

CS430-01

Introduction to Algorithms

1. Algorithm & Analysis

Michael Choi

Dept. of Computer Science

IIT

# Objectives

- Analyze techniques for solving problems
- Define an algorithm
- Define growth rate of an algorithm as a function of input size
- Define Worst case, average case, and best case complexity analysis of algorithms
- Classify functions based on growth rate
- Define growth rates: Big O, Theta, and Omega

# Roles of Algorithms in Computing

- What is algorithm?
  - A well defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output
  - A sequence of computational steps that transform the input into the output
  - A tool for solving a well-specified computational problem

# Roles of Algorithms in Computing

What is algorithm?

- Algorithms are the ideas behind computer programs
- An algorithm is the thing which stays the same whether the program is in Pascal running on a Cray computer or is in BASIC running on a MacPro.
- To be interesting, an algorithm has to solve a general, specified problem. An algorithmic problem is specified by describing the set of instances it must work on and what desired properties the output must have.

# Define a sorting problem

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

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

- Given the input sequence  $\langle 31, 41, 59, 26, 41, 58 \rangle$
- A sorting algorithm returns  
 $\langle 26, 31, 41, 41, 58, 59 \rangle$

We seek algorithms which are correct and efficient

# Correctness

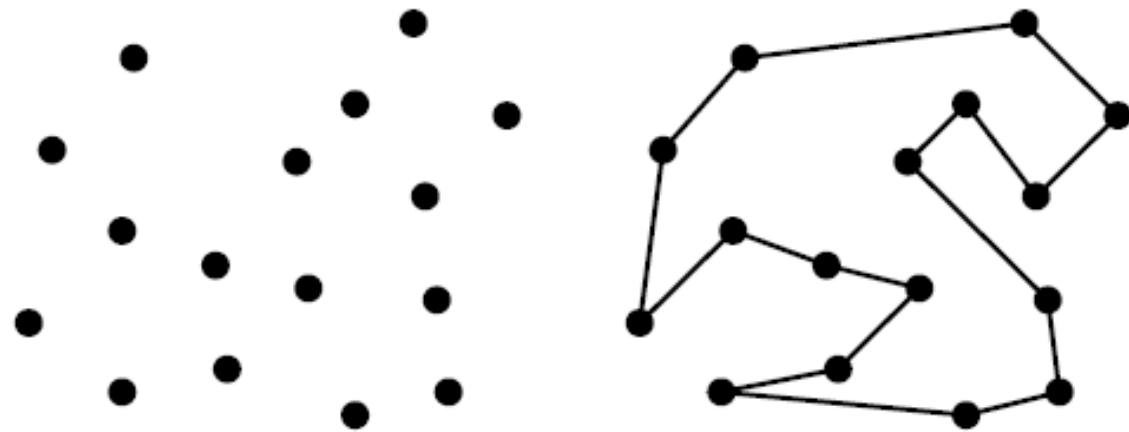
- For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem
- For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements

# Correctness is not obvious

The following problem arises often in manufacturing and transportation testing applications.

Suppose you have a robot arm equipped with a tool, say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points, so the robot visits (and solders) the first contact point, then visits the second point, third, and so forth until the job is done.

Since robots are expensive, we need to find the order which minimizes the time (ie. travel distance) it takes to assemble the circuit board.



You are given the job to program the robot arm. Give me an algorithm to find the best tour!



# Nearest Neighbor Tour

A very popular solution starts at some point  $p_0$  and then walks to its nearest neighbor  $p_1$  first, then repeats from  $p_1$ , etc. until done.

Pick and visit an initial point  $p_0$

$p = p_0$

$i = 0$

While there are still unvisited points

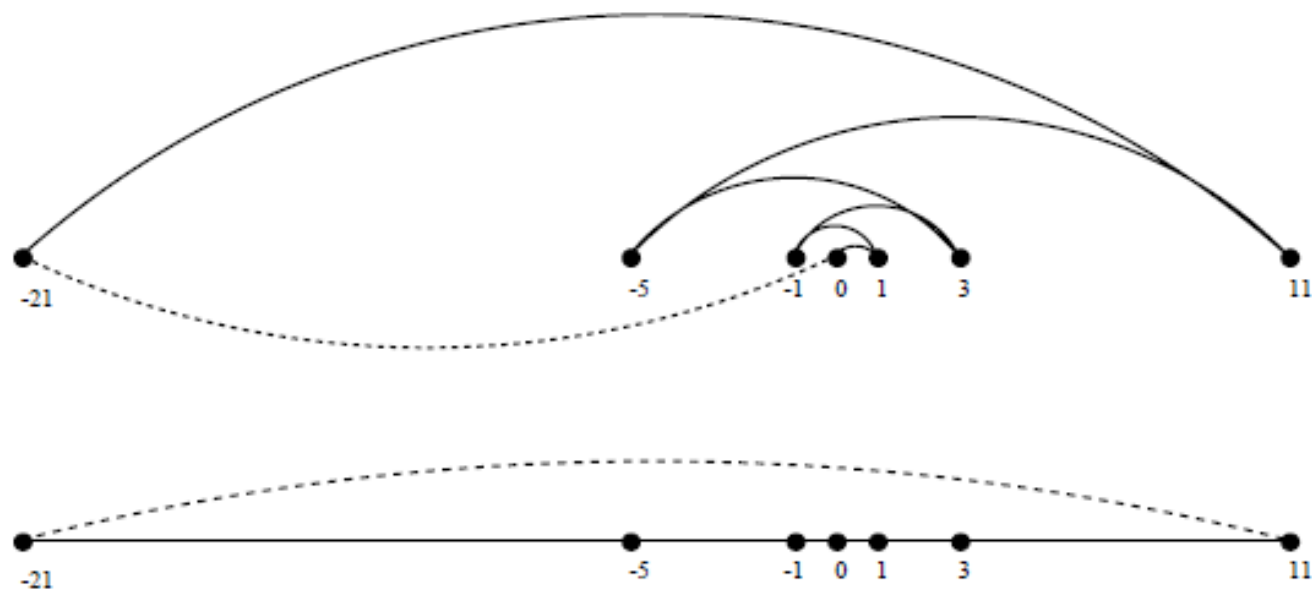
$i = i + 1$

    Let  $p_i$  be the closest unvisited point to  $p_{i-1}$

    Visit  $p_i$

Return to  $p_0$  from  $p_i$

This algorithm is simple to understand and implement and very efficient. However, it is **not correct!**



Always starting from the leftmost point or any other point will not fix the problem.

