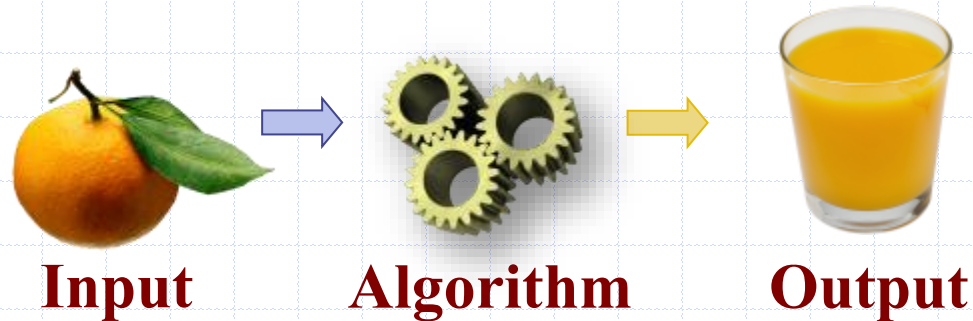


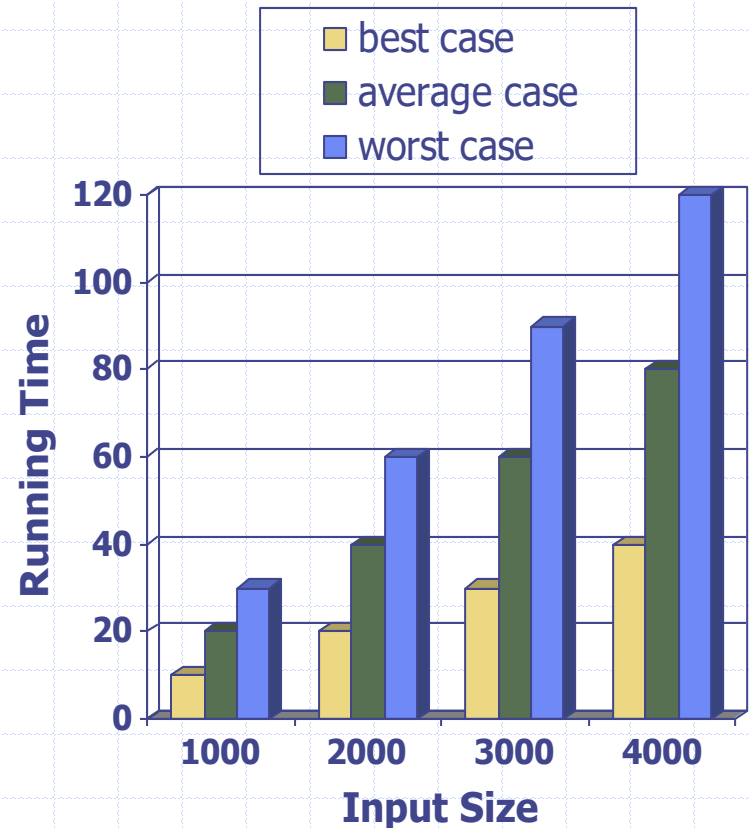
Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Analysis of Algorithms



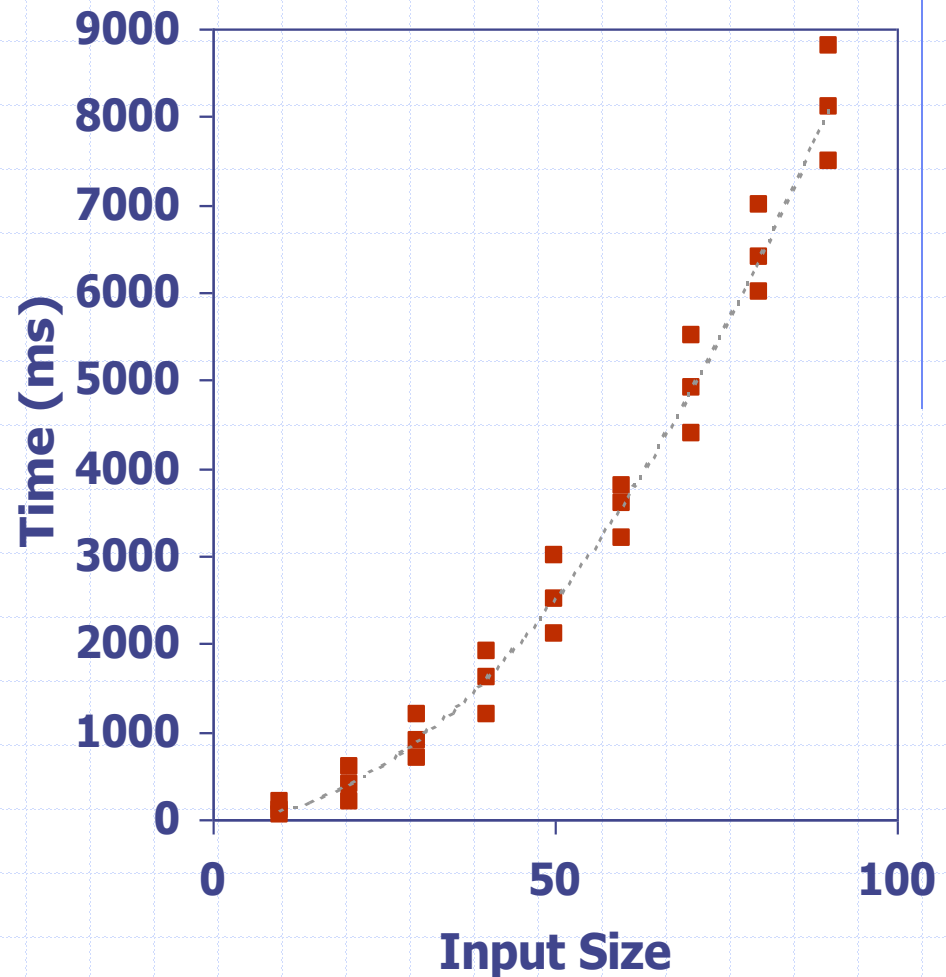
Running Time

- ❑ 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, noting the time needed:
- Plot the results



```
1 long startTime = System.currentTimeMillis();
2 /* (run the algorithm) */
3 long endTime = System.currentTimeMillis();
4 long elapsed = endTime - startTime;
```

```
// record the starting time
```

```
// record the ending time
```

