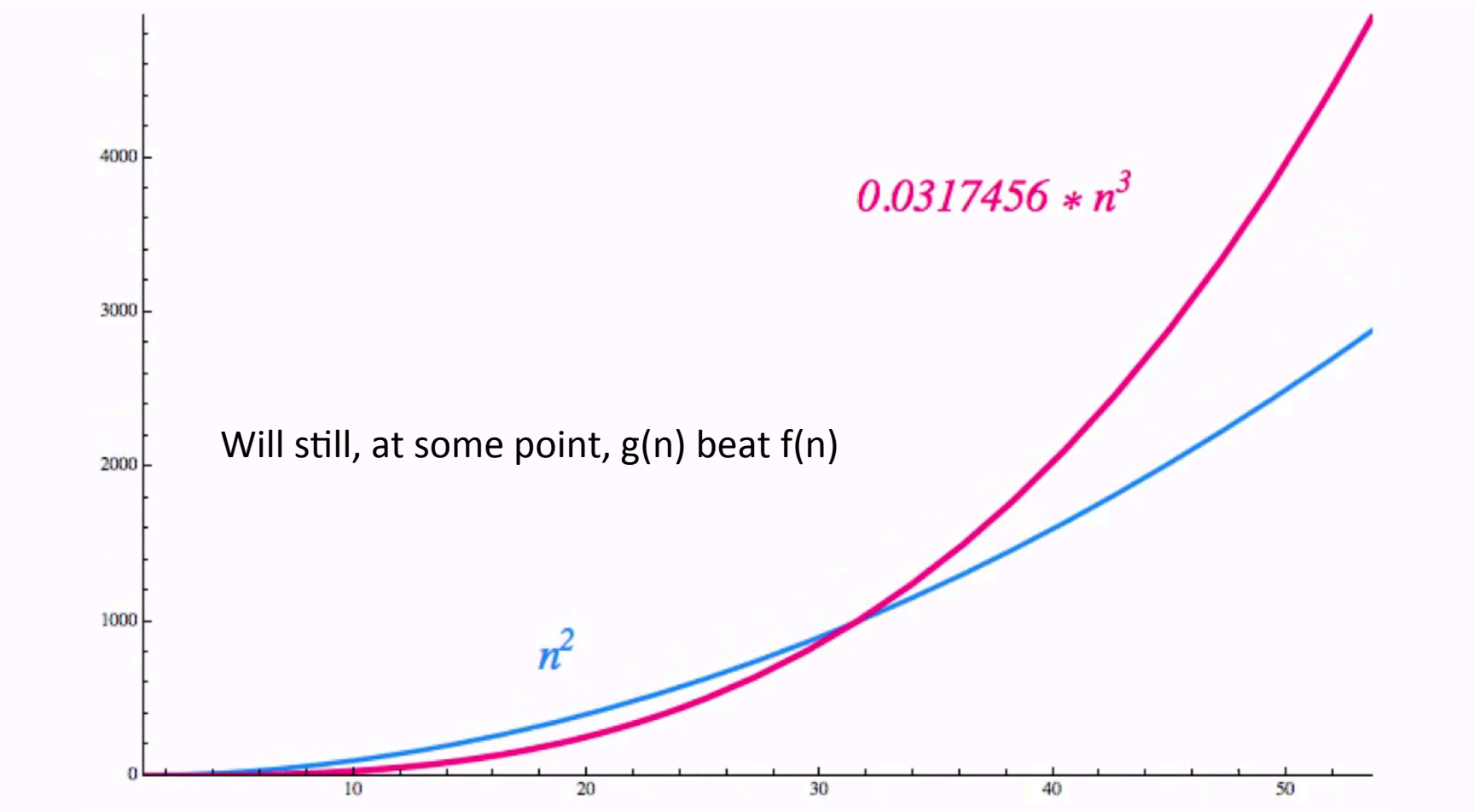
(1) f(n) is O(g(n))



(2.1) If k<0



(2.2) f(n) is O(g(n))



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

Therefore..

- From the example, regardless of what is the value of k, g(n) will always beats f(n).

f(n) is O(g(n))

