EECS 114:
Engineering Data Structures and Algorithms
Lecture 2

Instructor: Ryan Rusich
E-mail: rusichr@uci.edu
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

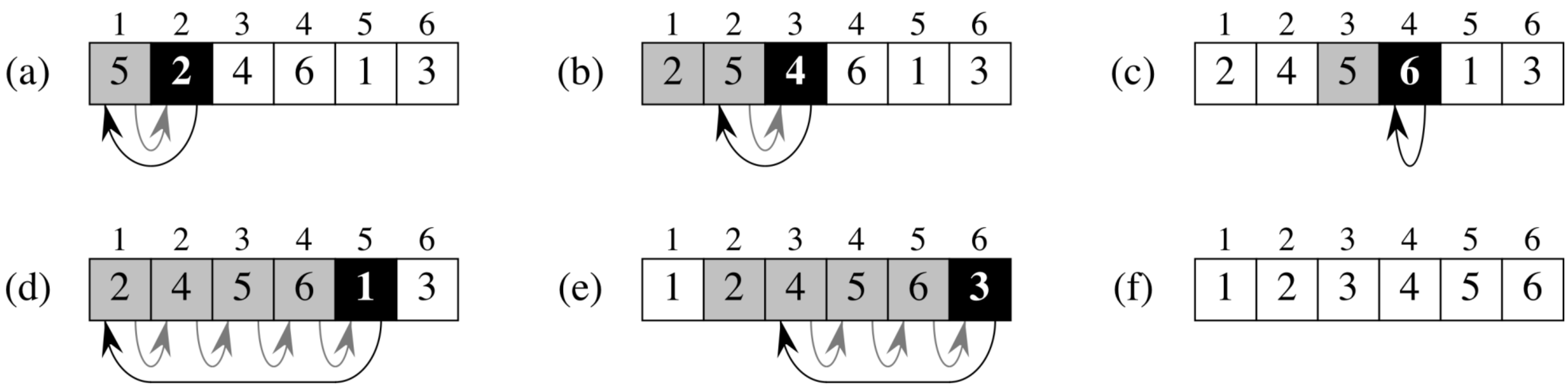
Sorting Problem

- **Input:** A sequence of *n* numbers $\{a_1, a_2, \ldots, a_n\}$
- **Output:** A permutation (reordering) $\{a'_1, a'_2, \ldots, a'_n\}$ of the input sequence such that $a'_I \leq a'_2 \leq \ldots \leq a'_n$

- **Problem statement** specifies in general terms desired input/output relationship.
- An **algorithm** is a tool for solving a well-specified **computational problem**.
- Can be several ways to solve particular problem

Insertion Sort

Example



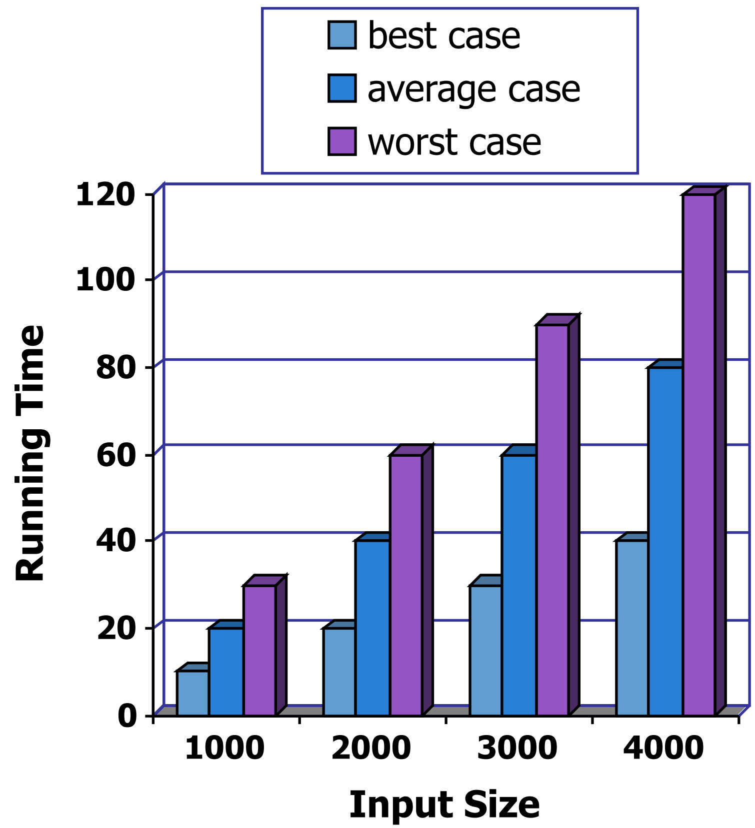
Insertion-Sort Pseudocode

INSERTION-SORT(A, n)

```
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

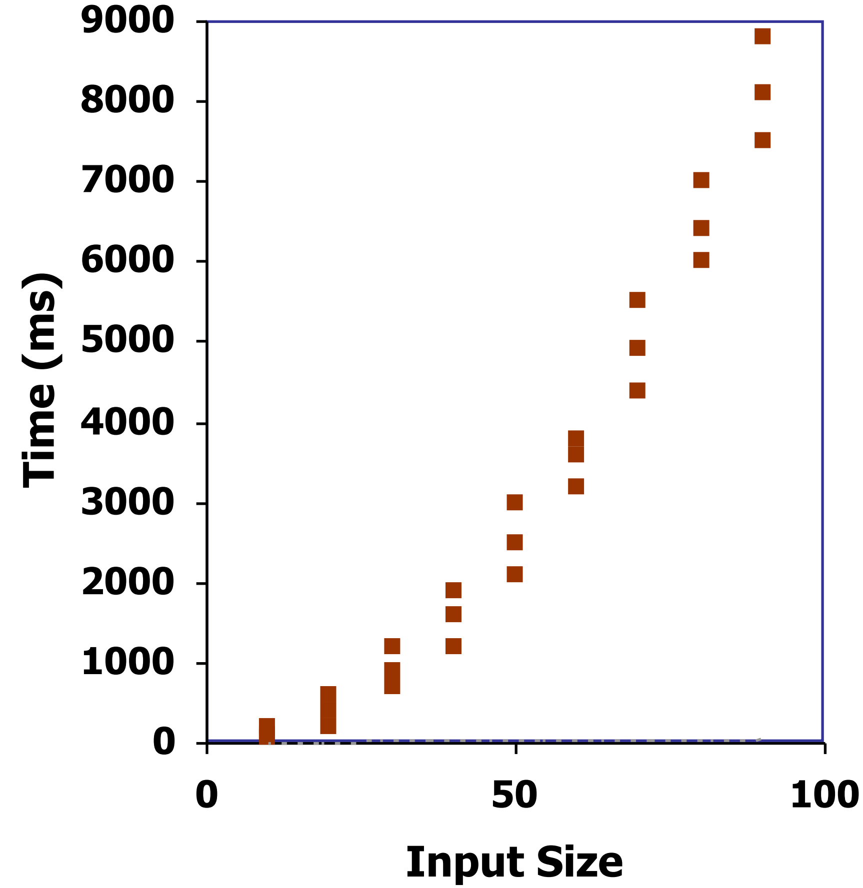
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
    - Easier to analyze
    - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Measure the runtime using *time*