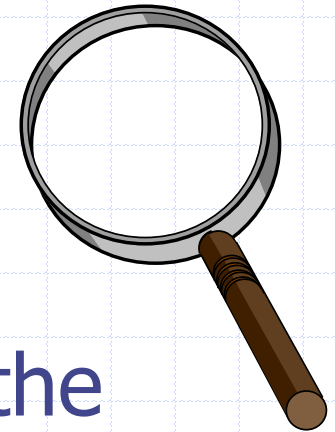
```
// compute the elapsed time
```

Limitations of Experiments

- ❑ It is necessary to implement the algorithm, which may be difficult
- ❑ Results may not be indicative of the running time on other inputs not included in the experiment.
- ❑ In order to compare two algorithms, the same hardware and software environments must be used



Theoretical Analysis



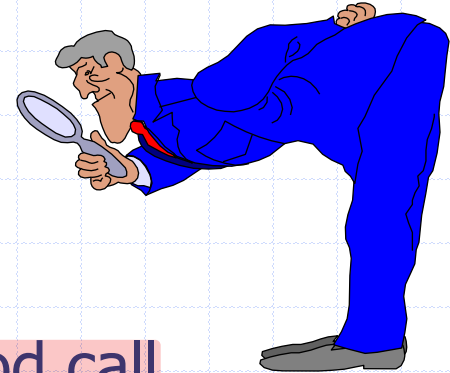
- ❑ Uses a high-level description of the algorithm instead of an implementation
- ❑ Characterizes running time as a function of the input size, n
- ❑ Takes into account all possible inputs
- ❑ Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

not high level (algorithm)

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Pseudocode Details



□ Control flow

- **if ... then ... [else ...]**
- **while ... do ...**
- **repeat ... until ...**
- **for ... do ...**
- Indentation replaces braces

□ Method declaration

Algorithm *method* (*arg* [, *arg*...])

Input ...

Output ...

□ Method call

method (*arg* [, *arg*...])

□ Return value

return *expression*

□ Expressions:

← Assignment

= Equality testing

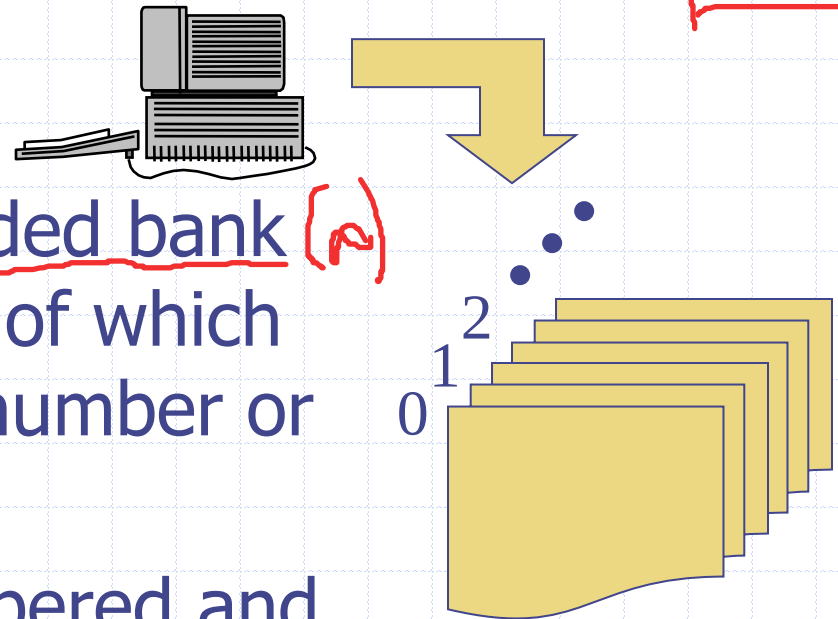
n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

how i can write in RAM

A RAM consists of

- A CPU
- An potentially unbounded bank (n) of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time nat

