Analysis of Algorithms

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

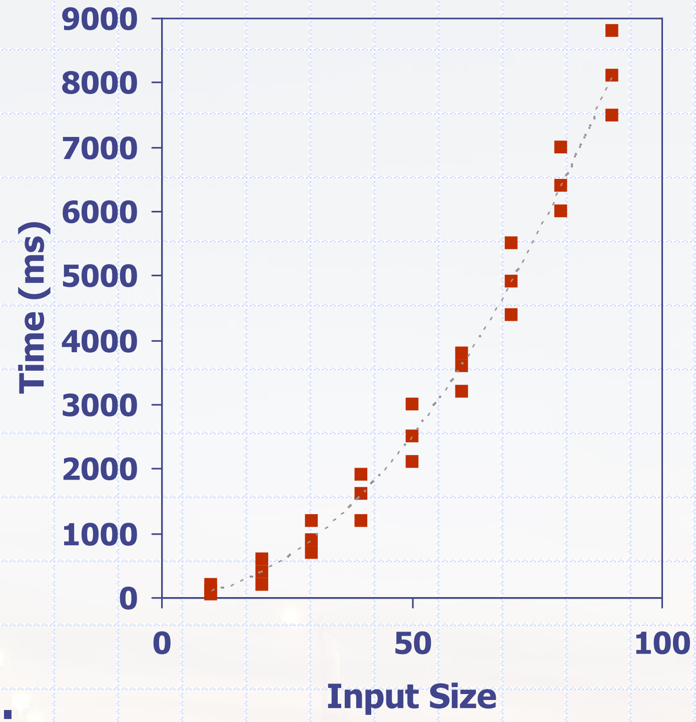
These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

How to Estimate Efficiency?

- Correctness of a method depends merely on whether the algorithm performs what it is supposed to do.
- Clearly, efficiency is hence different than correctness.
- Efficiency can be measured in many ways; i.e:
  - Less memory utilization
  - Faster execution time
  - Quicker release of allocated recourses
  - etc.
- How efficiency can then be measured?
  - Measurement should be independent of used software (i.e. compiler, language, etc.) and hardware (CPU speed, memory size, etc.)
  - Particularly, run-time analysis can have serious weaknesses

Experimental Studies

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time.
- Plot the results.



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult/costly.
- 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
- In some multiprogramming environments, such as Windows, it is very difficult to determine how long a single task takes (since there is so much going behind the scene).

How to Estimate Efficiency?

- Efficiency, to a great extent, depends on how the method is defined.
- An abstract analysis that can be performed by direct investigation of the method definition is hence preferred.
- Ignore various restrictions; i.e:
  - CPU speed
  - Memory limits; for instance allow an int variable to take any allowed integer value, and allow arrays to be arbitrarily large
  - etc.
- Since the method is now unrelated to specific computer environment, we refer to it as ***algorithm***.

Estimating Running Time

- How can we estimate the running/execution-time from algorithm's definition?

- Consider the number of executed statements, in a trace of the algorithm, as a measurement of running- time requirement.

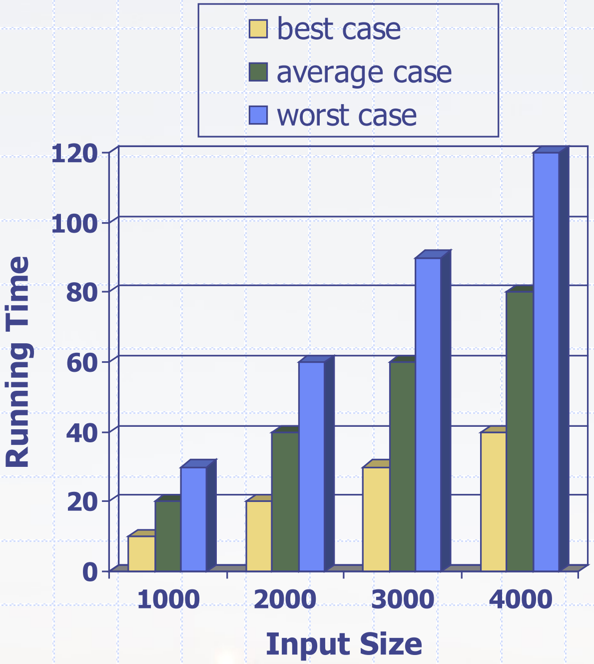
- This measurement can be represented as function of the "size" of the problem.

- The running time of an algorithm typically grows with the input size.

Estimating Running Time

- We focus on the worst case of running time since this is crucial to many applications such as games, finance, robotics, etc.

- Given a method of a problem of size *n*, find ***worstTime(n)*** , which is the maximum number of executed statements in a trace, considering all possible parameters/input values.



Estimating Running Time

Example:

Assume an array a [0 … n –1] of int, and assume the following trace:

```
for (int i = 0; i < n - 1; i++)
  if (a [i] > a [i + 1])
      System.out.println (i);
```

What is worstTime(*n*)?

Estimating Running Time

Statement	Worst Case Number of Executions
i = 0	1
i < n – 1	n
i++	n - 1
a[i] > a[i+1]	n - 1
System.out.println()	n - 1

- **That is, worstTime(*n*) is: $4n - 2$.**

Estimating Running Time

→ See *aboveMeanCount()* method

- **What is worstTime(n)?**

Estimating Running Time

worstTime(*n*) of ***aboveMeanCount()*** method

Statements	Worst # of Executions
double[] a, double mean (assignment of 1st & 2nd arguments)	1 + 1
n = a.length, count = 0	1 + 1
i = 0, return count	1 + 1
i < n	n + 1
i++	n
a[i] > mean	n
count++	n – 1
	= **4n + 6**

Pseudocode

- 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.

