Lect 01. Algorithms : Efficiency, Analysis, and Order

Spring, 2020

School of Computer Science & Engineering

Pusan National University

2

Algorithms and Problem

Algorithms(1)

- computer program
  - Be composed of individual modules, understandable by a computer, that solve specific tasks (such as sorting).
- The design of the individual modules that accomplish the specific tasks
  - These specific tasks are called **problems**.
- A problem is a question to which we seek an answer.

Algorithms(2)

- Procedures for solving the problem
- A step-by-step procedures designed carefully
- A series of procedures to change input to output(solution)

Algorithms(3)

- A problem may contain **variables** that **are not assigned specific values** in the statement of the problem. These variables are called **parameters** to the problem.
- Because a problem contains **parameters**, it represents a class of problems, one for each assignment of values to the parameters. Each specific assignment of values to the parameters is called an **instance** of the problem.

What is a problem?

- Definition
  - A mapping/relation between a set of input instances (domain) and an output set (range)
- Problem Specification
  - Specify what a typical input instance is
  - Specify what the output should be in terms of the input instance
- Example: Sorting
  - **Input**: A sequence of N numbers $a_1 \ldots a_n$
  - **Output**: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \leq \ldots \leq a_n$.

Problem

- Problem
  - A question to which we seek an answer.
- Parameters
  - **Variables** that are not assigned specific values in the statement of the problem.
- Instance of problem(input)
  - Each specific assignment of values to the parameters
- Solution of instance(output)
  - The answer to the question asked by the problem in that instance

Problem(Ex: search)

- Problem: Determine whether the number x is in the list S of n numbers. The answer is yes if x is in S and no if it is not.
- Parameters: S, n, x
- Example of instance: $S = [10,7,11,5,3,8]$, $n = 6$, $x = 5$
- Solution of instance: “yes”

Expression of Algorithms

- Methods for Expressing Algorithms
  - Implementations(c-like language)
  - Pseudo-code
  - English
- Writing clear and understandable proofs
- Not the specific language used but the clarity of your algorithm/proof

Two desired properties of algorithms

- Correctness
  - Always provides correct output when presented with legal input
- Efficiency
  - Computes correct output quickly given input

Correctness

- Example: Traveling Salesperson Problem (TSP)
- **Input**: A sequence of N cities with the distances d_{ij} between each pair of cities
- **Output**: a permutation (ordering) of the cities $<c_{1'}, \ldots, c_{n'}>$ that minimizes the expression

$$\Sigma_{j=1 \text{ to } n-1} d_{j',j'+1} + d_{n',1'}$$

- Which of the following algorithms is correct?
  - Nearest neighbor: Initialize tour to city 1. Extend tour by visiting nearest unvisited city. Finally return to city 1.
  - All tours: Try all possible orderings of the points selecting the ordering that minimizes the total length:

Efficiency

- Example: Odd Number Problem
- **Input**: A number *n*
- **Output**: Yes if *n* is odd, no if *n* is even
- Which of the following algorithms is most efficient?
  - Count up to that number from one and alternate naming each number as odd or even.
  - Factor the number and see if there are any twos in the factorization.
  - Keep a lookup table of all numbers from 0 to the maximum integer.
  - Look at the last bit (or digit) of the number.

Sequential vs Binary Search

- To compare Sequential Search to Binary Search,
  - Sequential Search **does *n* comparisons** to determine that *x* is not in the array of size *n*. If *x* is in the array, the number of comparisons is no greater than *n*.
  - A Binary Search is an algorithm for locating the position of an element in a **sorted list**. The method reduces the number of elements that need to be examined by two each time.
  - Binary Search appears to be much more efficient than Sequential Search.

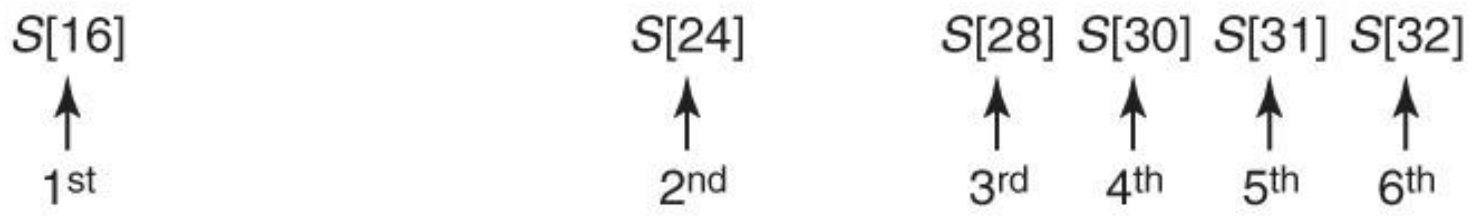


Figure 1.1: The array items that Binary Search compares with x when x is large than all the items in an array of size 32. The items are numbered according to the order in which they are compared.

Fibonacci Sequence

- By definition, the first two Fibonacci numbers are 0 and 1. Each remaining number is the sum of the previous two. A recurrence relation defines a function by means of an expression that includes one or more (smaller) instances of itself. An example of a recurrence is the Fibonacci sequence:

Fib(n) = Fib(n – 1) + Fib(n – 2) for n > 2;

Fib(1) = Fib(2) = 1

- By doing the examples in the book, we can see that this algorithm is extremely inefficient.

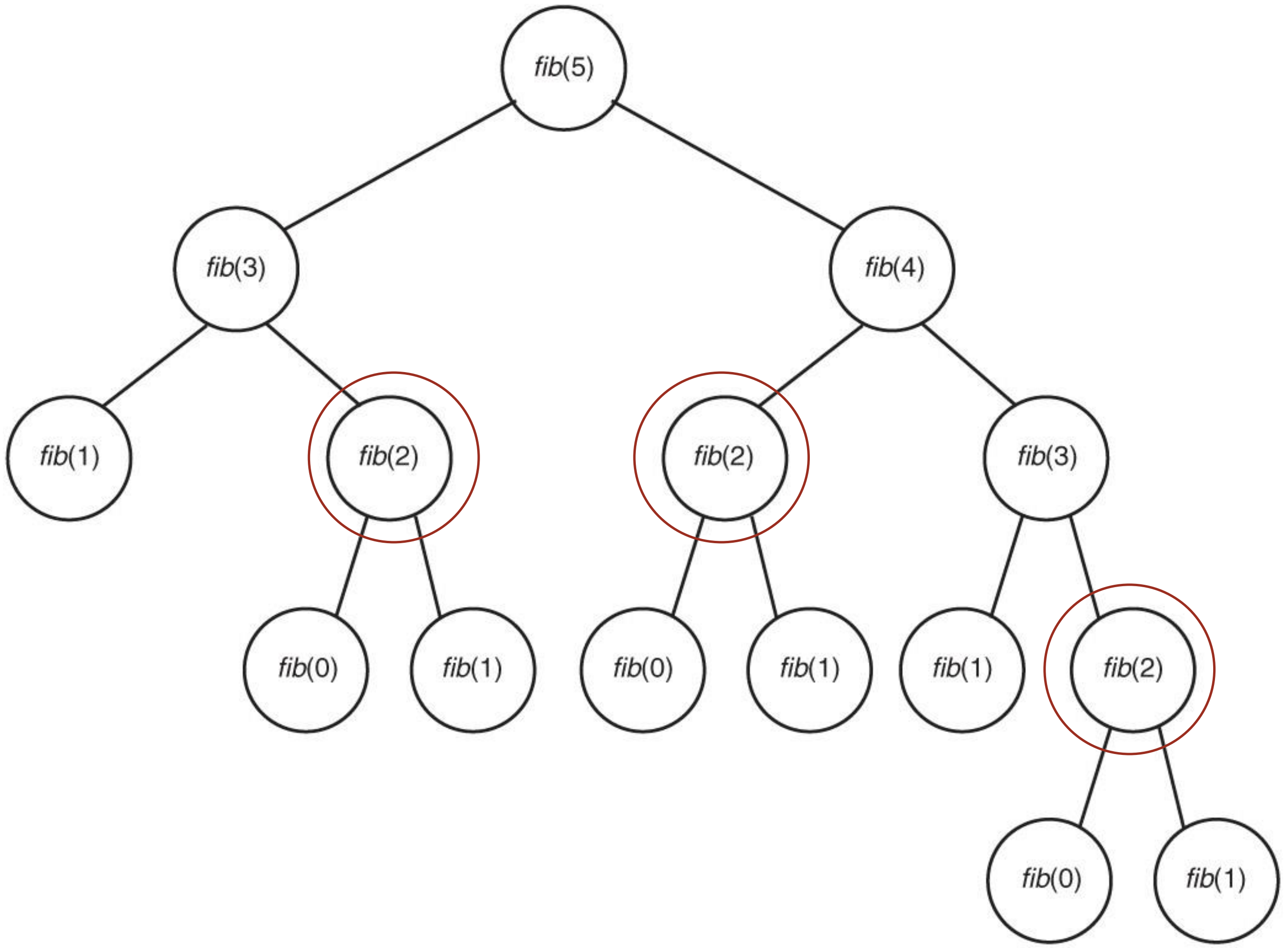


Figure 1.2: The recursion tree corresponding to Algorithm 1.6 when computing the fifth Fibonacci term.

