Lecture 14: Growth of Functions and Complexity

Dr. Chengjiang Long
Computer Vision Researcher at Kitware Inc.
Adjunct Professor at SUNY at Albany.
Email: **clong2@albany.edu**

Outline

- Introduction
- Big-O, Big-Omega and Big-Theta Notations
- Complexity of Algorithms

Outline

- **Introduction**
- Big-O, Big-Omega and Big-Theta Notations
- Complexity of Algorithms

What Is the Best?

- We saw that there are several algorithms to solve certain problems. Some works better than other. What does “better” mean? How to evaluate the algorithm?
- Speed depends on the computer and on the input data. It should be something universal, behavioral to represent the quality of an algorithm.

Worst-case Analysis

- *Recall:*

```
procedure max(a_1, a_2, ...., a_n: integers)
    max := a_1
    for i := 2 to n
        if max < a_i
                    then max := a_i
    return max{max is the largest element}
```

- Let consider one line as one executable instruction, then $f(n) = 2 + 3(n\text{-}1) = 3n - 1$ is the number of instructions (steps) of this algorithm.

Worst-case Analysis

- In fact, the number of instructions to be executed depends on the data. For the data set A = { 4, 3, 2, 1 } we have one assignment and 3 tests. For the data set A = { 1, 2, 3, 4 } we have an assignment after each test.
- This is called “worst-case scenario” and the algorithms should be analyzed under such heavy-duty load.

Big Data

- Worst-case is a qualitative parameter of the input data.
- The other challenge is size of the input data.
- For the purposes of Computer Science it is always of interest how algorithms manage growing data.
- Let's think about function $f(n) = 3n - 1$ as a function that represents how the algorithm reacts on increasing of size of the input data without bounds.

Big Data

- It is easy to simplify the function to the form of $f(n) = 3n$ because impact of 1 is practically nothing in case of large values of *n.*
- What is the meaning of coefficient 3? It is a result of our assumption that each line of the code is one instruction. To clean the function from the implementation details we have to end up with $f(n) = n$.

Asymptotic Behavior

- Such function represents the **asymptotic behavior** of the algorithms.
- Any algorithm that doesn't have any loops will have $f(n) = 1$, since the number of instructions it needs is just a constant (unless it uses recursion).

Asymptotic Behavior

- Any program with a single loop which goes from 1 to *n* will have f(*n*) = *n*, since it will do a constant number of instructions before the loop, a constant number of instructions after the loop, and a constant number of instructions within the loop which all run *n* times.