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

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

  var.method (*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions

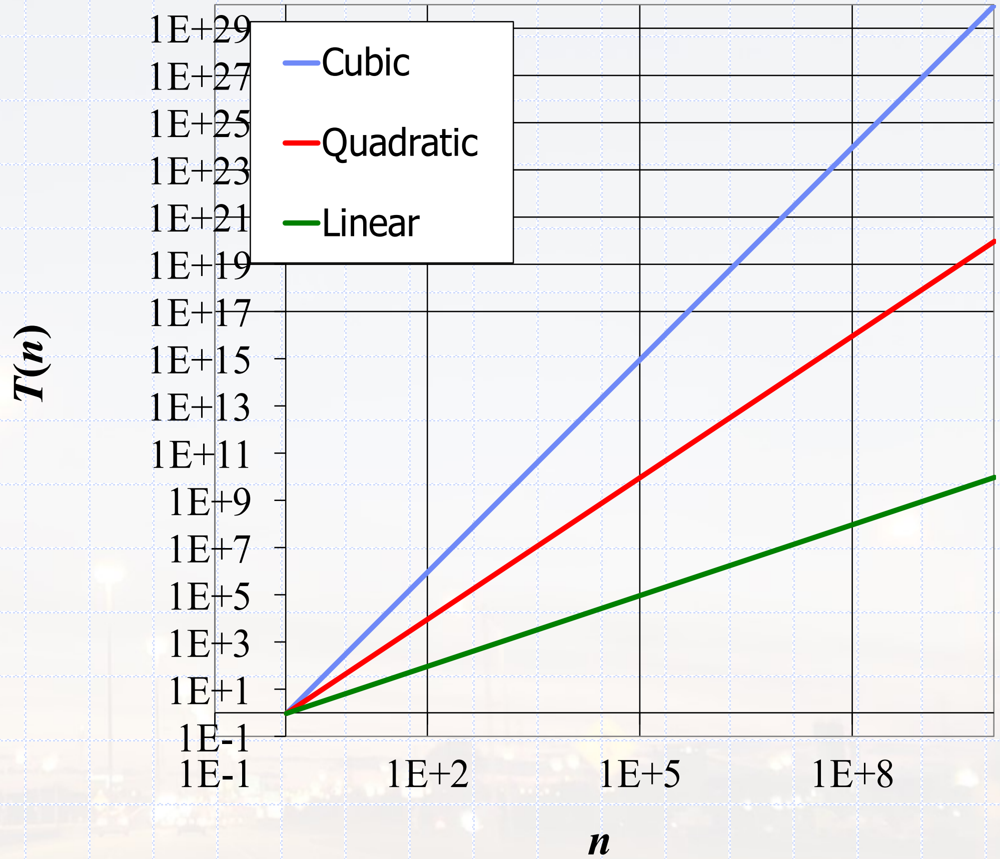
  $\leftarrow$ Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

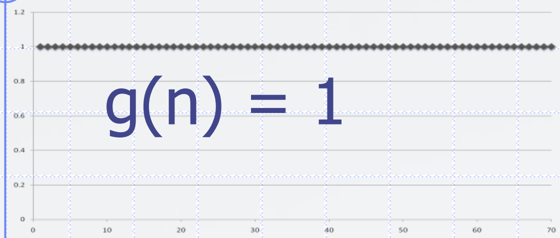
Seven Important Functions

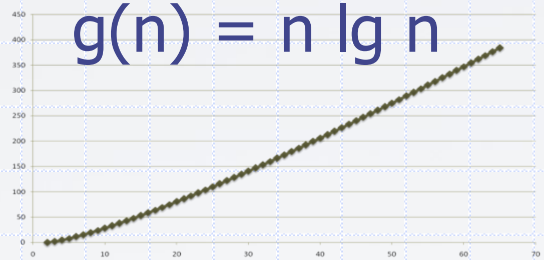
- Seven functions often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n

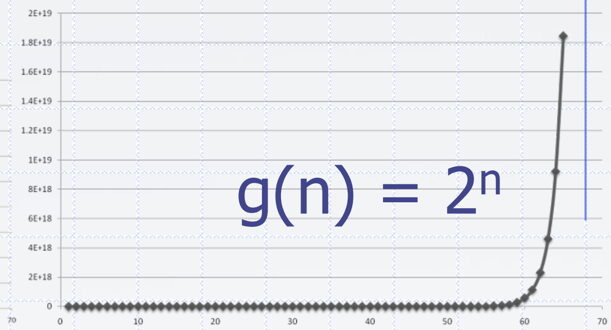


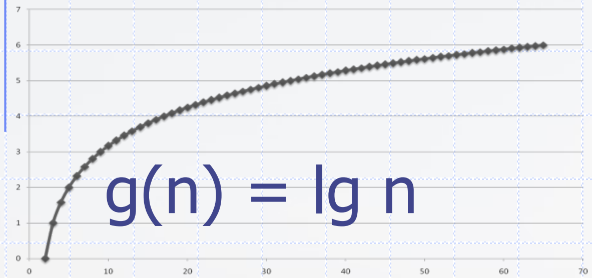
Functions Graphed Using "Normal" Scale

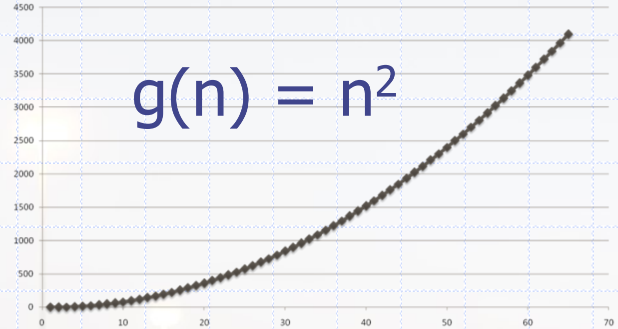
Slide by Matt Stallmann included with permission.