Fibonacci sequence

- Let's call T(n) the number of terms in the recursion tree for n.

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

  $> 2*T(n-2)$ .... $T(n-1)>T(n-2)$

  $> 2*2*T(n-4)$

  $> 2*2*2*T(n-6)$

  ...

  $> \underbrace{2*2* \ldots *2}_{\text{n/2 terms}} *T(0)$

  $T(n) > 2^{n/2}$

- See Table 1.2

Analysis of Algorithms

Analysis of Algorithms

- **Algorithm analysis** measures **the efficiency of an algorithm** as the **input size** becomes large.
- The critical resource for a program is most often its **running time**. Other factors should be taken into account, like the **space** required to run the program.

Analyzing algorithms

- The “process” of determining how much resources (time, space) are used by a given algorithm
- We want to be able to make quantitative assessments about the value (goodness) of one algorithm compared to another
- We want to do this WITHOUT implementing and running an executable version of an algorithm

Complexity Analysis

- A measure that is independent of the computer, the programming language, the programmer, and all the complex details of the algorithm.
- The running time of the algorithm increases with the size of the input. The total running time is proportional to how many times some basic operation is done
- Analyze an algorithm's efficiency be determining the **number of times some basic operation** is done as a function of the size of the input. The size of the input is called the **input size**.
- A **time complexity analysis** of an algorithm is the determination of how many times the basic operation is done for each value of the input size.
- **Memory(Space) complexity** is an analysis of how efficient the algorithm is in terms of memory. A **complexity function** can be any function that maps the positive integers to the nonnegative reals.

Measuring Complexity

- The running time of an algorithm is the function defined by the number of steps (or amount of memory) required to solve input instances of size n
  - F(1) = 3
  - F(2) = 5
  - F(3) = 7
  - …
  - F(n) = 2n+1
- Problem: Inputs of the same size may require different numbers of steps to solve

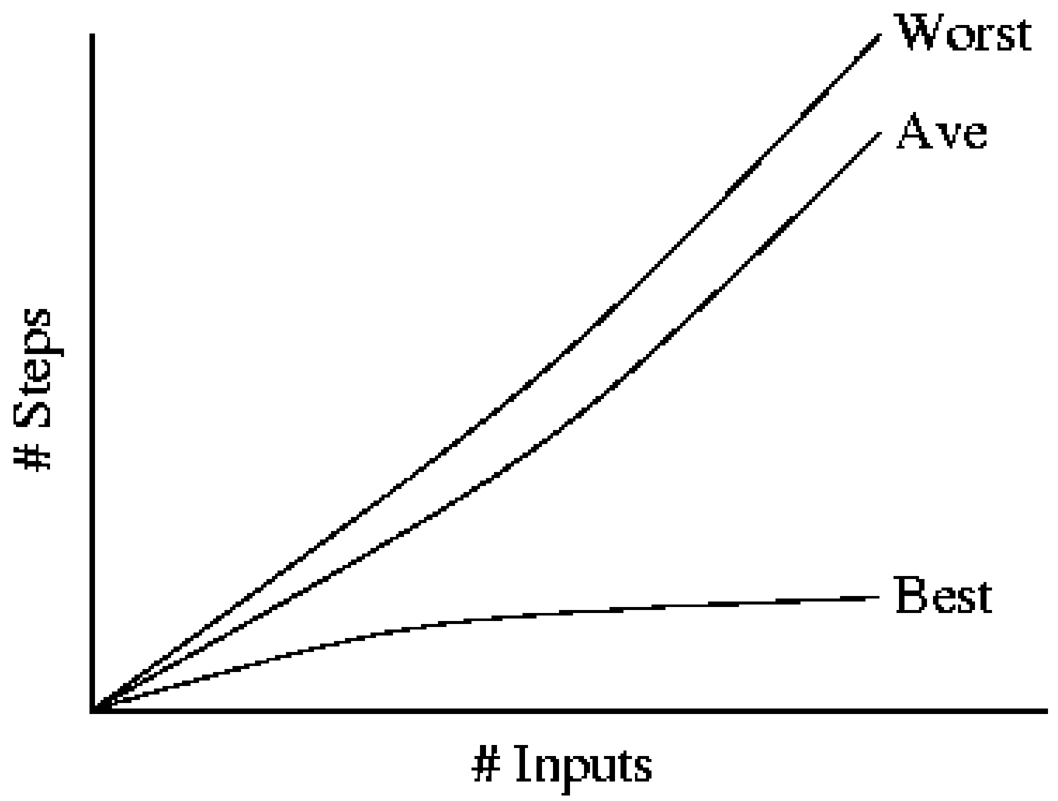
3 different analyses

- The *worst case running time* of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.
- The *best case running time* of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.
- The *average-case running time* of an algorithm is the function defined by an average number of steps taken on any instance of size n.
- Which of these is the best to use?

Average case analysis

- Drawbacks
  - Based on a probability distribution of input instances
    - The distribution may not be appropriate
    - Provides little consolation if we have a worst-case input
  - More complicated to compute than worst case running time
- Worst case running time is often comparable to average case running time (see next graph)
  - Counterexamples to above point:
    - Quicksort
    - simplex method for linear programming

Best, Worst, and Average Case



Worst case analysis

- Typically much simpler to compute as we do not need to "average" performance on many inputs
    - Instead, we need to find and understand an input that causes worst case performance
- Provides guarantee that is independent of any assumptions about the input
- Often reasonably close to average case running time
- The standard analysis performed

Complexity Analysis

- **T(n)** is defined as **the number of times** the algorithm does the basic operation for an instance of size n.
- T(n) is called the **every-case time complexity of the algorithm**, and the **determination of T(n)** is called an **every-case time complexity analysis**.
- **W(n)** is defined as **the maximum number of times** the algorithm will ever do its basic operation for an input size of n. So W(n) is called the **worst case time complexity of the algorithm**, and the determination of W(n) is called a worst-case time complexity analysis.
- **A(n)** is called the **average-case time complexity of the algorithm**, and the determination of A(n) is called an average-case time complexity analysis.
- **B(n)** is defined as **the minimum number of times the algorithm** will ever do its basic operation for an input size of n. So B(n) is called the best-case time complexity of the algorithm, and the determination of B(n) is called a best-case time complexity analysis.

27

Order

Order

- Algorithms with time complexities such as n and 100n are called **linear-time algorithms** because their time complexities are linear in the input size n, whereas algorithms with time complexities such as n^2 and $0.01n^2$ are called **quadratic-time algorithms** because their time complexities are quadratic in the input size n. Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm.

Motivation for Asymptotic Analysis

- An *exact computation* of worst-case running time can be difficult
  - Function may have many terms:
    - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75$
- An *exact computation* of worst-case running time is unnecessary
  - Remember that we are already approximating running time by using RAM model

Simplifications

- Ignore constants
  - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75$ becomes
  - $n^2 – n \log n + n - n^{2/3} + 1$
- Asymptotic Efficiency
  - $n^2 – n \log n + n - n^{2/3} + 1$ becomes n^2
- End Result: $\Theta(n^2)$

Why ignore constants?

