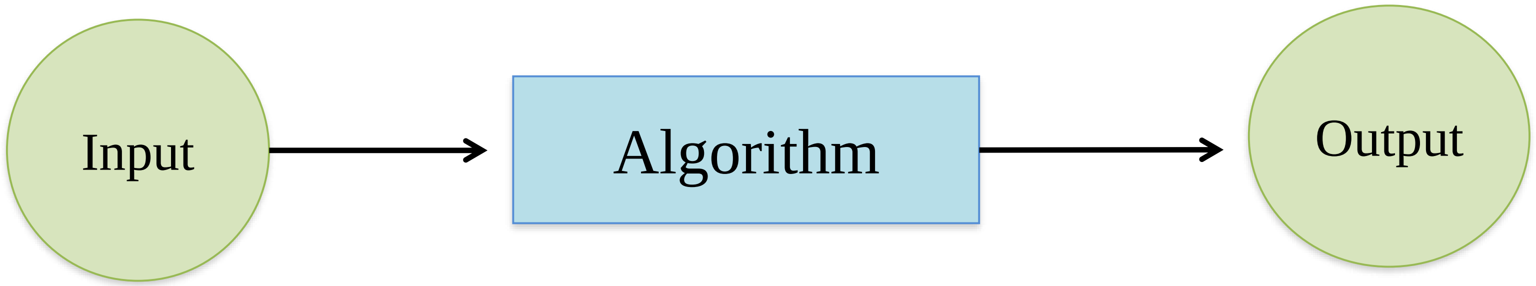
Analysis of Algorithms

- An algorithm is a step-by-step procedure for performing some task (ex: sorting a set of integers) in a finite amount of time.



- We are concerned with the following properties:
  - Correctness
  - Efficiency (how fast it is, how many resources it needs)

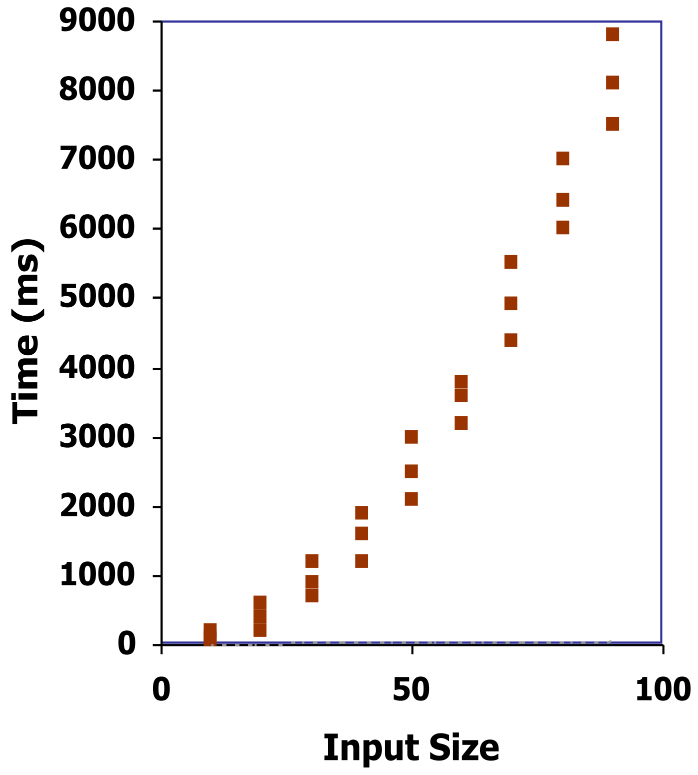
Running Time

- Running time is a natural measure of Efficiency.
  - So what would be the proper way of measuring it?
- We can do experiments, and then decide which algorithm is better.
  - If we have two algorithms for a problem, implement them and do several experiments on various input size.

Experimental Studies

- Run the program with inputs of varying size and composition
  - Use a method like std::clock() to get an accurate measure of the actual running time

- Plot the results



What is the problem of experimental studies?

- The running time is affected by the hardware (Processor, RAM, etc.) and software (Compiler, programing language, etc.)