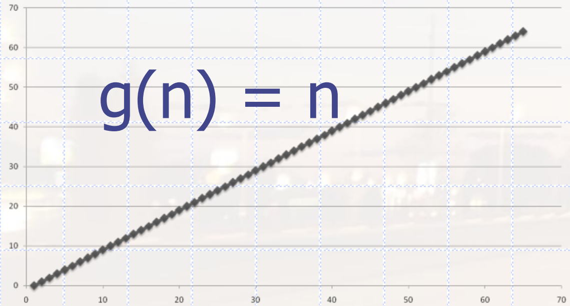


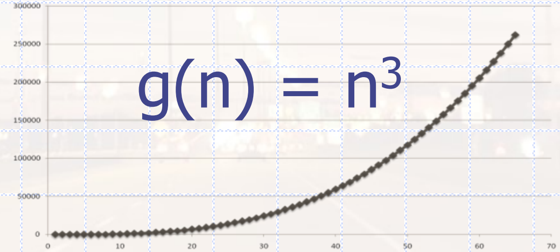






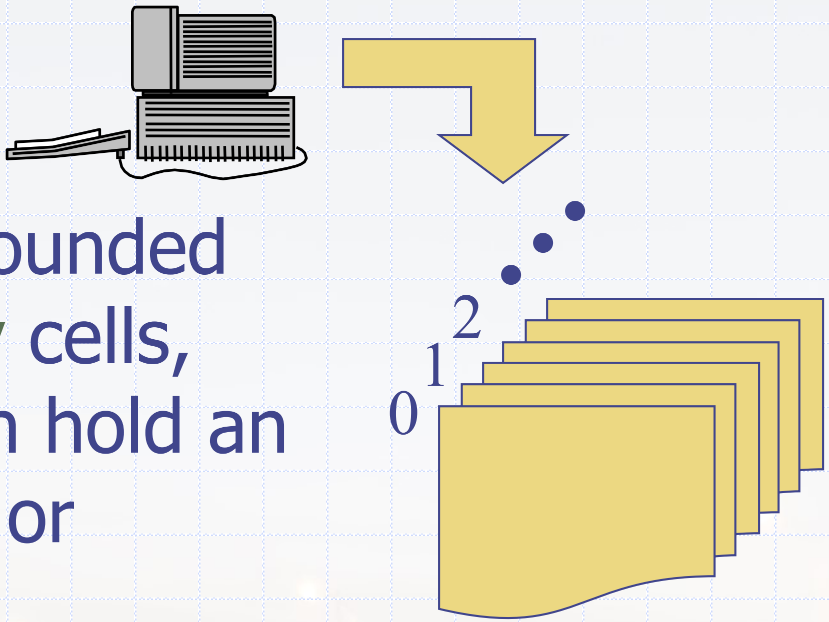






The Random Access Machine (RAM) Model

- A **CPU**.
- A potentially unbounded bank 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.

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

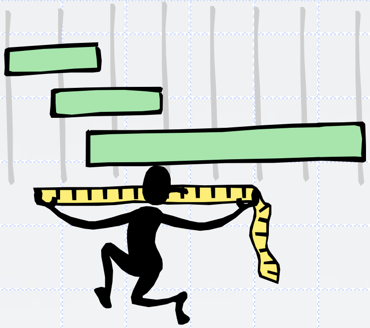
Counting Primitive Operations

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

Algorithm *compareValues*(A, n)
Input array A of n integers
Output display all elements larger than following ones

	# of operations
for $i \leftarrow 0$ **to** $n - 2$ **do**	$n - 1$
if $A[i] > A[i+1]$ **then**	$n - 1$
display i	$n - 1$
increment counter i	$n - 1$
Total	$4n - 4$

Estimating Running Time



- Algorithm ***compareValues*** executes $4\boldsymbol{n} - 4$ primitive operations in the worst case.
- Define:

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

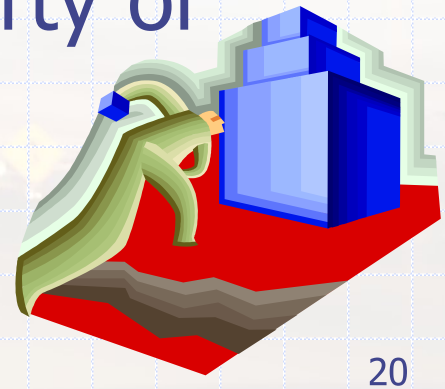
  $\boldsymbol{b}$ = Time taken by the slowest primitive operation
- Let $\boldsymbol{T}(\boldsymbol{n})$ be the running time of ***compareValues.*** Then

$$\boldsymbol{a}\,(4\boldsymbol{n} - 4) \le \boldsymbol{T}(\boldsymbol{n}) \le \boldsymbol{b}(4\boldsymbol{n} - 4)$$

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

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - 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 ***compareValues***



Slide by Matt Stallmann included with permission.

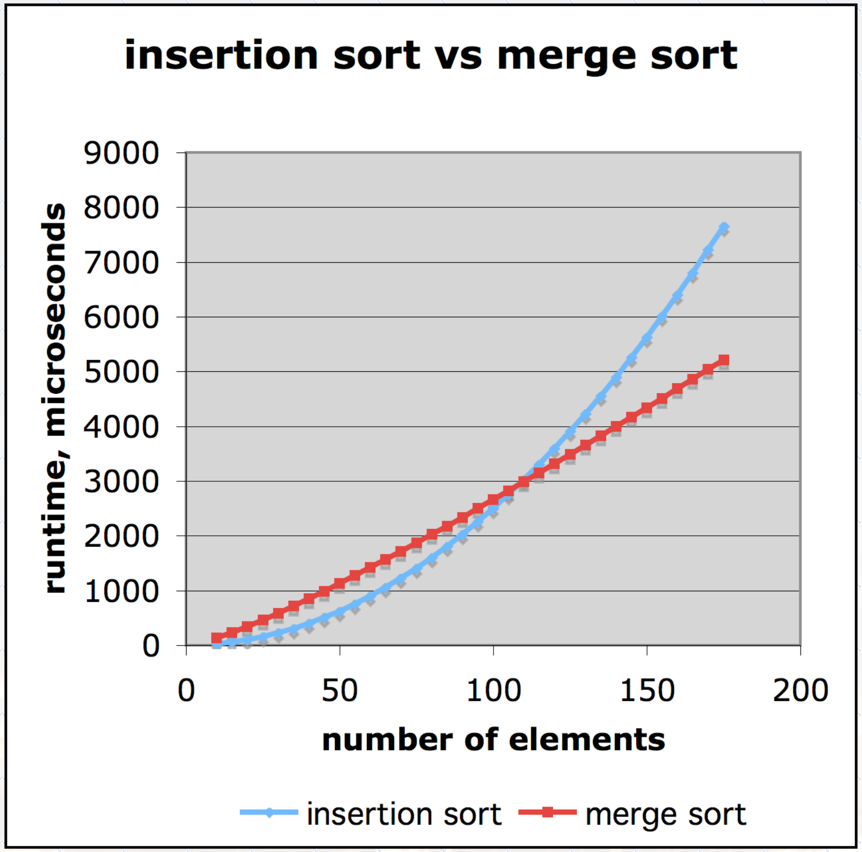
Why Growth Rate Matters

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

runtime quadruples when problem size doubles

Slide by Matt Stallmann included with permission.