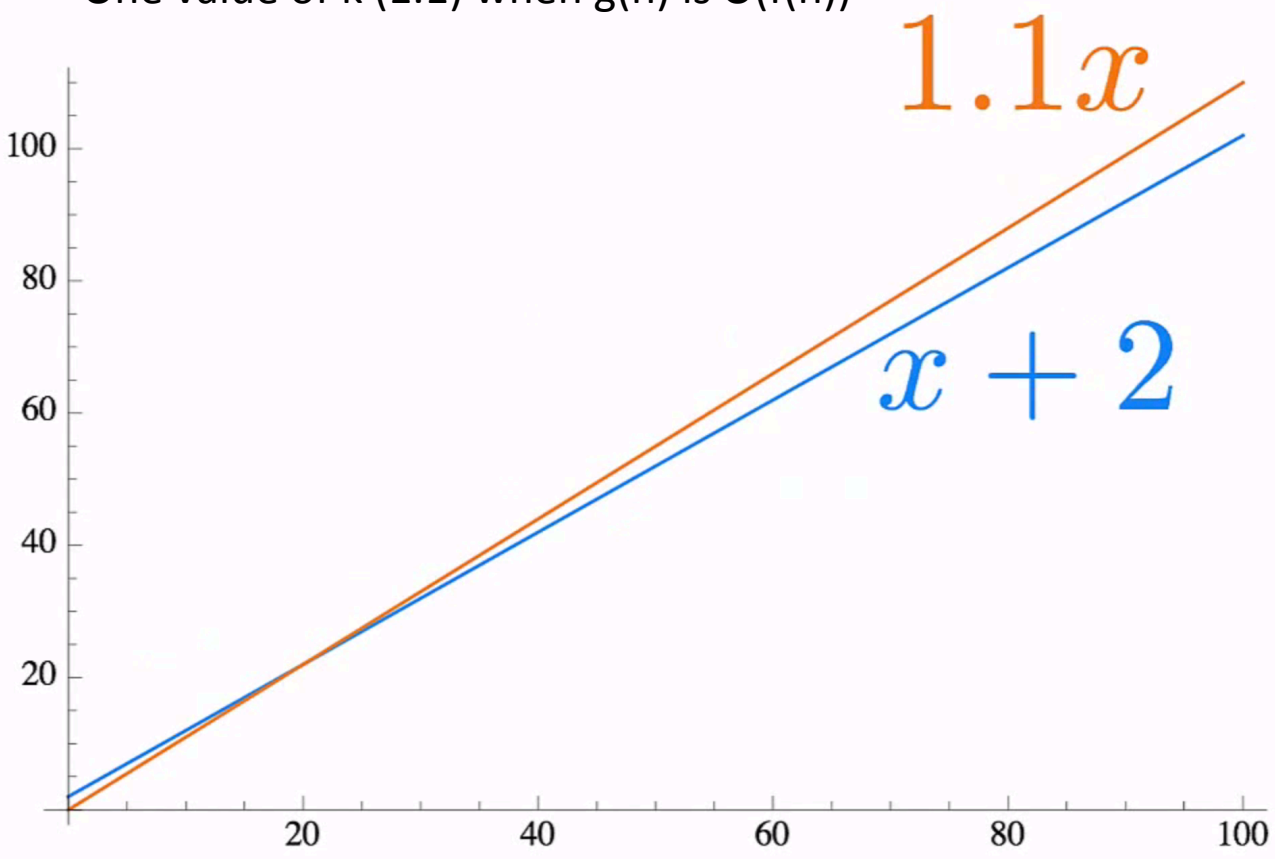
Question…

- Given the following two functions:
  - $f(n) = n$
  - $g(n) = n + 2$

1. Is $f(n) \in O(g(n))$?
2. Is $g(n) \in \Omega(f(n))$?
3. Is $g(n) \in O(f(n))$? $(n + 2 \leq n.k)$?

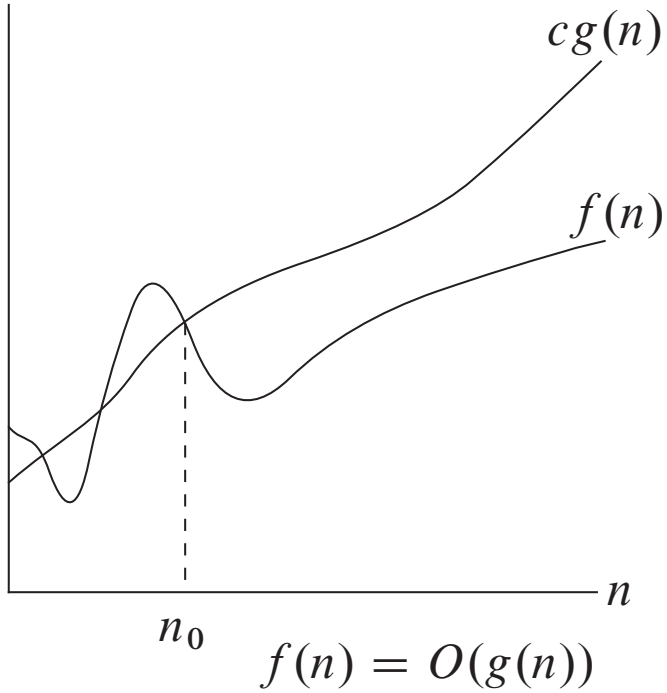
g(n) is O(f(n))

One value of k (1.1) when g(n) is O(f(n))



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).