- RAM model introduces errors in constants
  - Do all instructions take equal time?
  - Specific implementation (hardware, code optimizations) can speed up an algorithm by constant factors
  - We want to understand how effective an algorithm is independent of these factors
- Simplification of analysis
  - Much easier to analyze if we focus only on n^2 rather than worrying about 3.7 n^2 or 3.9 n^2

An Intuitive Introduction to Order

- Functions such as $5n^2$ and $5n^2 + 100$ are called **pure quadratic functions** because they contain no linear term, whereas a function such as $0.1n^2 + n + 100$ is called a **complete quadratic function** because it contains a linear term.
- We should always be able to throw away low-order terms when classifying complexity functions. For example, it seems that we should be able to classify $0.1n^3 + 10n^2 + 5n + 25$ with **pure cubic** functions.
- When an algorithm's time complexity is in $\theta(n^2)$, the algorithm is called a **quadratic-time algorithm** or a $\theta(n^2)$ algorithm.

An Intuitive Introduction to Order

- Examples of complexity categories:

  $\Theta(\lg(n)), \theta(n), \theta(n \lg(n)), \theta(n^2), \theta(n^3), \theta(2^n)$

- There is more information in knowing a time complexity than in simply knowing its order. If it takes the same amount of time to process basic operations and execute the overhead instructions in both algorithms, then the quadratic-time algorithm is more efficient in instances smaller than 10,000. If the application never requires instances larger than this, the quadratic-time algorithm should be implemented. When it is difficult to determine time complexities exactly, we have to be content with determining only the order.

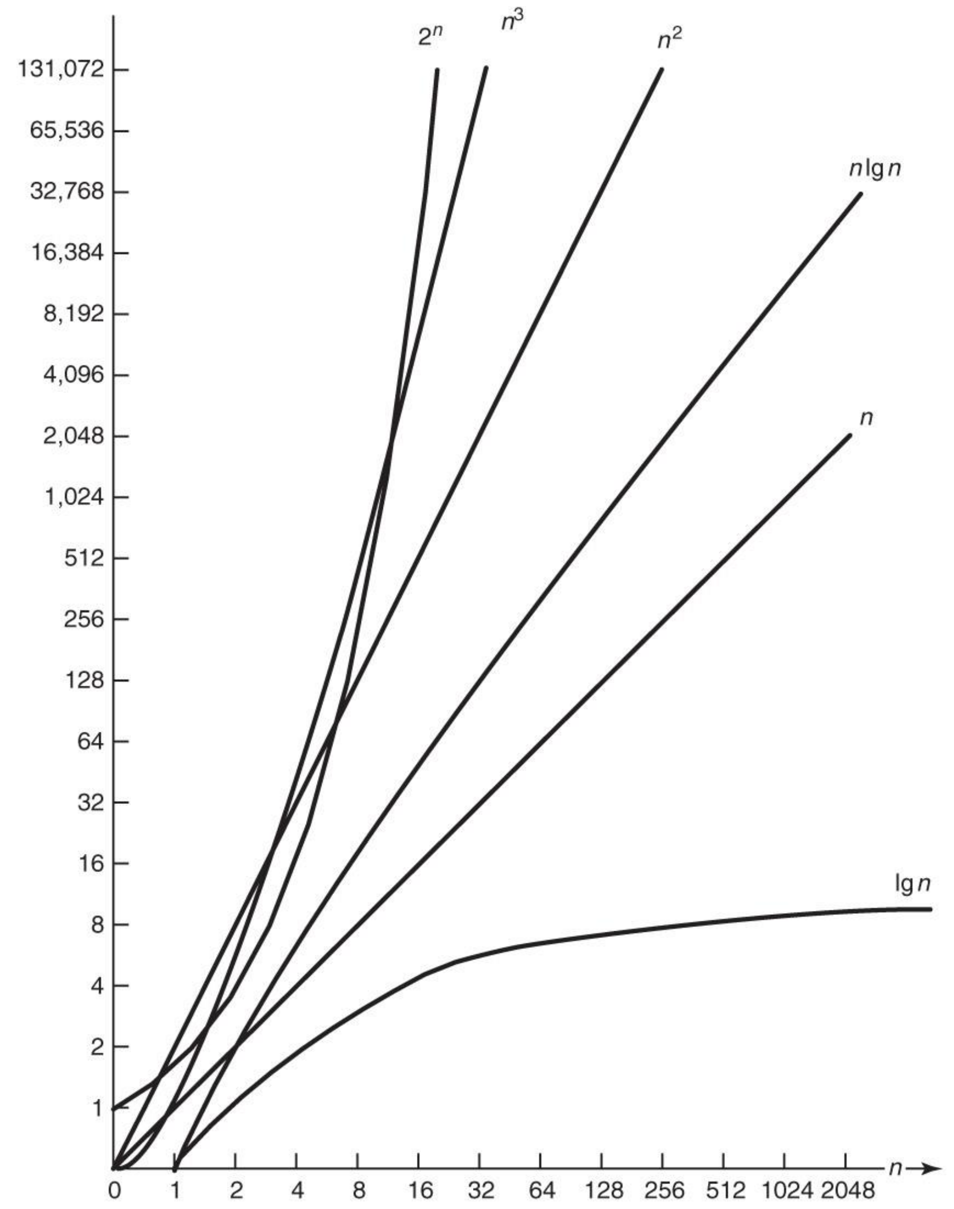


Figure 1.3: Growth rates of some common complexity functions.

Asymptotic Analysis

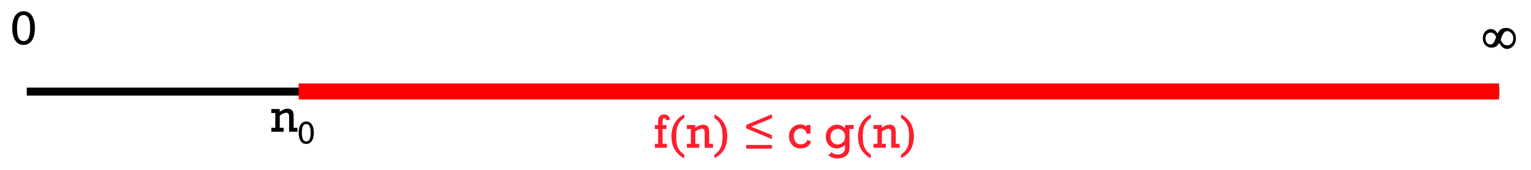
- We focus on the infinite set of large n ignoring small values of n
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

"Big Oh" Notation

- $O(g(n))$ =

  $\{f(n)$: there exist positive constants c and n_0 such that $\forall\ n \geq n_0,\ 0 \leq f(n) \leq c\ g(n)\ \}$

  - What are the roles of the two constants?
    - n_0:
    - c:



Set Notation Comment

- $O(g(n))$ is a set of functions.
- However, we will use one-way equalities like
  $n = O(n^2)$
- This really means that function n belongs to the set of functions $O(n^2)$
- Incorrect notation: $O(n^2) = n$
- Analogy
  - "A dog is an animal" but not "an animal is a dog"

Three Common Sets

- **$f(n) = O(g(n))$** means $c \times g(n)$ is an *Upper Bound* on $f(n)$
- **$f(n) = \Omega(g(n))$** means $c \times g(n)$ is a *Lower Bound* on $f(n)$
- **$f(n) = \Theta(g(n))$** means $c_1 \times g(n)$ is an *Upper Bound* on $f(n)$ *and* $c_2 \times g(n)$ is a *Lower Bound* on $f(n)$
- These bounds hold for all inputs beyond some threshold n_0.