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

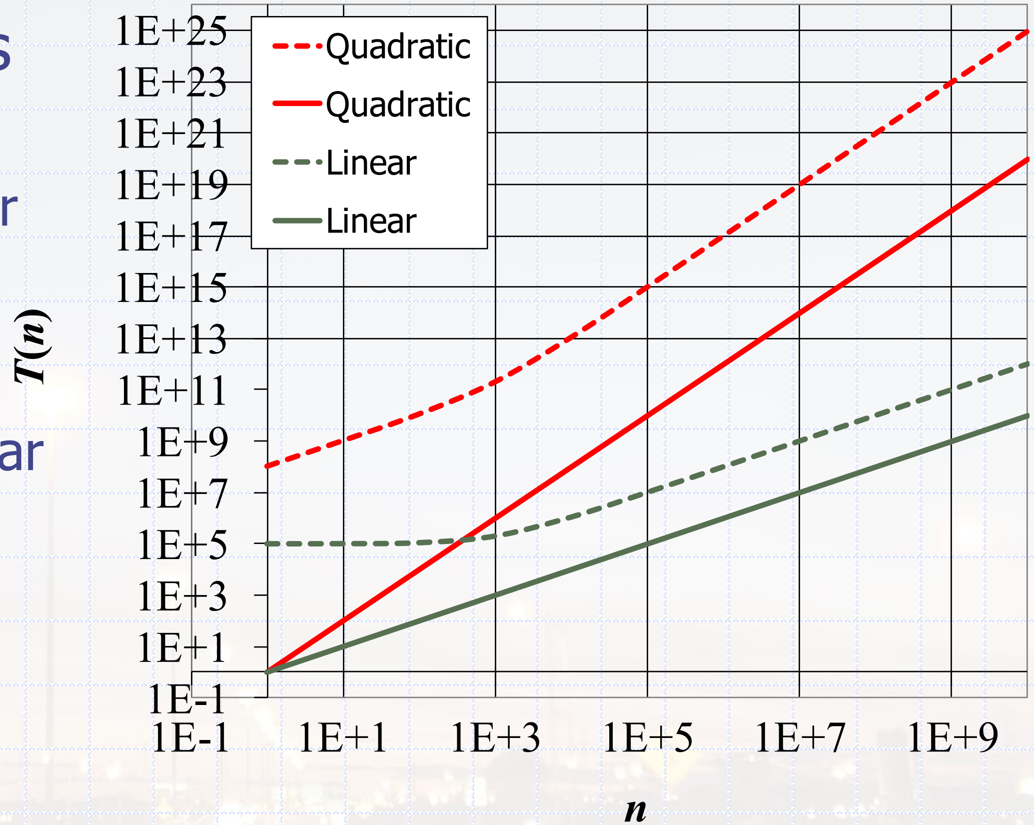
while

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

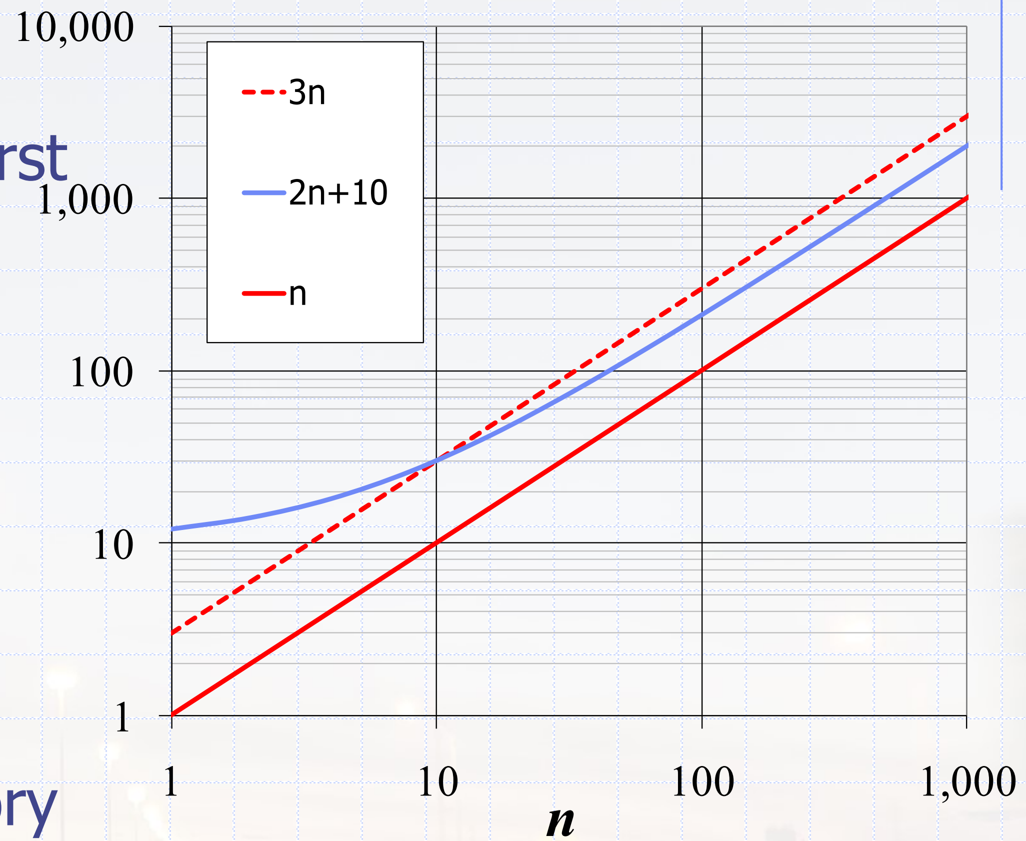
Constant Factors

- 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



Big-O Notation

- We do NOT need to calculate the exact worst time since it is only an approximation of time requirements.
- Instead, we can just approximate that time by means of "Big-O" notation.
- That is quite satisfactory since it gives us approximation of an approximation!



Big-O Notation

- The basic idea is to determine an *upper bound* for the behavior of the algorithm/function.

- In other words, to determine how bad the performance of the algorithm can get!

- If some function $g(n)$ is an upper bound of function $f(n)$, then we say that $f(n)$ is Big-O of $g(n)$.

Big-O Notation

- Specifically, Big-O is defined as follows: 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

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

- The idea is that if $f(n)$ is $O(g(n))$ then it is bounded above (cannot get bigger than) some constant times $g(n)$.

Big-O Notation

- Further, by a standard abuse of notation, we often associate a function with the value is calculates.

- For instance, if $g(n) = n^3$, for $n = 0, 1, 2, \ldots,$ then instead of saying that $f(n)$ is $O(g(n))$, we say $f(n)$ is $O(n^3)$.

Big-O Notation

- **Example 1:** What is O() if $f(n) = 2n + 10$?
  - $2n \le 2n$ for $n \ge 0$
  - $10 \le 10n$ for $n \ge 1$

  So, for any $n \ge 1$
  - $2n + 10 \le 12n$ ➔ consider $c = 12, n_0 = 1$ ➔ $g(n) = n$

  Consequently, the above $f(n)$ is $O(n)$.

Big-O Notation

- In general, if *f(n) is a polynomial function, which is of the form:*

$$a_i n^i + a_{i-1} n^{i-1} + a_{i-2} n^{i-2} + \ldots + a_1 n + a_0$$

Then, we can directly establish that $f(n)$ is $O(n^i)$.