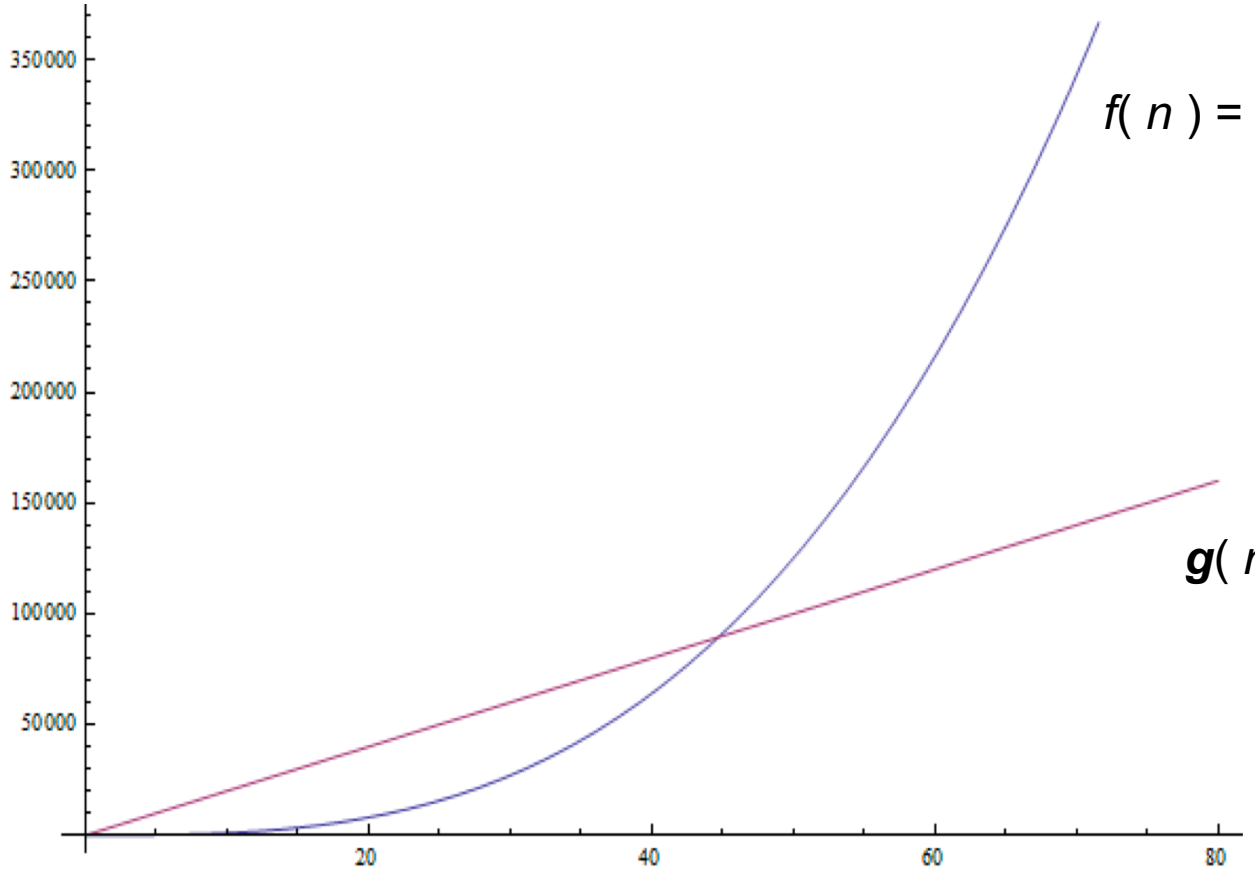
Asymptotic Behavior, cont'd

- Simple programs can be analyzed by counting the nested loops of the program:
  - A single loop over n items yields f(n) = n.
  - A loop within a loop yields f(n) = n^2.
  - A loop within a loop within a loop yields $f(n) = n^3$.
- Given a series of the loops that are sequential, the **slowest** of them determines the asymptotic behavior of the whole algorithm. Two nested loops followed by a single loop is asymptotically the same as the nested loops alone, because the nested loops dominate the simple loop.

Example

- **Problem:** Find the asymptotic behavior of the following function:
- $f(n) = n^3 + 1999n + 1337$
- **Solution:** $f(n) = n^3$
- Even though the factor in front of n is quite large, we can still find a large enough n so that n^3 is bigger than $1999n$. As we're interested in the behavior for very large values of n, we only keep n^3
- The n^3 function, drawn in blue, becomes larger than the $1999n$ function, drawn in purple, after $n = 45$. After that point it remains larger for ever.

Example


350000
300000
250000
200000
150000
100000
50000
20
40
60
80
f(n) = n^3
g(n) = 1999n

Outline

- Introduction
- **Big-O, Big-Omega and Big-Theta Notations**
- Complexity of Algorithms

