

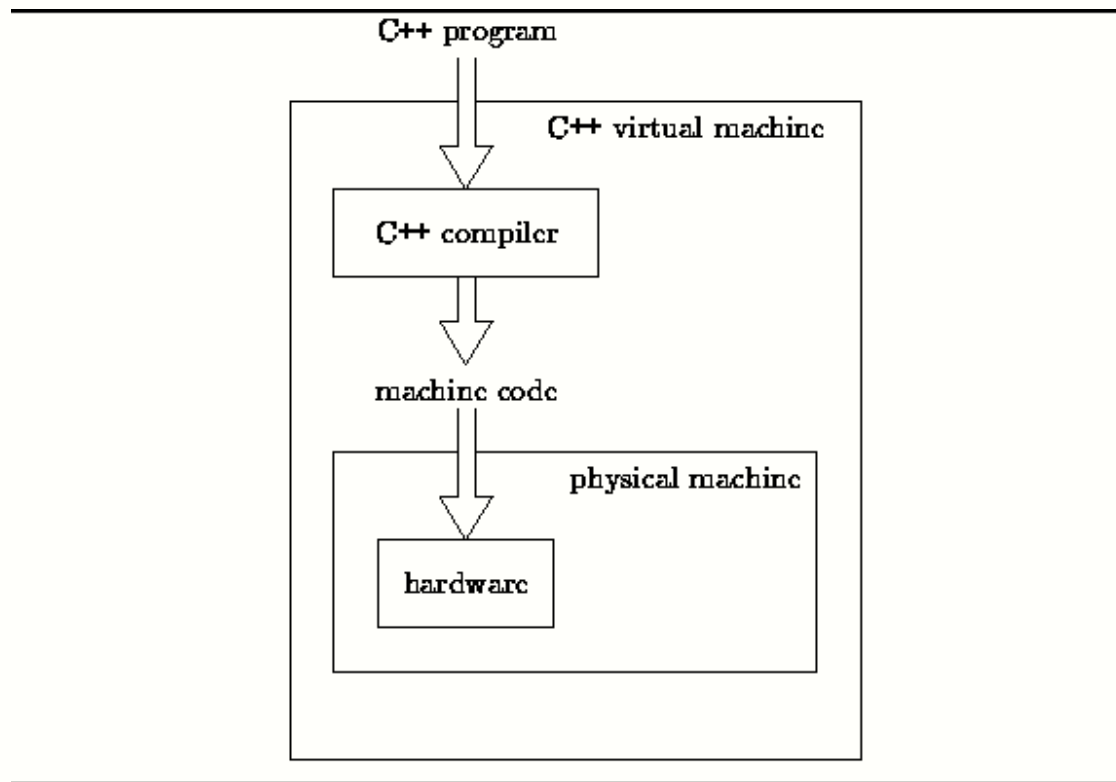
- What is an algorithm?
  - An algorithm is a step-by-step procedure for accomplishing some task.
  - An algorithm can be given in many ways. For example, it can be written down in English (or French, or any other ``natural" language).
  - However, we are interested in algorithms which have been precisely specified using an appropriate mathematical formalism--such as a programming language.

- What do we mean by analyzing one?
  - Study the specification of the algorithm and to draw conclusions about how the implementation of that algorithm--the program--will perform in general.
- What can we analyze?
  - determine the running time of a program as a function of its inputs;
  - determine the total or maximum memory space needed;
  - determine the total size of the program code;
  - determine whether the program correctly computes the desired result;
  - determine the complexity of the program--e.g., how easy is it to read, understand, and modify; and,
  - determine the robustness of the program--e.g., how well does it deal with unexpected or erroneous inputs?

- In this class, we are concerned primarily with
  - Running time (time complexity)
  - Memory space (space complexity)
- Many factors that affect the running time of a program:
  - the algorithm itself,
  - the input data, and
  - the computer system used to run the program.
    - the hardware:
      - processor used (type and speed),
      - memory available (cache and RAM), and
      - disk available;
    - the programming language in which the algorithm is specified;
    - the language compiler/interpreter used; and
    - the computer operating system software.

# A Detailed Model of the Computer

- We consider the implementation of the C++ programming language as a kind a ``virtual C++ machine"



- **Axiom 1**

The time required to fetch an integer operand from memory is a constant,  $T_{\text{fetch}}$ , and the time required to store an integer result in memory is a constant,  $T_{\text{store}}$

- **Axiom 2**

The times required to perform elementary operations on integers, such as addition,  $T_{+}$ , subtraction,  $T_{-}$ , multiplication,  $T_{\times}$ , division,  $T_{/}$ , and comparison,  $T_{<}$  are all constants.

- Running time for

`y = y + 1;`

is

$$2T_{\text{fetch}} + T_{\text{store}} + T_{+}$$

- Running time for

`y += 1;`

`++y;`

`y++;`

is also  $2T_{\text{fetch}} + T_{\text{store}} + T_{+/+=/++}$

# The Basic Axioms

- **Axiom 3**

The time required to call a function is a constant,  $T_{\text{call}}$ , and the time required to return from a function is a constant,  $T_{\text{return}}$

- When a function is called, certain housekeeping operations need to be performed.
  - saving the return address so that program execution can resume at the correct place after the call,
  - saving the state of any partially completed computations so that they may be resumed after the call,
  - the allocation of a new execution context (stack frame or activation record ).
- Conversely, on the return from a function, all of this work is undone.

- **Axiom 4**

The time required to pass an integer argument to a function or procedure is the same as the time required to store an integer in memory,  $T_{\text{store}}$ .