Proof:

Choose

- $n_0 = 1$, and
- $c = |a_i| + |a_{i-1}| + |a_{i-2}| + \ldots + |a_1| + |a_0|$

Big-O Notation

- **Example 2:** What is O() if

$$f(n) = 3n^4 + 6n^3 + 10n^2 + 5n + 4?$$

  - $3n^4 \leq 3n^4$ for $n \geq 0$
  - $6n^3 \leq 6n^4$ for $n \geq 0$
  - $10n^2 \leq 10n^4$ for $n \geq 0$
  - $5n \leq 5n^4$ for $n \geq 0$
  - $4 \leq 4n^4$ for $n \geq 1$

  So, for any $n \geq 1$

  - $3n^4 + 6n^3 + 10n^2 + 5n + 4 \leq 28n^4$ ➔ consider $c = 28$, $n_0 = 1$ ➔ $g(n) = n^4$

  Consequently, the above $f(n)$ is $O(n^4)$.

Big-O Notation

- When determining O(), we can (and actually do) ignore the logarithmic base.

  <u>Proof:</u>

  Assume that $f(n)$ is $O(\log_a n)$, for some positive constant a.

  - Then $f(n) \le C * \log_a n$
    for some positive constant C and some $\boldsymbol{n_0 \le n}$
  - By logarithmic fundamentals,
    $\log_a n = \log_a b * \log_b n$, for any $n > 0$
  - Let $C1 = C * \log_a b$. Then for all $n \ge n_0$
    $f(n) \le C * \log_a n = C * \log_a b * \log_b n = C1 * \log_b n$
    $\Rightarrow f(n)$ is $O(\log_b n)$.

Big-O Notation

- **Example 3:** What is O() if

$$f(n) = 3 \log n + 5$$

  - $3 \log n \leq 3 \log n$ for $n \geq 1$
  - $5 \leq 5 \log n$ for $n \geq 2$

So, for any $n \geq 2$

- $3 \log n + 5 \leq 8 \log n$ ➔ consider $c = 8, n_0 = 2$
  ➔ $g(n) = \log n$

Consequently, the above $f(n)$ is $O(\log n)$.

Big-O Notation

- Nested loops are significant when estimating O().
- **Example 4:**

  Consider the following loop segment, what is O()?

```
for (int i = 0; i < n; i++ )
  for (int j = 0; j < n; j++ )
      ..............
```

  - The outer loop has $1 + (n + 1) + n$ executions.
  - The inner loop has $n(1 + (n + 1) + n)$ executions.
  - Total is: $2n^2 + 4n + 2 \rightarrow O(n^2)$.

Hint: As seen in Example 2 for polynomial functions

Big-O Notation

- **Important Note:** Big-O only gives an upper bound of the algorithm.
- However, if $f(n)$ is $O(n)$, then it is also $O(n + 10)$, $O(n^3)$, $O(n^2 + 5n + 7)$, $O(n^{10})$, etc.
- We, generally, choose the smallest element from this hierarchy of orders.

- For example, if $f(n) = n + 5$, then we choose $O(n)$, even though $f(n)$ is actually also $O(n \log n)$, $O(n^4)$, etc.
- Similarly, we write $O(n)$ instead of $O(2n + 8)$, $O(n - \log n)$, etc.

Big-O Notation

- Elements of the Big-O hierarchy can be as:

  $O(1) \subset O(\log n) \subset O(n^{1/2}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \dots\dots\dots \subset O(2^n) \subset \dots\dots\dots$

Where the symbol "$\subset$", indicates "is contained in".

Big-O Notation

- The following table provides some examples:

Sample Functions	**Order of O()**
$f(n) = 3000$	$O(1)$
$f(n) = (n * \log_2(n+1) + 2) / (n+1)$	$O(\log n)$
$f(n) = (500 \log_2 n) + n / 100000$	$O(n)$
$f(n) = (n * \log_{10} n) + 4n + 8$	$O(n \log n)$
$f(n) = n * (n + 1) / 2$	$O(n^2)$
$f(n) = 3500\, n^{100} + 2^n$	$O(2^n)$

Big-O Notation

- **Warning:** One danger of Big-O is that it can be misleading when the values of n are small.

- For instance, consider the following two functions $f(n)$ and $g(n)$ for $n \geq 0$

$$f_1(n) = 1000\, n \rightarrow f_1(n) \text{ is hence } O(n)$$

$$f_2(n) = n^2 / 10 \rightarrow f_2(n) \text{ is hence } O(n^2)$$

However, and despite of the fact that $f_2(n)$ has a higher/worst order than the one of $f_1(n)$, $f_1(n)$ is actually greater than $f_2(n)$, for all values of n less than 10,000!

Finding Big-O Estimates Quickly

- **Case 1:** Number of executions is independent of n ➔ $O(1)$
- Example:

```
// Constructor of a Car class
  public Car(int nd, double pr)
  {
   numberOfDoors = nd;
   price = pr;
  }
```

Example:

```
for (int j = 0; j < 10000; j++ )
  System.out.println(j);
```

Finding Big-O Estimates Quickly

- **Case 2:** The splitting rule ➔ $O(\log n)$
- Example:

```
while(n > 1)
{
  n = n / 2;
  …;
}
```

Example:

See the binary search method in Recursion6.java & Recursion7.java

Finding Big-O Estimates Quickly

- **Case 3:** Single loop, dependent on n → $O(n)$
- Example:

```
for (int j = 0; j < n; j++ )
    System.out.println(j);
```

Note: It does NOT matter how many simple statement (i.e. no inner loops) are executed in the loop. For instance, if the loop has k statements, then there is $k*n$ executions of them, which will still lead to $O(n)$.

Finding Big-O Estimates Quickly

- **Case 4:** Double looping dependent on n & splitting
  ➔ $O(n \log n)$
- Example:

```
for (int j = 0; j < n; j++ )
{
  m = n;
  while (m > 1)
  {
       m = m / 2;
       …;
       // Does not matter how many statements are here
  }
}
```

Finding Big-O Estimates Quickly

- **Case 4:** Double looping dependent on n
  $\rightarrow O(n^2)$
- Example:

```
for (int i = 0; i < n; i++ )
        for (int j = 0; j < n; j++ )

        {

                …;
                // Does not matter how many statements are here
        }
```

Finding Big-O Estimates Quickly

- **Case 4 (Continues):** Double looping dependent on n $\rightarrow O(n^2)$
- Example:

```
for (int i = 0; i < n; i++ )
        for (int j = i; j < n; j++ )
        {
                …;
                // Does not matter how many statements are here
        }
```

The number of executions of the code segment is as follows:

$$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

Which is: $n(n + 1) / 2 = \frac{1}{2} n^2 + \frac{1}{2} n \rightarrow O(n^2)$

Finding Big-O Estimates Quickly

- **Case 5:** Sequence of statements with different $O(\ )$
  $O(g_1(n)) + O(g_2(n)) + \ldots = O(g_1(n) + g_2(n) + \ldots)$
- Example:

```
for (int i = 0; i < n; i++ )
{       ...
}
for (int i = 0; i < n; i++ )
    for (int j = i; j < n; j++ )
        {       ...
        }
```

The first loop is $O(n)$ and the second is $O(n^2)$. The entire segment is hence $O(n) + O(n^2)$, which is equal to $O(n + n^2)$, which is in this case $O(n^2)$.

