

## Chapter 2

# Analysis of Algorithms

# Chapter Scope

- Efficiency goals
- The concept of algorithm analysis
- Big-Oh notation
- The concept of asymptotic complexity
- Comparing various growth functions

# Algorithm Efficiency

- The efficiency of an algorithm is usually expressed in terms of its use of CPU time
- The analysis of algorithms involves categorizing an algorithm in terms of efficiency
- An everyday example: washing dishes
- Suppose washing a dish takes 30 seconds and drying a dish takes an additional 30 seconds
- Therefore,  $n$  dishes require  $n$  minutes to wash and dry

# Algorithm Efficiency

- Now consider a less efficient approach that requires us to redry all previously washed dishes after washing another one

$$= n * (30 \text{ seconds wash time}) + \sum_{i=1}^n (i * 30)$$

$$\begin{aligned} \text{time}(n \text{ dishes}) &= 30n + \frac{30n(n+1)}{2} \\ &= 15n^2 + 45n \text{ seconds} \end{aligned}$$

# Problem Size

- For every algorithm we want to analyze, we need to define the size of the problem
- The dishwashing problem has a size  $n$  – number of dishes to be washed/dried
- For a search algorithm, the size of the problem is the size of the search pool
- For a sorting algorithm, the size of the program is the number of elements to be sorted

# Growth Functions

- We must also decide what we are trying to efficiently optimize
  - *time complexity* – CPU time
  - *space complexity* – memory space
- CPU time is generally the focus
- A growth function shows the relationship between the size of the problem ( $n$ ) and the value optimized (time)

# Asymptotic Complexity

- The growth function of the second dishwashing algorithm is

$$t(n) = 15n^2 + 45n$$

- It is not typically necessary to know the exact growth function for an algorithm
- We are mainly interested in the *asymptotic complexity* of an algorithm – the general nature of the algorithm as  $n$  increases

# Asymptotic Complexity

- Asymptotic complexity is based on the *dominant term* of the growth function – the term that increases most quickly as  $n$  increases
- The dominant term for the second dishwashing algorithm is  $n^2$ :

Number of dishes ( $n$ )	$15n^2$	$45n$	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000



# Big-Oh Notation

- The coefficients and the lower order terms become increasingly less relevant as  $n$  increases
- So we say that the algorithm is *order*  $n^2$ , which is written  $O(n^2)$
- This is called *Big-Oh notation*
- There are various Big-Oh categories
- Two algorithms in the same category are generally considered to have the same efficiency, but that doesn't mean they have equal growth functions or behave exactly the same for all values of  $n$

# Big-Oh Categories

- Some sample growth functions and their Big-Oh categories:

Growth Function	Order	Label
$t(n) = 17$	$O(1)$	constant
$t(n) = 3\log n$	$O(\log n)$	logarithmic
$t(n) = 20n - 4$	$O(n)$	linear
$t(n) = 12n \log n + 100n$	$O(n \log n)$	$n \log n$
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	quadratic
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	cubic
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponential

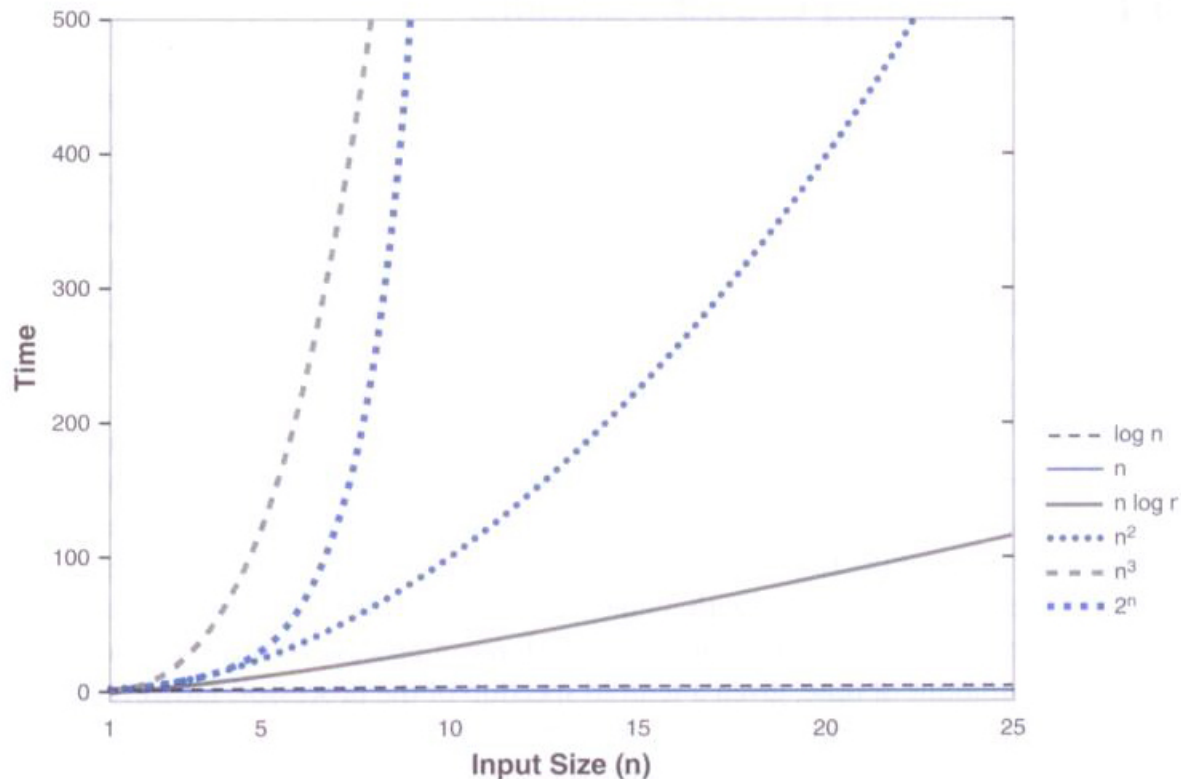
# Comparing Growth Functions

- You might think that faster processors would make efficient algorithms less important
- A faster CPU helps, but not relative to the dominant term