A Rigorous Introduction to Order

- 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 ≥ N,**

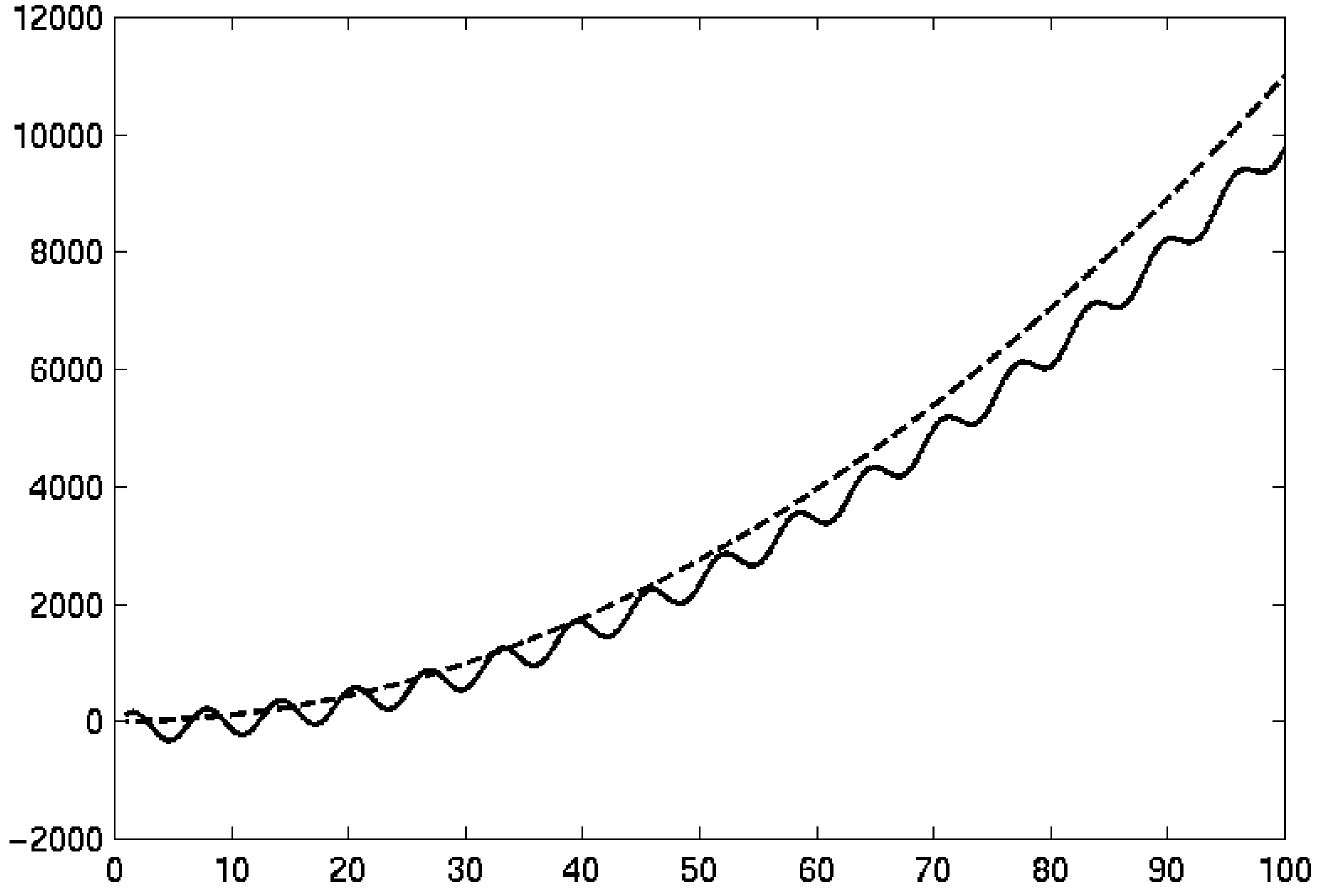
  $$\mathbf{g(n) \leq c \times f(n)}$$

  If g(n) ϵ O(f(n)), we say that g(n) is big O of f(n).

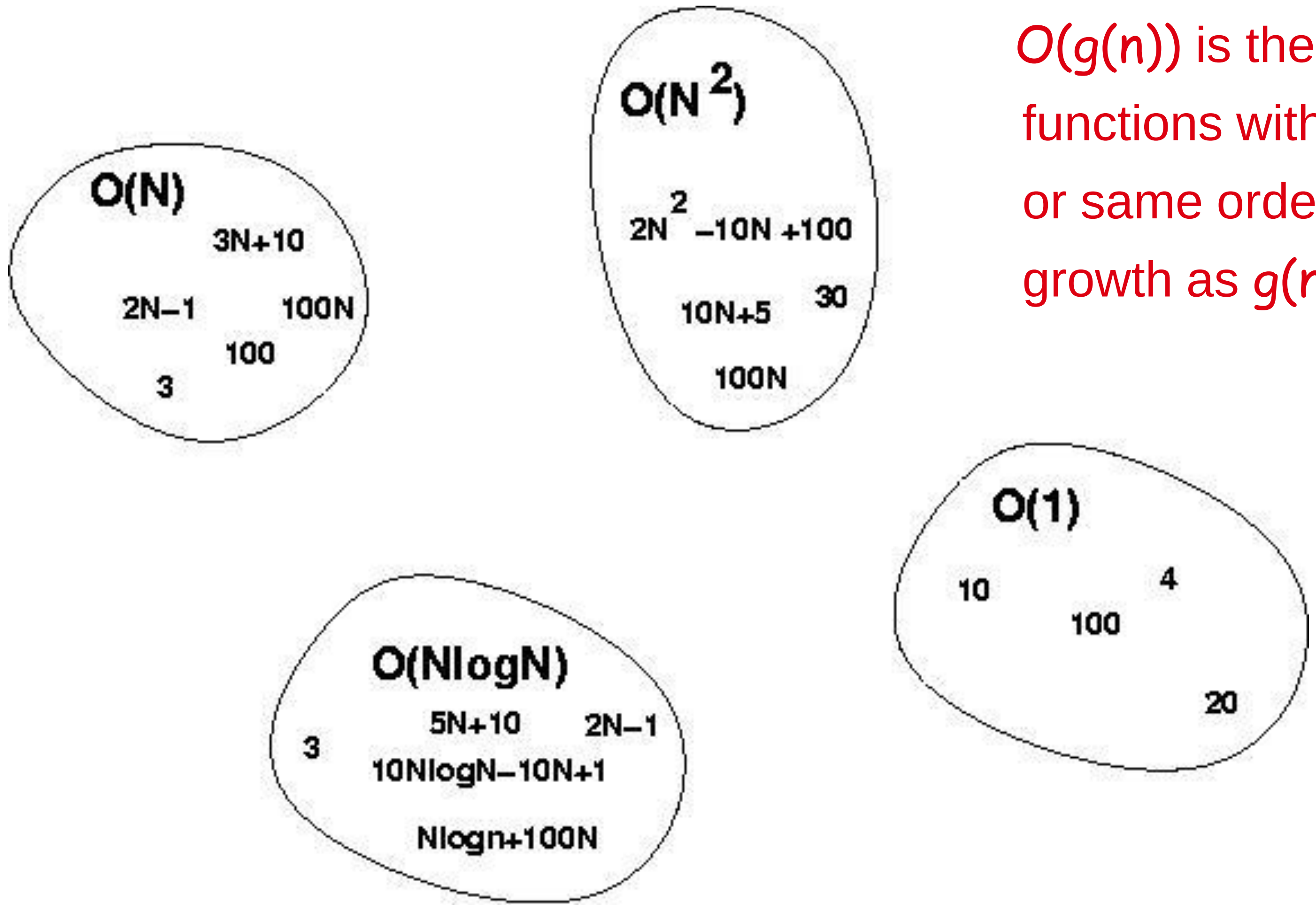
- "Big O" describes the **asymptotic behavior** of a function because it is concerned only with eventual behavior. We say that "big O" puts an asymptotic upper bound on a function.