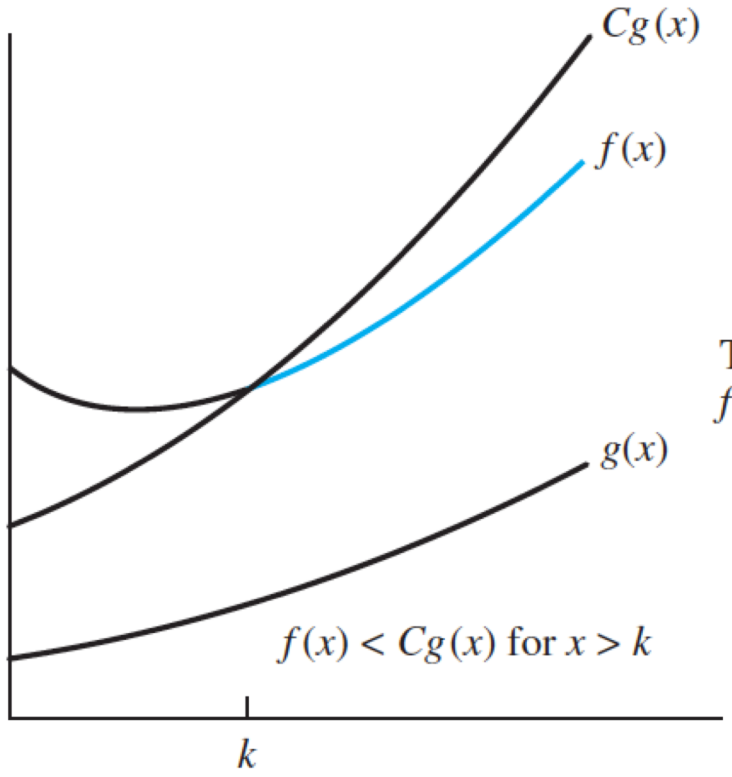
Big-*O* Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that *f*(*x*) is *O*(*g*(*x*)) if there are constants *C* and *k* such that

$$|f(x)| \leq C|g(x)|$$

  whenever $x > k$.
- This is read as "*f*(*x*) is big oh of *g*(*x*)" or "*g* asymptotically dominates *f*."
- The constants C and k are called **witnesses** to the relationship *f*(*x*) is *O*(*g*(*x*)). Only one pair of witnesses is needed.