# Closest Pair Tour

Always walking to the closest point is too restrictive, since that point might trap us into making moves we don't want.

Another idea would be to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch to be formed, until we have a single chain with all the points in it.

Let  $n$  be the number of points in the set

$d = \infty$

For  $i = 1$  to  $n - 1$  do

    For each pair of endpoints  $(x, y)$  of partial paths

        If  $\text{dist}(x, y) \leq d$  then

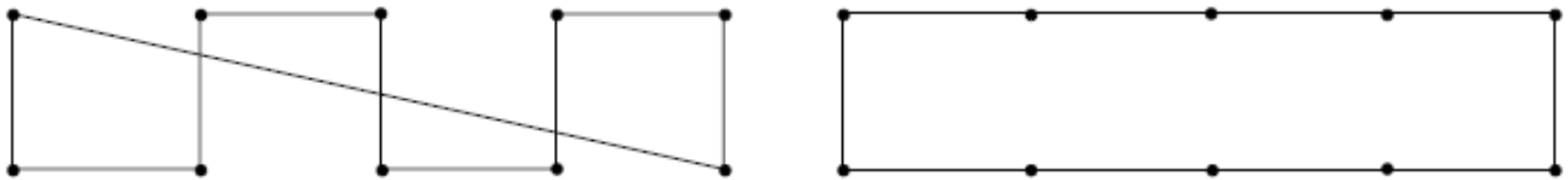
$x_m = x, y_m = y, d = \text{dist}(x, y)$

        Connect  $(x_m, y_m)$  by an edge

Connect the two endpoints by an edge.

# Closest Pair Tour

Although it works correctly on the previous example, other data causes trouble:



This algorithm is **not correct**!

# A Correct Algorithm

We could try all possible orderings of the points, then select the ordering which minimizes the total length:

$d = \infty$

For each of the  $n!$  permutations  $\Pi_i$  of the  $n$  points

    If ( $cost(\Pi_i) \leq d$ ) then

$d = cost(\Pi_i)$  and  $P_{min} = \Pi_i$

Return  $P_{min}$

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour.

Because it tries all  $n!$  permutations, it is extremely slow, much too slow to use when there are more than 10-20 points.

No efficient, correct algorithm exists for the *traveling salesman problem*, as we will see later.

# Expressing Algorithms

: We need some way to express the sequence of steps comprising an algorithm.

In order of increasing precision, we have English, pseudocode, and real programming languages. Unfortunately, ease of expression moves in the reverse order.

I prefer to describe the *ideas* of an algorithm in English, moving to pseudocode to clarify sufficiently tricky details of the algorithm.

# Methodology

- Approach to solving a problem
- Independent of Programming Language
- Independent of Style
- Sequential Search versus Binary Search
- Which technique results in the most efficient solution?

# Problem?

- A question to which an answer is sought
- Parameters
  - Input to the problem
  - Instance: a specific assignment of values to the input parameters
- Algorithm:
  - Step-by-step procedure
  - Solves the Problem
- RAM (Random Access Machine) model
  - One processor, one instruction at a time, no concurrency allows
  - No memory hierarchy concern



# The problem of sorting

**Input:** array  $A[1 \dots n]$  of numbers.

**Output:** permutation  $B[1 \dots n]$  of  $A$  such that  $B[1] \leq B[2] \leq \dots \leq B[n]$ .

e.g.  $A = [7, 2, 5, 5, 9.6] \rightarrow B = [2, 5, 5, 7, 9.6]$

How can we do it efficiently ?

# Why Sorting?

- Obvious applications
  - Organize an MP3 library
  - Maintain a telephone directory
- Problems that become easy once items are in sorted order
  - Find a median, or find closest pairs
  - Binary search, identify statistical outliers
- Non-obvious applications
  - Data compression: sorting finds duplicates
  - Computer graphics: rendering scenes front to back

# Insertion sort

INSERTION-SORT ( $A, n$ )  $\triangleright A[1 \dots n]$   
for  $j \leftarrow 2$  to  $n$   
    insert key  $A[j]$  into the (already sorted) subarray  $A[1 \dots j-1]$   
    by pairwise key-swaps down to its  
    new location