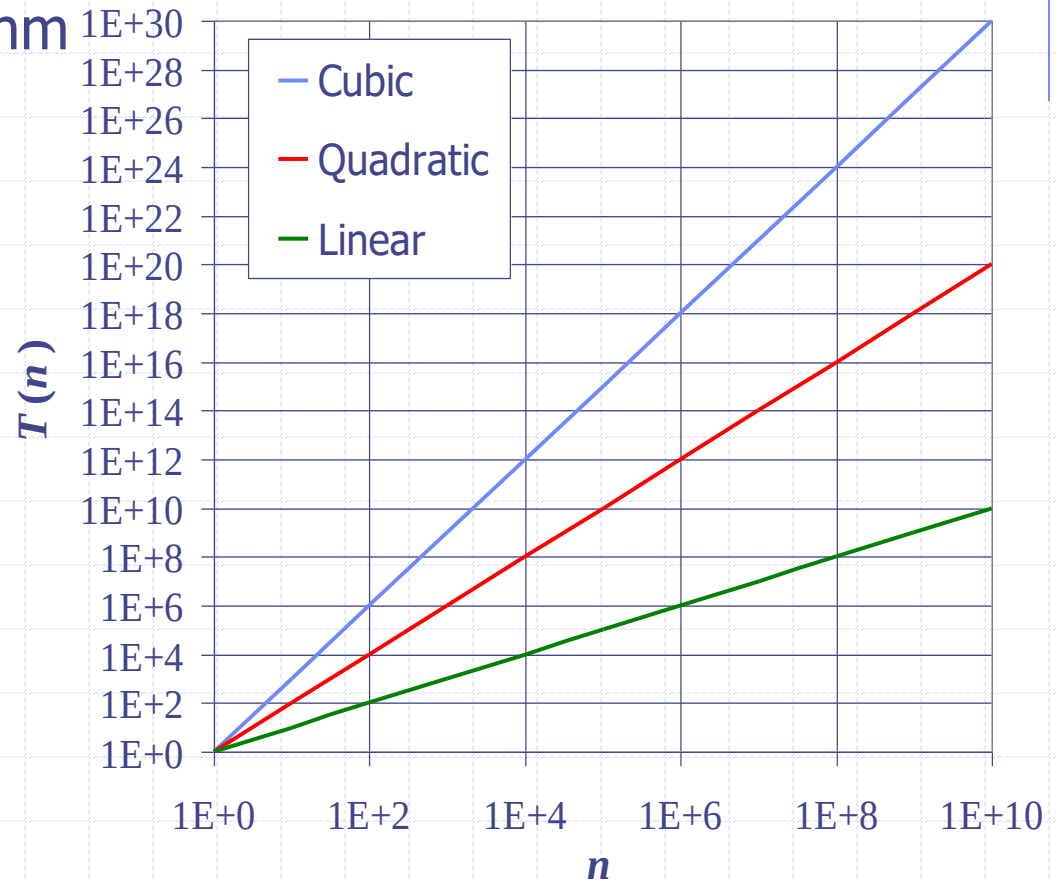


Seven Important Functions

Seven functions that often appear in algorithm analysis:

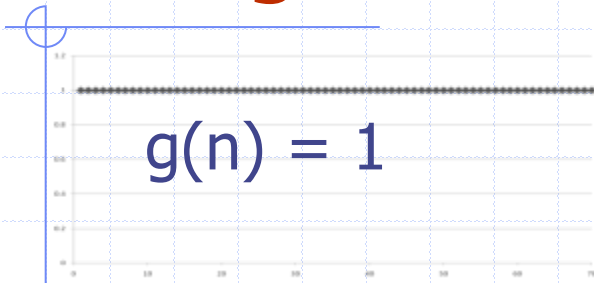
- Constant ≈ 1
- Logarithmic $\approx \log n$
- Linear $\approx n$
- N-Log-N $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Exponential $\approx 2^n$

□ In a log-log chart, the slope of the line corresponds to the growth rate

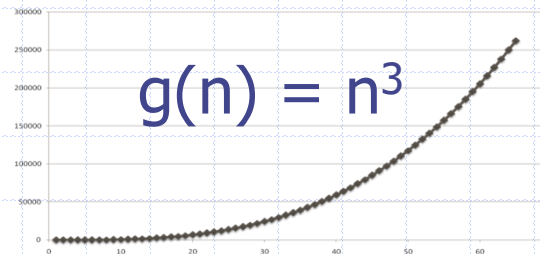
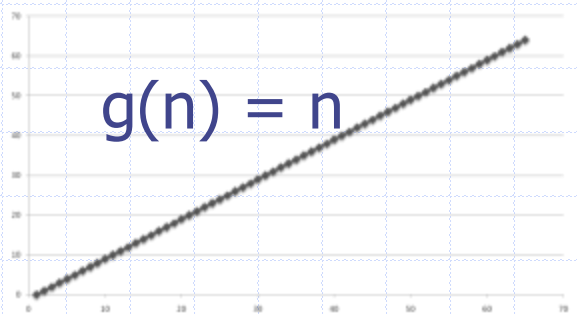
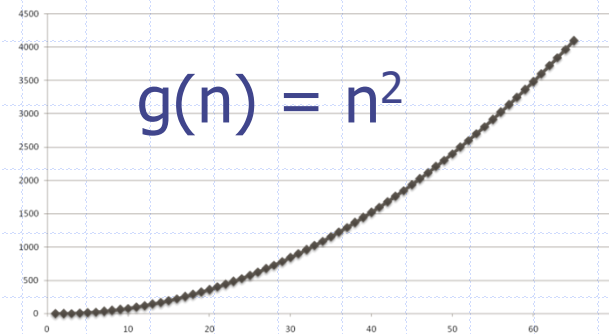
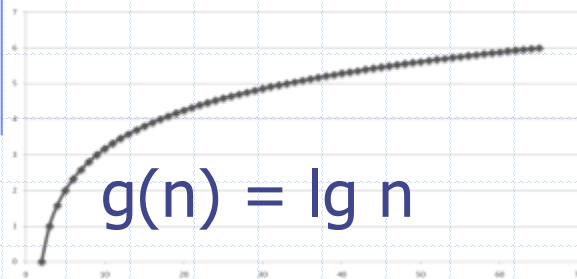
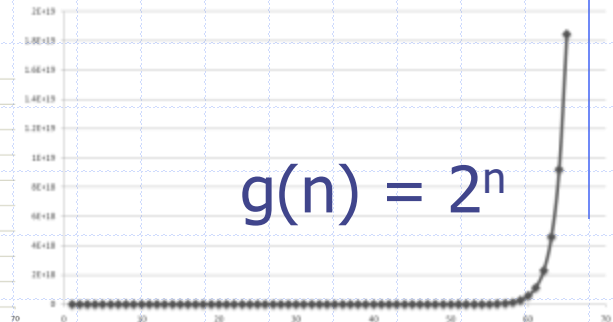
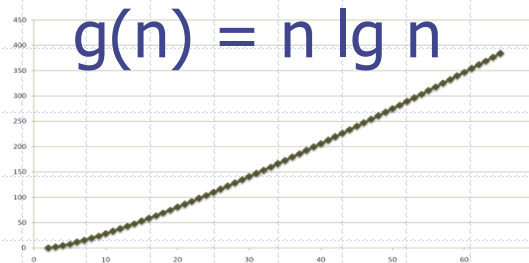