- Running time for

$$y = f(x)$$

is

$$T_{\text{fetch}} + 2T_{\text{store}} + T_{\text{call}} + T_{\text{return}} + T_{f(x)}$$



# Running time

```

1 unsigned int Sum (unsigned int n)
2 {
3     unsigned int result = 0;
4     for (unsigned int i = 1; i <= n; ++i)
5         result += i;
6     return result;
7 }

```

statement	time	code
3	$\tau_{\text{fetch}} + \tau_{\text{store}}$	result = 0
4a	$\tau_{\text{fetch}} + \tau_{\text{store}}$	i = 1
4b	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n + 1)$	i <= n
4c	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	++i
5	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	result += i
6	$\tau_{\text{fetch}} + \tau_{\text{return}}$	return result
TOTAL	$(6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + 2\tau_{+}) \times n$	
	$+ (5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{\text{return}})$	

- $T(n) = t_1 + t_2 \cdot n$

Where

$$t_1 = 5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{<}} + \tau_{\text{return}}$$

$$t_2 = 6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{<}} + 2\tau_{\text{+}}$$

- **Axiom 5.**

The time required for the *address calculation* implied by an array subscripting operation, e.g.,  $a[i]$ , is a constant,  $T_{[.]}$ . This time does not include the time to compute the subscript expression, nor does it include the time to access (i.e., fetch or store) the array element.

Thus the running for

$y = a[i]$

is

$$3T_{\text{fetch}} + T_{[.]} + T_{\text{store}}$$

## A Simplified Model of the Computer

- The detailed model of the computer given in the previous section is based on a number of different timing parameters ( $T_{\text{fetch}}$ ,  $T_{\text{store}}$ ,  $T_{\text{call}}$ ,  $T_{\text{return}}$ ,  $T_{f(x)}$  ... )
- Keeping track of all of the parameters during the analysis is rather burdensome
- In a real machine, each of these parameters will be a multiple of the basic clock cycle of the machine (typically between 2 and 10 ns.),  $T$ .

For example,

$$T_{\text{fetch}} = k * T$$

## A Simplified Model of the Computer

- The simplified model eliminates all of the arbitrary timing parameters in the detailed model.
- Two simplifying assumptions:
  - All timing parameters are expressed in units of clock cycles. In effect,  $T=1$ .
  - The proportionality constant,  $k$ , for all timing parameters is assumed to be the same:  $k=1$ .
- The effect of these two assumptions is that we no longer need to keep track of the various operations separately.
- To determine the running time of a program, we simply count the total number of operations ( or cycles).

# Running time

statement	time	code
3	$\tau_{\text{fetch}} + \tau_{\text{store}}$	result = 0
4a	$\tau_{\text{fetch}} + \tau_{\text{store}}$	i = 1
4b	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n + 1)$	i <= n
4c	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	++i
5	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	result += i
6	$\tau_{\text{fetch}} + \tau_{\text{return}}$	return result
TOTAL	$(6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + 2\tau_{+}) \times n$	
	$+ (5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{\text{return}})$	

# Running time

statement	time	code
3	2	result = 0
4a	2	i = 1
4b	$3*(n+1)$	i <= n
4c	$4*n$	++i
5	$4*n$	result += i
6	2	return result
TOTAL	$11*n+9$	

- $T(n) = t_1 + t_2 \cdot n$

Where

$$t_1 = 5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{<}} + \tau_{\text{return}}$$

$$t_2 = 6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{<}} + 2\tau_{\text{+}}$$

are replaced by

$$t_1 = 9$$

$$t_2 = 11$$

$$T(n) = 9 + 11 \cdot n$$



## In class exercise (I)

- Write a program to calculate and return the sum of a 1-dimensional array of  $n$ .
- Use the detailed model to find out the running time of the above algorithm:  $T(n)$
- Use the simplified model to find out the simplified running time  $T(n)$

---

```

1 int Horner (int a [], unsigned int n, int x)
2 {
3     int result = a [n];
4     for (int i = n - 1; i >= 0; --i)
5         result = result * x + a [i];
6     return result;
7 }

```

---

statement	detailed model	simple model	big oh
3	$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$	5	$O(1)$
4a	$2\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}}$	4	$O(1)$
4b	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n + 1)$	$3n+3$	$O(n)$
4c	$(2\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}}) \times n$	$4n$	$O(n)$
5	$(5\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{+} + \tau_{\times} + \tau_{\text{store}}) \times n$	$9n$	$O(n)$
6	$\tau_{\text{fetch}} + \tau_{\text{return}}$	2	$O(1)$
TOTAL	$(9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{[\cdot]}$ $+ \tau_{+} + \tau_{\times} + \tau_{-}) \times n$ $+ (8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[\cdot]} + \tau_{-} + \tau_{<} + \tau_{\text{return}})$	$16n+14$	$O(n)$

---

```

1 unsigned int FindMaximum (unsigned int a [], unsigned int n)
2 {
3     unsigned int result = a [0];
4     for (unsigned int i = 1; i < n; ++i)
5         if (a [i] > result)
6             result = a [i];
7     return result;
8 }

```

---

statement

time

3

$$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$$

4a

$$\tau_{\text{fetch}} + \tau_{\text{store}}$$

4b

$$(2\tau_{\text{fetch}} + \tau_{<}) \times n$$

4c

$$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times (n - 1)$$

5

$$(4\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{<}) \times (n - 1)$$

6

$$(3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}) \times ?$$

7

$$\tau_{\text{fetch}} + \tau_{\text{store}}$$

# Asymptotic Notation

- Considering two algorithms,  $A$  and  $B$ , for solving a given problem.
- The running times of each of the algorithms are determined to be  $T_A(n)$  and  $T_B(n)$ , where  $n$  is the size of the problem
- Can we simply compare  $T_A(n)$  and  $T_B(n)$  to determine which algorithm is better?

- No.