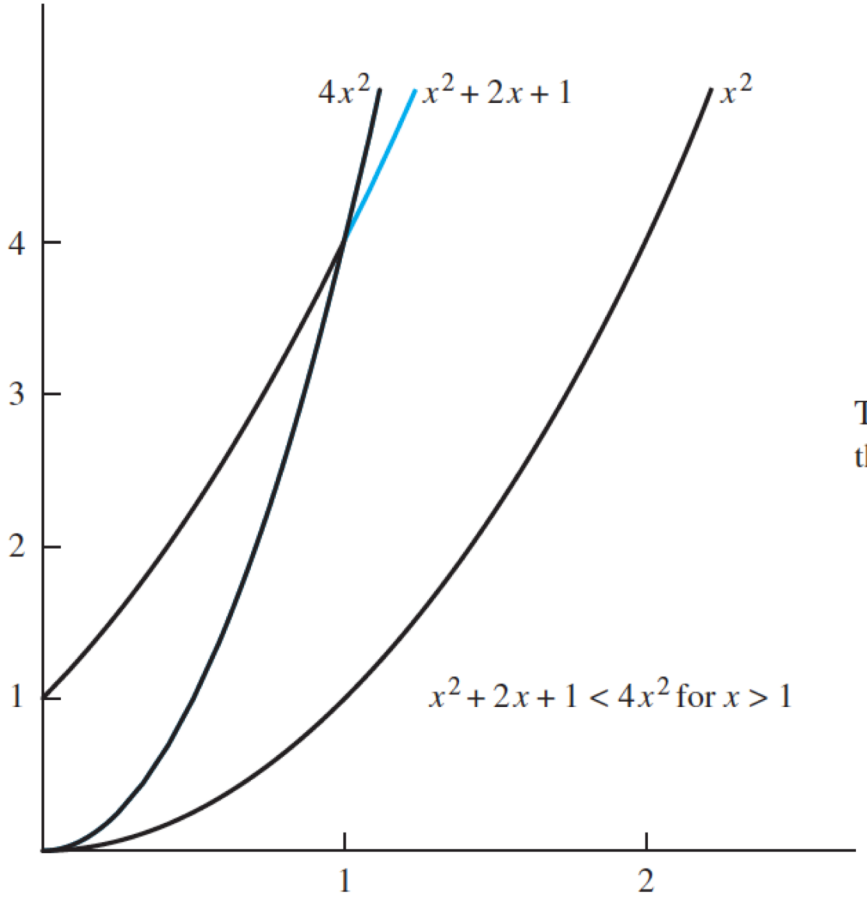
Big-O Notation



The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

FIGURE 2 **The Function $f(x)$ is $O(g(x))$.**

Example



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

FIGURE 1 **The Function $x^2 + 2x + 1$ is $O(x^2)$.**

Important Notes about Big-*O* Notation

- If one pair of witnesses is found, then there are infinitely many pairs. We can always make the *k* or the *C* larger and still maintain the inequality

$$|f(x)| \leq C|g(x)|$$

- You may see " $f(x) = O(g(x))$" instead of " $f(x)$ is $O(g(x))$" but it is not an equation. It is ok to write $f(x) \in O(g(x))$, because $O(g(x))$ represents the set of functions that are $O(g(x))$.

- Usually, we will drop the absolute value sign since we will always deal with functions that take on positive values.

Practice

- **Problem**: Use big-*O* notation to estimate the sum of the first *n* positive integers.
- **Solution**:

$$1 + 2 + \cdots + n \leq n + n + \cdots n = n^2$$

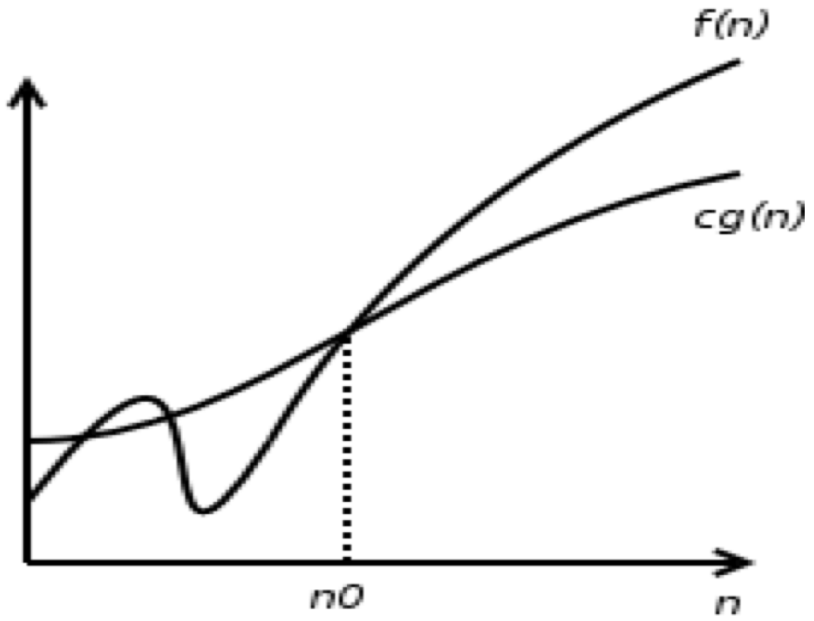
$$1 + 2 + \ldots + n \text{ is } O(n^2) \text{ taking } C = 1 \text{ and } k = 1.$$

Big-Omega Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$
- if there are constants *C* and *k* such that $|f(x)| \geq C|g(x)|$
- when $x > k$.
- We say that "*f*(*x*) is big-Omega of *g*(*x*)."

Big-Omega Notation

- Big-*O* gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-Omega tells us that a function grows at least as fast as another.
- $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. This follows from the definitions.