Functions Graphed Using “Normal” Scale

Slide by Matt Stallmann included with permission.



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



Primitive Operations



- Basic computations performed by an algorithm
 - Identifiable in pseudocode
 - Largely independent from the programming language
 - Exact definition not important (we will see why later)
 - Assumed to take a constant amount of time in the RAM model
- Examples:
 - Evaluating an expression
 - Assigning a value to a variable
 - Indexing into an array
 - Calling a method
 - Returning from a method

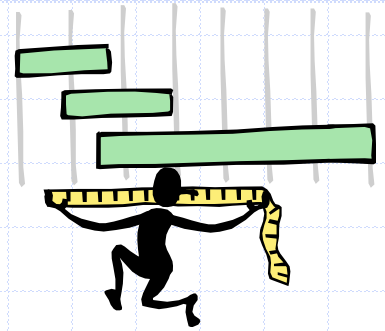
Counting Primitive Operations

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

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[] data) {
3      int n = data.length;      |+|=2
4      double currentMax = data[0]; 2          // assume first entry is biggest (for now)
5      for (int j=1; j < n; j++) 1 + 2(n-1)    // consider all other entries
6          if (data[j] > currentMax) 2(n-1)    // if data[j] is biggest thus far...
7              currentMax = data[j]; 0 → 2(n-1) // record it as the current max
8      return currentMax; 1
9  }
```

- Step 3: 2 ops, 4: 2 ops, 5: $2(n-1)+1$ ops, 6: $2(n-1)$ ops, 7: 0 to $2(n-1)$ ops, 8: 1 op

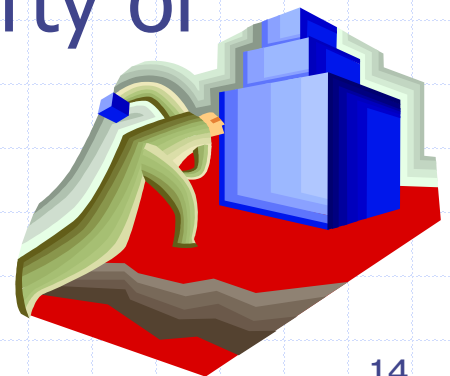
Estimating Running Time



- Algorithm **arrayMax** executes $6n$ primitive operations in the worst case, $4n + 2$ in the best case. Define:
 - a = Time taken by the fastest primitive operation
 - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of **arrayMax**. Then
$$a(4n + 2) \leq T(n) \leq b(6n)$$
- Hence, the running time $T(n)$ is bounded by two linear functions $\implies$ linear

Growth Rate of Running Time

- Changing the hardware/ software environment
 - 1) ■ Affects $T(n)$ by a constant factor, but
 - 2) ■ Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax

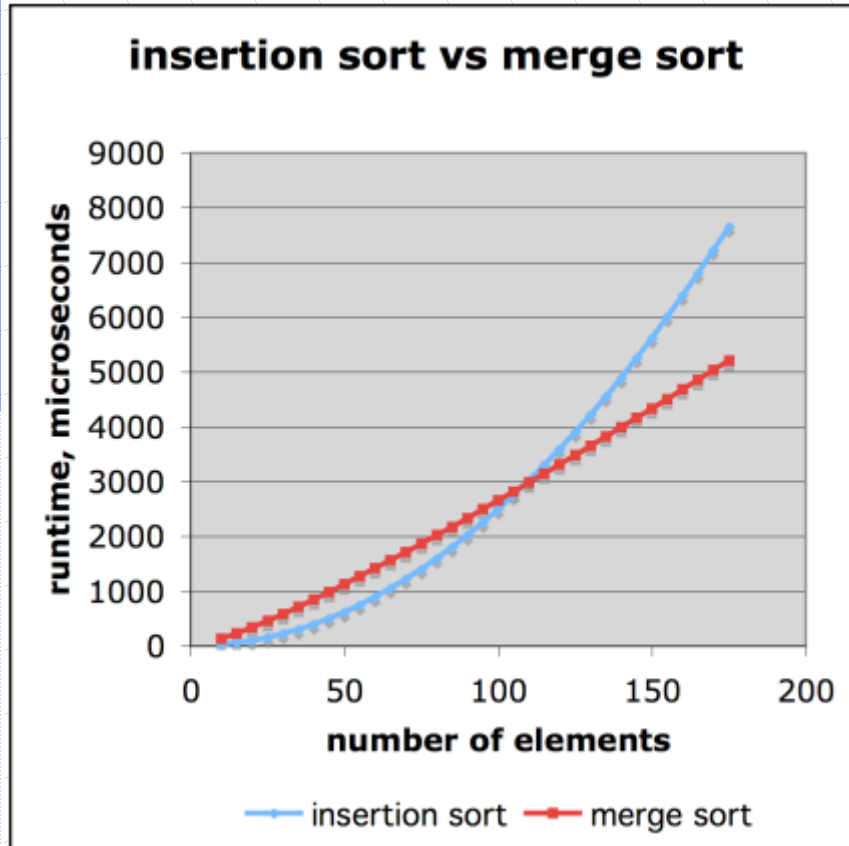


Why Growth Rate Matters

if runtime is...	time for $n + 1$	time for $2n$	time for $4n$