- Because it is not true that one of the functions is less than or equal the other over the entire range of problem sizes
- the comparison result might be determined by the problem size  $n$
- And we do not always have knowledge of problem size  $n$  beforehand

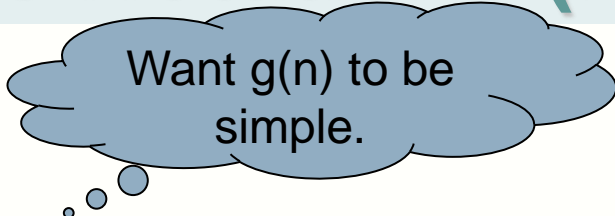
- Here, we consider the *asymptotic* behavior of the two functions for very large problem sizes.

# Asymptotic Analysis

- **Asymptotic analysis**
  - based on the idea that as the problem size grows, the complexity will eventually settle down to a proportionality to some known and simple function.
- **Three asymptotic notations**
  - Big O-Notation (upper bound)
  - Big  $\Omega$ -Notation (lower bound)
  - Big  $\Theta$ -Notation (tight bound)

# Asymptotic Notations (Big O)

## Big O-Notation



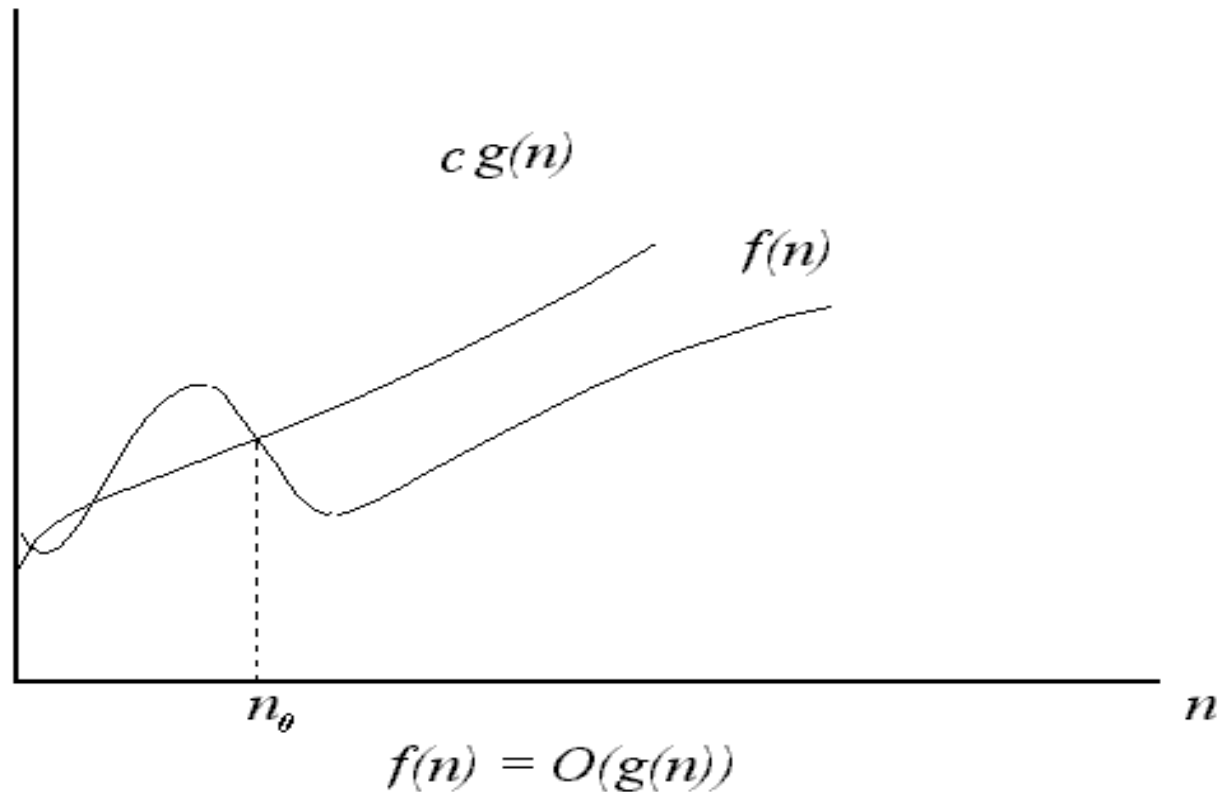
Want  $g(n)$  to be simple.

- For a given function  $g(n)$ ,  $\mathbf{O}(g(n))$  denote the set of functions such that,

$\mathbf{O}(g(n)) = \{f(n) \mid \text{if and only if there exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$

- We write  
 $f(n) = \mathbf{O}(g(n))$   
indicates a function  $f(n)$  is a member of  $\mathbf{O}(g(n))$ .
- $g(n)$  is *asymptotically upper bound* for  $f(n)$ .

# Asymptotic Notations (Big O)



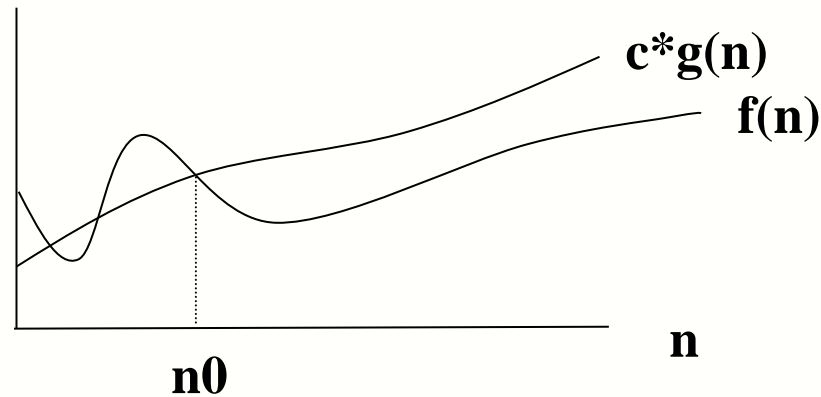
**Big O notation characterizes the asymptotic behavior of  $f(n)$  by providing an upper bound  $c \cdot g(n)$  on the rate at which  $f(n)$  grows as  $n \geq n_0$ .**