Limitations of Experiments

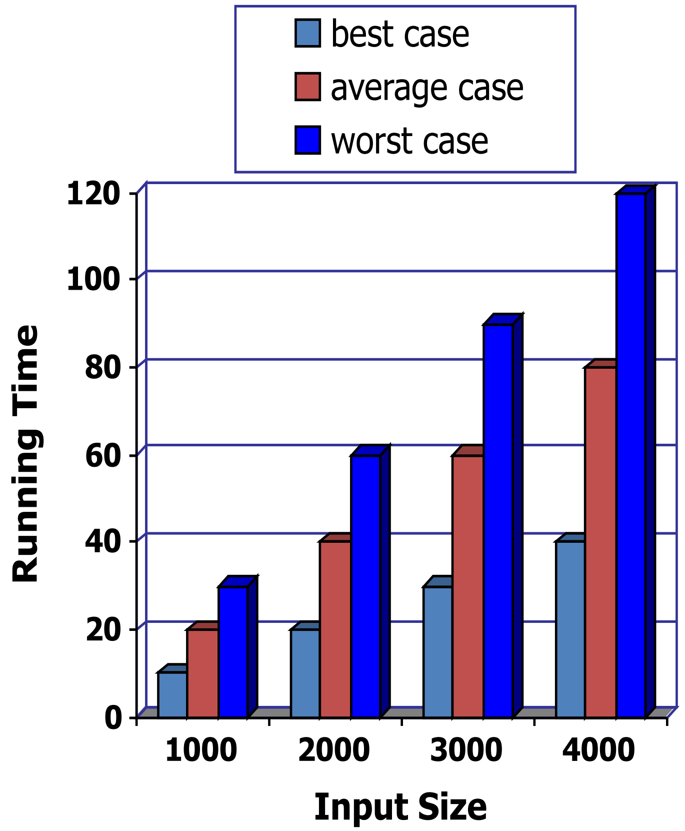
- Need to implement the algorithm
  - may be difficult
- Experiments done on a limited set of test inputs
  - may not be indicative of running times on other inputs not included in the experiment
- Difficult to compare
  - same hardware and software environments must be used

Running time

- We need another way to measure to the running time of an algorithm which:
  - Considers all possible inputs.
  - Be independent from hardware and software.

Running Time

- 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



Theoretical Analysis

- Uses pseudocode, a high-level description of the algorithm
  - no implementation necessary
- Takes into account all possible inputs
- Characterizes running time by $f(n)$, a function of the input size n
  - allows us to evaluate the speed of an algorithm independent of hardware/software environment

Pseudocode

- Mixture of natural language and high-level programming constructs that describe the main ideas behind an algorithm implementation
- Preferred notation for describing algorithms
- Hides program design issues

```
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***

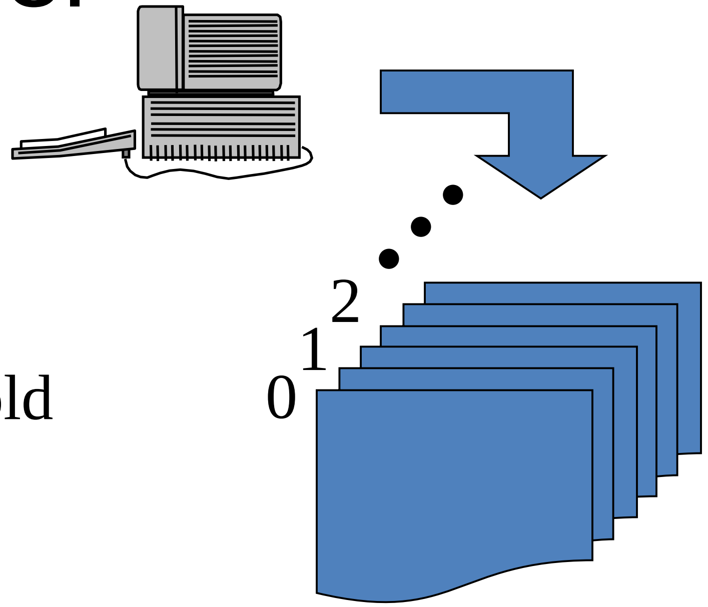
- Expressions

  $\leftarrow$ or := Assignment (like = in C++)

  = Equality testing (like == in C++)

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model



- Views a computer as:
  - a **CPU**, with
  - 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.

Random Access refers to ability of CPU to access arbitrary memory cell with one primitive operation

Primitive Operations

- Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (we'll see why later)

- Assumed to take a constant amount of time in the RAM model

- Includes:
  - 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 arrayMax(A, n)
    currentMax ← A[0]
    for i ← 1 to n – 1 do
        if A[i] > currentMax then
            currentMax ← A[i]
        { increment counter i }
    return currentMax
```

An algorithm to find the maximum number in array.