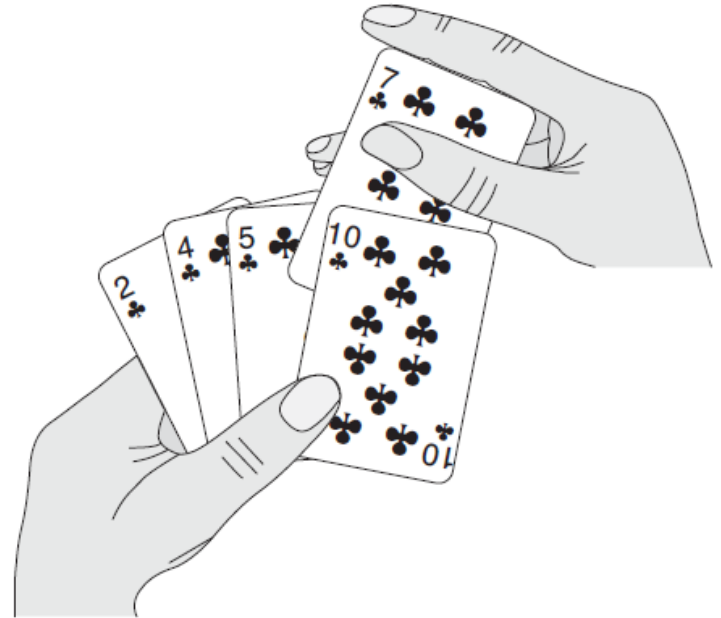
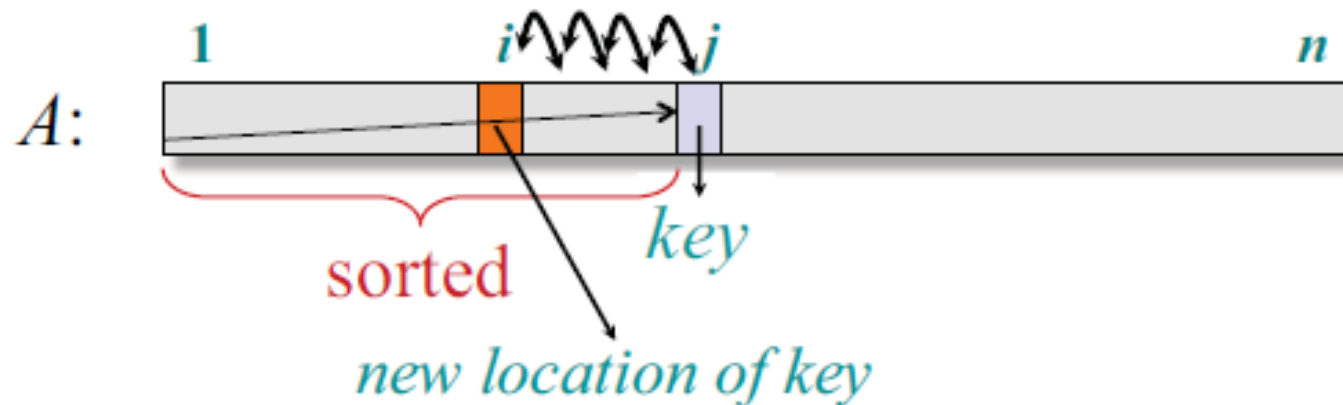
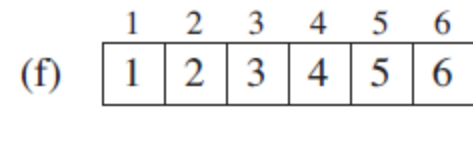
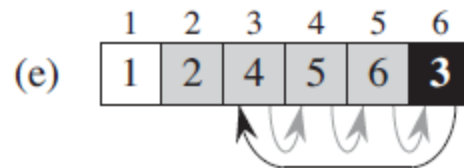
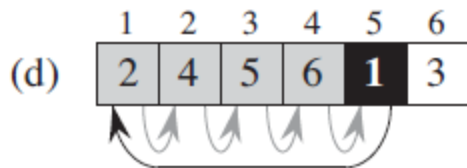
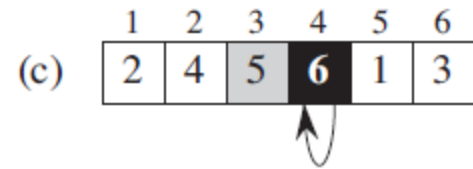
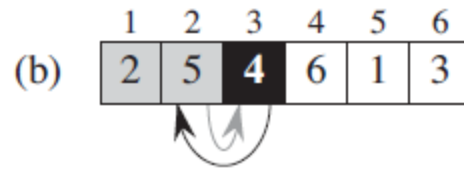
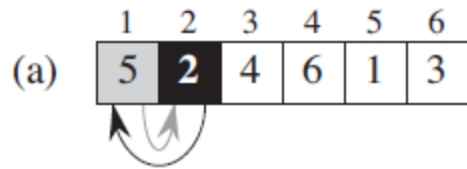


Illustration of iteration  $j$



# Insertion Sort



# Example of insertion sort

8 2 4 9 3 6



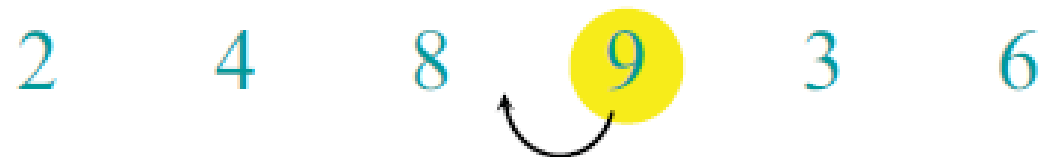
2 8 4 9 3 6



2 8 4 9 3 6



2 4 8 9 3 6



2 4 8 9 3 6



2 3 4 8 9 6



2 3 4 6 8 9 *done*

## INSERTION-SORT( $A$ )

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

**Initialization:** It is true prior to the first iteration of the loop.

**Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.

**Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

## ***insertionSort***

```
for count going from 1 through SIZE - 1  
    insertElement(0, count)
```

### ***InsertElement(startIndex, endIndex)***

```
Set finished to false
```

```
Set current to endIndex
```

```
Set moreToSearch to true
```

```
while moreToSearch AND NOT finished
```

```
    if values[current] < values[current - 1]
```

```
        swap(values[current], values[current - 1])
```

```
        Decrement current
```

```
        Set moreToSearch to (current does not equal startIndex)
```

```
    else
```

```
        Set finished to true
```

(a)

values	
[0]	36
[1]	10
[2]	24
[3]	6
[4]	32

(b)

values	
[0]	36
[1]	10
[2]	24
[3]	6
[4]	32

→

values	
[0]	10
[1]	36
[2]	24
[3]	6
[4]	32

(c)