Asymptotic Algorithm Analysis

- In computer science and applied mathematics, asymptotic analysis is a way of describing limiting behaviours (may approach ever nearer but never crossing!).

- Asymptotic analysis of an algorithm determines the running time in big-O notation.
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We then express this function with big-O notation

Asymptotic Algorithm Analysis

- Example:
  - We determine that algorithm ***compareValues*** executes at most $4n - 4$ primitive operations
  - We say that algorithm ***compareValues*** "runs in $O(n)$ time", or has a "complexity" of $O(n)$

Note: Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.

Asymptotic Algorithm Analysis

- If two algorithms *A* & *B* exist for solving the same problem, and, for instance, A is $O(n)$ and B is $O(n^2)$, then we say that *A* is asymptotically better than *B* (although for a small time *B* may have lower running time than *A*).

- To illustrate the importance of the asymptotic point of view, let us consider three algorithms that perform the same operation, where the running time (in μs) is as follows, where n is the size of the problem:
  - Algorithm1: $400n$
  - Algorithm2: $2n^2$
  - Algorithm3: 2^n

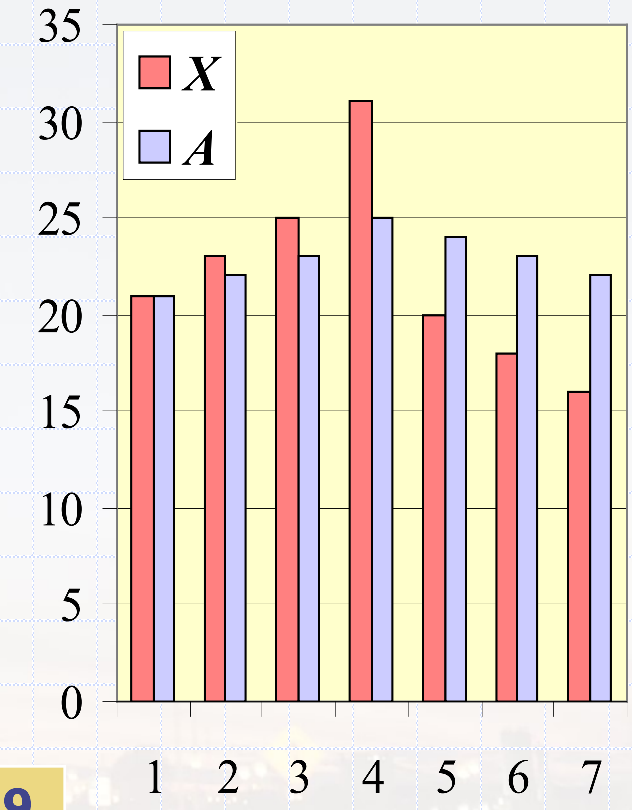
Asymptotic Algorithm Analysis

- Which of the three algorithms is faster?
  - *Notice that Algorithm1 has a very large constant factor compared to the other two algorithms!*

Running Time (μs)	**Maximum Problem Size (n) that can be solved in:**		
	1 Second	1 Minute	1 Hour
Algorithm 1 $400n$	2,500 (400 * 2,500 = 1000,000)	150,000	9 Million
Algorithm 2 $2n^2$	707 (2 * 707^2 ≈ 1000,000)	5,477	42,426
Algorithm 3 2^n	19 (only 19, since 2^{20} would exceed 1000,000)	25	31

Asymptotic Algorithm Analysis

- Let us further illustrate asymptotic analysis with two algorithms that would compute prefix averages.
- Given an array X storing n numbers, we need to construct an array A such that:
  - A[i] is the average of $X[0] + X[1] + \ldots + X[i]$)



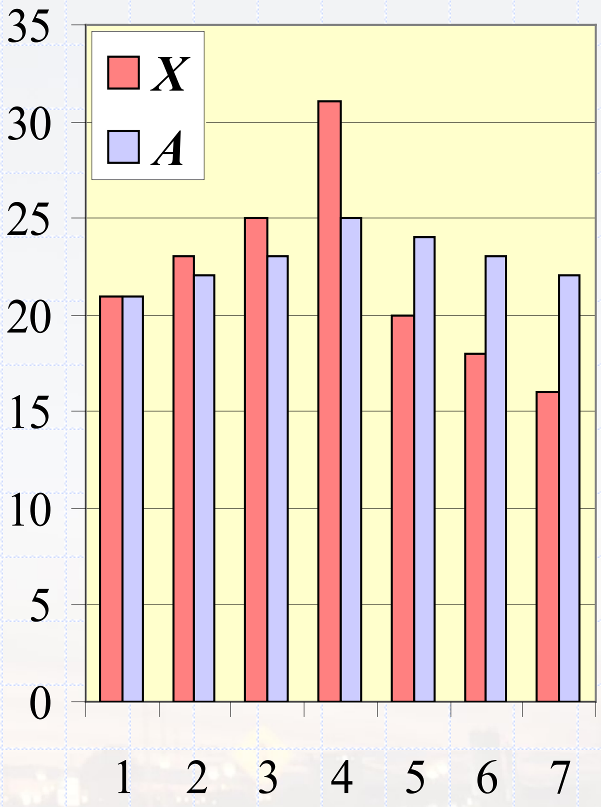
X	10	16	4	18	7	23	27	39
A	10	13	10	12	11	13	15	18

Asymptotic Algorithm Analysis

- That is, the i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

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

- Computing prefix average has applications to financial analysis; for instance the average annual return of a mutual fund for the last year, three years, ten years, etc.



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time (n^2) by applying the definition

Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	# of operations
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Prefix Averages (Quadratic)

- Hence, to calculate the sum n integers, the algorithm needs (from the segment that has the two loops) $n(n+1)/2$ operations.