Counting Primitive Operations

- Primitive Operations:

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

And at most

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2$$

Estimating Running Time

- Algorithm ***arrayMax*** executes $7n - 2$ primitive operations in the worst 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

$$a(7n - 2) \leq T(n) \leq b(7n - 2)$$

Hence, the running time $T(n)$ is bounded by two linear functions.

Growth Rate of Running Time

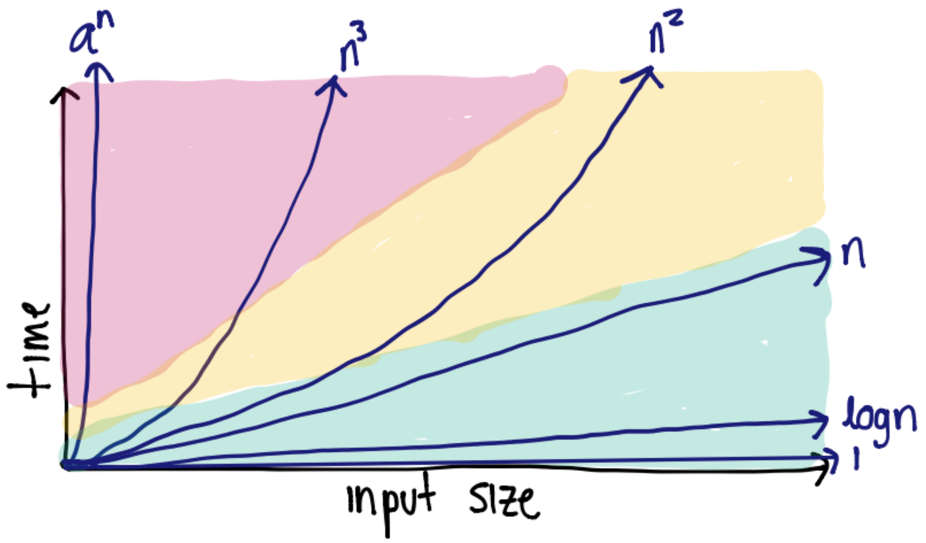
- Changing the hardware/software environment
  - affects $T(n)$ by a constant factor, but
  - does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***

Mathematical Review

- Logarithm: $\log_b a = c \;\; if \; b^c = a$
- Polynomial function: A polynomial can have constants, variables and exponents.
- Example: $4n^3 + 10n^2 + 1000n + 99$
  - Has 4 terms, but only one variable (n)
  - The Degree is **3** (which is the largest exponent)
- Example: $10mn^4 + 5m^2 + n + 3m + 13$
  - Has 5 terms, two variables (n and m)
  - The Degree is **5** (which is the largest exponent)

Growth Rates

Constant	$\approx \boldsymbol{1}$	
Logarithmic	$\approx \log \boldsymbol{n}$	
Linear	$\approx \boldsymbol{n}$	
Quadratic	$\approx \boldsymbol{n}^2$	
Cubic	$\approx \boldsymbol{n}^3$	
Polynomial	$\approx \boldsymbol{n}^k$	(for $k \geq 1$)
Exponential	$\approx \boldsymbol{a}^{\boldsymbol{n}}$	$(a \geq 1)$