Algorithm	Time Complexity	Max Problem Size Before Speedup	Max Problem Size After Speedup
A	$n$	$s_1$	$10s_1$
B	$n^2$	$s_2$	$3.16s_2$
C	$n^3$	$s_3$	$2.15s_3$
D	$2^n$	$s_4$	$s_4 + 3.3$

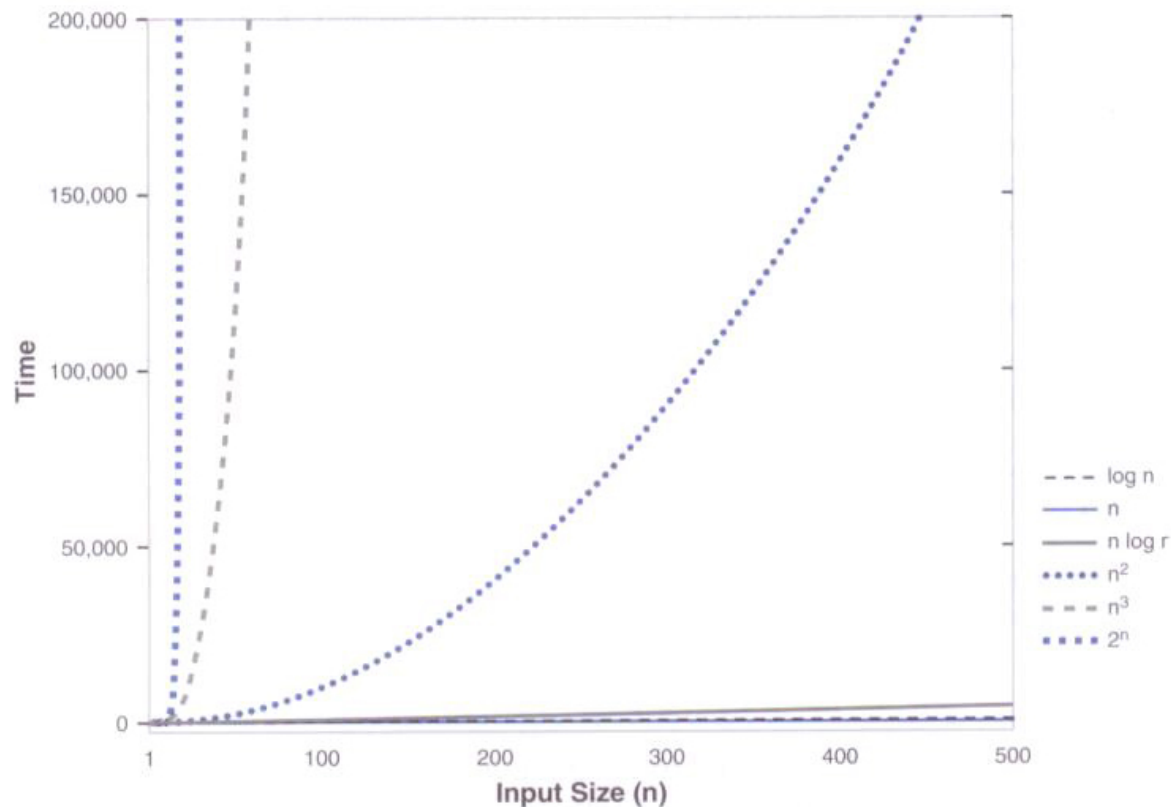
# Comparing Growth Functions

- As  $n$  increases, the various growth functions diverge dramatically:



# Comparing Growth Functions

- For large values of  $n$ , the difference is even more pronounced:



# Analyzing Loop Execution

- First determine the order of the body of the loop, then multiply that by the number of times the loop will execute

```
for (int count = 0; count < n; count++)  
    // some sequence of  $O(1)$  steps
```

- $N$  loop executions times  $O(1)$  operations results in a  $O(n)$  efficiency

# Analyzing Loop Execution

- Consider the following loop:

```
count = 1;
while (count < n)
{
    count *= 2;
    // some sequence of O(1) steps
}
```

- The loop is executed  $\log_2 n$  times, so the loop is  $O(\log n)$

# Analyzing Nested Loops

- When loops are nested, we multiply the complexity of the outer loop by the complexity of the inner loop

```
for (int count = 0; count < n; count++)  
    for (int count2 = 0; count2 < n; count2++)  
    {  
        // some sequence of O(1) steps  
    }
```

- Both the inner and outer loops have complexity of  $O(n)$
- The overall efficiency is  $O(n^2)$



# Analyzing Method Calls

- The body of a loop may contain a call to a method
- To determine the order of the loop body, the order of the method must be taken into account
- The overhead of the method call itself is generally ignored