-

O(g(n))


12000
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
50
60
70
80
90
100

Big-O Visualization



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Example

Algorithm 1	**Cost**	*Algorithm 2*	**Cost**
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...			
arr[N-1] = 0;	c_1		
	-----------		-------------
$c_1+c_1+\ldots+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$	

- Both algorithms are of the same order: *O(N)*

Example (cont'd)

Algorithm 3	***Cost***
`sum = 0;`	c_1
`for(i=0; i<N; i++)`	c_2
`  for(j=0; j<N; j++)`	c_2
`      sum += arr[i][j];`	c_3

$$c_1 + c_2 \, x \, (N+1) + c_2 \, x \, N \, x \, (N+1) + c_3 \, x \, N^2 = O(N^2)$$

Examples

- $2n^2 = O(n^3)$: $2n^2 \le cn^3 \Rightarrow 2 \le cn \Rightarrow c = 1$ and $n_0 = 2$
- $n^2 = O(n^2)$: $n^2 \le cn^2 \Rightarrow c \ge 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2+1000n = O(n^2)$:

$$1000n^2+1000n \le 1000n^2+ n^2 = 1001n^2 \Rightarrow c=1001 \text{ and } n_0 = 1000$$

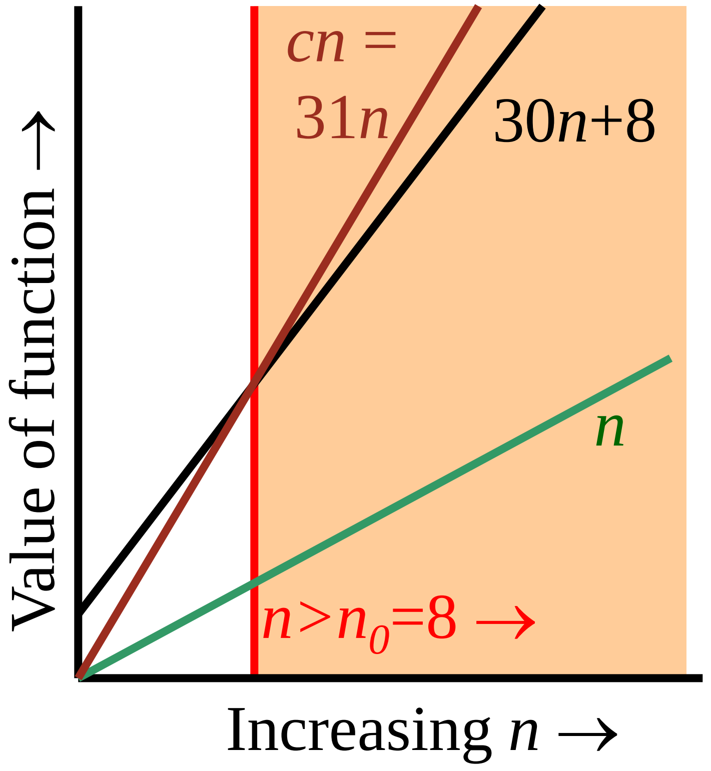
- $n = O(n^2)$: $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \leq cn$, $\forall n > n_0$.
    - Let $c=31$, $n_0=8$. Assume $n>n_0=8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.

Big-O example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.



$\boxed{30n+8 \in O(n)}$

No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2$

    for all $n \geq 5$

    $n_0 = 5$ and $c = 101$ is a solution
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$

    for all $n \geq 1$

    $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

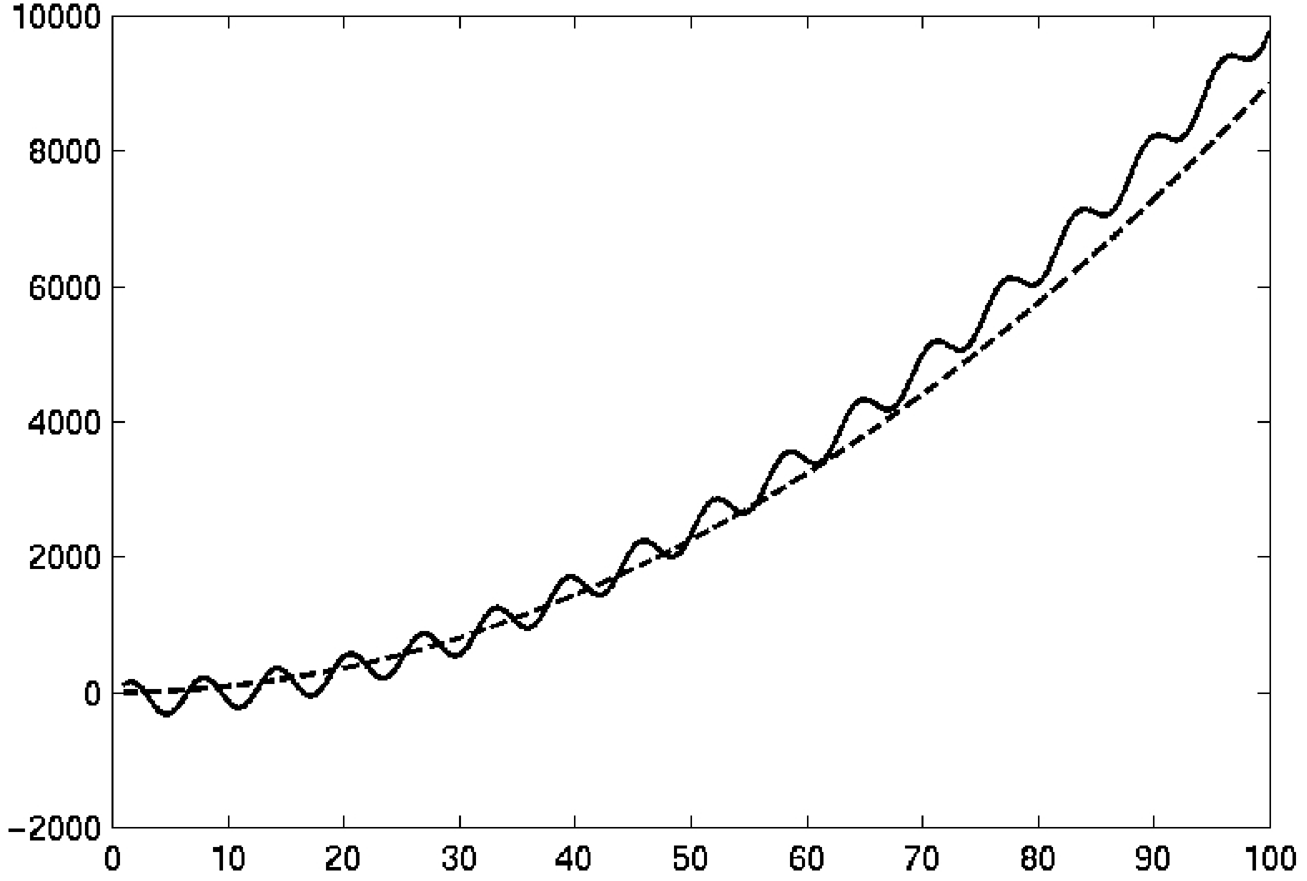
A Rigorous Introduction to Order

- Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like "big O", this **is a measure of the algorithm's growth rate**, and it works for any resource, but we most often measure the least amount of time required. **The lower bound of an algorithm** is denoted by the symbol **Ω**. The following is the definition for Ω:
- For a given complexity function f(n), **Ω(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 ≥ N,**

$$g(n) \geq c \times f(n).$$

If $g(n) \in \Omega(f(n))$, we say that g(n) is omega of f(n)

$\Omega(g(n))$



Examples

- $5n^2 = \Omega(n)$

  $\exists\ c, n_0$ such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \quad \Rightarrow c = 1$ and $n_0 = 1$

- $100n + 5 \neq \Omega(n^2)$

  $\exists\ c, n_0$ such that: $0 \leq cn^2 \leq 100n + 5$

  $100n + 5 \leq 100n + 5n\ (\forall\ n \geq 1) = 105n$

  $cn^2 \leq 105n \quad \Rightarrow n(cn - 105) \leq 0$

  Since n is positive $\Rightarrow cn - 105 \leq 0 \qquad \Rightarrow n \leq 105/c$