Growth rate is not affected by

- constant factors or
- lower-order terms

Ex: $10^2\boldsymbol{n} + 10^5$ is a linear function

Ex: $10^5\boldsymbol{n}^2 + 10^8\boldsymbol{n}$ is a quadratic function

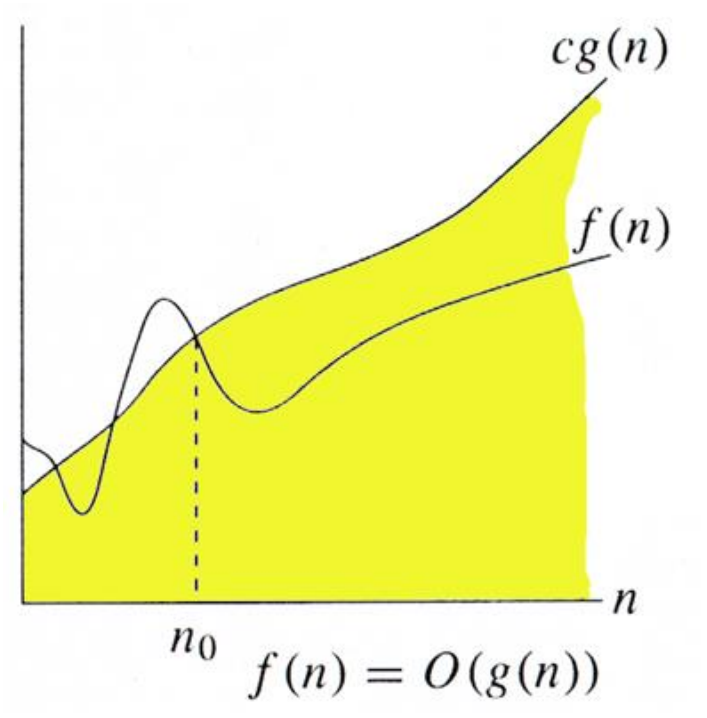
Asymptotic Complexity

- Worst case running time of an algorithm as a function of input size n **for large n**.
- Expressed using only the **highest-order term** in the expression for the exact running time.
  - Instead of exact running time, say $O(n^2)$
- Written using asymptotic notation (***O***, Ω, Θ, o, ω)
  - Ex: $f(n) = O(n^2)$
  - Describes how $f(n)$ grows in comparison to n^2
- The notations describe different rate-of-growth relations between the defining function and the defined **set** of functions

O-notation

For functions $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{ f(n) :$
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$
we have $0 \leq f(n) \leq cg(n) \}$



Technically, $f(n) \in O(g(n))$.
Older usage, $f(n) = O(g(n))$.

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$

Examples

$O(g(n)) = \{ f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ $\}$

- ***O(n)***
  - $f(n)=7n+3$
  - $f(n) = 2n + 10$
  - $f(n) = n + 1$
  - $f(n) = 10000n$
  - $f(n) = 10000n + 300$
- ***O(n²)***
  - $f(n) = n^2 + 1$
  - $f(n) = n^2 + n$
  - $f(n) = 10000n^2 + 10000n + 300$
  - $f(n) = n^{1.99}$

- The function $\boldsymbol{n}^2$ is not $\boldsymbol{O}(\boldsymbol{n})$
  – the inequality $\boldsymbol{n}^2 \leq \boldsymbol{cn}$ cannot be satisfied since c is constant

Big-Oh Rules

- Drop lower-order terms
  - Ex: if $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$
- Drop constant factors, using the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- See Theorem 1.7 in your book

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - Find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Ex:
  - ***arrayMax*** executes at most $7\boldsymbol{n} - 1$ primitive operations
  - ***arrayMax*** "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

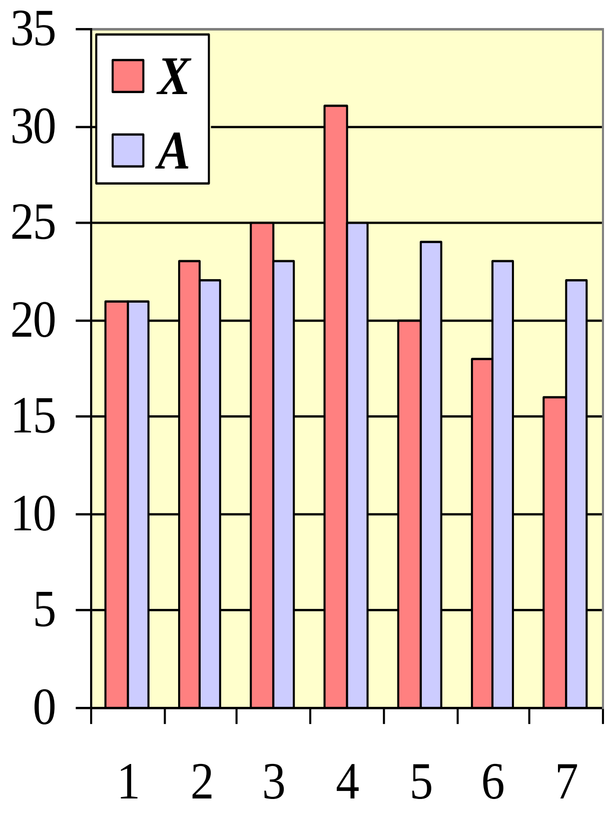
Ex: Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.