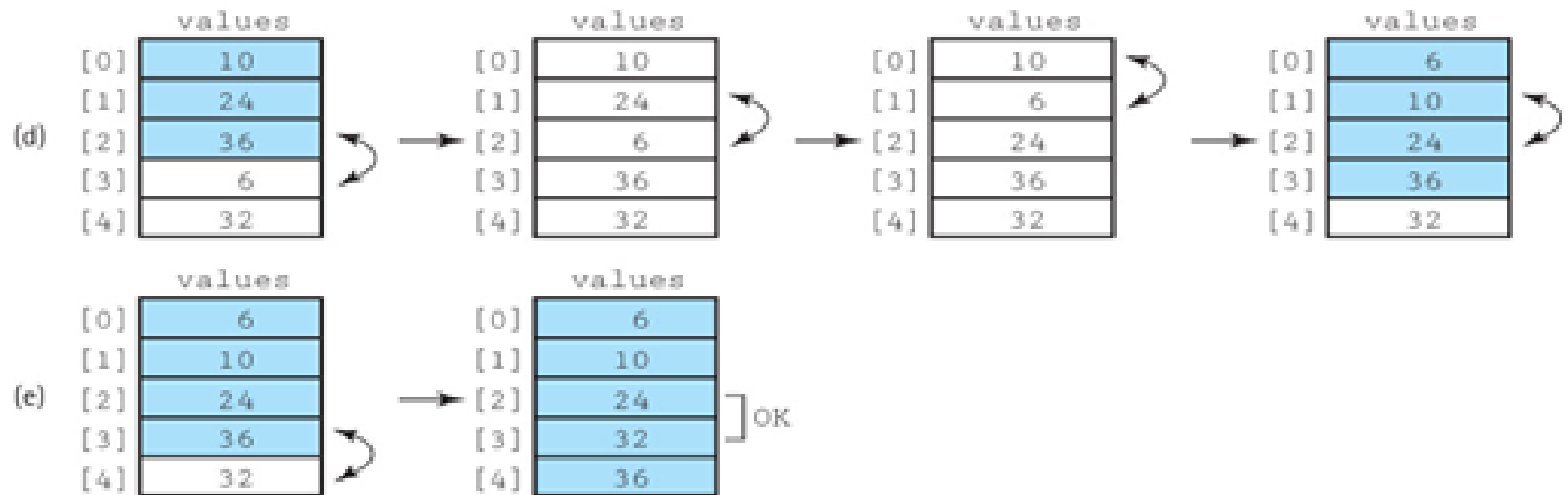
values	
[0]	10
[1]	36
[2]	24
[3]	6
[4]	32

→

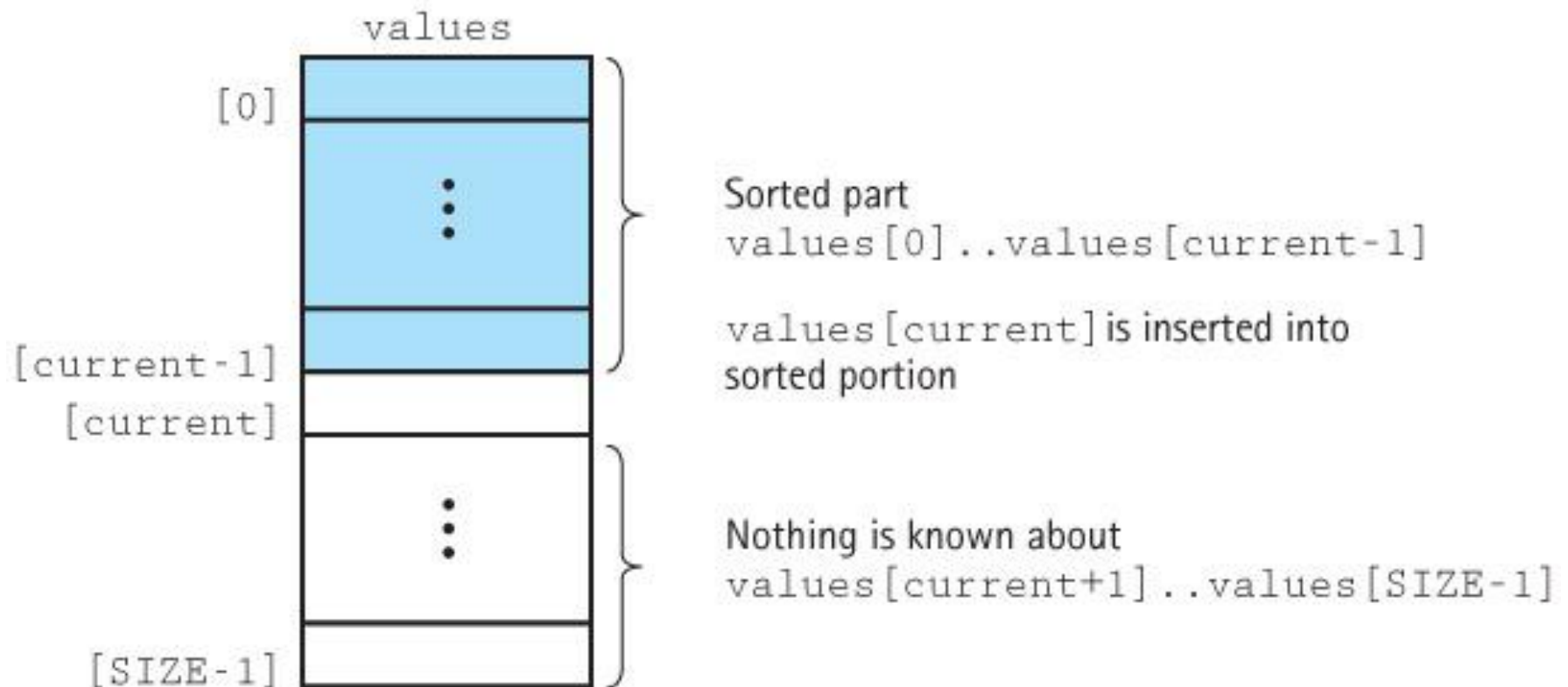
values	
[0]	10
[1]	24
[2]	36
[3]	6
[4]	32