  $\Rightarrow$ contradiction: n cannot be smaller than a constant

- $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

A Rigorous Introduction to Order

- For a given complexity function f(n),

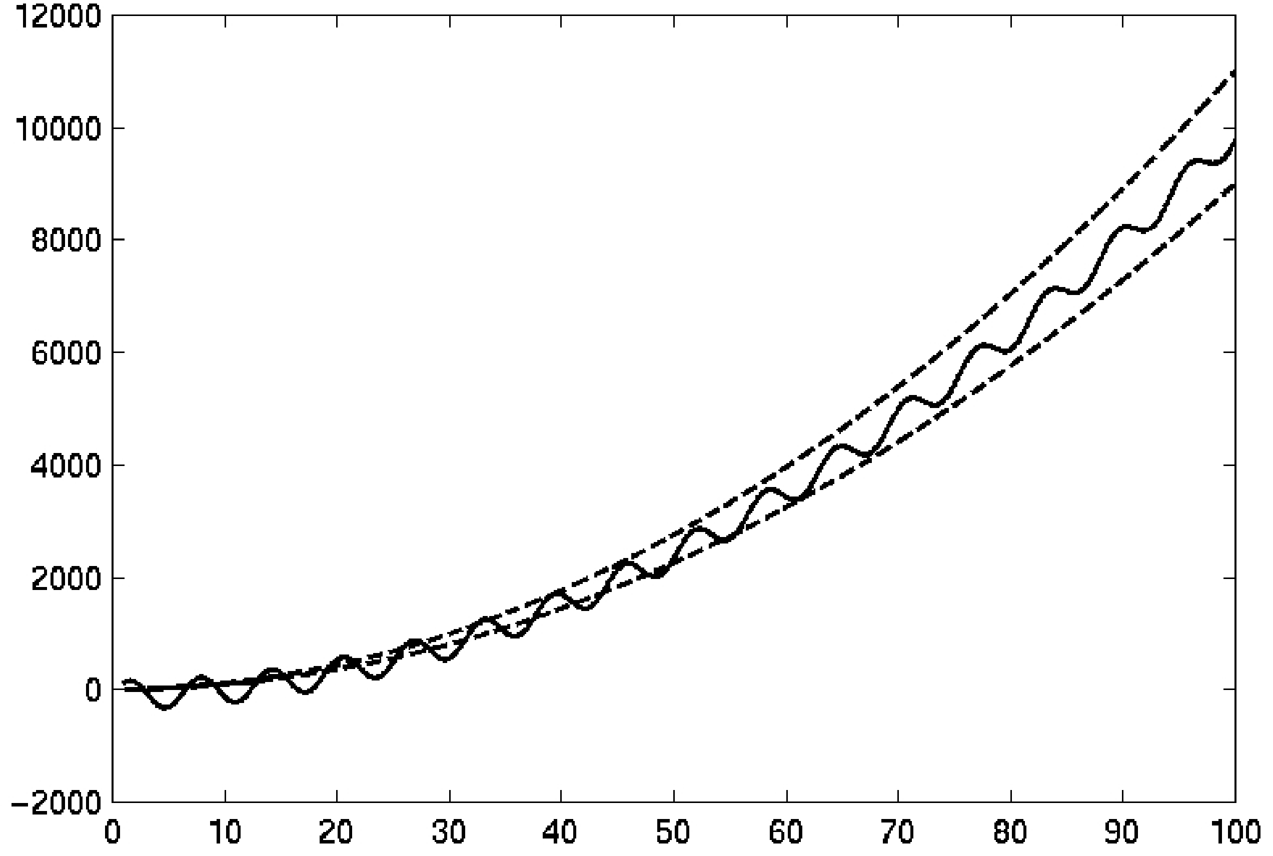
  $\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

- This means that **θ(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 ≥ N,**

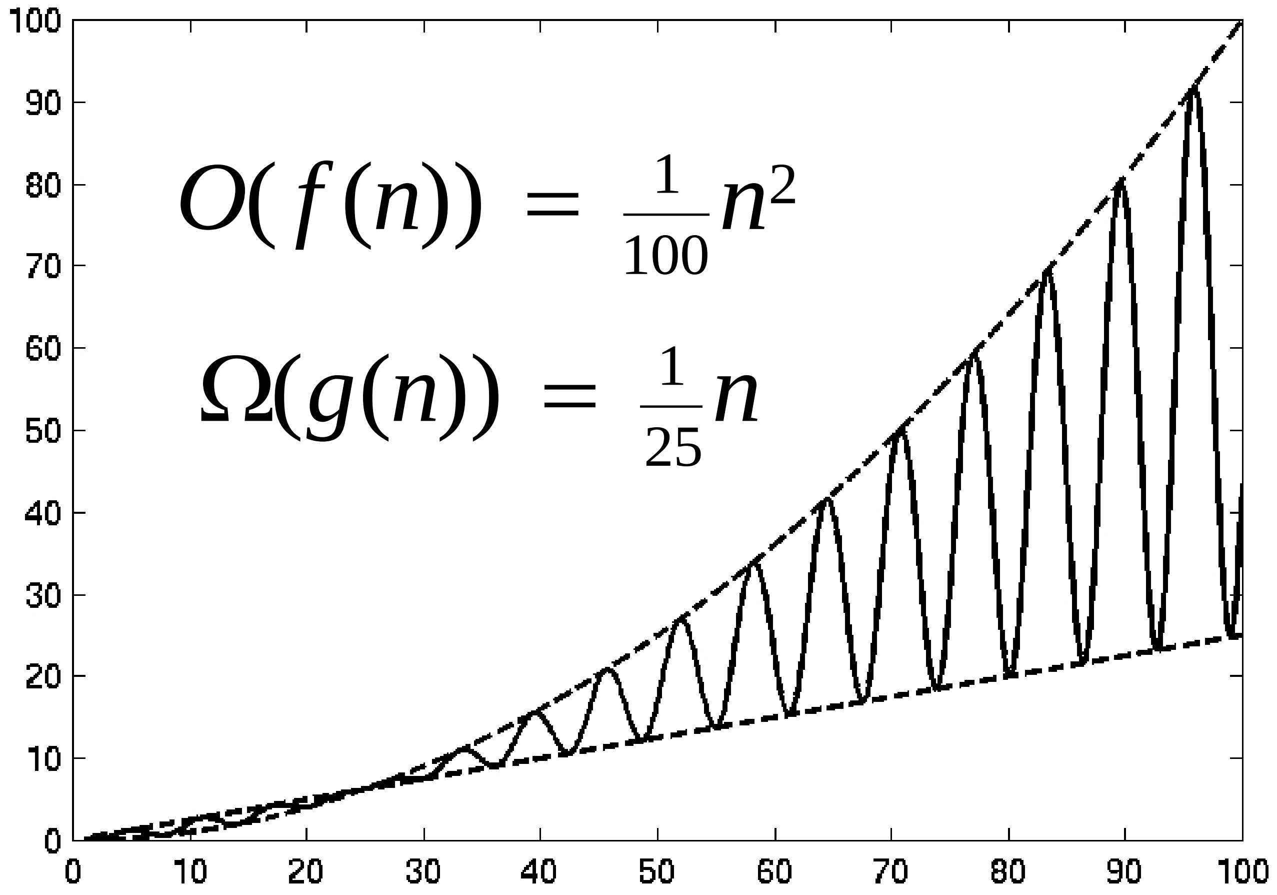
  $$c \times f(n) \le g(n) \le d \times f(n).$$

If g(n) ϵ θ(f(n)), we say that g (n) is **order** of f(n).