- The ***i***-th prefix average of an array ***X*** is average of the first ($\boldsymbol{i}$ + 1) elements of ***X***:

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

- Prefix average has applications in economic and statistics



Prefix Averages V1 $O(n^2)$ - Quadratic!

The following algorithm computes prefix averages by applying the definition

```
Algorithm prefixAverages1(X, n)                    rough # operations
    Input array X of n integers
    Output array A of prefix averages of X
    A ← new array of n integers                    n
    for i ← 0 to n – 1 do                          n
        s ← X[0]                                   n
        for j ← 1 to i do                          1 + 2 + … + (n-1)
            s ← s + X[j]                           1 + 2 + … + (n-1)
        A[i] ← s / (i + 1)                         n
    return A                                       1
```

Prefix Averages V2

O(n) - Linear!

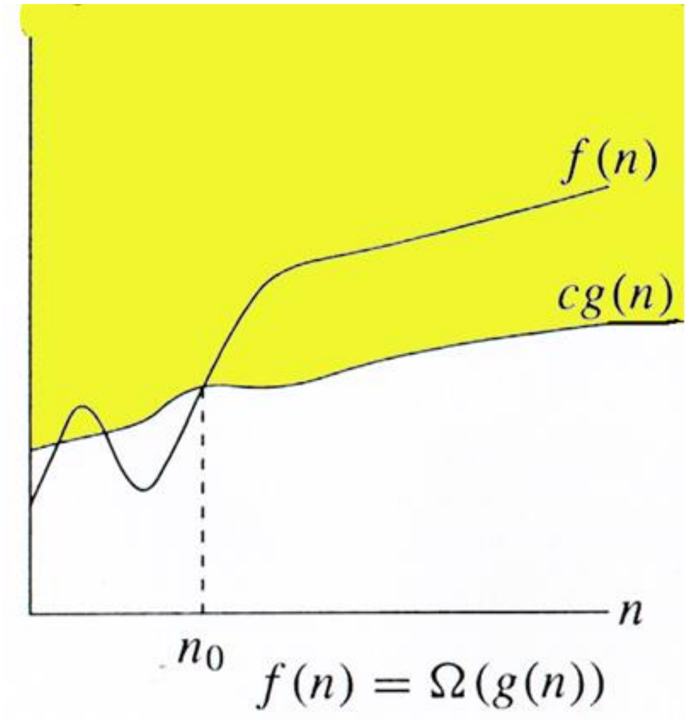
- The following algorithm computes prefix averages by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          rough # operations
    A ← new array of n integers                            n