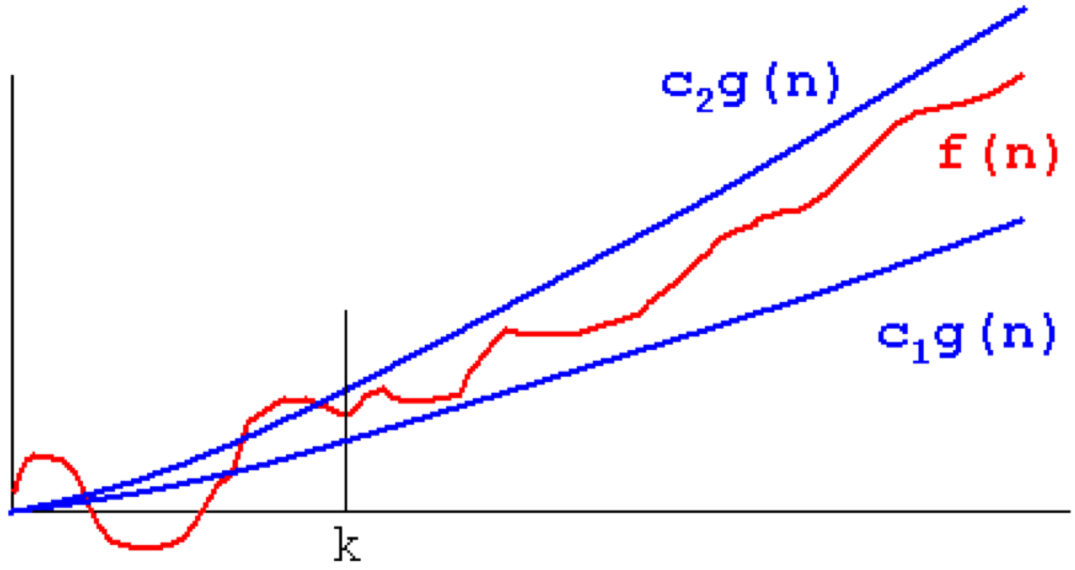


Big-Theta Notation

- Find the Asymptotic Behavior:
- $f(n) = n^6 + 3n$ $\quad n^6 + 3n \in \Theta(n^6)$
- $f(n) = 2^n + 12$ $\quad 2^n + 12 \in \Theta(2^n)$
- $f(n) = 3^n + 2^n$ $\quad 3^n + 2^n \in \Theta(3^n)$
- $f(n) = n^n + n$ $\quad n^n + n \in \Theta(n^n)$
- Once we found asymptotic behavior *g* for the function *f* we denote this by
  - $f(n)$ is $\Theta(g(n))$ ”f is theta of g".
- There is an alternative notation: $2n \in \Theta(n)$ pronounced as "two n is theta of n” and means that if the number of instructions of the algorithm is $2n$, then the asymptotic behavior of the algorithm is described by n.

Big-Theta Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. The function $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$