OK





# Insertion Sort Snapshot



# Insertion Sort Code

```
static void insertElement(int startIndex, int endIndex)
// Upon completion, values[0]..values[endIndex] are sorted.
{
    boolean finished = false;
    int current = endIndex;
    boolean moreToSearch = true;
    while (moreToSearch && !finished)
    {
        if (values[current] < values[current - 1])
        {
            swap(current, current - 1);
            current--;
            moreToSearch = (current != startIndex);
        }
        else
            finished = true;
    }
}

static void insertionSort()
// Sorts the values array using the insertion sort algorithm.
{
    for (int count = 1; count < SIZE; count++)
        insertElement(0, count);
}
```

# Insertion Sort Analysis

- The general case for this algorithm mirrors the `selectionSort` and the `bubbleSort`, so the general case is  $O(N^2)$ .
- But `insertionSort` has a “best” case: The data are already sorted in ascending order
  - `insertElement` is called  $N$  times, but only one comparison is made each time and no swaps are necessary.
- The maximum number of comparisons is made only when the elements in the array are in reverse order.

# Sequential Search vs Binary Search – Worst Case

- Input Array  $S$  size  $n$
- $X \notin S$
- Sequential Search:  $n$  operations
- Binary Search:  $\lg n + 1$  operations

# Complexity Analysis

- Define Basic Operation
- Count the number of times the basic operation executes for each value of the input size
- Maybe dependent on input size (sequential search)
- Every-case time complexity analysis

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

## Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

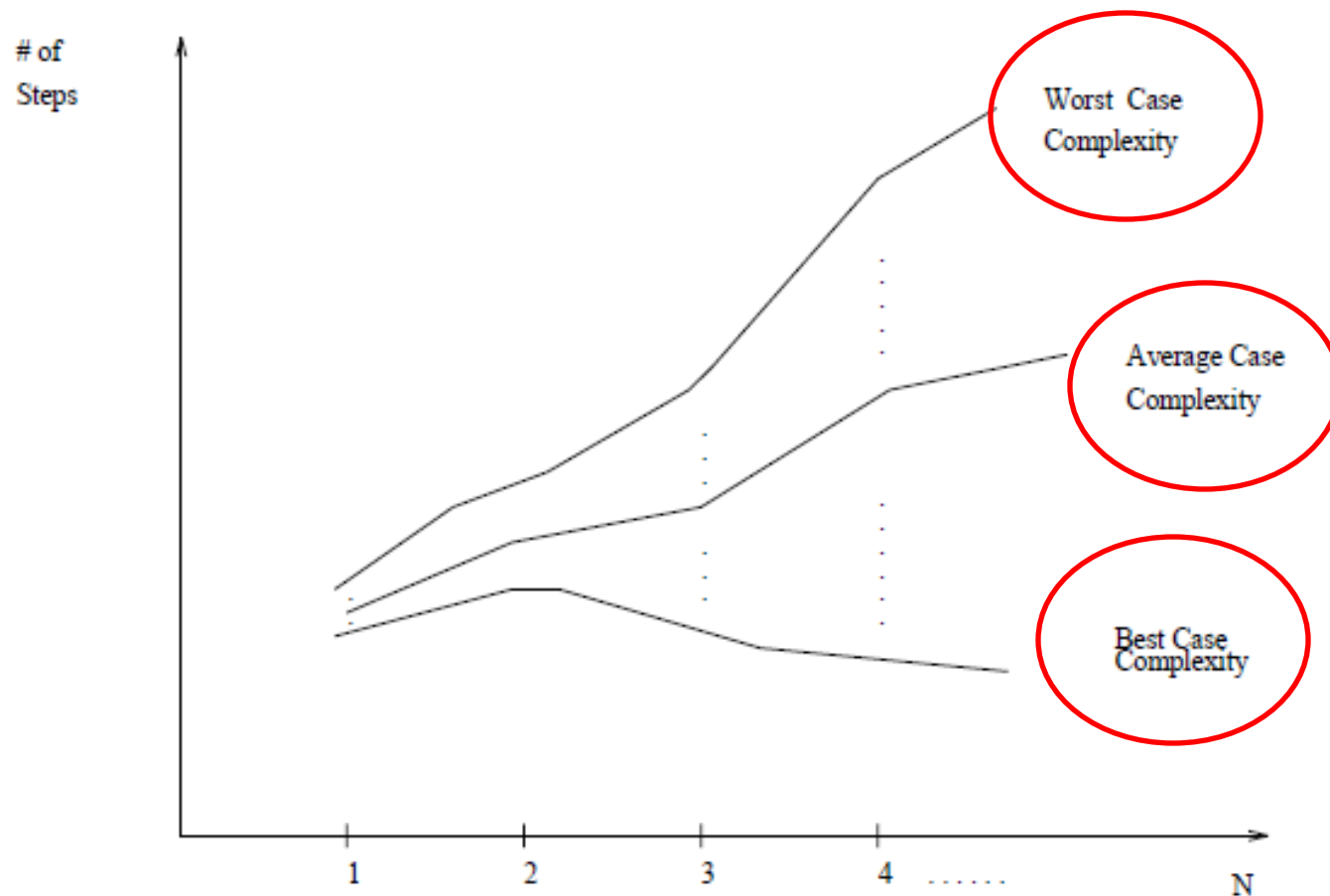


The term asymptotic means approaching a value or curve arbitrarily closely (i.e., as some sort of **limit** is taken).  
A **line** or **curve** that is asymptotic to given **curve C** is called the **asymptote** of C .

suppose that we are interested in the properties of a function  $f(n)$  as  $n$  becomes very large. If  $f(n) = n^2 + 3n$ , then as  $n$  becomes very large, the term  $3n$  becomes insignificant compared to  $n^2$ . The function  $f(n)$  is said to be "*asymptotically equivalent* to  $n^2$ , as  $n \rightarrow \infty$ ". This is often written symbolically as  $f(n) \sim n^2$ , which is read as " $f(n)$  is asymptotic to  $n^2$ ".

# Best, Worst, and Average-Case

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size  $n$ .



The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size  $n$ .