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c n$	$c (n + 1)$	$2c n$	$4c n$
$c n \lg n$	$\sim c n \lg n + c n$	$2c n \lg n + 2c n$	$4c n \lg n + 4c n$
$c n^2$	$\sim c n^2 + 2c n$	$4c n^2$	$16c n^2$
$c n^3$	$\sim c n^3 + 3c n^2$	$8c n^3$	$64c n^3$
$c 2^n$	$c 2^{n+1}$	$c 2^{2n}$	$c 2^{4n}$

runtime
quadruples
when
problem
size doubles

Comparison of Two Algorithms



insertion sort is

$$n^2 / 4$$

merge sort is

$$2 n \lg n$$

sort a million items?

insertion sort takes
roughly 70 hours

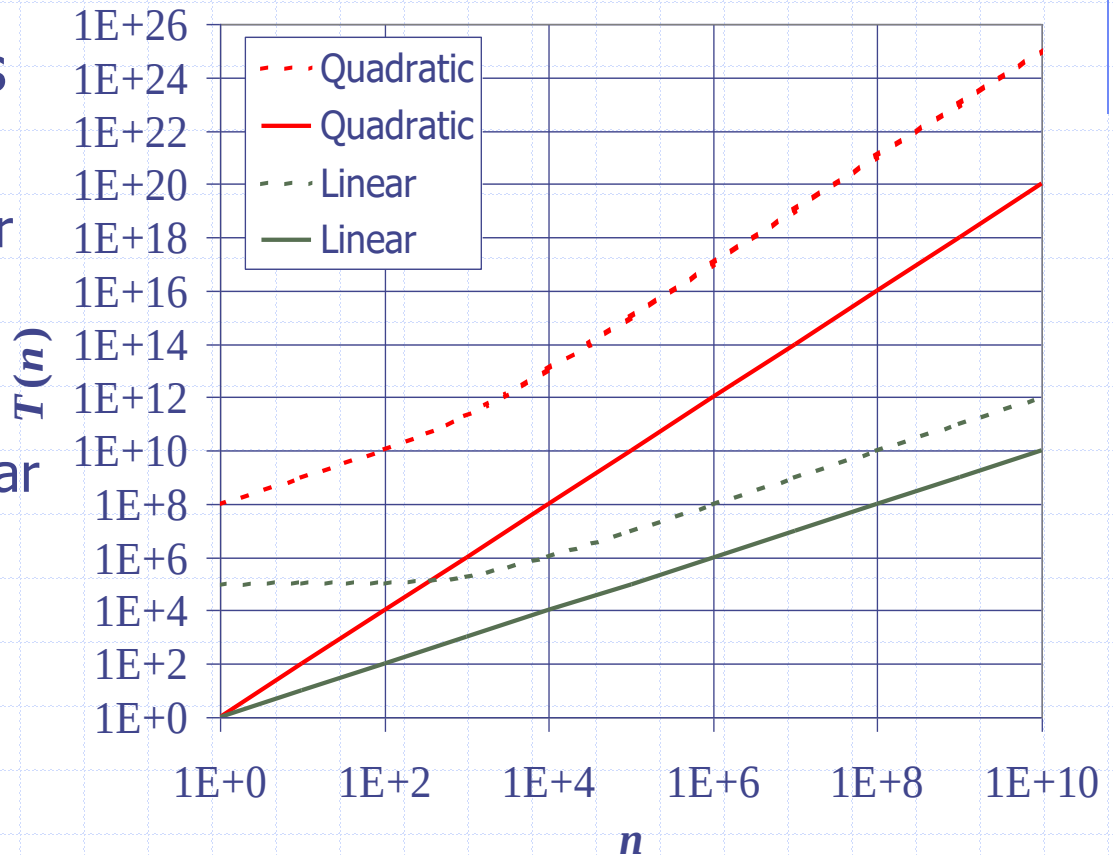
while

merge sort takes
roughly 40 seconds

This is a slow machine, but if
100 x as fast then it's 40 minutes
versus less than 0.5 seconds

Constant Factors $\frac{5n}{5n^2 + 30n} \rightarrow O(n^2)$

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



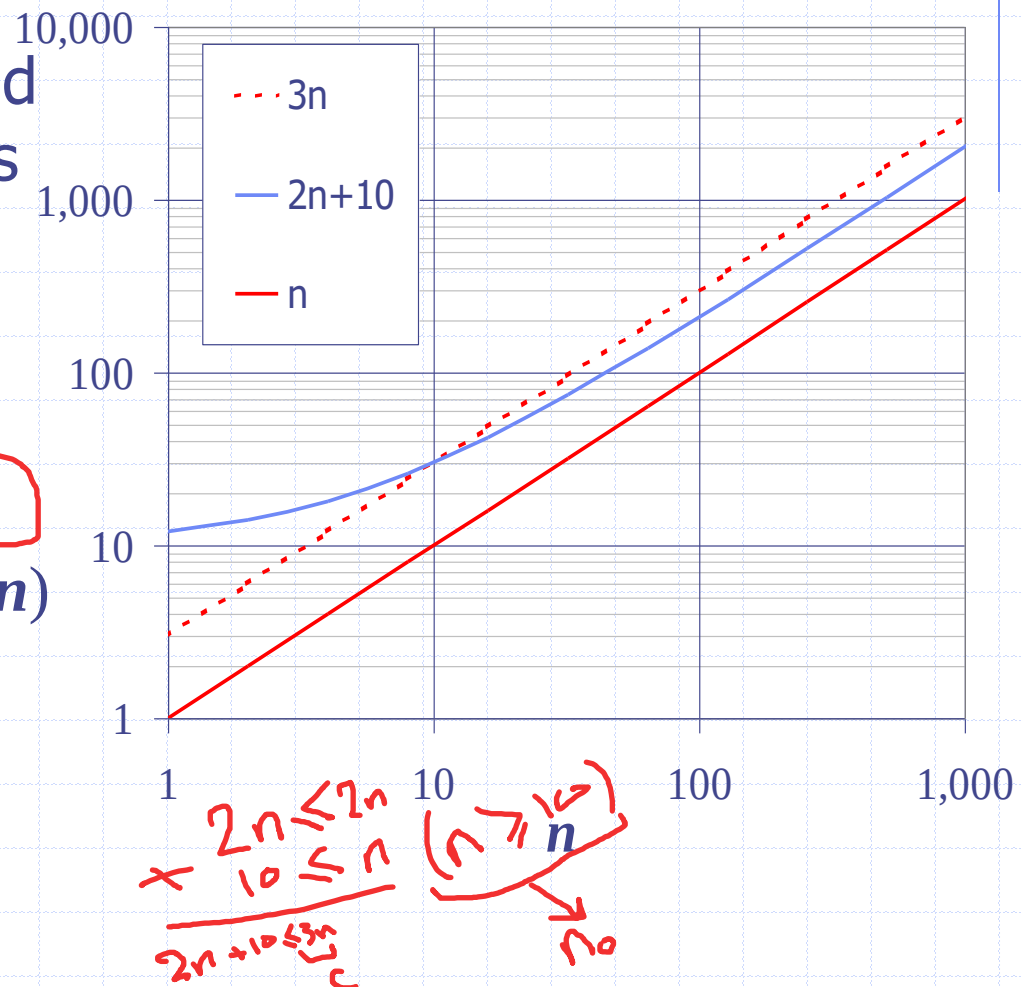
Big-Oh Notation

- 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

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$

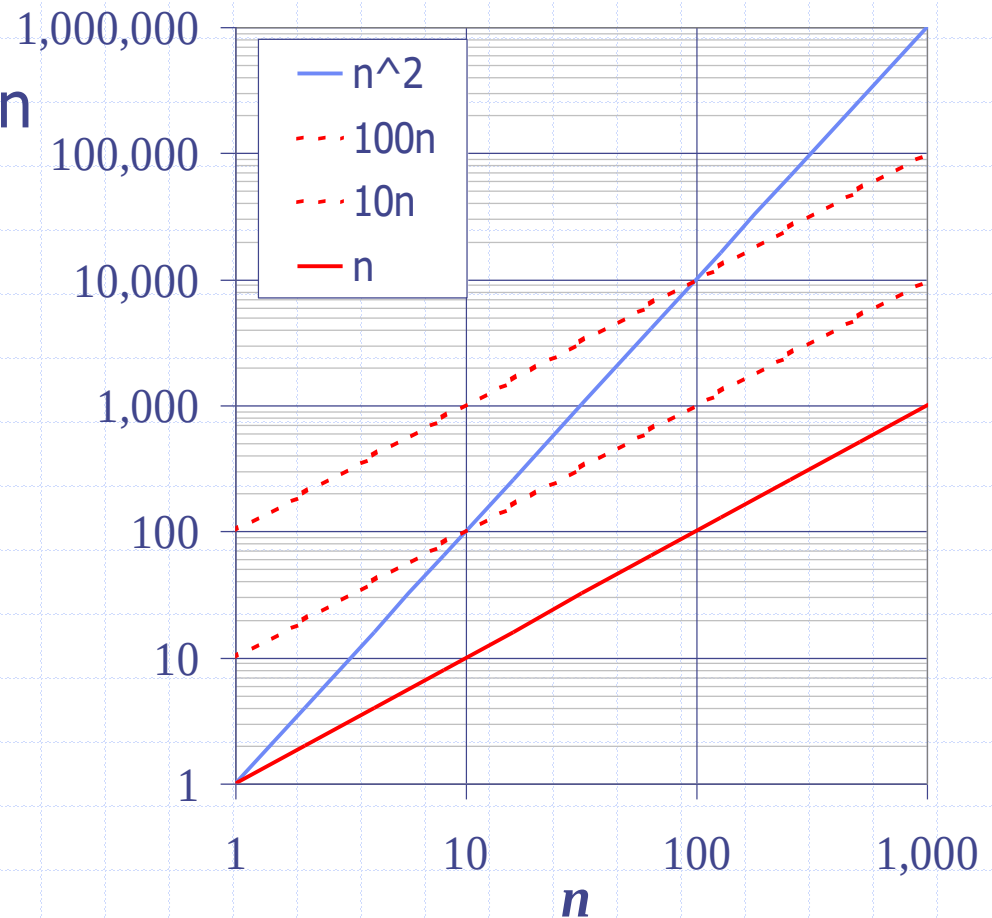
- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$

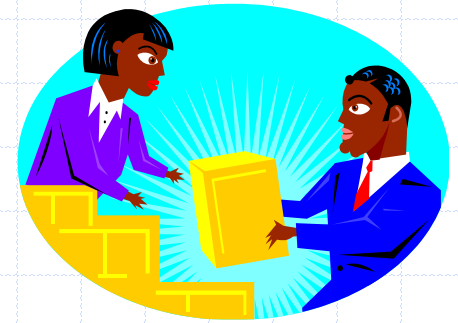


Big-Oh Example

□ Example: the function n^2 is not $O(n)$

- $n^2 \leq cn$
- $n \leq c$
- 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 $n_0 = 1$

□ $3n^3 + 20n^2 + 5$ $c = 26 \Rightarrow$

$3n^3 + 20n^2 + 5$ 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 $n_0 = 21$~~

□ $3 \log n + 5$

$3 \log n + 5$ is $O(\log n)$

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$ $|\log| = 0$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement “ $f(n)$ is $O(g(n))$ ” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

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

$$\begin{aligned} n^4 + n^3 + \dots &= O(n^4) \\ 70n^3 \times 50n^2 &= O(n^5) \end{aligned}$$

- 3) □ Use the smallest possible class of functions

- Say “ $2n$ is $O(n)$ ” instead of “ $2n$ is $O(n^2)$ ”

- Use the simplest expression of the class

- Say “ $3n + 5$ is $O(n)$ ” instead of “ $3n + 5$ is $O(3n)$ ”

Big-Oh Rules [Ext. Material]