# Big-Oh Notation: Asymptotic Upper Bound

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

- if  $f(n) \leq c \cdot g(n)$  for all  $n > n_0$ , where  $c$  &  $n_0$  are constants  $> 0$



- Example:  $T(n) = 2n + 5$  is  $O(n)$ . Why?
  - $2n+5 \leq 3n$ , for  $n \geq 5$  and  $c = 3$
- $T(n) = 5 \cdot n^2 + 3 \cdot n + 15$  is  $O(n^2)$ . Why?
  - $5 \cdot n^2 + 3 \cdot n + 15 \leq 6 \cdot n^2$ , for  $n \geq 6$  and  $c = 6$

Ex) Show  $f(n) = 8n+128 = O(n^2)$

Sol)

- Based on the Big-O notation, we need to find an positive integer  $n_0$  and a constant  $c>0$  such that for all integers  $n \geq n_0$ ,  $f(n) \leq c \cdot (n^2)$ .

Ex) Show  $f(n) = 8n+128 = O(n^2)$

Sol)

• Let us choose  $c=1$

$$\begin{aligned} f(n) \leq c \cdot n^2 & \implies f(n) \leq n^2 \\ & \implies 8n + 128 \leq n^2 \\ & \implies 0 \leq n^2 - 8n - 128 \\ & \implies 0 \leq (n-16)(n+18) \end{aligned}$$

- We know that  $n+18 > 0$  for all  $n$ .
- To make  $n - 16 \geq 0$ , i.e.,  $n \geq 16$ ,
- So if we let  $n_0 = 16$ , then we have  $n \geq n_0$ .

- Hence when  $c = 1$  and  $n_0 = 16$ ,

$$f(n) \leq c \cdot n^2$$

for all integers  $n \geq n_0$

- According to the big O definition, we have

$$f(n) = 8n + 128 = O(n^2)$$

## Another Example

*Ex) Show  $f(n)=8n+128 = O(n)$*

- According to definition, we need to find an integer  $n_0 > 0$  and a constant  $c > 0$  such that for all integers  $n \geq n_0$ ,  $f(n) \leq c \cdot n$ .

- Let us choose  $c=9$

$$\begin{aligned}f(n) \leq c \cdot n & \implies f(n) \leq 9 \cdot n \\& \implies 8n + 128 \leq 9 \cdot n \\& \implies 0 \leq n - 128 \\& \implies 128 \leq n\end{aligned}$$

Let  $n_0 = 128$

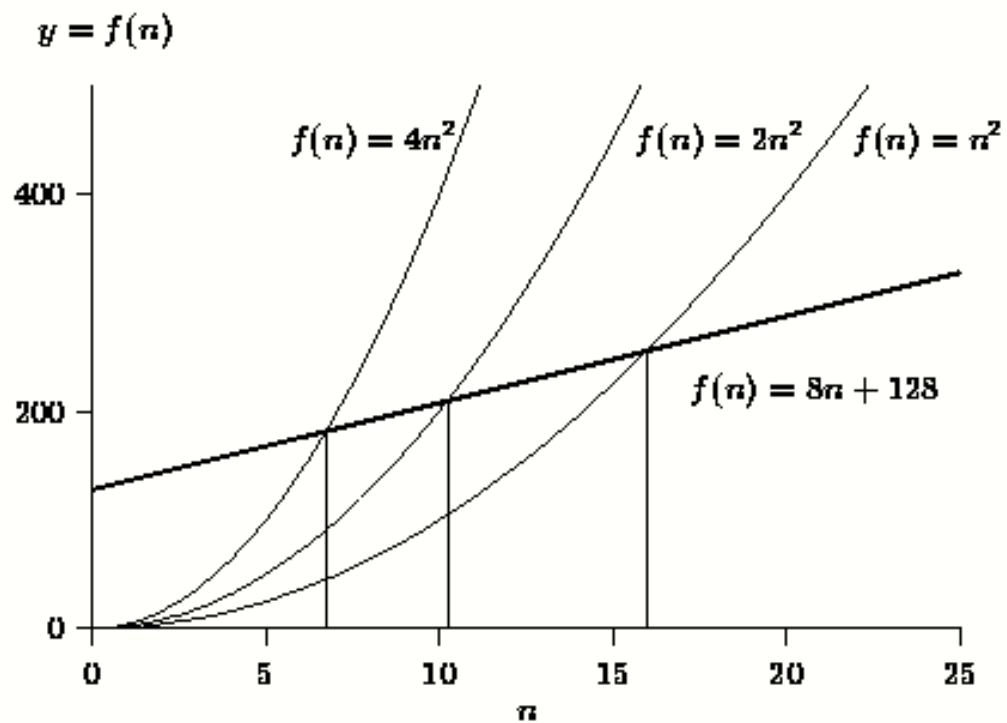
- Therefore when we have  $c = 9$  and  $n_0 = 128$ ,

$$f(n) \leq c \cdot n$$

for all integers  $n \geq n_0$

- According to big O definition, we have

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



## Tight Big O Bounds

- Big O notation characterizes the asymptotic behavior of a function by providing an upper bound on the rate at which the function grows as  $n$  gets large.
- Unfortunately, the notation does not tell us how close the actual behavior of the function is to the bound. i.e., the bound might be very close (tight) or it might be overly conservative (loose).
- When we use Big O analysis, we implicitly agree that the function we choose is the smallest one which still satisfies the definition.