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Good range of inputs?
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
  - Look at large input sizes
- Takes into account all possible inputs
- Evaluates algorithm independent of hardware, implementation, input set, etc.
- Count operations not actual clock time

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm ***printArray***(*A*, ***n***)
    i ← 0
    while ***i*** < n **do**
        cout << A[i] << endl
        i ++

1 assignment
n + 1 comparisons
n outputs
n increments

1 + (n+1) + n + n = 3n + 2 operations
Proportional to n, more items = more time

Counting Primitive Operations

- In class exercise

Algorithm $foo(n)$

$x \leftarrow 0, y \leftarrow 0$

while $x < \mathrm{n}$ **do**

$x{++}$

while $y < \mathrm{n}$ **do**

$y{++}$

$y = 0$

Estimating Running Time

- Algorithm ***printArray*** executes $3\boldsymbol{n} + 2$ primitive operations in the worst case. Define:

  $\boldsymbol{a}$ = Time taken by the fastest primitive operation

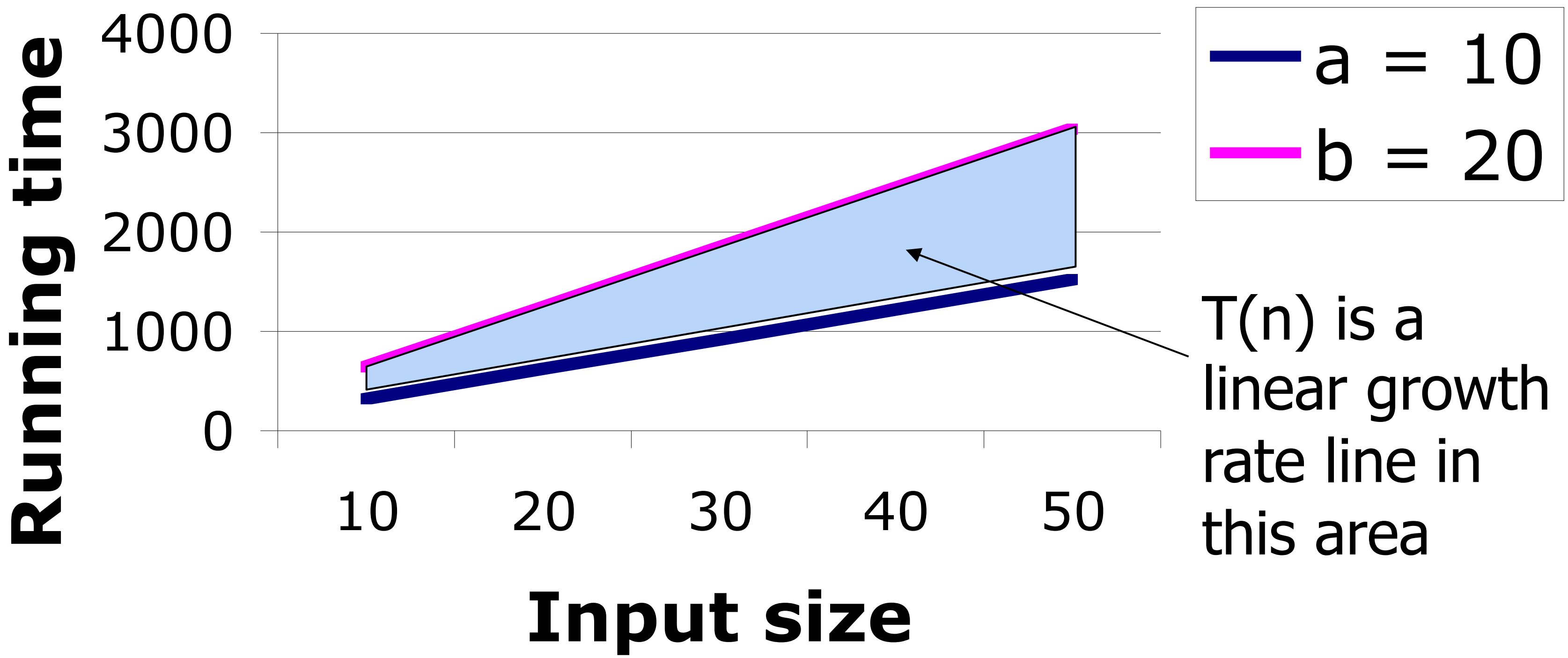
  $\boldsymbol{b}$ = Time taken by the slowest primitive operation

- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***printArray.*** Then

$$\boldsymbol{a}\,(3\boldsymbol{n} + 2) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(3\boldsymbol{n} + 2)$$

- The running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions

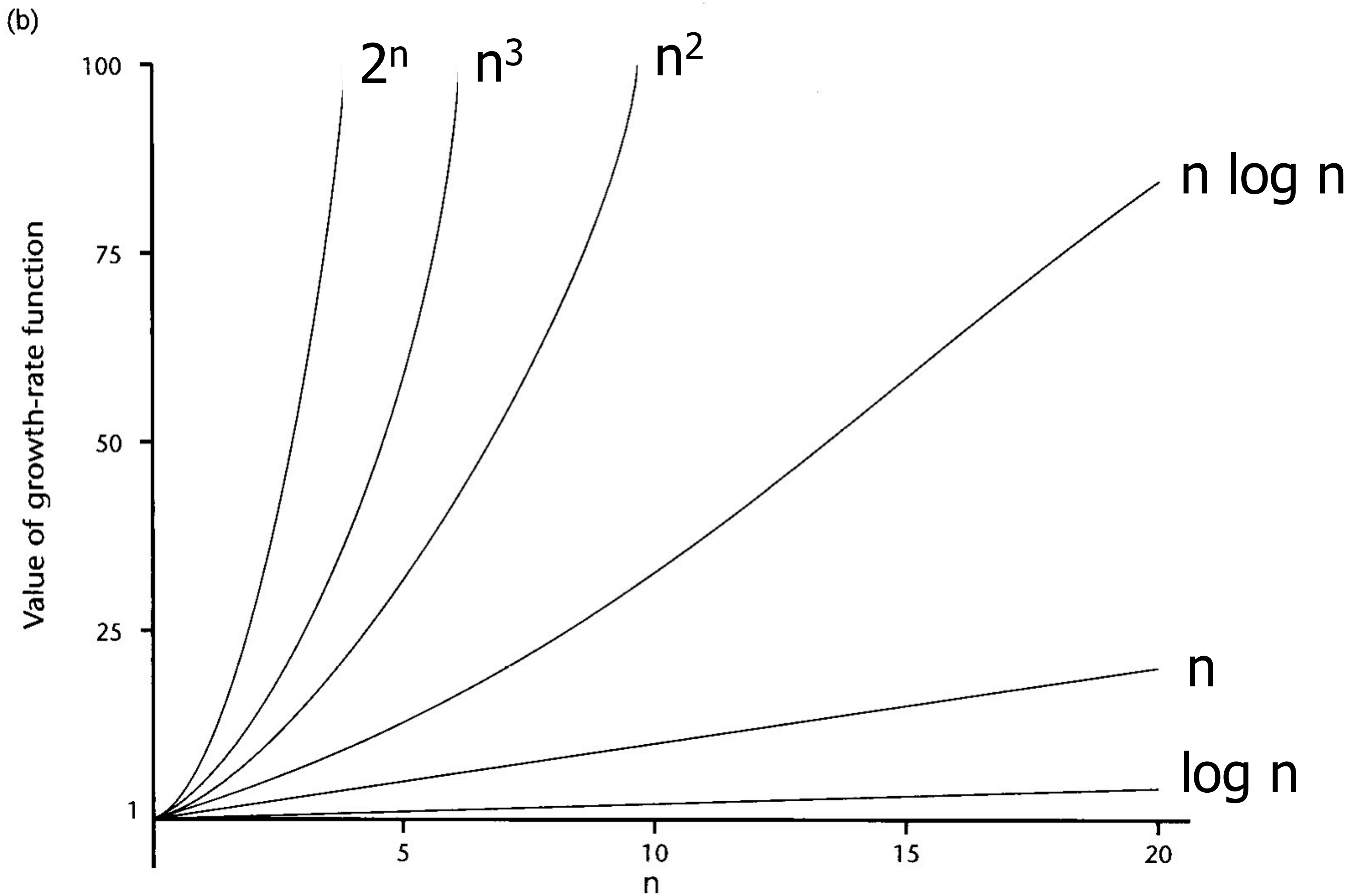
Growth Rate of Running Time



Growth Rates

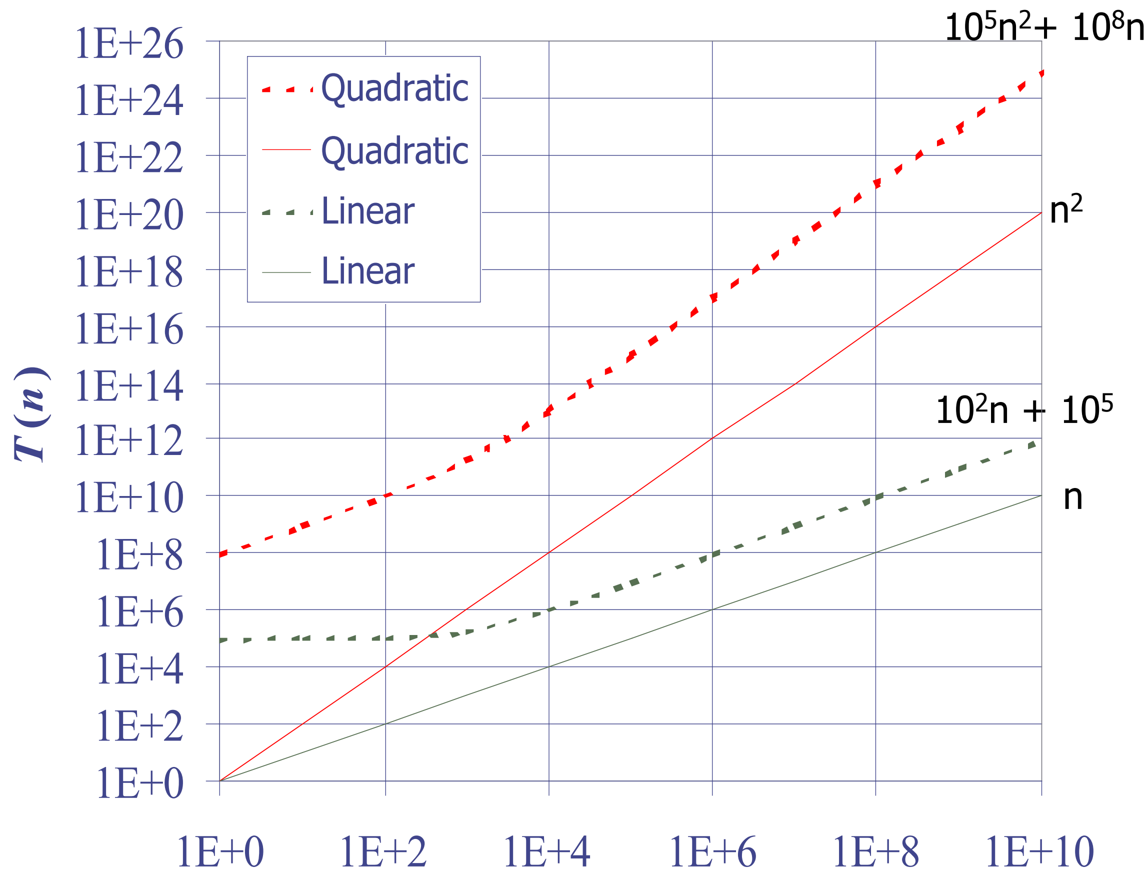
- Growth rates of functions in order of increasing growth rate:
  - Constant ≈ ***1***
  - Logarithmic ≈ ***log n*** *(log base 2)*
  - Linear ≈ ***n***
  - ***n log n*** *(log base 2)*
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\boldsymbol{2}^{\boldsymbol{n}}$

Growth Rates



Constant Factors and Low-Order Terms

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Constant Factors and Low-Order Terms

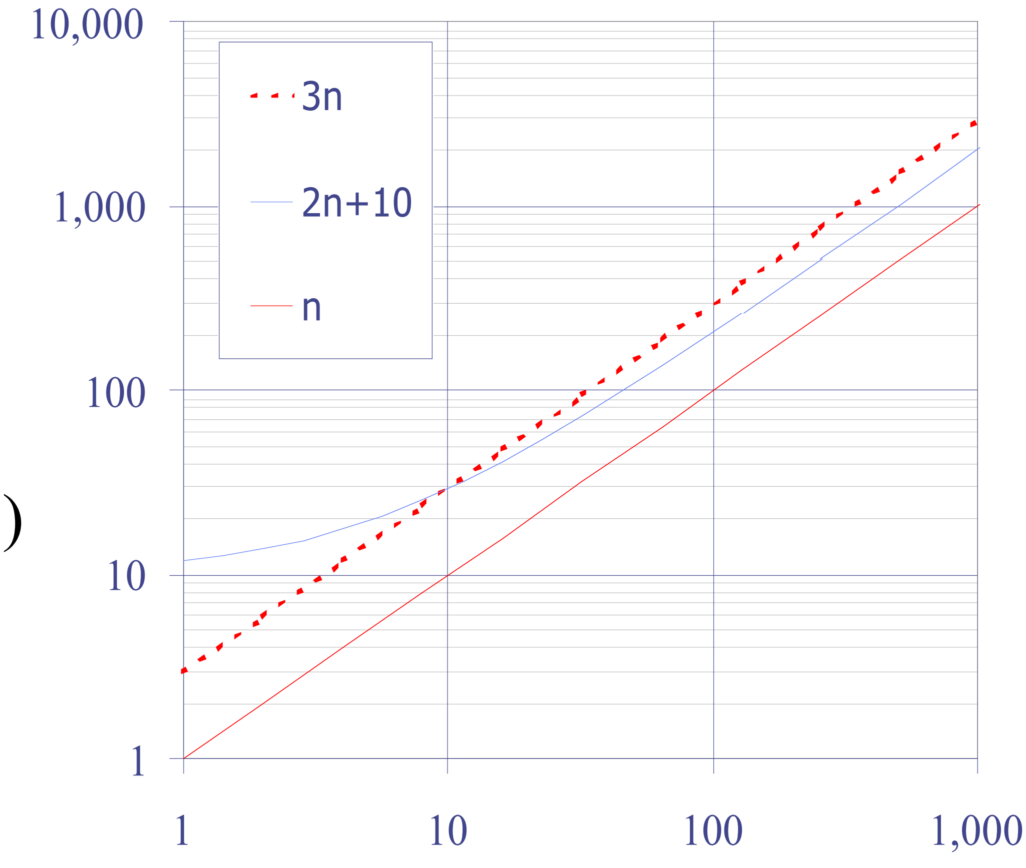