Big-Theta Notation

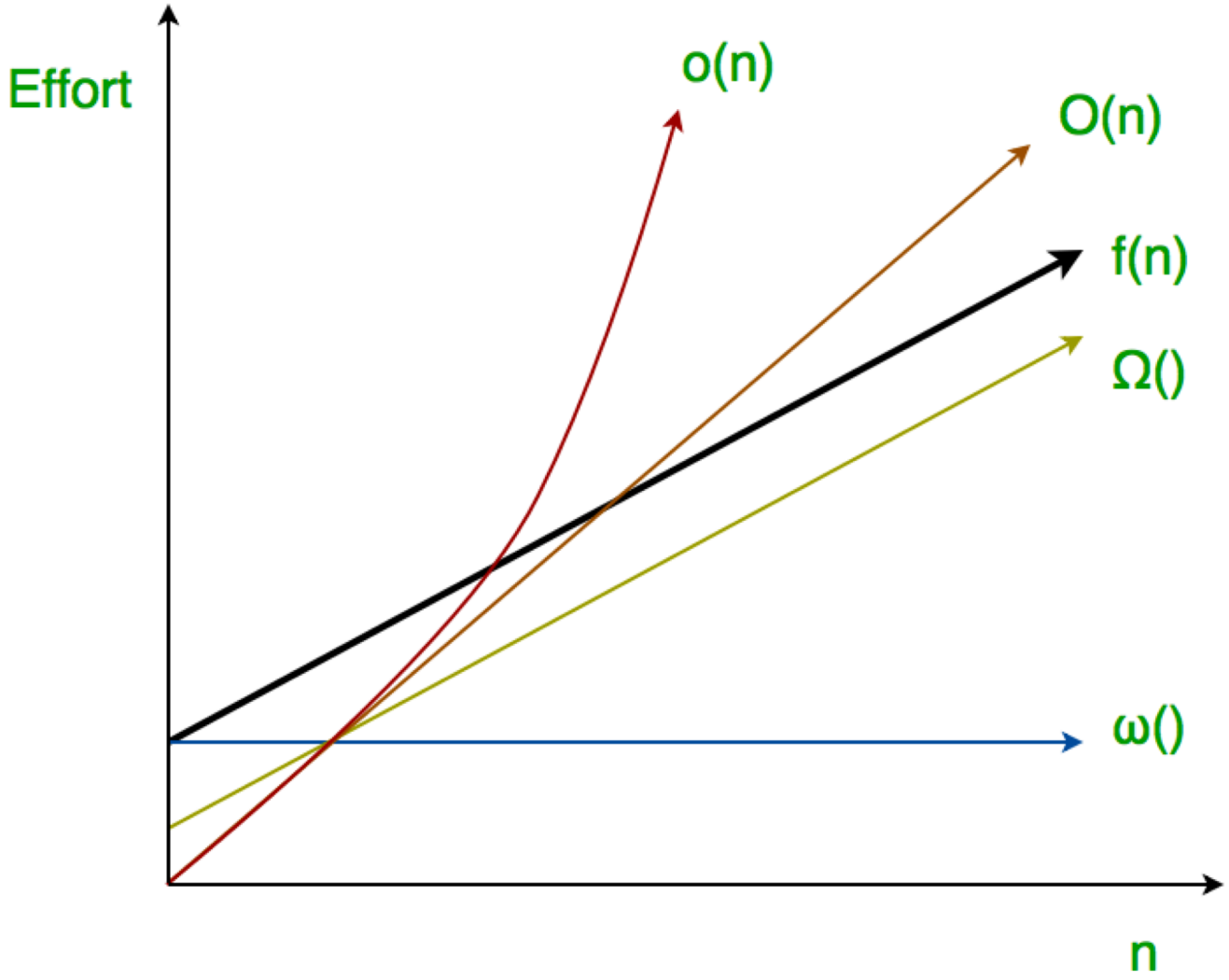
- We say that "f is big-Theta of $g(x)$" and also that "$f(x)$ is of *order* $g(x)$" and also that "$f(x)$ and $g(x)$ are of the *same order*."

- $f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1 , C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ if $x > k$. This follows from the definitions of big-*O* and big-Omega.

Little - Oh and Little - Omega

- If $f(x)$ is $O(g(x))$, but not $\Theta(g(x))$, then $f(x)$ is said to be $o(g(x))$,
  and it is read as "f(x) is little-oh of g(x)."
- For example, x is $o(x^2)$, x^2 is $o(2^x)$, 2^x is $o(x!)$.
- Similarly for little-omega ϖ .

Big-O, Big-Omega and Big-Theta



Outline

- Introduction
- Big-O, Big-Omega and Big-Theta Notations
- **Complexity of Algorithms**

Time Complexity

- Big Theta defines time complexity. An algorithm with Θ($g(n)$) is of complexity $g(n)$.
- There are special names for algorithms depending of the complexity
  - Θ(1) is a constant-time algorithm,
  - Θ(n) is a linear algorithm,
  - Θ(n^2) is a quadratic algorithm,
  - Θ(log(n)) is a logarithmic algorithm.
- Programs with a bigger Θ run slower than programs with a smaller Θ.

Comparison of Complexities

- Suppose we have three algorithms with complexities given by
  - the linear function *n*, drawn in green at the top, grows much faster than
  - the $\sqrt{n}$ function, drawn in red in the middle, which, in turn, grows much faster than
  - the log(*n*) function drawn in blue at the bottom of the plot.
- The conclusion is “execution time of a linear algorithm grows much faster than others with increasing of a size of data.”