# Asymptotic Notations (Big O)

Ex) Show that  $f(n) = 17n^2 - 5 = O(n^2)$

Sol)

- Based on big-O notation, we need find two integer  $c$  and  $n_0$  such that

$$17n^2 - 5 \leq c n^2 \text{ for all } n \geq n_0$$

- With  $c = 17$  and  $n_0 = 1$

$$17n^2 - 5 \leq 17 n^2 \text{ for all } n \geq 1$$

$$\therefore 17n^2 - 5 = O(n^2)$$

# Asymptotic Notations (Big O)

Ex) Show that  $f(n) = 35n^3 + 100 = O(n^3)$

Sol)

- Based on the Big-O notation, we need find two integer  $c$  and  $n_0$  such that

$$35n^3 + 100 \leq c n^3 \text{ for all } n \geq n_0$$

- With  $c = 36$  and  $n_0 = 5$

$$35n^3 + 100 \leq 36 n^3 \text{ for all } n \geq 5$$

$$\therefore 35n^3 + 100 = O(n^3)$$

# Asymptotic Notations (Big O)

Ex) Show  $f(n) = 6 \cdot 2^n + n^2 = O(2^n)$

Sol)

- Based on the Big-O notation, we need find two integer  $c$  and  $n_0$  such that

$$6 \cdot 2^n + n^2 \leq c \cdot 2^n \text{ for all } n \geq n_0$$

- With  $c = 7$  and  $n_0 = 5$

$$6 \cdot 2^n + n^2 \leq 7 \cdot 2^n \text{ for all } n \geq 5$$

$$\therefore 6 \cdot 2^n + n^2 = O(2^n)$$

## Conventions for Writing Big O Expressions

- When writing big O expressions to drop all but the most significant terms.

$$O(n^2 + n \cdot \log(n) + n)$$

is replaced by

$$O(n^2)$$

- Drop constant coefficients.

$$O(3 \cdot n^2)$$

is replaced by

$$O(n^2)$$

- Use 1 to represent constant

$$O(1024)$$

is replaced by

$$O(1)$$

- If we have a tight bound, use it.