- Examples
  - $2n$ and $100n$ have the same relative growth rates
  - $10n$ and $10n + 4$ have the same relative growth rates
  - $3n^2 + 10n + 7$ and n^2 have the same relative growth rates
  - $10000n + 1000$ and n have the same relative growth rates

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

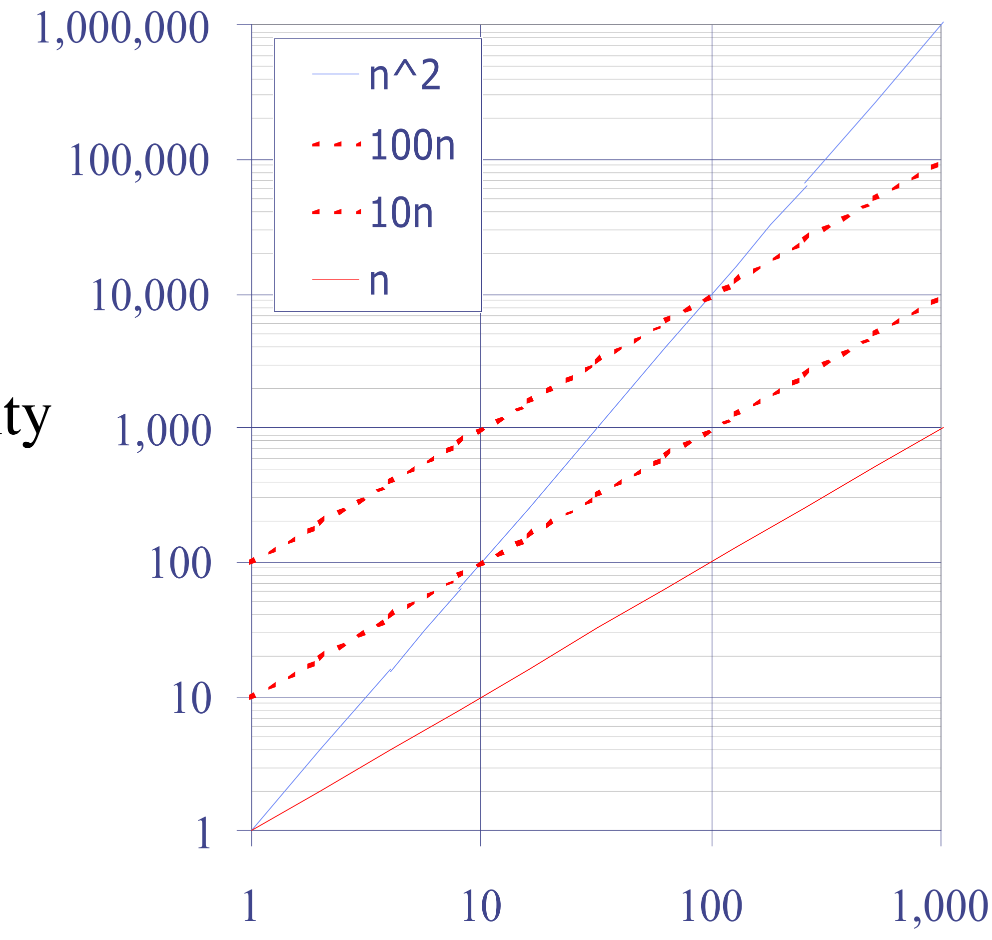
  $$f(n) \le cg(n) \quad \text{for } n \ge n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

$7n-2$

$7n-2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq cn$ for $n \geq n_0$

this is true for $c = 7$ and $n_0 = 1$

$3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 21$

$3 \log n + \log \log n$

$3 \log n + \log \log n$ is $O(\log n)$

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \log n$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 2$

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
  - $f(n) = 4n^4 + n^3 \Rightarrow O(n^4)$
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- You can combine growth rates