Formal justification

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

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

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

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

$3n + 2 \leq cn$; $c = 4$ – *where, it is an instance where the statement holds*

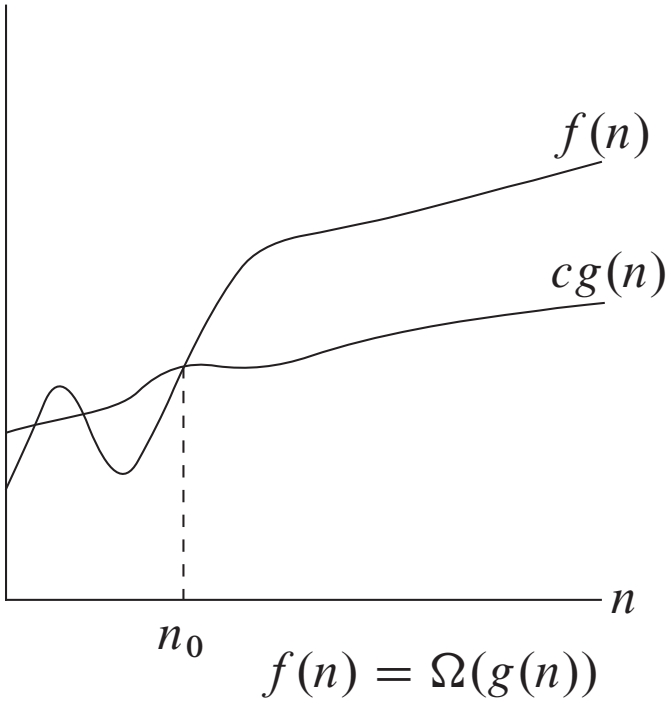
$= 3n + 2 \leq 4n$

$n \geq 2$

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).



Formal justification

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

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

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

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

$3n + 2 \geq cn$; $c = 1$ – *where, it is an instance where the statement holds*

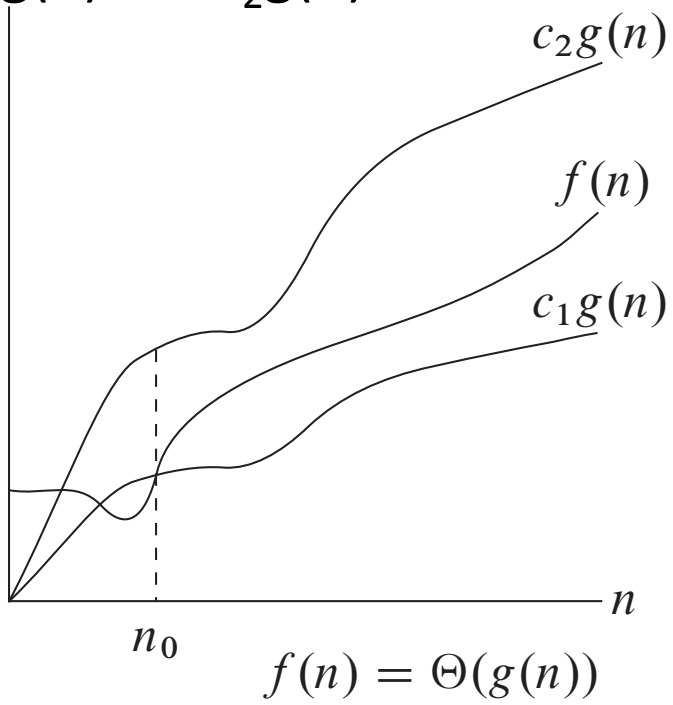
$= 3n + 2 \geq n$

$n \geq 1$

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

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.



Formal justification

$c_1 g(n) \leq f(n) \leq c_2 g(n)$; $n \geq n_0$, $n_0 \geq 1$ 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) $f(n) \geq c_1 g(n)$; $n_0 \geq 1$ and $c > 0$
$3n + 2 \geq c_1 n$; $c_1 = 1$ – *where, it is an instance where the statement holds*
$= 3n + 2 \geq n$
$n \geq 1$

(2) $f(n) \leq c_2 g(n)$; $n_0 \geq 1$ and $c > 0$
$3n + 2 \leq c_2 n$; $c = 4$ – *where, it is an instance where the statement holds*
$= 3n + 2 \geq n$
$n \geq 2$

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

Analysis - revisit

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

Time complexity analysis of algorithm

- Types of algorithm
  - iterative
  - recursive
- If a program has no iteration/recursive – running time is constant
  - O(1)
- Problems that can be solve with iterative, can be solve using recursion as well, and vice versa.
- Time complex analysis is different.

Time Complexity Analysis

- [Recap] 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	600,00
1000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

Time Complexity Analysis

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

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

- Changing of units does not effect 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:

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

[Recap] Time complexity analysis using cost (Iterative)

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	$n-1$

[Recap] Time complexity

- Sum of cost for every statement.

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

[Recap] Best-case and worst-case

- For each j = 2, 3, …, n, we then find that A[i] ≤ *key* in line 5 when i has its initial value of j-1. Thus, $t_j = 1$ for j = 2, 3, …, n, and the best-case running time is:

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

T(n) – a linear function

- If the array is in reverse sorted order - that is, in decreasing order – produce the worst case results. We must compare each element A[j] with each element in the entire sorted subarray A[1, …, j – 1], and so $t_j = j$ for j = 2, 3, …, n.We find that in the worst case, the running time of INSERTION-SORT is:

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

T(n^2) – a quadratic function

Simpler way..