Proposition 4.8: *If $f(n)$ is a polynomial of degree d , that is,*

$$f(n) = a_0 + a_1n + \cdots + a_dn^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Justification: Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq \cdots \leq n^d$; hence,

$$a_0 + a_1n + a_2n^2 + \cdots + a_dn^d \leq (|a_0| + |a_1| + |a_2| + \cdots + |a_d|)n^d.$$

Comparative Analysis

[Ext. Material]

We illustrate the growth rates of the seven functions in Table 4.3.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 4.3: Selected values of fundamental functions in algorithm analysis.

Comparative Analysis

[Ext. Material]

We further illustrate the importance of the asymptotic viewpoint in Table 4.4.

Running Time (μ s)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

کون کونسا؟
Task 10⁶ & 4000 n

Table 4.4: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

Some Words of Caution

[Ext. Material]

- The use of the big-Oh and related notations can be somewhat misleading should the constant factors they “hide” be very large.
- For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n \log n$, we should prefer the $O(n \log n)$ -time algorithm, even though the linear-time algorithm is asymptotically faster.
- This preference is because the constant factor, 10^{100} , which is called “one googol,” is believed by many astronomers to be an upper bound on the number of atoms in the
- observable universe.
- So we are unlikely to ever have a real-world problem that has this number as its input size.

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
 - We find the worst-case number of primitive operations executed as a function of the input size
 - We express this function with big-Oh notation
- Example:
 - We say that algorithm arrayMax “runs in $O(n)$ time”
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

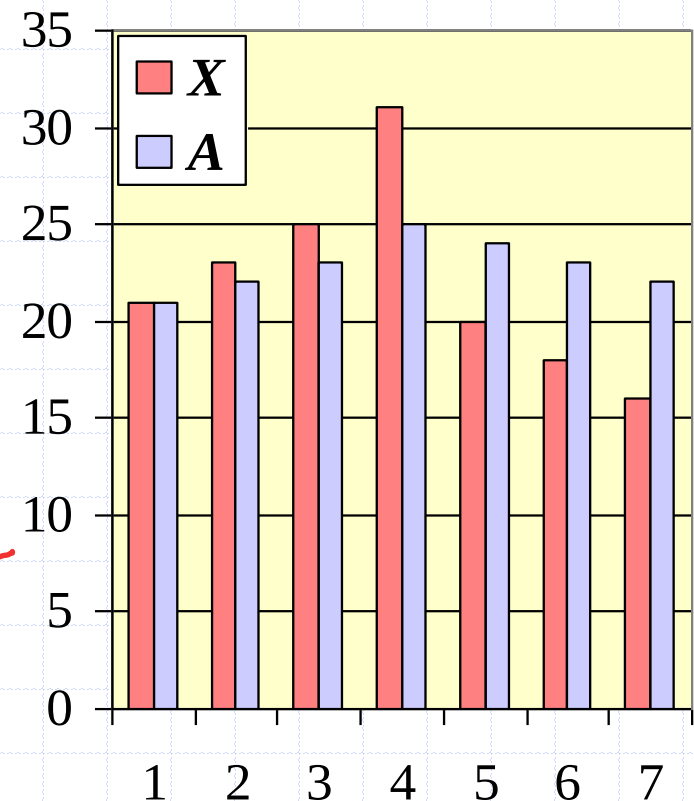
ایہو تخمینہ کرنا ہے

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages زي قصة البورصة
- The i -th prefix average of an array X is average of the first $(i + 1)$ elements of X :