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  - $O(n) + O(n^3 + 5) = O(n + n^3 + 5) = O(n^3)$

Rules for Analyzing Running Time

- Loops
  - The running time of a loop is at most the running time of the statements inside the loop times the number of iterations

```
for ( x = 0; x < N; x ++ ) {
        statement 1
        statement 2
        …
        statement c
}
```

N iterations
c statements

$O(cN) = O(N)$

Rules for Analyzing Running Time

- Nested Loops - analyze inside out
  - The running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops

```
for ( x = 0; x < N; x ++ ) {
        for ( y = 0; y < N; y ++ ) {
                statement 1
        }
}
```

N*N iterations
1 statements

$O(N*N) = O(N^2)$

Rules for Analyzing Running Time

- Consecutive statements
  - Sum them - largest one dominates

```
statement 1
statement 2
for ( x = 0; x < N; x ++ ) {
        statement 3
}
```

2 statements

N iterations

O(2+N) = O(N)

Rules for Analyzing Running Time

- if/else statements
  - The running time is never more than the running time of the test plus the larger of the running times of S1 and S2

```
if ( condition )          O(running time of condition)
        S1                              +
else                      max ( O(running time of S1),
        S2                        O(running time of S2) )
```

Analyzing Running Time

- In class exercise - give the O-notation running time of the following code

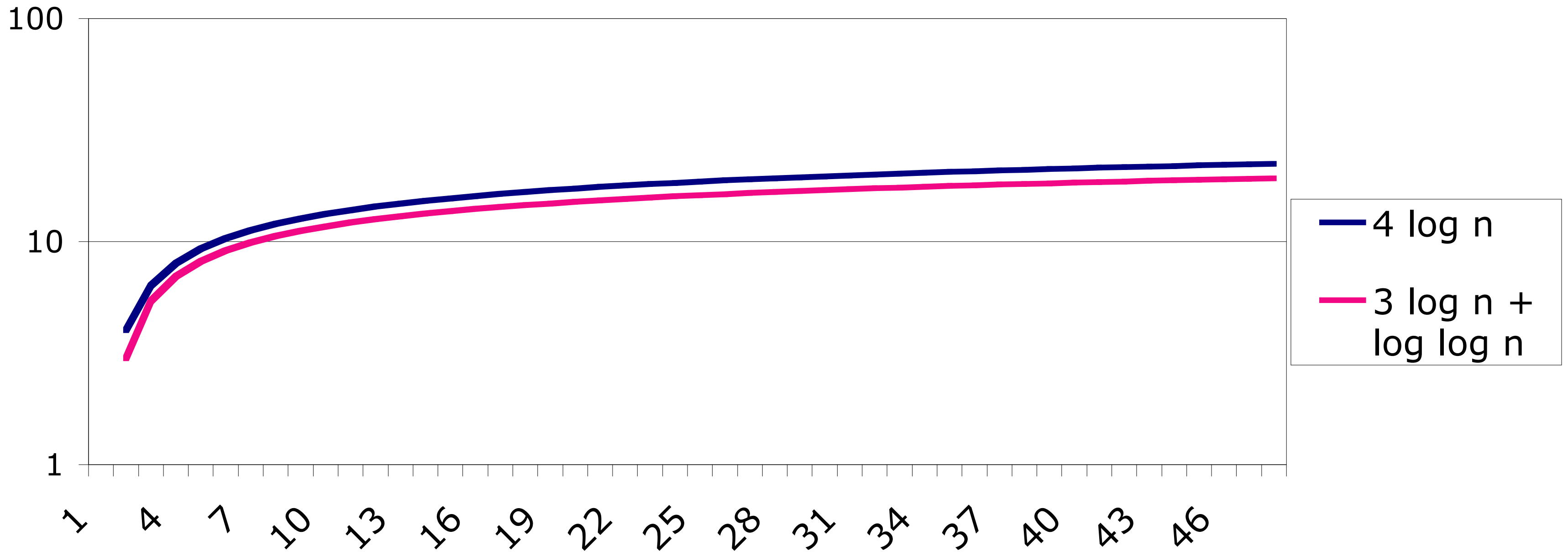
```
for ( x = 0; x < N; x ++ )
        array[x] = x*N;

for (x = 0; x < N; x ++ ) {
        if ( x < (N/2) )
                cout << array[x];
        else
                for (  y = 0; y < N; y ++ )
                        cout << y*array[x];
}
```

Calculating O-notation

- In class exercise - Give the O-notation for the following functions:
  - $n + \log(n) =$
  - $8n \log(n) + n^2 =$
  - $6n^2 + 2^n + 300 =$
  - $n + n \log(n) + \log(n) =$
  - $40 + 8n + n^7 =$

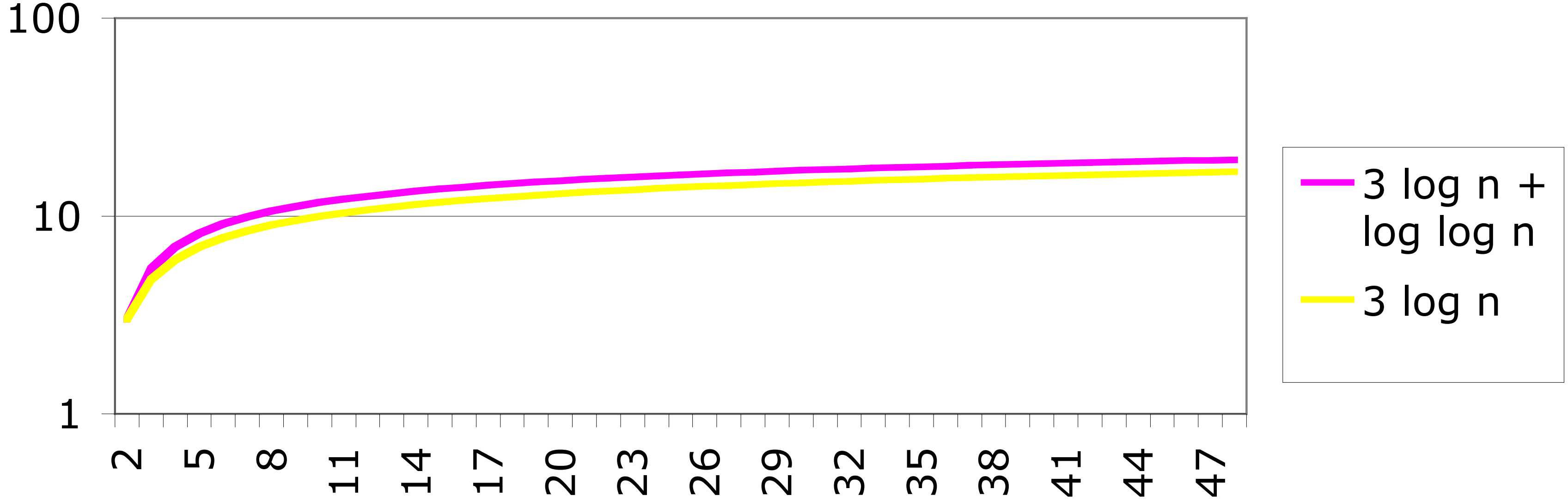