- Determine higher order term by calculating the time a statement is executed:
  - operation
  - comparison
  - loop
  - pointer reference
  - function calls

Example 1

- Consider the following iterative program:

```
function1 (){
   for (int i = 1; i <= n; i ++) {
      printf("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:

```
function2(){
for (int i = 1; i <=n; i ++) {
          for (int j = 1; j <=n; j ++) {
          printf("Hello world");
        }
    }
```

Outer 1: n times

Inner loop: n times

Complexity = $O(n^2)$

Example 3

- Consider the following iterative program:

```
function3(){
for (int i = n/2; i <=n; i ++) {
   for (int j <= n/2; j <=n; j ++) {
      for (int k = 1; k <= n; k * 2) {
         printf("Hello world");
         }
      }
   }
}
```

Loop 1: n/2 times

Loop 2: n/2 times

Loop 3: $\log_2 n$

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

Recursive Algorithm

- Recurrences go hand in hand with the divide-and-conquer paradigm
  - because they give a natural way to characterize the running times of divide-and-conquer algorithms.
- A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Merge-sort algorithm

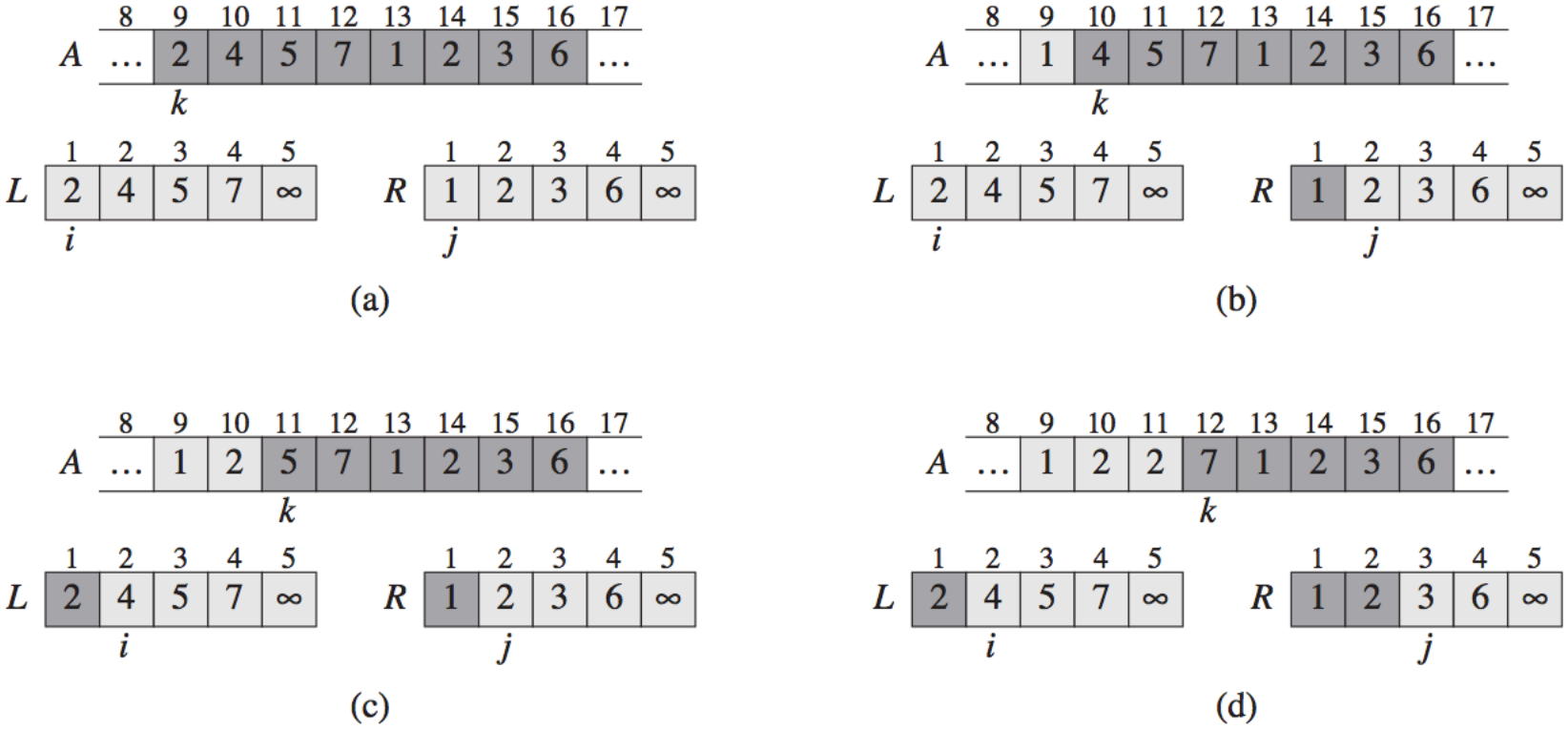
- Line 1 computes the length n1 of the subarray A[p..q], and line 2 computes the length n_2 of the subarray A[q+1..r] .
- We create arrays L and R ("left"and"right"), of lengths n_1 +1 and n_2 + 1, respectively, in line 3; the extra position in each array will hold the sentinel (∞).
- The for loop of lines 4–5 copies the subarray A[p..q] into L[1..n_1] , and the for loop of lines 6–7 copies the subarray A[q+1..r] into R[1..n_2].
- Lines 8–9 put the sentinels at the ends of the arrays L and R.

MERGE(A, p, q, r)

```
1  n1 = q − p + 1
2  n2 = r − q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

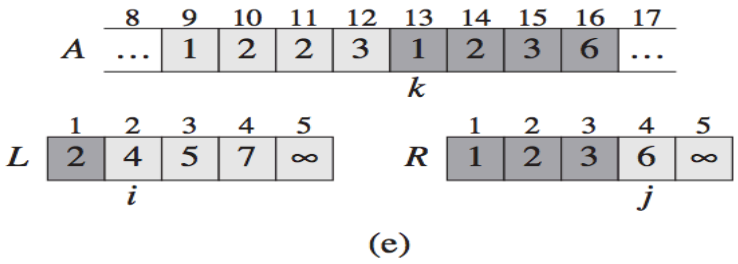
Merge-sort algorithm

- Line 10-17 is illustrated as follows (1):