$$A[i] = (X[0] + X[1] + \dots + X[i]) / (i + 1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition

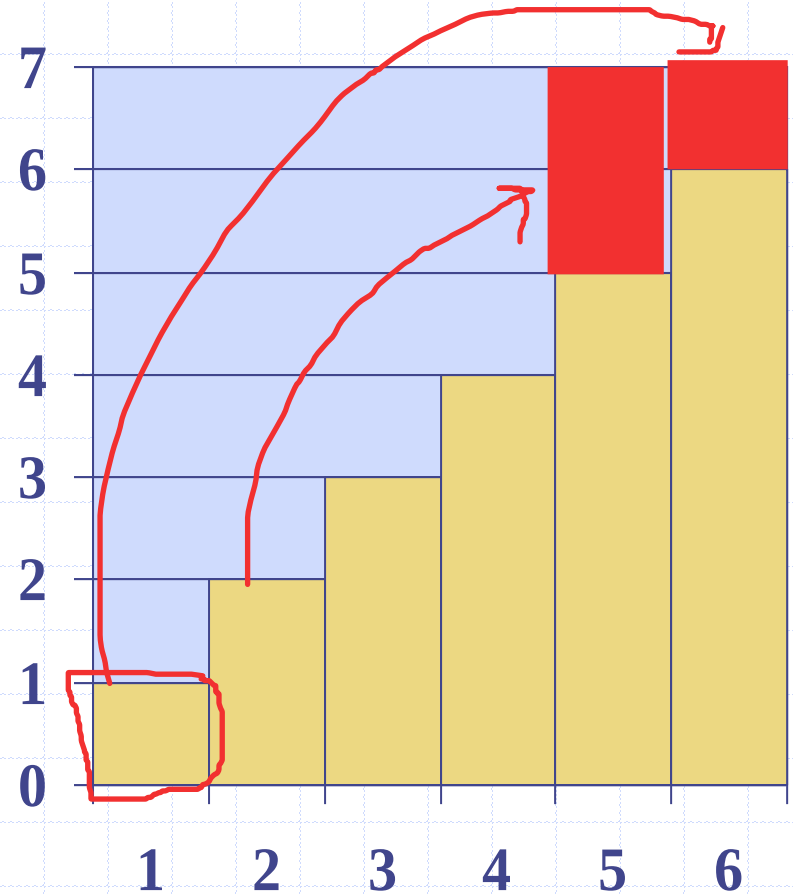
```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[] prefixAverage1(double[] x) {
3      int n = x.length;
4      double[] a = new double[n]; // filled with zeros by default
5      for (int j=0; j < n; j++) {
6          double total = 0; // begin computing x[0] + ... + x[j]
7          for (int i=0; i <= j; i++) {
8              total += x[i];
9              a[j] = total / (j+1);
10         }
11     return a;
12 }
```

Handwritten annotations:

- Red numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 are written next to the corresponding line numbers.
- Red annotations for the inner loop: $1 + \sum_{i=0}^j 1$ (circled) and $2 \times \left(\frac{n(n+1)}{2}\right)$ (circled).
- Red arrow pointing from the inner loop to the outer loop.
- Red arrow pointing from the inner loop to the text "the first for".
- Red text "بتكرري زي ال" (I repeat like this) is written next to the inner loop.
- Red text "2n" is written next to the inner loop.
- Red text "2n" is written next to the inner loop.
- Red text "2n" is written next to the inner loop.

Arithmetic Progression

- The running time of `prefixAverage1` is $O(1 + 2 + \dots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
 - There is a simple visual proof of this fact
- Thus, algorithm `prefixAverage1` runs in $O(n^2)$ time



Prefix Averages 2 (Linear)

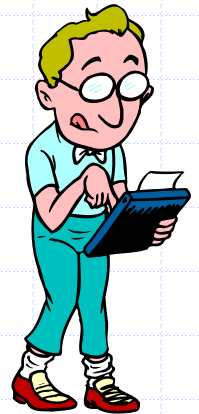
The following algorithm uses a running summation to improve the efficiency

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[] prefixAverage2(double[] x) {
3      int n = x.length; 2
4      double[] a = new double[n]; 2 // filled with zeros by default
5      double total = 0; 1 // compute prefix sum as x[0] + x[1] + ...
6      for (int j=0; j < n; j++) { 1 + 2n
7          total += x[j]; 2n 2n // update prefix sum to include x[j]
8          a[j] = total / (j+1); 2n 2n // compute average based on current sum
9      }
10     return a; 1
11 }
```