- In other words, ***prefixAverages1*** is $O(1+2+\ldots+n) = O(n(n+1)/2) \rightarrow O(n^2)$

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time (n) by keeping a running sum

Algorithm *prefixAverages2(X, n)*	
Input array X of n integers	
Output array A of prefix averages of X	# of operations
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

- Algorithm *prefixAverages2* runs in $O(n)$ time, which is clearly better than *prefixAverages1*.

Big-Omega, Big-Theta & *Plain English!*

- In addition to Big-O, there are two other notations that are used for algorithm analysis: Big-Omega and Big-Theta.
- While Big-O provides an upper bound of a function, Big-Omega provides a lower bound.
- In other words, while Big-O indicates that an algorithm behavior "cannot be any worse than", Big-Omega indicates that it "cannot be any better than".

Big-Omega, Big-Theta & *Plain English!*

- Logically, we are often interested in worst-case estimations, however knowing the lower bound can be significant when trying to achieve an optimal solution.

- **big-Omega**

Big-Omega is defined as follows: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants c and n_0 such that

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

Big-Omega, Big-Theta & *Plain English!*

- Example: $3n \log n + 2n$ is $\Omega(n \log n)$

  Proof:

  - $3n \log n + 2n \geq 3n \log n \geq n \log n$

    for every $n \geq 1$

- Example: $3n \log n + 2n$ is $\Omega(n)$

  Proof:

  - $3n \log n + 2n \geq 2n \geq n$

    for every $n \geq 1$

Big-Omega, Big-Theta & *Plain English!*

- Notice that: … $\Omega(2^n) \subset \Omega(n^3) \subset \Omega(n^2) \subset \Omega(n \log n) \subset \Omega(n) \subset \Omega(n^{1/2}) \subset \ldots \subset \Omega(\log n) \subset \ldots \subset \Omega(1)$

Where the symbol "$\subset$", indicates "is contained in".

- It should be noted that in "many" cases, a method that is $O(\,)$ is also $\Omega(\,)$.

Big-Ω

- **<u>Example 1 (Revised):</u>** What are O() and Ω() if $f(n) = 2n + 10$?

→ ***As seen before, the method is O(n).***

- Now,
  - $2n + 10 \geq 2n \geq n \quad$ for $n \geq 0$

  So, for any $n \geq 0$

  - $2n + 10 \geq n$ ➔ consider $c = 1, n_0 = 0$ ➔ $\Omega(n) = n$

  Consequently, the above $f(n)$ is $O(n)$, and is also $\Omega(n)$.

Big-Ω

- **Example 2 (Revised):** What are O() and Ω() if

$$f(n) = 3n^4 + 6n^3 + 10n^2 + 5n + 4?$$

As seen before, the method is $O(n^4)$.

- Now,
  - $3n^4 + 6n^3 + 10n^2 + 5n + 4 \geq 3n^4$ for $n \geq 0$
  - $3n^4 \geq n^4$ for $n \geq 0$

  So, for any $n \geq 0$

  - $3n^4 + 6n^3 + 10n^2 + 5n + 4 \geq n^4$

    ➔ consider $c = 1$, $n_0 = 0$ ➔ $g(n) = n^4$

  Consequently, the above $f(n)$ is $O(n^4)$ and is also $\Omega(n^4)$.

Big-Ω

- However, Big-O and Big-Omega are distinct.
- That is, there are cases when a function may have different O( ) and Ω( ).

- A simple, and somehow artificial, proof of that can be provided as follows:
  - Assume $f(n) = n$ for $n = 0, 1, 2, \ldots$
  - Clearly $f(n)$ is $O(n)$, and hence is $O(n^2)$
  - Yet, $f(n)$ is NOT $\Omega(n^2)$
  - Also, since $f(n)$ is $\Omega(n)$, it is also $\Omega(1)$
  - Yet, $f(n)$ is NOT $O(1)$

Big-Ω

- In fact, as seen, the hierarchy of $\Omega()$ is just the reverse of the one of O()

- For example, the following code segment:

```
for (int j = 0; j < n; j++)
    System.out.println (j);
```

is: **$O(n)$, $O(n \log n)$, $O(n^2)$, …**

And is also:

$\Omega(n)$, $\Omega(\log n)$, $\Omega(1)$

Big-O, Big-Omega, Big-Theta &*Plain English!*

- Big-O provides and upper bound of a function, while Big-Ω provides a lower bound of it.

- In many, if not most, cases, there is often a need of one function that would serve as both lower and upper bounds; that is Big-Theta (Big-Θ).

Big-O, Big-Omega, Big-Theta & *Plain English!*

big-Theta

- Big-Theta is defined as follows: Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$ if there are positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for } n \geq n_0$$

- Simply put, if a function is $f(n)$ is $\Theta(g(n))$, then it is bounded above and below by some constants time $g(n)$; in other words, it is, roughly, bounded above and below by $g(n)$.
- → Notice that if $f(n)$ is $\Theta(g(n))$, then it is hence both $O(g(n))$ and $\Omega(g(n))$.

Big-Θ

- **Example 2 (Revised Again):** What is Θ() of the following function:

$$f(n) = 3n^4 + 6n^3 + 10n^2 + 5n + 4?$$

As seen before, the function is $O(n^4)$ and also is $\Omega(n^4)$

➔ Hence, it is $\Theta(n^4)$.

Big-O, Big-Ω & Big-Θ

Quick Examples

- **$5n^2$ 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 $f(n) \geq c \bullet g(n)$ 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 $f(n) \geq c \bullet g(n)$ 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 $\mathrm{f}(n) \leq c \bullet g(n)$ for $n \geq n_0$

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

- Notice that **$5n^2$ is NOT $\Theta(n)$** since it is not **O(n)**

Plain English

- Sometimes, it might be easier to just indicate the behavior of a method through natural language equivalence of Big-Θ.

- For instance
  - if $f(n)$ is $\Theta(n)$, we indicate that f is "linear in n".
  - if $f(n)$ is $\Theta(n^2)$, we indicate that f is "quadratic in n".