O-notation

- **big-Oh**
  - f(n) is O(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$
  - f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)
  - Example: ***3 log n + log log n = O(log n)*** for c = 4 and n ≥ 2



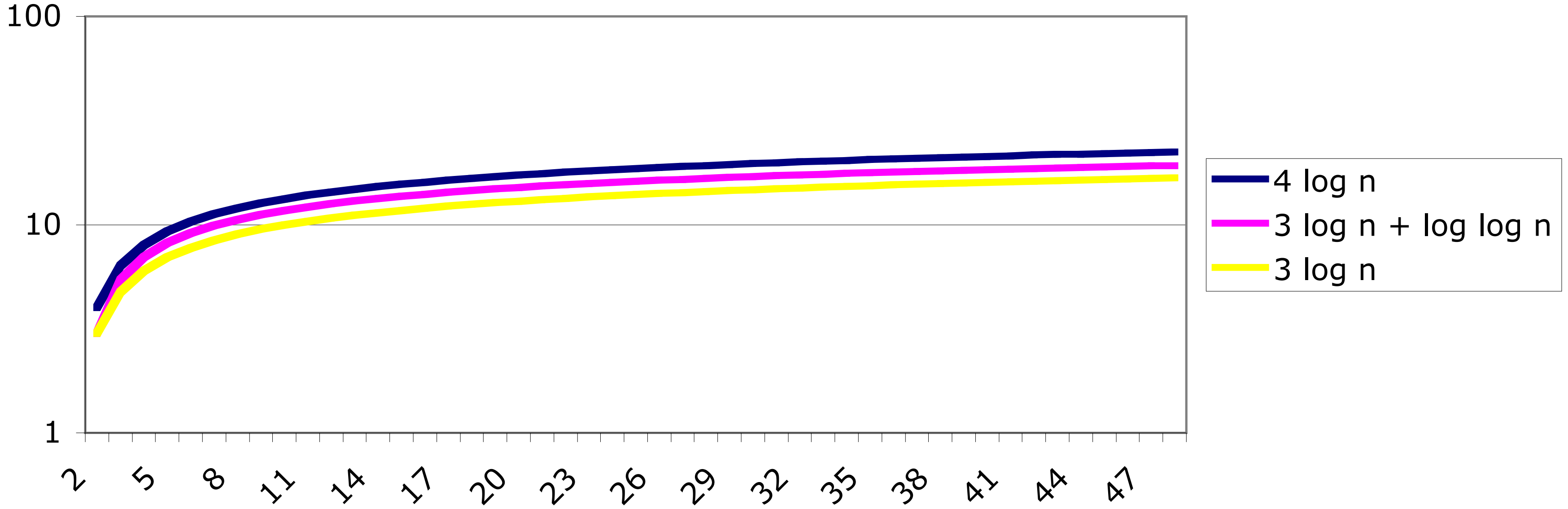
Ω-notation

- **big-Omega**
  - f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$
  - f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)
  - Example: ***3 log n + log log n = Ω(log n)*** for c = 3 and n ≥ 2

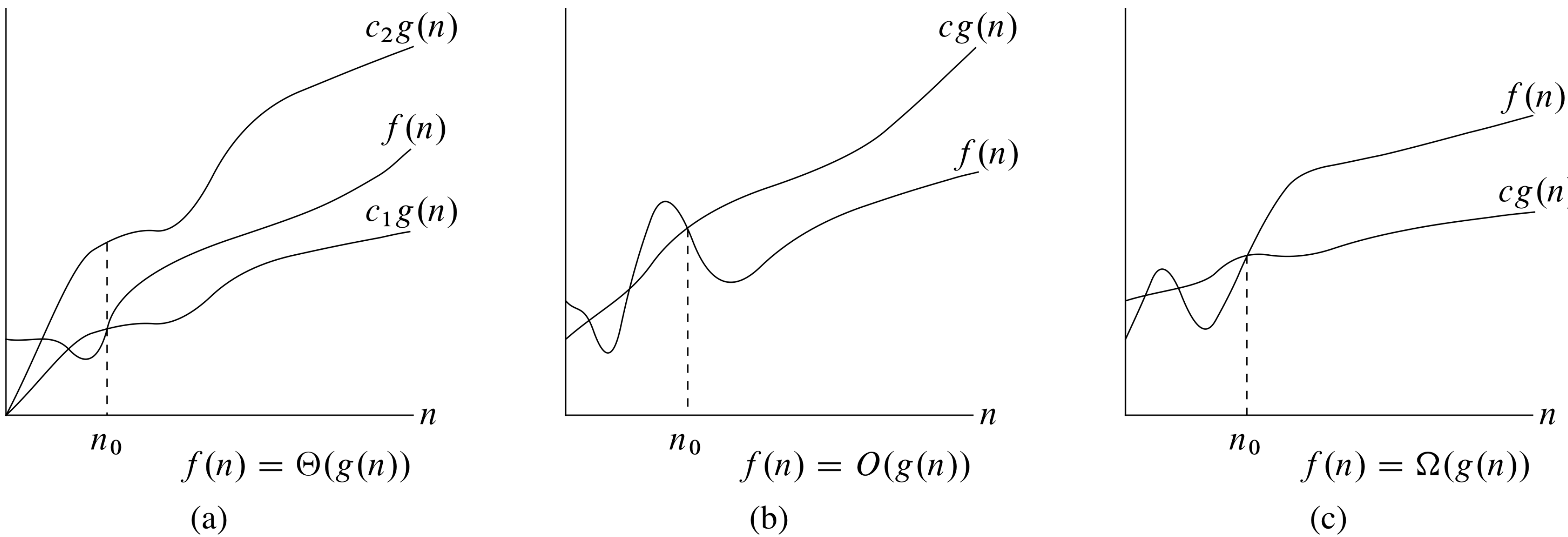


Θ-notation

- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c'>0$ and $c''>0$ and an integer constant $n_0 \geq 1$ such that $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$
  - f(n) is $\Theta(g(n))$ if f(n) is asymptotically **equal** to g(n)
  - Example: ***3 log n + log log n = Θ log n)*** for c'=3 and c''=4 and $n \geq 2$



Θ, O, and Ω Notations



Insertion-Sort Pseudocode Analysis

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j - 1]$.	0	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i + 1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i + 1] = key$	c_8	$n - 1$

Other Notations - Review

- **little-oh**
  - f(n) is o(g(n)) if, **for any** constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) < cg(n)$ for $n \geq n_0$

  f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

- **little-omega**
  - f(n) is ω(g(n)) if, **for any** constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) > cg(n)$ for $n \geq n_0$

  f(n) is ω(g(n)) if is asymptotically **strictly greater** than g(n)

Array Operation Running Times

- Unsorted insert
  - O(1) - add to end
- Sorted insert
  - O(N) - shift items
- Number of items
  - O(1) - have to keep counter
- Sorted Remove
  - O(N) - shift items
- Unsorted Remove
  - O(1) - move last
- Linear search
  - O(N)

Space Complexity

- Similar to determining Big-Oh
- Give upper bound on space required based on the input size
  - Constant factors and low-order terms are not significant