Handwritten notes: $6n+7$ (boxed)

Algorithm **prefixAverage2** runs in $O(n)$ time!

Math you need to Review



- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

□ Properties of powers:

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c \cdot \log_a b}$$

□ Properties of logarithms:

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x a = a \log_b x$$

$$\log_b a = \log_x a / \log_x b$$

Relatives of Big-Oh



big-Omega Vs big-O(n)

- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

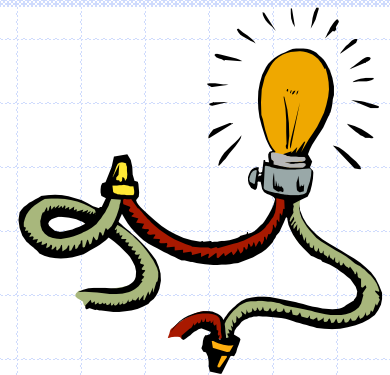
$$f(n) \geq c g(n) \text{ for } n \geq n_0$$

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

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

Intuition for Asymptotic Notation



big-Oh $g(n)$ is big for $f(n)$

- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

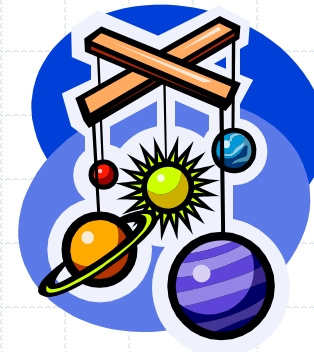
big-Omega

- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$

big-Theta

- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

Example Uses of the Relatives of Big-Oh



■ $5n^2$ is $\Omega(n^2)$

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c g(n)$ for $n \geq n_0$

let $c = 5$ and $n_0 = 1$



■ $5n^2$ is $\Omega(n)$

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c g(n)$ for $n \geq n_0$

let $c = 1$ and $n_0 = 1$

■ $5n^2$ is $\Theta(n^2)$

$f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c g(n)$ for $n \geq n_0$

Let $c = 5$ and $n_0 = 1$