$\Theta(g(n))$


12000
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
50
60
70
80
90
100

O(f(n)) and Ω(g(n))



$O(f(n)) = \frac{1}{100}n^2$
$\Omega(g(n)) = \frac{1}{25}n$
0
10
20
30
40
50
60
70
80
90
100

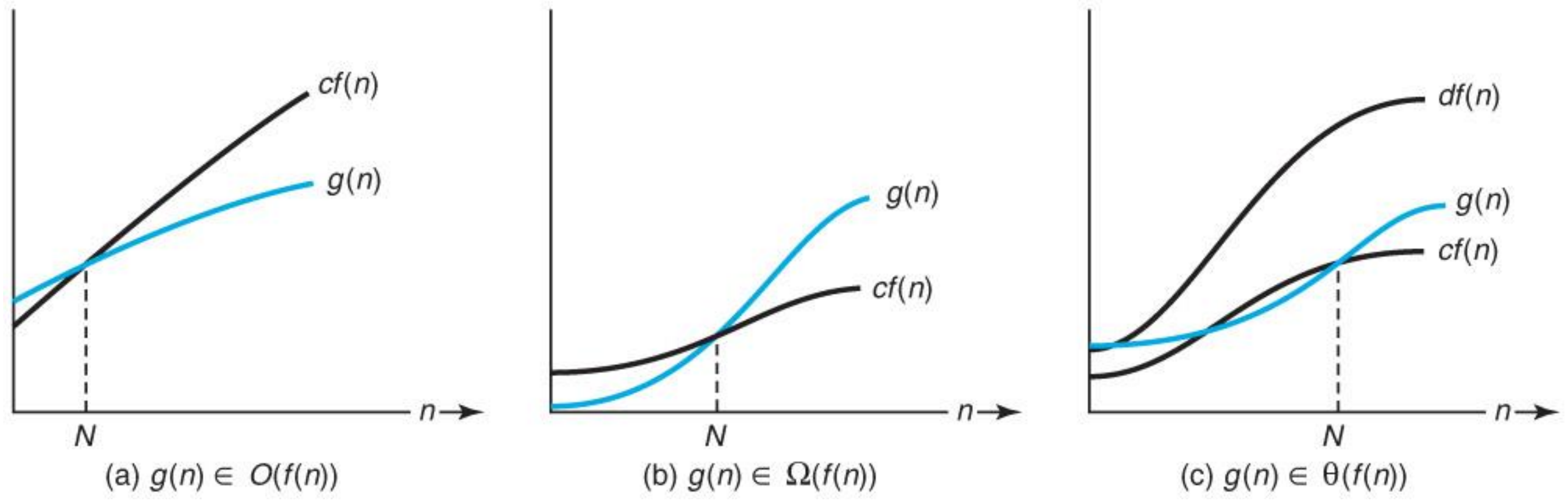


Figure 1.4: Illustrating "big O", Ω and Θ

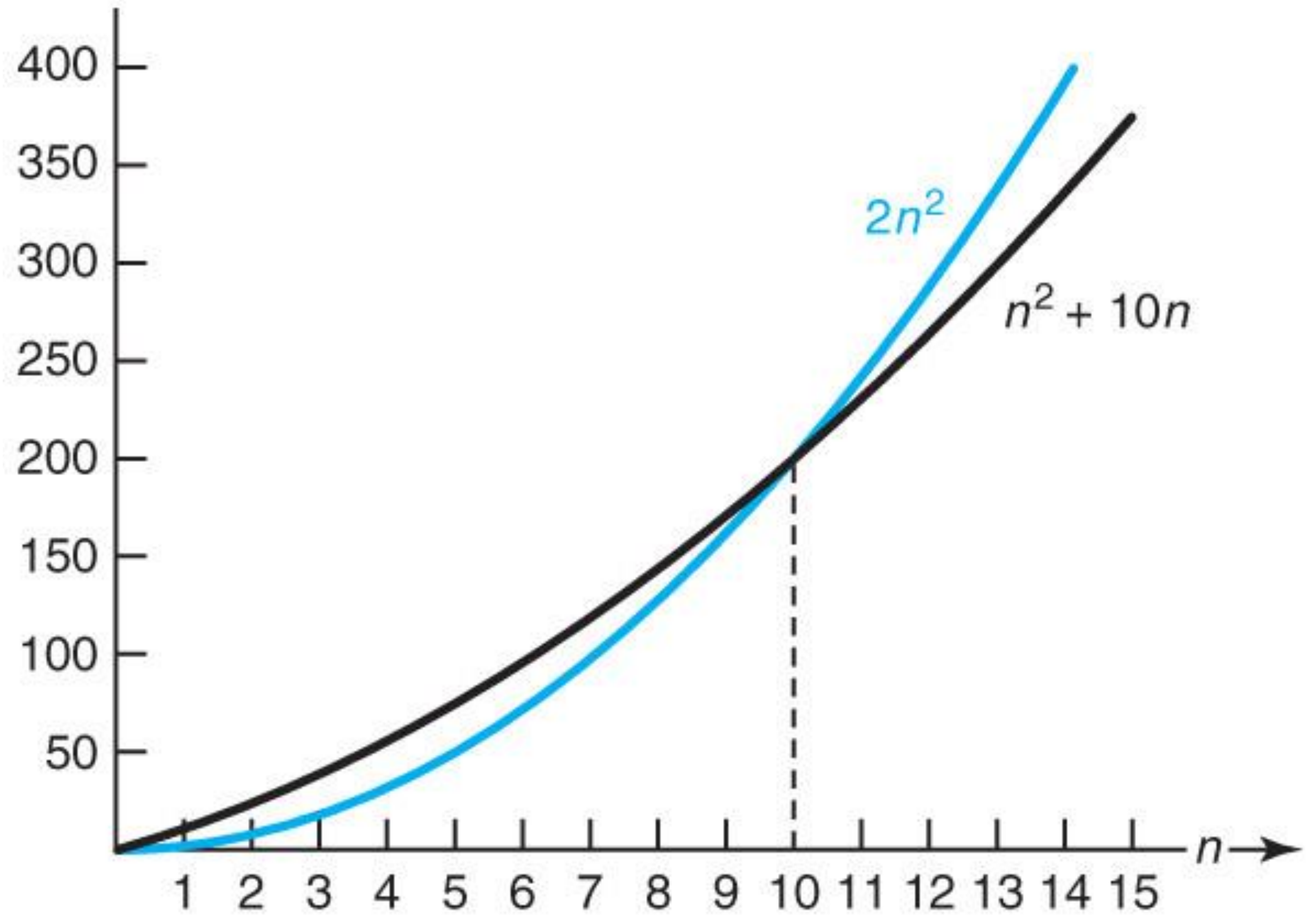


Figure 1.5: The function n2 + 10n evenually stays beneath the function 2n2

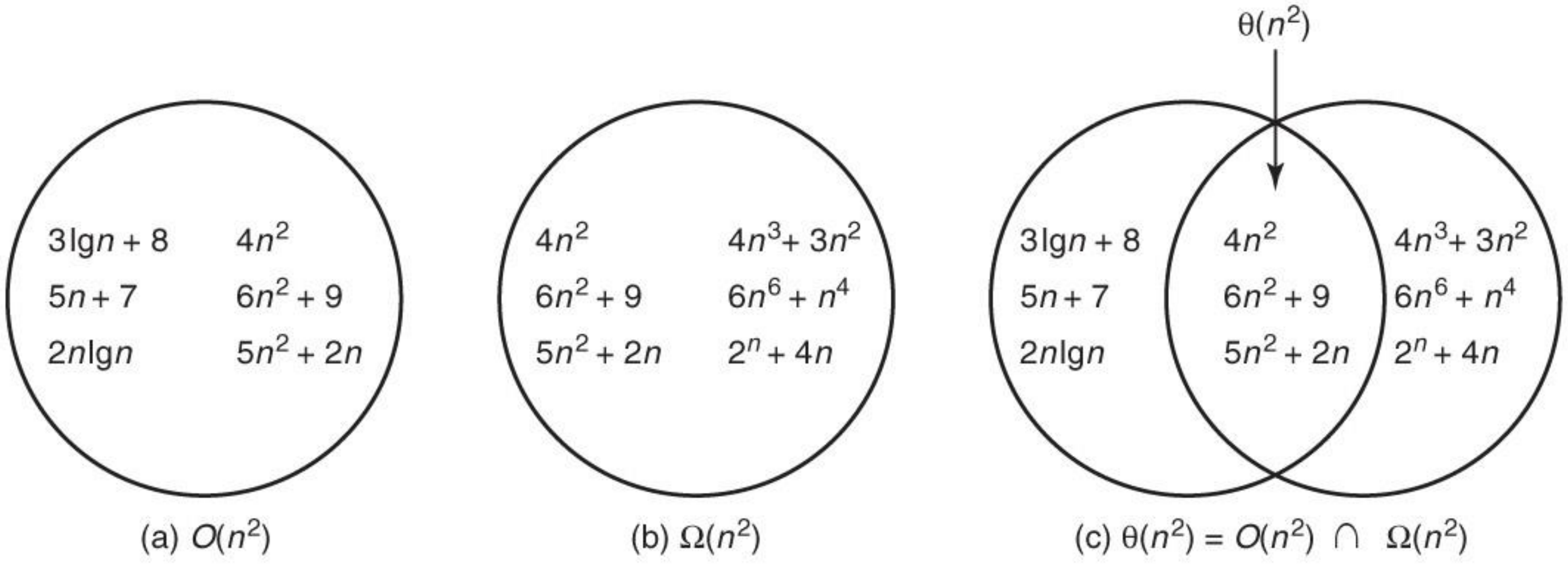


Figure 1.6: The sets O (n2), Ω (n2)and Θ (n2). Some exemplary members are shown.