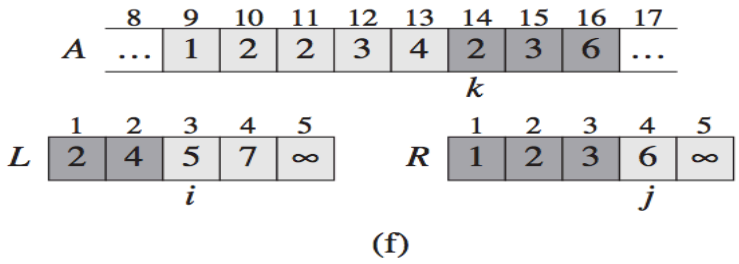


Merge-sort algorithm

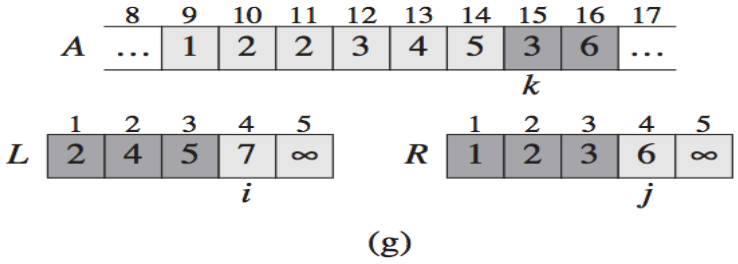
- Line 10-17 is illustrated as follows (2):



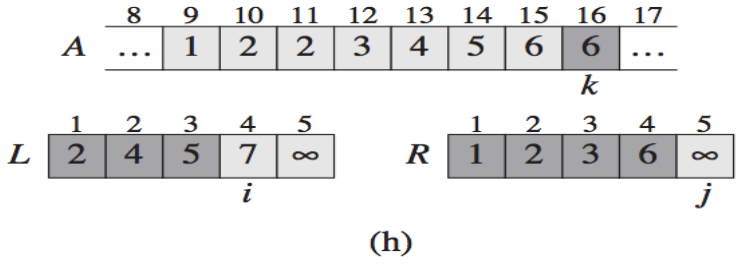
(e)



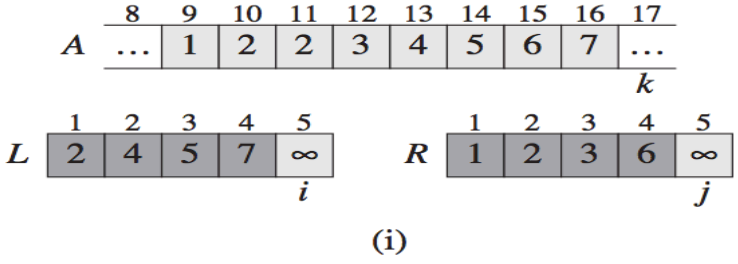
(f)



(g)



(h)



(i)