# Big O related theorems

- Theorem # 1 and proof
- Theorem # 2 and proof
- Theorem # 3
- Theorem # 4

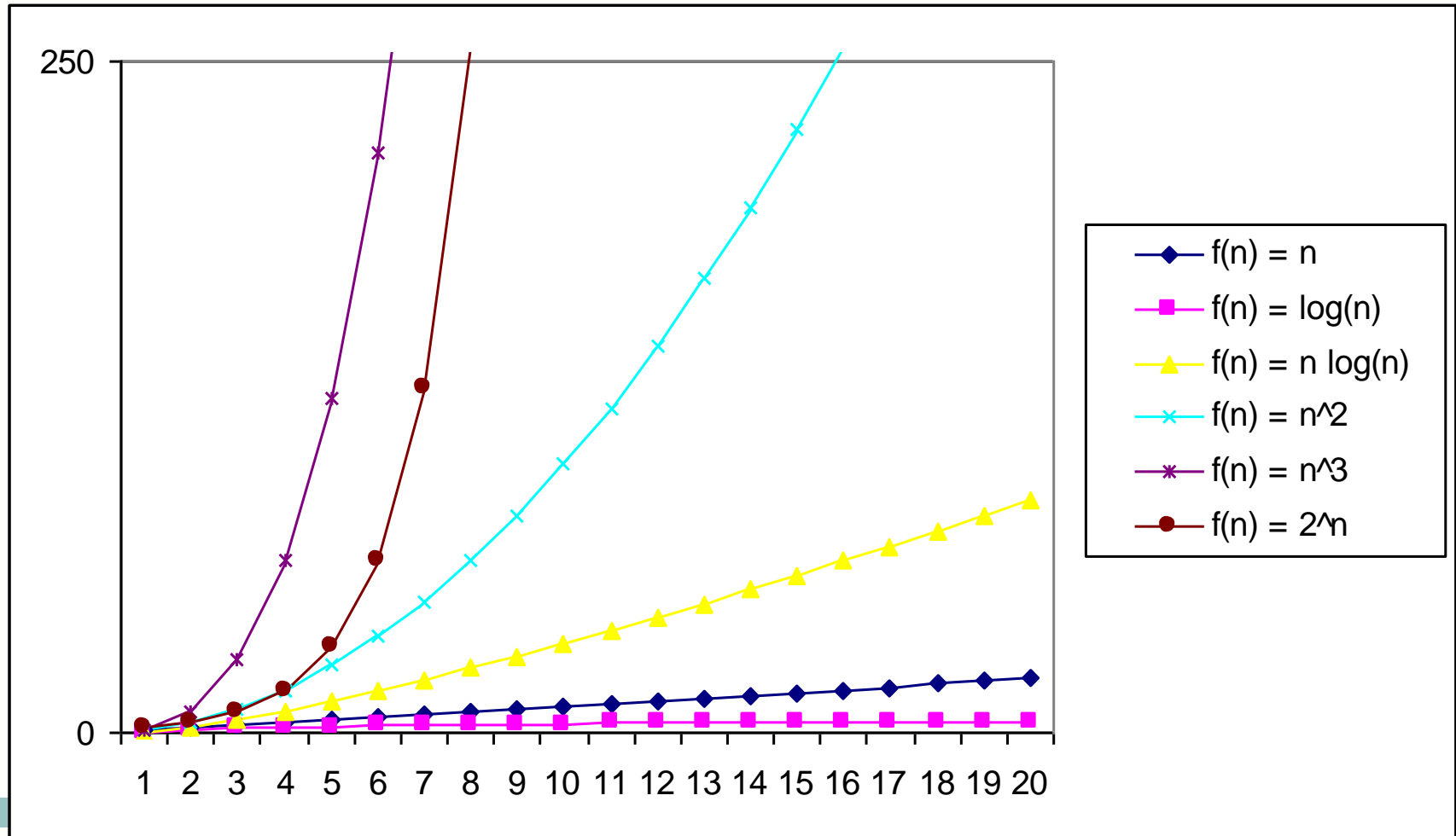
# Common Functions

Increasing cost

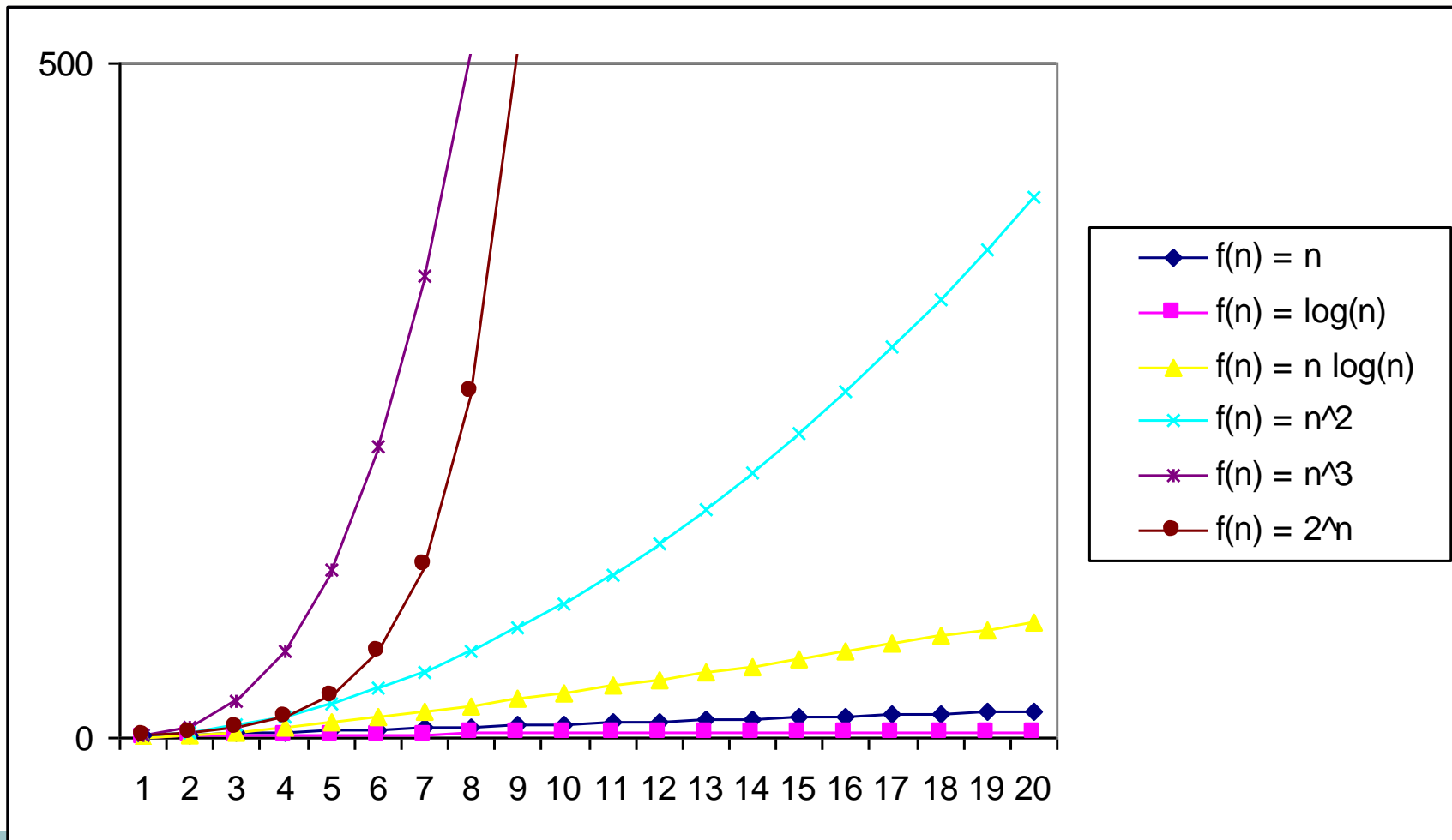
Polynomial time

Name	Big-O	Comment
Constant	$O(1)$	Can't beat it!
Log log	$O(\log \log N)$	Extrapolation search
Logarithmic	$O(\log N)$	Typical time for <b>good</b> searching algorithms
Linear	$O(N)$	This is about the fastest that an algorithm can run given that we need $O(n)$ just to read the input
N logN	$O(N \log N)$	Most sorting algorithms
Quadratic	$O(N^2)$	Acceptable when the data size is small ( $N < 10000$ )
Cubic	$O(N^3)$	Acceptable when the data size is small ( $N < 1000$ )
Exponential	$O(2^N)$	Only good for really small input sizes ( $n \leq 20$ )