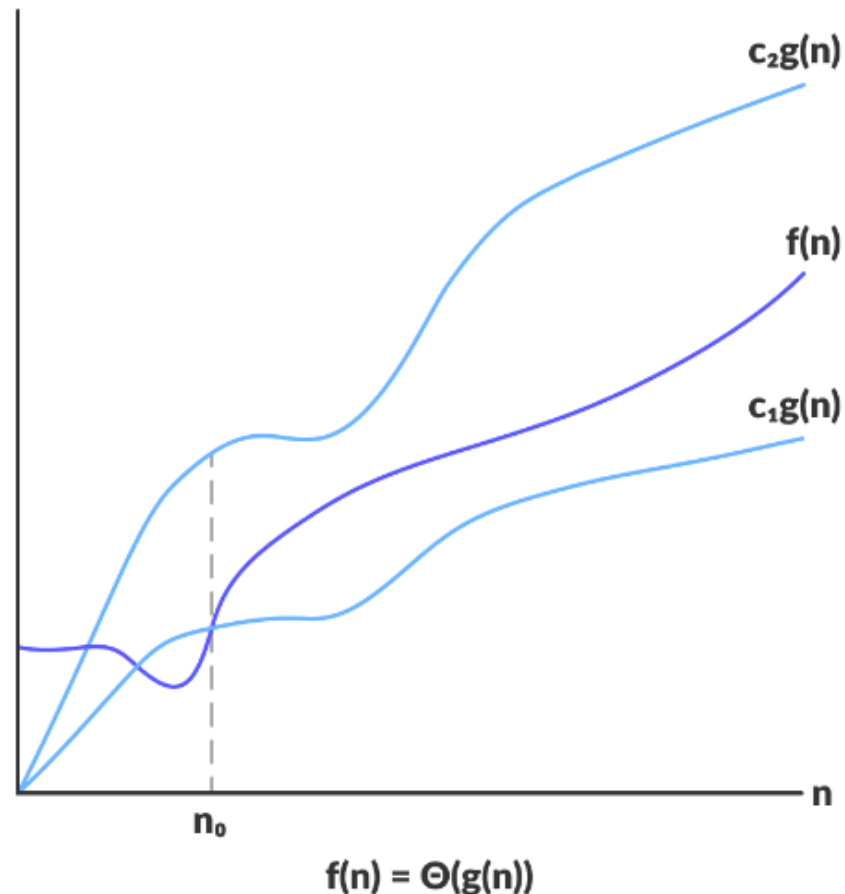
The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size  $n$ .

Each of these complexities defines a numerical function  
– time vs. size!

## Theta Notation ( $\Theta$ -notation)

Theta notation encloses the function from above and below.

Since, it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.



For a function  $g(n)$ ,  $\theta(g(n))$  is given by the relation:

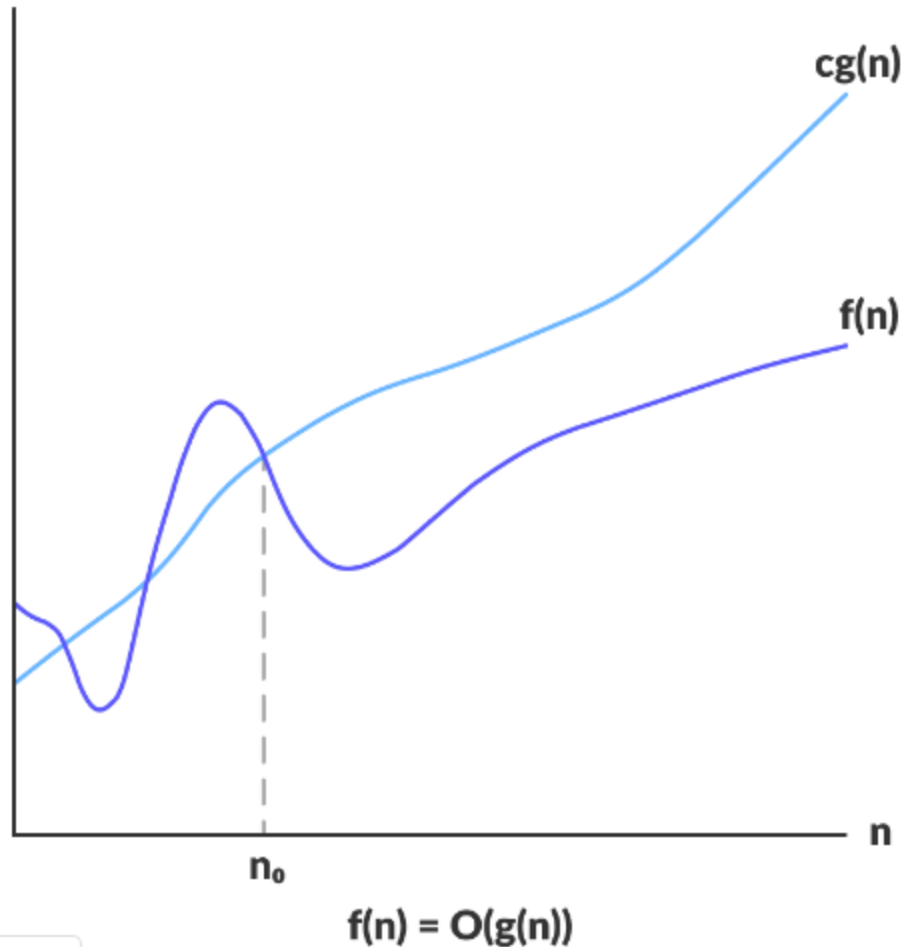
$$\theta(g(n)) = \{ f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a function  $f(n)$  belongs to the set  $\theta(g(n))$  if there exist positive constants  $c_1$  and  $c_2$  such that it can be sandwiched between  $c_1 g(n)$  and  $c_2 g(n)$ , for sufficiently large  $n$ .

If a function  $f(n)$  lies anywhere in between  $c_1 g(n)$  and  $c_2 g(n)$  for all  $n \geq n_0$ , then  $f(n)$  is said to be asymptotically tight bound.

## Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.



$$O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

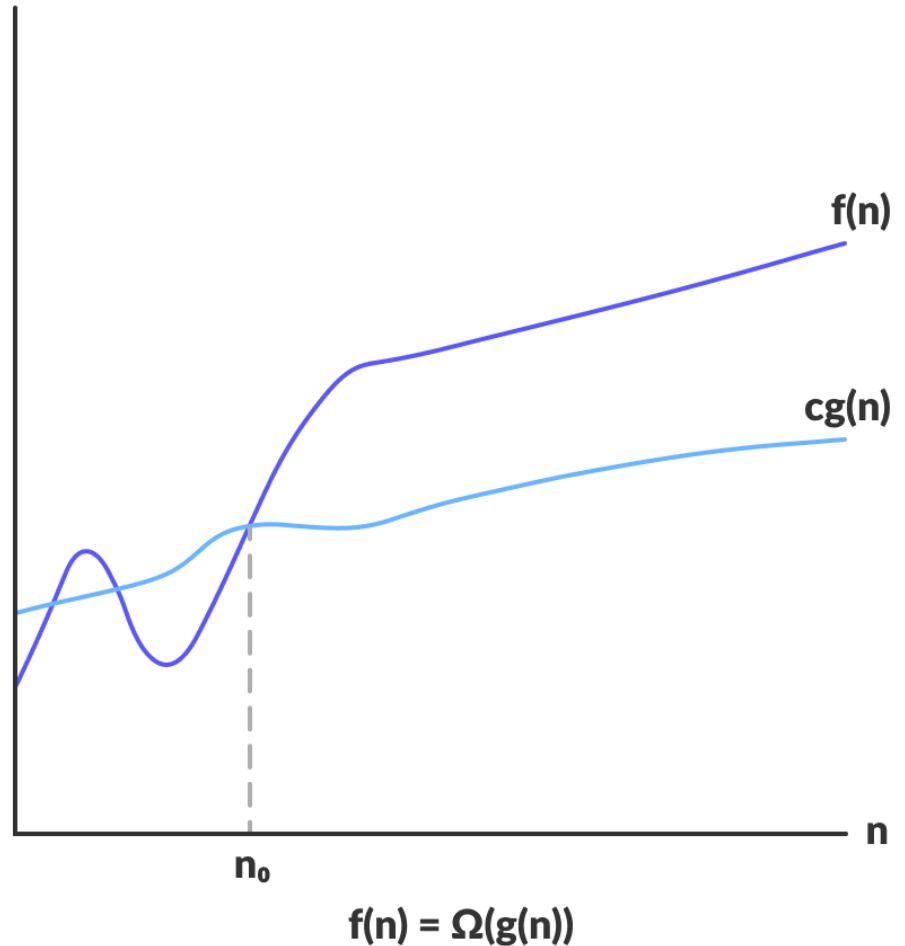
The above expression can be described as a function  $f(n)$  belongs to the set  $O(g(n))$  if there exists a positive constant  $c$  such that it lies between  $0$  and  $cg(n)$ , for sufficiently large  $n$ .

For any value of  $n$ , the running time of an algorithm does not cross time provided by  $O(g(n))$ .

Since it gives the worst case running time of  $n$  algorithm, it is widely used to analyze an algorithm as we are always interested in worst case scenario.

## Omega Notation ( $\Omega$ -notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.





$$\Omega(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

The above expression can be described as a function  $f(n)$  belongs to the set  $\Omega(g(n))$  if there exists a positive constant  $c$  such that it lies above  $cg(n)$ , for sufficiently large  $n$ .

For any value of  $n$ , the minimum time required by the algorithm is given by Omega  $\Omega(g(n))$ .