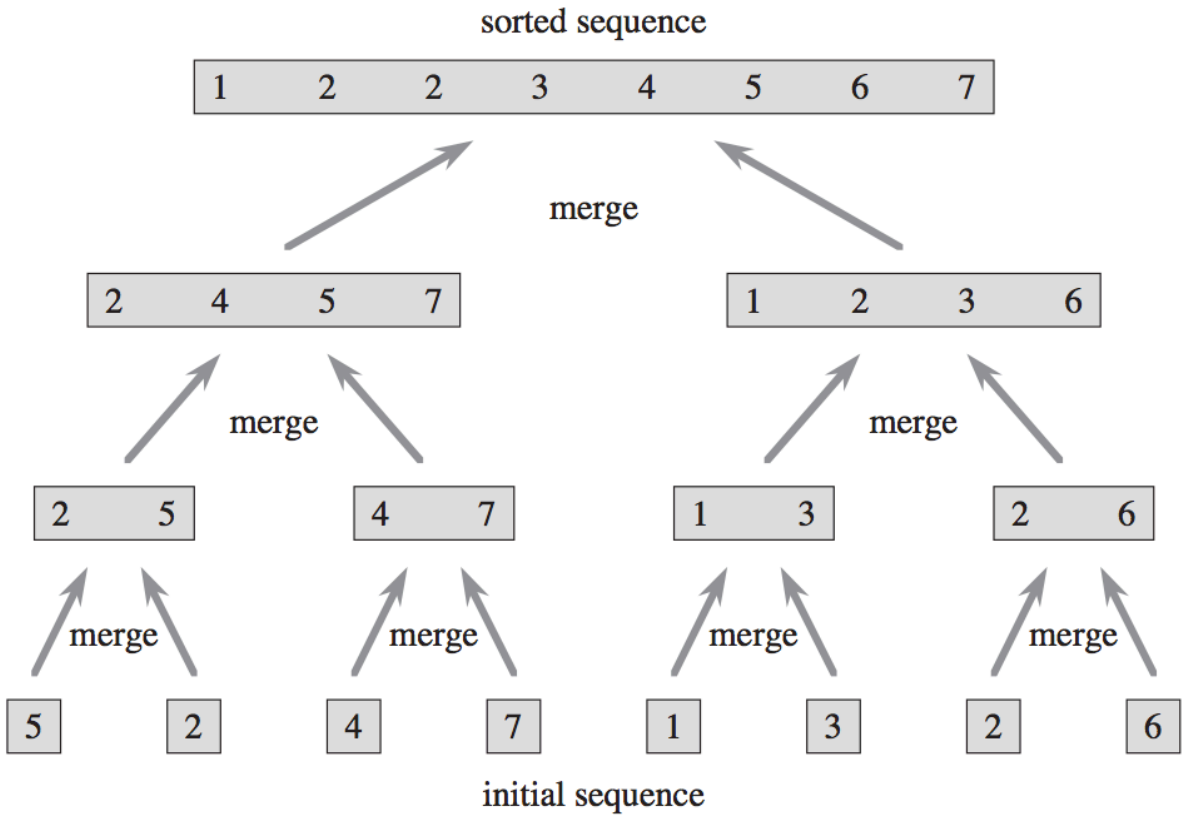
Merge-sort as subroutine

- The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray A[p..r]. If p ≥ r, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions A[p..r] into two subarrays:
  - A[p..q], containing [n/2] elements,
  - A[q+1..r], containing [n/2] elements.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Merge-sort as subroutine

- The operation of merge sort on the array A.
- The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.



Analysing time complexity (recursive)

- To calculate the T(n) for recursion (3 step):
  - **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, D(n) = Θ(1).
  - **Conquer:** We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
  - **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time Θ(n) and so C(n) = D(n).
    - D(n) time to divide the problem into subproblems
    - C(n) time to combine the solutions to the subproblems into the solution to the original problem

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

T(n) = Θ (n lg n)

Methods for solving recurrences

- There are three methods for solving recurrences:
  - ***Substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
  - R***ecursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
  - ***Master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

    - where a ≥ 1, b > 1, and f .n/ is a given function

Substitution Method

- The ***substitution method*** for solving recurrences comprises two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.

Example (1)

- The following is a recursive function A

```
A(n){
 if (n > 1)
    return (A(n-1))
}
```

- Using substitution method, find the order of function A.

Solution

$$T(n) = 1 + T(n-1) \quad -(1)$$

$$T(n-1) = 1 + T(n-2) \quad -(2)$$

$$T(n-2) = 1 + T(n-3) \quad -(3)$$

(2) in (1)

$$T(n) = 1 + 1 + T(n-2)$$

$$= 2 + T(n-2) \quad \text{- put in (3)}$$

$$= 3 + T(n-3)$$

$$\vdots$$

$$= k + T(n-k) \quad -$$

$$= n-1 + T(n-(n-1))$$

$$= n-1 + T(1)$$

$$= n-1+1$$

$$= n$$

Find the stopping value.

$T(n) = 1 + T(n-1)$; $n > 1$

$\therefore$ stopping value

$T(n) = 1$; $n = 1$

$n - k = 1$

$\therefore k = n-1$

$$T(n) = O(n)$$

Example (2)

- Given the recursive function B is:

$$T(n) = n + T(n-1)$$

- Using substitution method, find the order of function B.

Solution

$$T(n) = n + T(n-1) \quad ; n > 1 \quad \text{— (1)}$$
$$= 1 \quad ; n = 1 \quad \text{— Base condition}$$

$$T(n-1) = (n-1) + T(n-2) \quad \text{— (2)}$$
$$T(n-2) = (n-2) + T(n-3) \quad \text{— (3)}$$

(2) in (1)

$$T(n) = n + (n-1) + T(n-2) \quad \text{— put in (3)}$$
$$= n + (n-1) + (n-2) + T(n-3)$$
$$\vdots$$
$$= n + (n-1) + (n-2) + \cdots + (n-k) + T(n-(k+1))$$

$T(n-(k+1)) = 1$

$$= n + (n-1) + (n-2) + (n-(n-2)) + T(n-(n-2)+1)$$
$$= n + (n-1) + (n-2) + 2 + 1$$
$$= \frac{n(n+1)}{2} \quad \text{— sum of natural numbers}$$
$$= \frac{n^2 + n}{2}$$
$$= n^2$$

$n - (k+1) = 1$

$k = n - 2$

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

Recursion Tree Method

- In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- To determine the total cost, sum the costs within each level of the tree to obtain a set of per-level costs is calculated, and then we sum all the per-level costs.
- A recursion tree is best used to generate a good guess, which you can then verify by the substitution method.