# Asymptotic Complexity

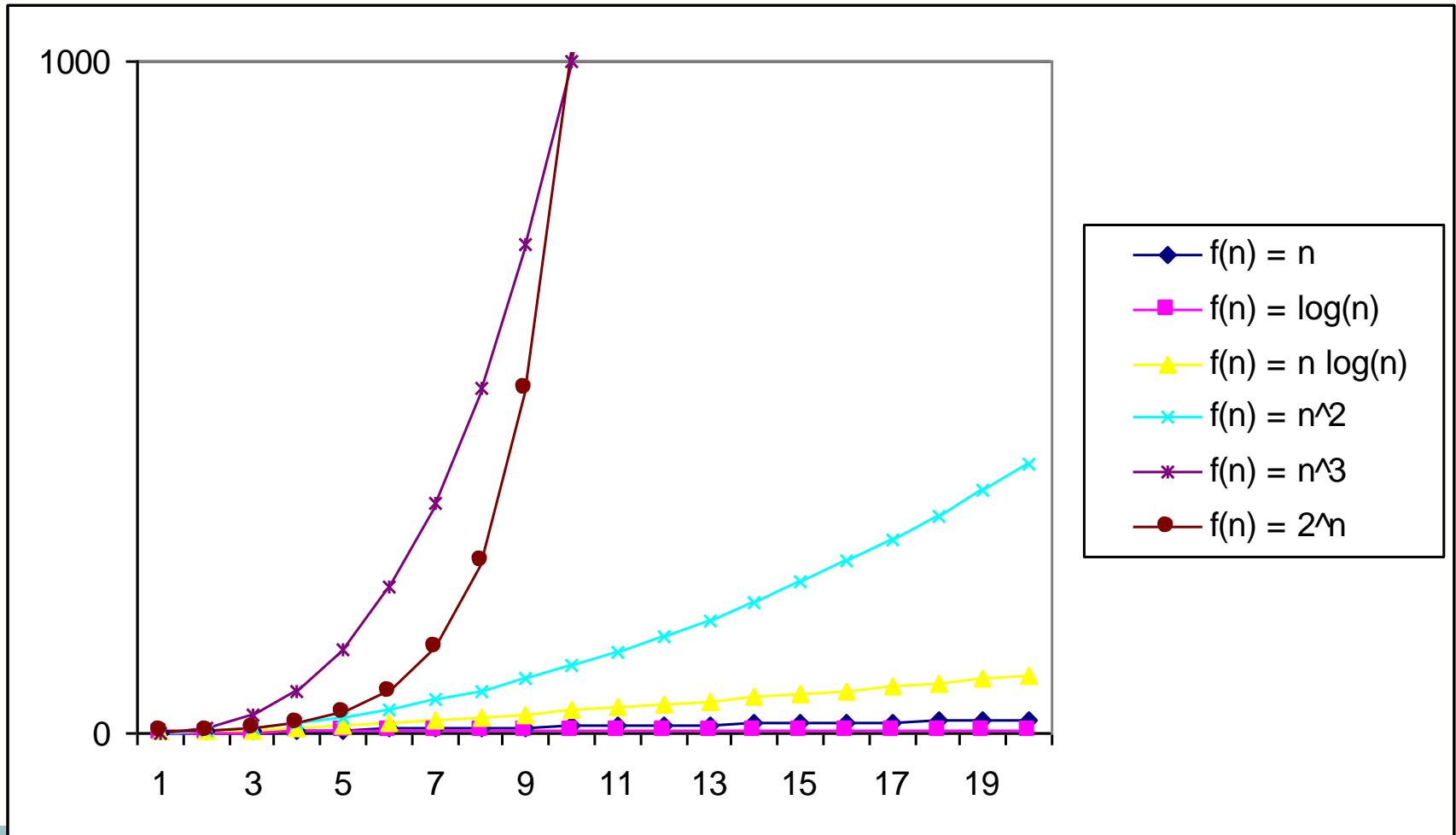


# Asymptotic Complexity

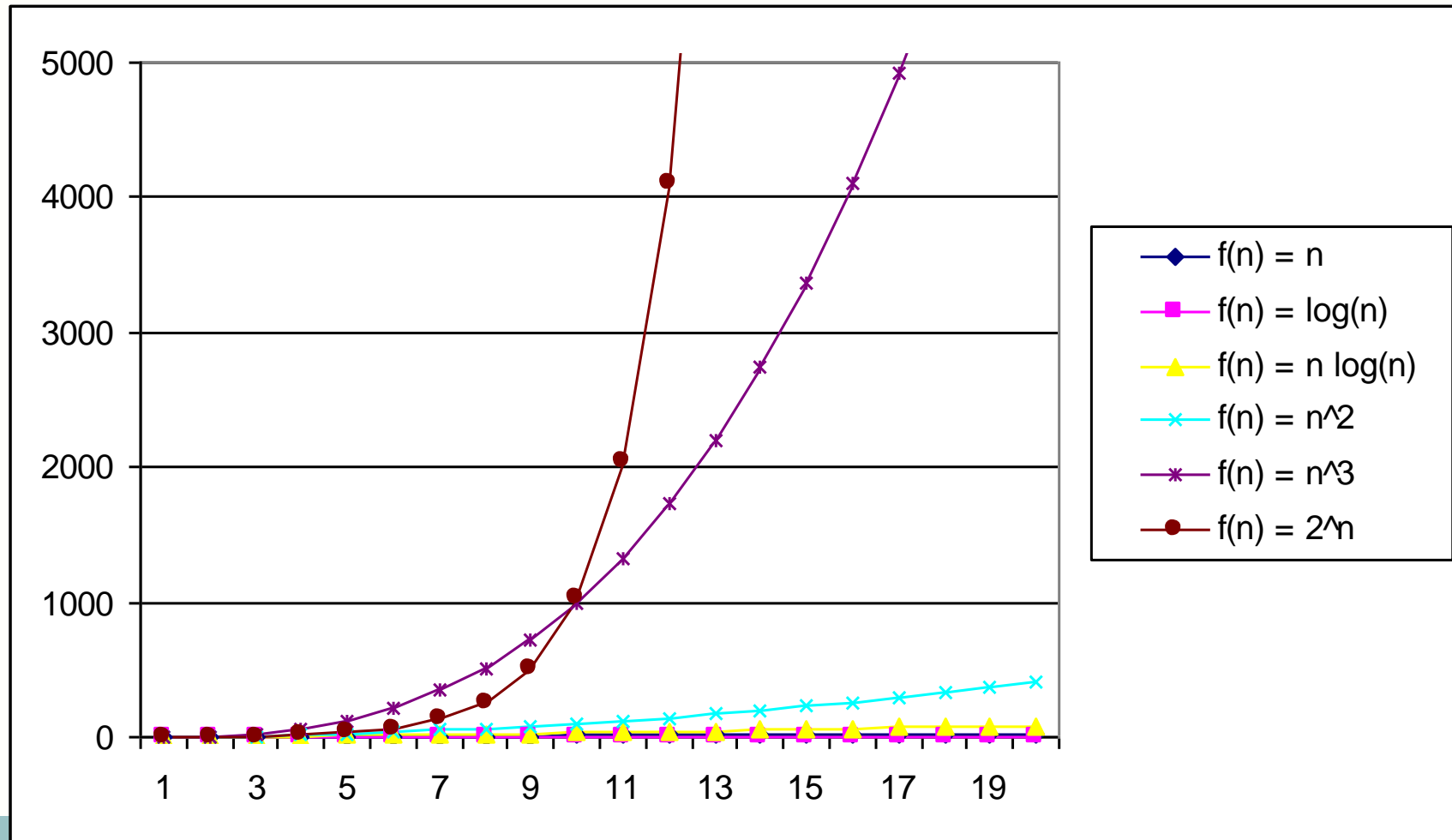




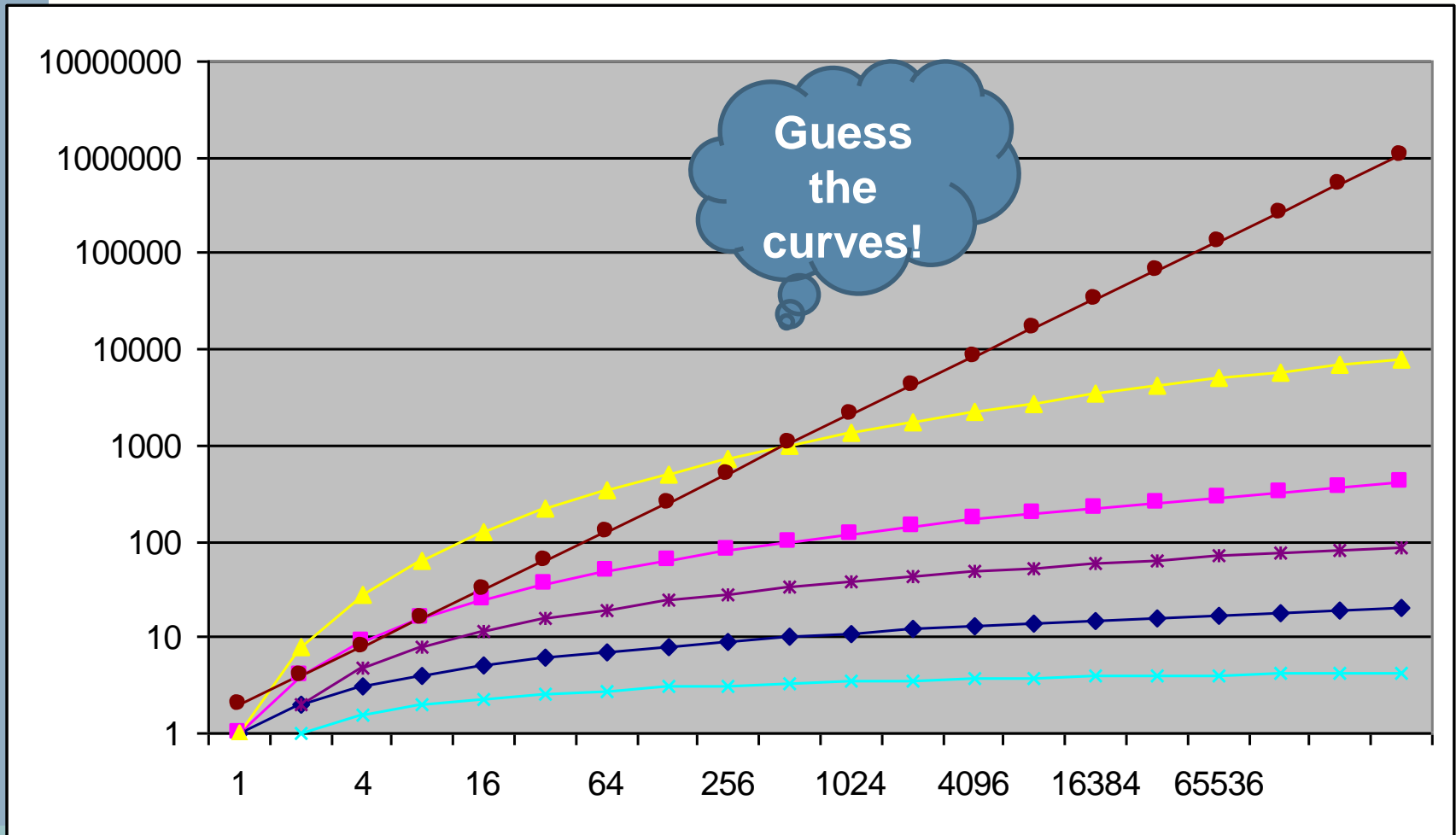
# Asymptotic Complexity



# Asymptotic Complexity



# Asymptotic Complexity



## Tips to guide your intuition:

- Think of  $O(g(N))$  as “greater than or equal to”  $f(N)$ 
  - Upper bound: “grows faster than or same rate as”  $f(N)$
- Think of  $\Omega(g(N))$  as “less than or equal to”  $f(N)$ 
  - Lower bound: “grows slower than or same rate as”  $f(N)$
- Think of  $\Theta(g(N))$  as “equal to”  $f(N)$ 
  - “Tight” bound: same growth rate

*(True for large  $N$  and ignoring constant factors)*