Plain English

- The following table shows the English-language equivalence of Big-Θ

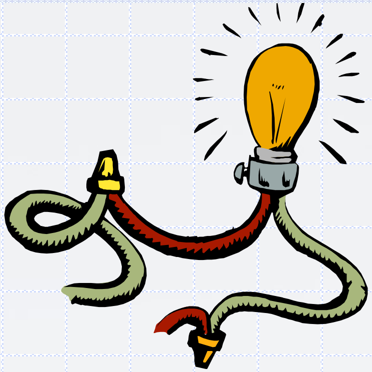
Big-Θ	English
$\Theta(c)$ for some constant $c \geq 0$	Constant
$\Theta(\log n)$	Logarithmic in *n*
$\Theta(n)$	Linear in *n*
$\Theta(n \log n)$	Linear-logarithmic in *n*
$\Theta(n^2)$	Quadratic in *n*

Big-O

- We may just prefer to use plain English, such as "linear", "quadratic", etc..
- However, in practice, there are MANY occasions when all we specify is an upper bound to the method, which is namely:

Big-O

Intuition for Asymptotic Notation



Big-O

- $f(n)$ 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 $g(n)$

Big-O and Growth Rate

- The big-O 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-O 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-O and Growth Rate

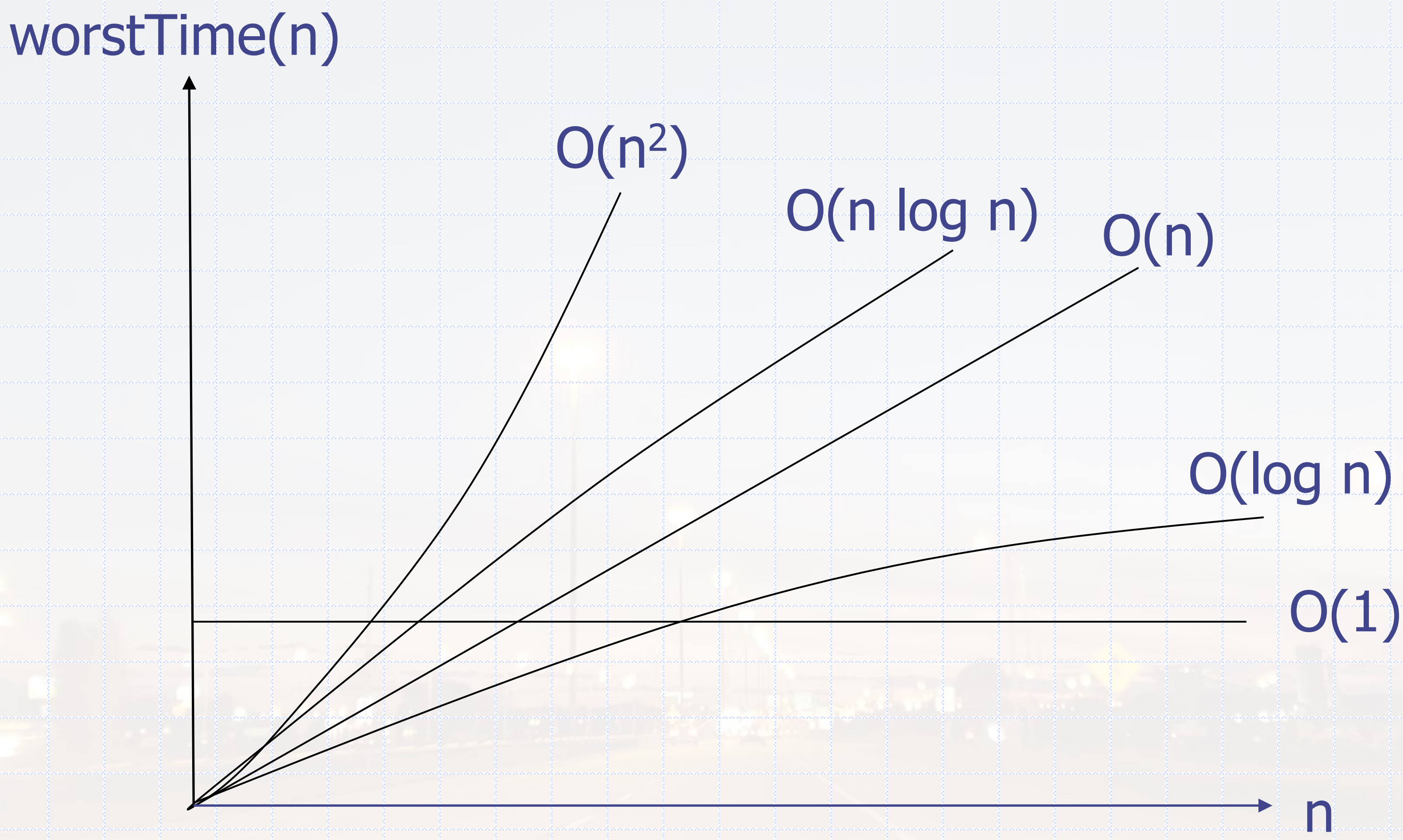
- Again, we are specifically interested in how rapidly a function increases based on its classification.

- For instance, suppose that we have a method whose ***worstTime()*** estimate is linear in n, what will be the effect of doubling the problem size?
  - $worstTime(n) \approx c * n$, for some constant *c*, and sufficiently large value of *n*
  - If the problem size doubles then $worstTime(n) \approx c * 2n \approx 2 * worstTime(n)$
- In other words, if n is doubled, then worst time is doubled.

Big-O and Growth Rate

- Now, suppose that we have a method whose ***worstTime()*** estimate is quadratic in n, what will be the effect of doubling the problem size?
  - $worstTime(n) \approx c * n^2$
  - If the problem size doubles then $worstTime(2n) \approx c * (2n)^2 = c * 4 * n^2 \approx 4 * worstTime(n)$
- In other words, if n is doubled, then worst time is quadrupled.

Big-O and Growth Rate



Big-O and Growth Rate

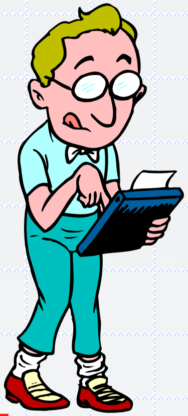
- Again, remember that the Big-O differences eventually dominate constant factors.
- For instance if n is sufficiently large, $100\ n \log n$ will still be smaller than $n^2 / 100$.
- So, the relevance of Big-O, Big-Ω or Bog-Θ may actually depend on how large the problem size may get (i.e. $100{,}000$ or more in the above example).

Big-O and Growth Rate

- The following table provides estimate of needed execution time for various functions of n, if $n = 1000{,}000{,}000$, running on a machine executing $1000{,}1000$ statements per second.

Function o f n	**Time Estimate**
$\log_2 n$	**.0024 Seconds**
n	**17 Minutes**
$n \log_2 n$	**7 Hours**
n^2	**300 Years**

Math you need to Review



- Summations
- Logarithms and Exponents

- **properties of logarithms:**

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

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

  $\log_b xa = a\log_b x$

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

  $\mathrm{Log}_2 2n = \log_2 n + 1$

  $\mathrm{Log}_2 4n = \log_2 n + 2$

- **properties of exponentials**:

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

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

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

  $b = a^{\log_a b}$

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

- Proof techniques
- Basic probability