Example (1)

- Given the recursive function A is:

$$T(n) = 2T(n/2) + C$$

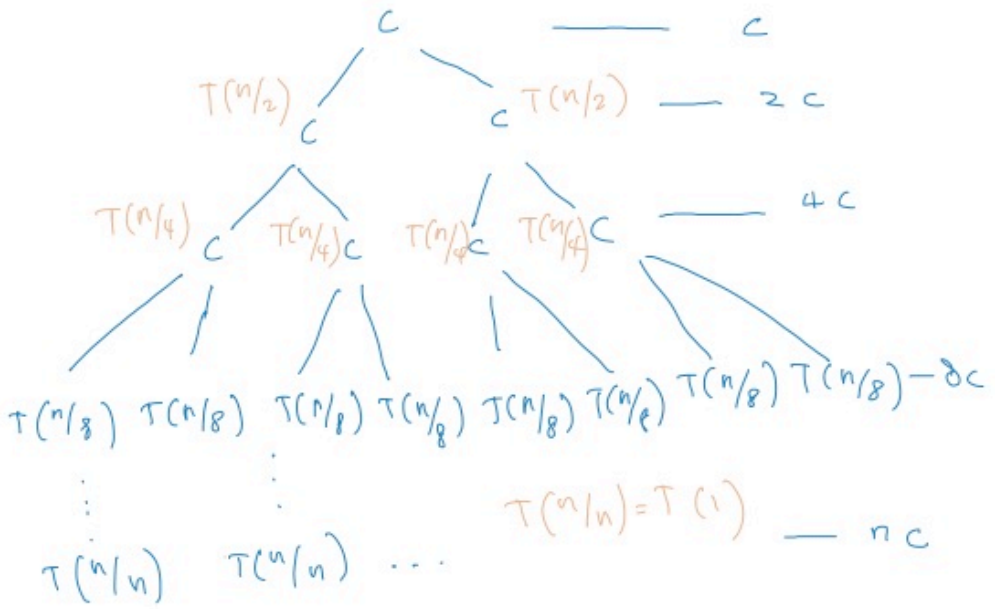
- Using recursion tree method, find the order of function A.

Solution

$T(n) = 2T(n/2) + c \quad ; n > 1$

$= c \quad ; n = 1$

$T(n)$ is to do c (constant) amount of work by breaking it down into 2 smaller problem.



$T(n) = c + 2c + 4c + 8c \cdots + nc$

$= c(1 + 2 + 4 + 8 \cdots + n)$ — assume $n = 2^k$

$= c(2^0 + 2^1 + 2^2 + 2^3 \cdots 2^k$

$= c\left(\frac{1(2^{k+1} - 1)}{(2-1)}\right) = c(2n - 1)$

$= n$

$T(n) = O(n)$

Example (2)

- Given the recursive function B is:

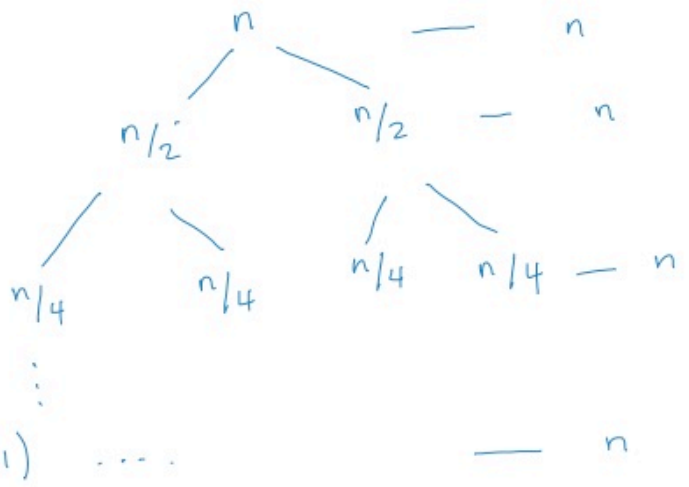
$$T(n) = 2T(n/2) + n$$

- Using recursion tree method, find the order of function B.

Solution

$$T(n) = 2T(n/2) + n \;;\; n>1$$
$$= 1 \;;\; n=1$$

$T(n/2) = 2T(n/4) + n/2$



$n/2^0, n/2^1, n/2^2, n/2^3 \ldots n/2^k$

level 0 1 2 3 k+1

The levels - $n = 1$ is at $k+1$

$n/2^k$

$n = 2^k$

$k = \log n$

$= (\log n + 1)\, n$ (times)

$= n \log n$

$T(n) = O(n \log n)$

Master Method

- The master method provides a “cookbook” method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

  - where a ≥ 1 and b > 1 are constants and f(n) is an asymptotically positive function.

Master Theorem

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Easier alternative definition

$$T(n) = aT(n/b) + \Theta(n^k \log^p n)$$

$a \geq 1$, $b > 1$, $k \geq 0$ and p is real numbers

(1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

(2) if $a = b^k$

a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$

c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

(3) if $a < b^k$

a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

b) if $p < 0$, then $T(n) = O(n^k)$

Example (1)

- Using the masters theorem, solve the following problem:

$$T(n) = 3T(n/2) + n^2$$

Solution

$$T(n) = 3T(n/2) + n^2$$

$a = 3$, $b = 2$, $k = 2$ $p = 0$

$a \quad b^k$

$3 < 4$; $p = 0$ — Case 3(a)

$$T(n) = \Theta(n^2 \log^0 n)$$

$$= \Theta(n^2)$$

Example (2)

- Using the masters theorem, solve the following problem:

$$T(n) = 4T(n/2) + n^2$$

Solution

$$T(n) = 4T(n/2) + n^2$$

$a = 4, \; b = 2, \; k = 2, \; p = 0$

$a \quad b^k$

$4 = 4 \; ; \; p = 0$ — Case 2 (a)

$$T(n) = \Theta(n^{\log_4 2} \log n)$$

$$= \Theta(n^2 \log n)$$

Example (3)

- Using the masters theorem, solve the following problem:

$$T(n) = 6T(n/3) + n^2 \log n$$

Solution

$$T(n) = 6T(n/3) + n^2 \log n$$

$a = 6$, $b = 3$, $k = 2$ $p = 1$

$6 < 9$; $p = 1$ — case 3 (a)

$\Theta(n^2 \log n)$

Example (4)

- Using the masters theorem, solve the following problem:

$$T(n) = 0.5T(n/2) + 1/n$$

Solution

- a <= 1, therefore masters theorem cannot be applied to solve the problem.