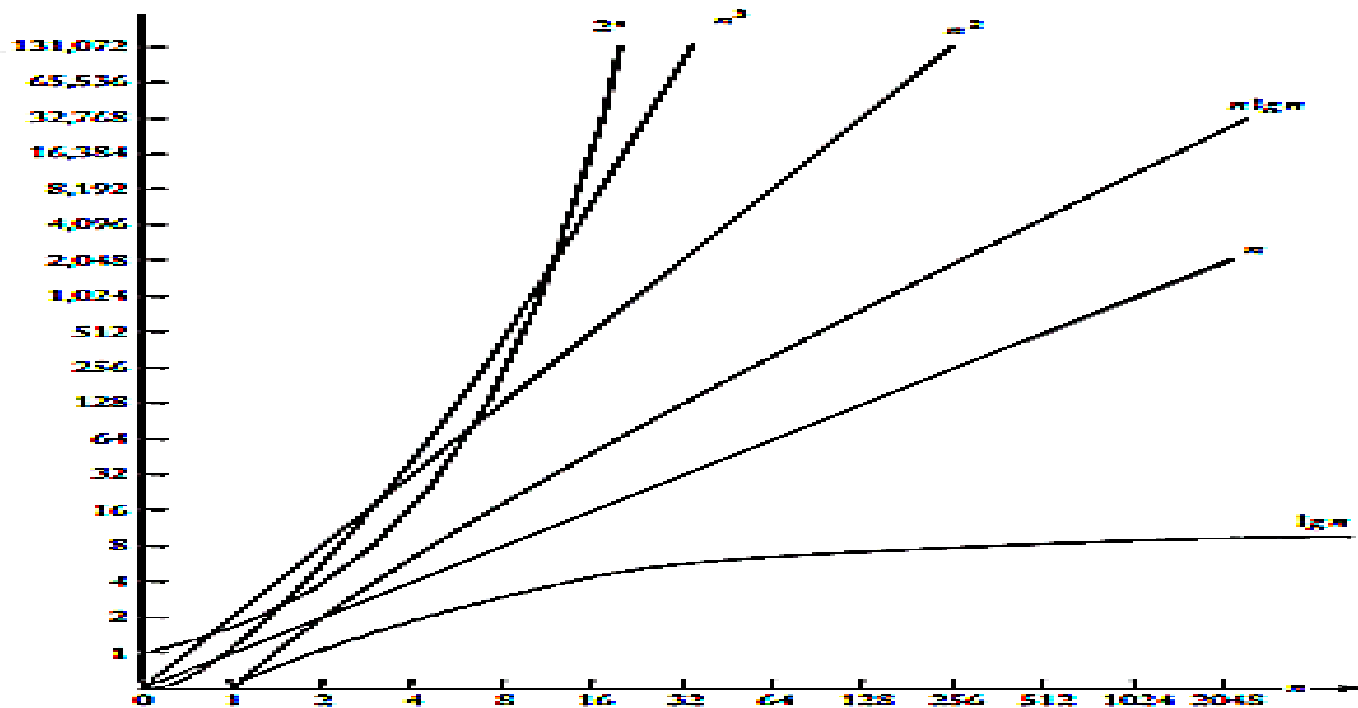
# Complexity Analysis – Large $n$

- Worst Case
- Every Case
- Average Case
- Best Case

# Order – Classes of Functions – Growth Rate

- $\Theta(f)$  – At the rate of  $f$
- $O(f)$  – At most as fast as  $f$
- $\Omega(f)$  – At least as fast as  $f$
- $n$  grows more slowly than  $n^3 \Rightarrow n \in O(n^3)$
- $n^3$  grows faster than  $n \Rightarrow n^3 \in \Omega(n)$
- By definition  $n$  and  $2n$  grow at the same rate  
 $\Rightarrow 2n \in \Theta(n)$

# Growth Rates of Common Complexity Functions



# Big O

- For a given complexity function  $f(n)$ ,  $O(f(n))$  is the set of complexity functions  $g(n)$  for which there exists some positive real constant  $c$  and some nonnegative integer  $N$  such that for all  $n \geq N$ ,
- $g(n) \leq c \times f(n)$
- $g(n) \in O(f(n))$

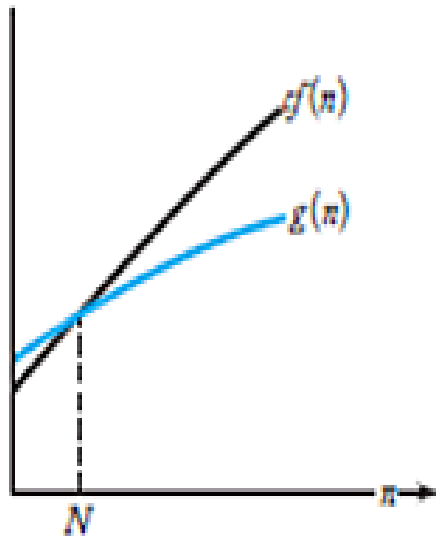
# Omega

- For a given complexity function  $f(n)$ ,  $\Omega(f(n))$  is the set of complexity functions  $g(n)$  for which there exists some positive real constant  $c$  and some nonnegative integer  $N$  such that, for all  $n \geq N$ ,
- $g(n) \geq c \times f(n)$ .
- $g(n) \in \Omega(f(n))$

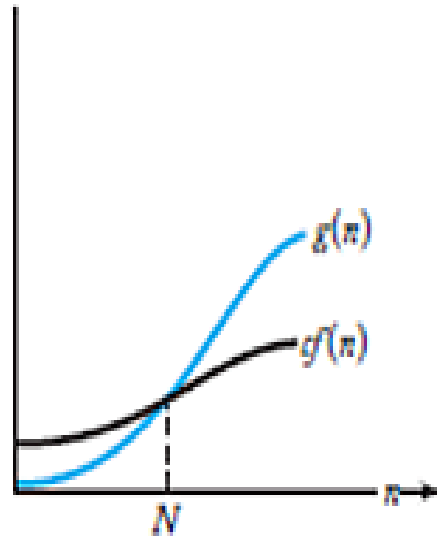
# Theta

- For a given complexity function  $f(n)$ ,
- $\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- This means that  $\theta(f(n))$  is the set of complexity functions  $g(n)$  for which there exists some positive real constants  $c$  and  $d$  and some nonnegative integer  $N$  such that, for all  $n \geq N$ ,
- $c \times f(n) \leq g(n) \leq d \times f(n)$ .
- $g(n) \in \theta(f(n))$

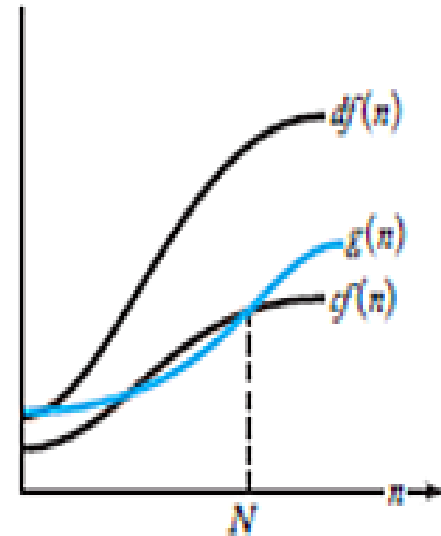
# Big O, Omega, Theta



(a)  $g(n) \in O(f(n))$



(b)  $g(n) \in \Omega(f(n))$



(c)  $g(n) \in \Theta(f(n))$



# Limit Definitions for Big O, Theta, and Omega

- $\lim_{n \rightarrow \infty} g(n)/f(n) = c \Rightarrow g(n) \in \theta(f(n))$  for  $c > 0$  and  $c \neq \infty$
- $\lim_{n \rightarrow \infty} g(n)/f(n) = c \Rightarrow g(n) \in O(f(n))$  where  $c \in$  Set of all positive real numbers union 0
- $\lim_{n \rightarrow \infty} g(n)/f(n) = \infty$  OR  $\lim_{n \rightarrow \infty} g(n)/f(n) = c > 0 \Rightarrow g(n) \in \Omega(f(n))$