Comparison of Complexities

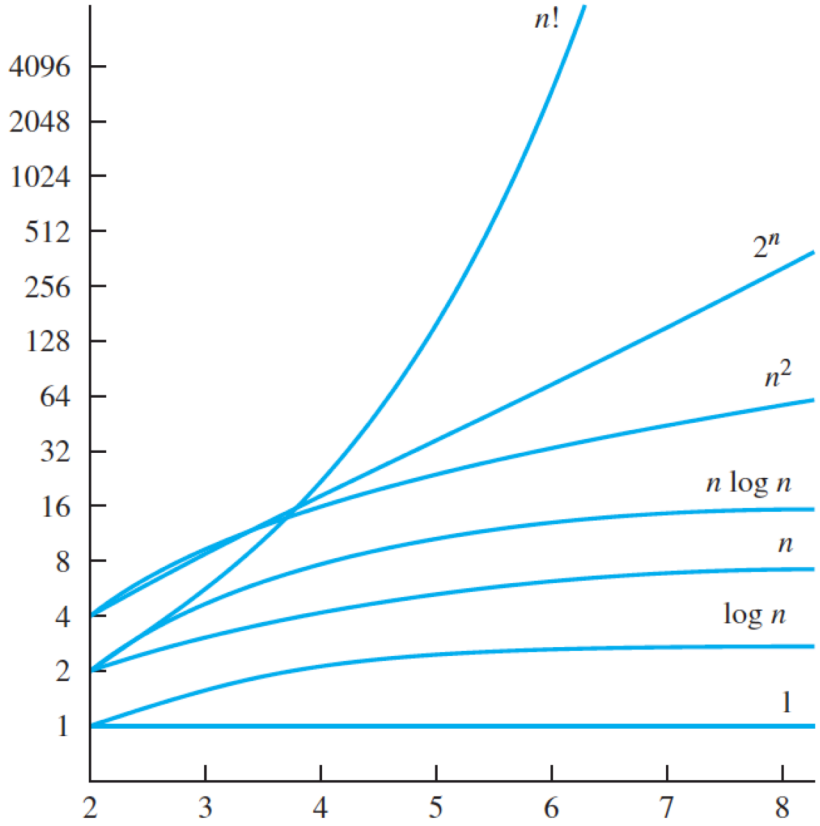


FIGURE 3 **A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.**

Complexity of the Binary Search

- Let's assume, for simplicity, that the array is always cut in exactly a half.
- The number of elements to search in each iteration:
  - 0^{th} iteration: n
  - 1^{st} iteration: n / 2
  - 2^{nd} iteration: n / 4
  - 3^{rd} iteration: n / 8

  ...
  - i^{th} iteration: $n / 2^i$

  ...
  - last iteration: 1

Complexity of the Binary Search, cont'd

- The worst-case scenario is the the value we're looking for is the last or does not exist.
- The number of iteration for the worst-case is a solution of the equation:

  $1 = n / 2^i$

  $2^i = n$

  $i = \log_2(n)$
- Therefore, the complexity of binary search is $\Theta(\log_2(n))$.
- This last result allows us to compare binary search with linear search. Clearly, as log(n) is much smaller than n, it is reasonable to conclude that binary search is a much faster method to search within an array then linear search, so it may be advisable to keep our arrays sorted if we want to do many searches within them.

Estimates of the Complexity

- In some many cases it is not straightforward to find the complexity of algorithm.
- **Example**: suppose that number of iteration of the algorithm is given by the factorial function

$$f(n) = n! = 1 \times 2 \times \cdots \times n \; .$$

- **Solution**: we may estimate the upper bound of the complexity of such algorithm if there is a function that grows at least not slower than *f*.

$$n! = 1 \times 2 \times \cdots \times n \; \leq \; n \times n \times \cdots \times n \; = \; n^n$$

or

$$n! \text{ is } O(n^n) \text{ taking } C = 1 \text{ and } k = 1.$$

Next class

- Topic: Integers and Division
- Pre-class reading: Chap 4.1