Inadmissible equation

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$

  a is not a constant; the number of subproblems should be fixed

- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

  non-polynomial difference between f(n) and $n^{\log_b a}$ (see below)

- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

  a<1 cannot have less than one sub problem

- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

  f(n) which is the combination time is not positive

- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

  case 3 but regularity violation.

These equation cannot be solve using Masters Theorem!

In the second inadmissible example above, the difference between $f(n)$ and $n^{\log_b a}$ can be expressed with the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{\frac{n}{\log n}}{n^{log_2 2}} = \frac{n}{n \log n} = \frac{1}{\log n}$. It is clear that $\frac{1}{\log n} < n^{\epsilon}$ for any constant $\epsilon > 0$. Therefore, the difference is not polynomial and the Master Theorem does not apply.

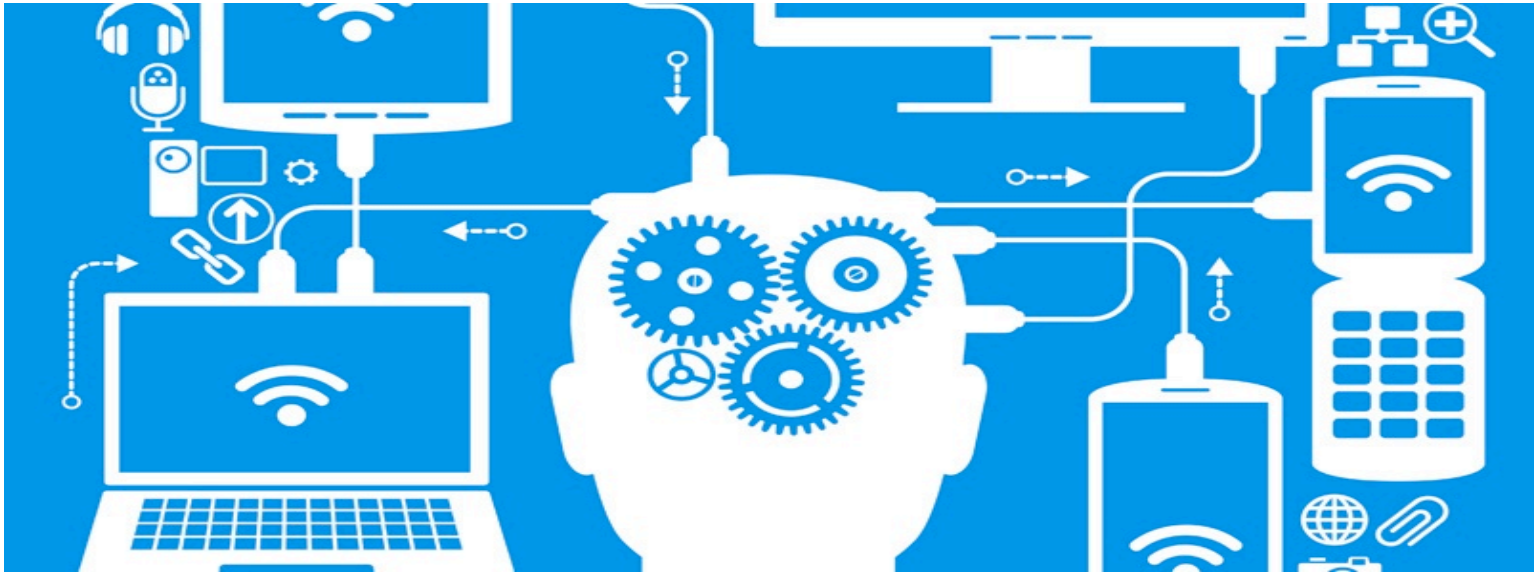
Key points

- Algorithm plays a major role in computing and has profound impact to the computing solution and performance.
- Insertion algorithm is an iterative algorithm while merge sort is a kind of recursive algorithm.
- Growth of function is the measurement for algorithms running time.
- The analysis of time complexity between the two types of algorithm is different.
- For recurrences, there are three methods:
  - substitution
  - recursion tree
  - master

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
- Time complexity analysis of iterative program
  - https://www.youtube.com/watch?v=FEnwM-iDb2g&index=2&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0
- Time complexity analysis of recursive program
  - https://www.youtube.com/watch?v=gCsfk2ei2R8&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0&index=3
- Masters Theorem
  - https://www.youtube.com/watch?v=lPUhHmgrpik&list=PLEbnTDJUr_IeHYw_sfBOJ6gk5pie0yP-0&index=5

In the next lecture..



Lecture 2: Sorting Algorithm