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \le \frac{1}{2} n^2 \ \forall n \ge 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \ge \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \ (\forall n \ge 2) = \frac{1}{4} n^2 \Rightarrow \quad c_1 = \frac{1}{4}$

- $n \ne \Theta(n^2)$: $c_1 n^2 \le n \le c_2 n^2$

$\Rightarrow$ only holds for: $n \le 1/c_1$

Examples

- $6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

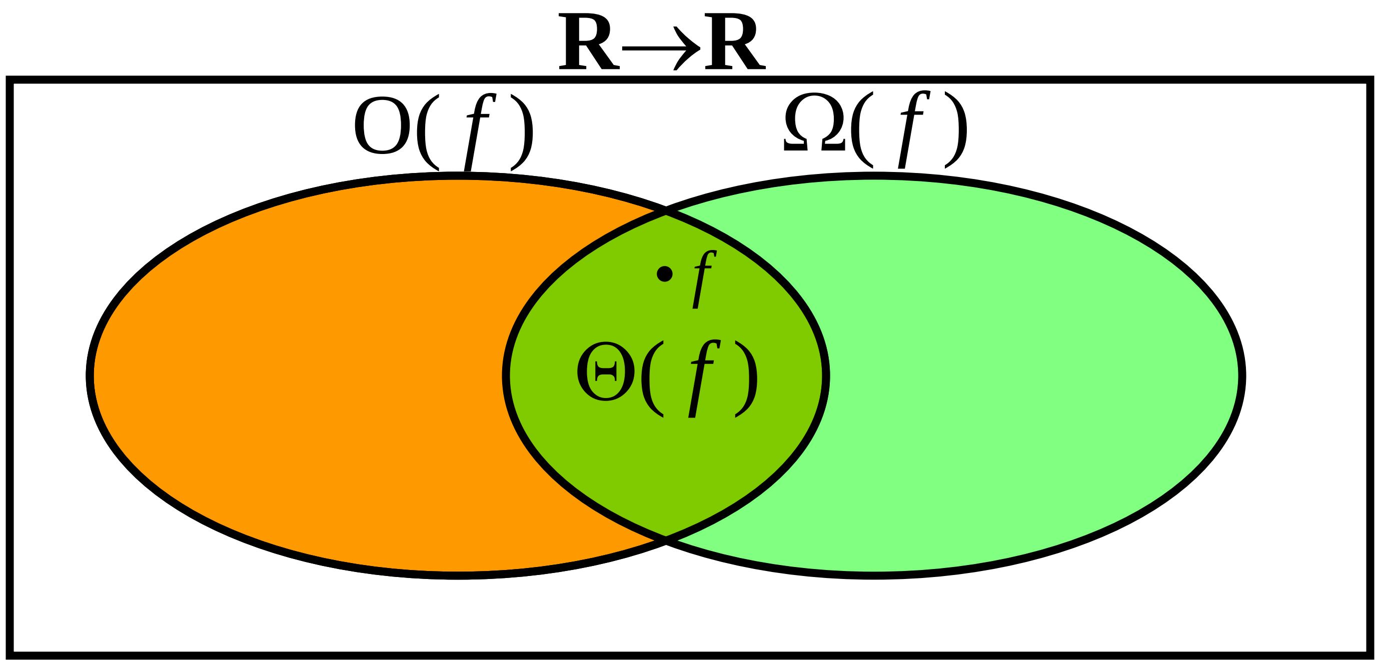
  $\Rightarrow$ only holds for: $n \leq c_2 / 6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

  $\Rightarrow c_2 \geq n/\log n, \forall\ n \geq n_0$ – impossible

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Example

- $f(n) = 3n^2 - 100n + 6$

	c	n_0
$3n^2 - 100n + 6 = O(n^2)$		
$3n^2 - 100n + 6 = O(n^3)$		
$3n^2 - 100n + 6 \neq O(n)$		
$3n^2 - 100n + 6 = \Omega(n^2)$		
$3n^2 - 100n + 6 \neq \Omega(n^3)$		
$3n^2 - 100n + 6 = \Omega(n)$		
$3n^2 - 100n + 6 = \Theta(n^2)$?		
$3n^2 - 100n + 6 = \Theta(n^3)$?		
$3n^2 - 100n + 6 = \Theta(n)$?		

Proof

- Proof by contradiction: assume something is true, and then do manipulations that lead to a result that is not true.

- For a given complexity function f(n), o(f(n)) is the set of all complexity functions g(n) satisfying the following: For every positive real constant c there exists a nonnegative integer N such that, for all $n \geq N$,

  $$g(n) \leq c \times f(n).$$

If $g(n) \in o(f(n))$, we say that g (n) is small o of f(n).

Logarithms and properties

- In algorithm analysis we often use the notation "log n" without specifying the base

Binary logarithm	$\lg n = \log_2 n$
Natural logarithm	$\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

More Examples

- For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct.
  - $f(n) = \log n^2$; $g(n) = \log n + 5$ **$f(n) = \Theta\ (g(n))$**
  - $f(n) = n$; $g(n) = \log n^2$ **$f(n) = \Omega(g(n))$**
  - $f(n) = \log \log n$; $g(n) = \log n$ **$f(n) = O(g(n))$**
  - $f(n) = n$; $g(n) = \log^2 n$ **$f(n) = \Omega(g(n))$**
  - $f(n) = n \log n + n$; $g(n) = \log n$ **$f(n) = \Omega(g(n))$**
  - $f(n) = 10$; $g(n) = \log 10$ **$f(n) = \Theta(g(n))$**
  - $f(n) = 2^n$; $g(n) = 10n^2$ **$f(n) = \Omega(g(n))$**
  - $f(n) = 2^n$; $g(n) = 3^n$ **$f(n) = O(g(n))$**

Properties

- *Theorem:*

$$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Asymptotic Notations in Equations

- On the right-hand side
  - $\Theta(n^2)$ stands for some anonymous function in $\Theta(n^2)$

  $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means:

  There exists a function $f(n) \in \Theta(n)$ such that

  $$2n^2 + 3n + 1 = 2n^2 + f(n)$$

- On the left-hand side

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

  No matter how the anonymous function is chosen on the left-hand side, there is a way to choose the anonymous function on the right-hand side to make the equation valid.

Common Summations

- Arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- Geometric series:

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

  - Special case: $|x| < 1$:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

- Harmonic series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- Other important formulas:

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Mathematical Induction

- A powerful, rigorous technique for proving that a statement S(*n*) is true for *every* natural number *n*, no matter how large.

- Proof:
  - **Basis step**: prove that the statement is true for n = 1
  - **Inductive step:** assume that S(n) is true and prove that S(n+1) is true for all n ≥ 1

- Find case n "within" case n+1

Example

- Prove that: $2n + 1 \leq 2^n$ for all $n \geq 3$
- **Basis step:**
  - $n = 3$: $2 * 3 + 1 \leq 2^3 \Leftrightarrow 7 \leq 8$ TRUE
- **Inductive step:**
  - Assume inequality is true for n, and prove it for (n+1):

$2n + 1 \leq 2^n$ must prove: $2(n + 1) + 1 \leq 2^{n+1}$

$2(n + 1) + 1 = (2n + 1) + 2 \leq 2^n + 2 \leq$

$\leq 2^n + 2^n = 2^{n+1}$, since $2 \leq 2^n$ for $n \geq 1$

HW

- HW#1.
  - Exercise
    - Sec. 1.1 – 5
    - Sec. 1.4 – 15
    - Additional exercise – 26, 27, 30
  - Due Date : 2020. 3. 28.