    s ← 0                                                  1
    for i ← 0 to n − 1 do                                  n
        s ← s + X[i]                                       n
        A[i] ← s / (i + 1)                                 n
    return A                                               1
```

Ω-notation

For functions $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{ f(n) :$
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$
we have $0 \leq cg(n) \leq f(n)\}$



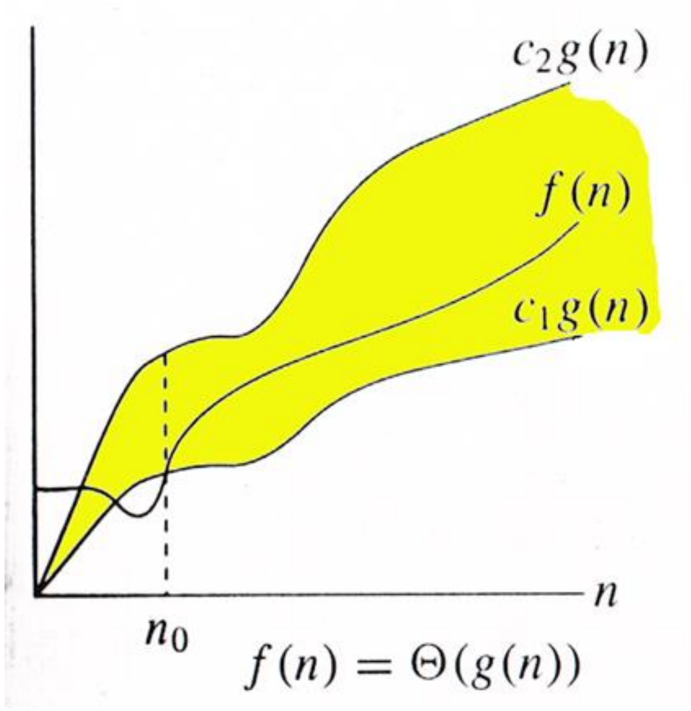
Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$

Θ-notation

For functions $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

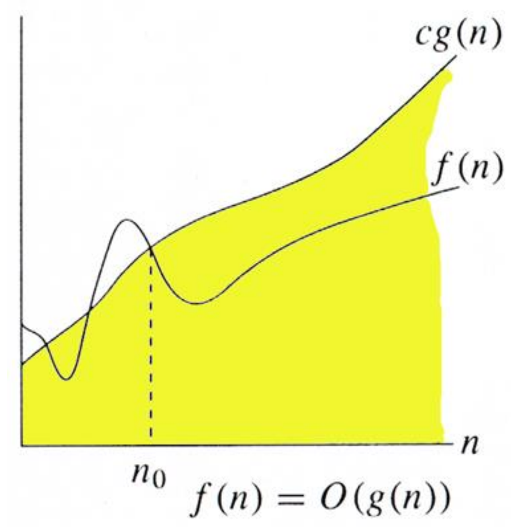
$\Theta(g(n)) = \{ f(n) :$
$\exists$ positive constants c_1, c_2, and n_0,
such that $\forall n \geq n_0$
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$



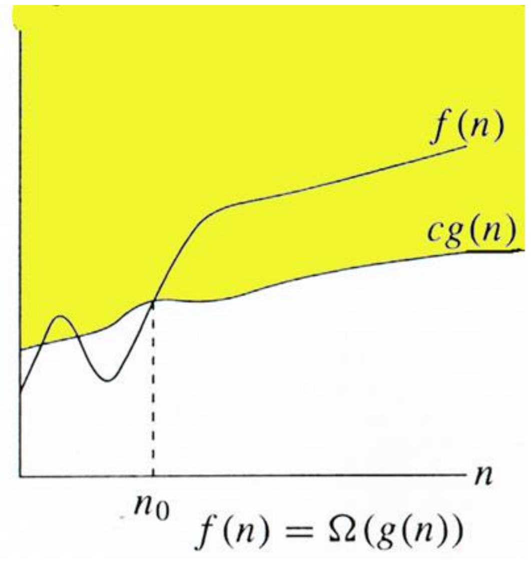
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for $f(n)$

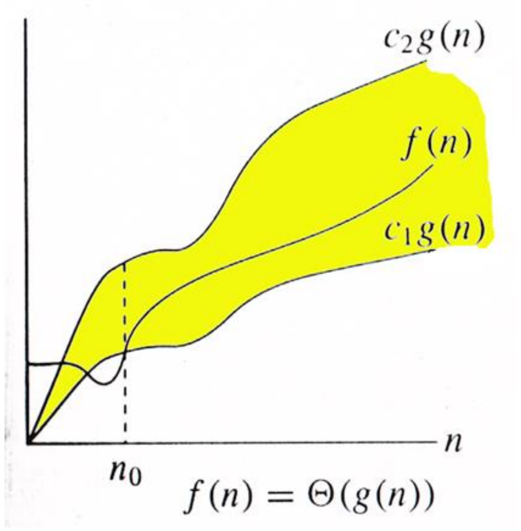
Relationship between *O*, Ω, *Θ*



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

Relatives of O and Ω

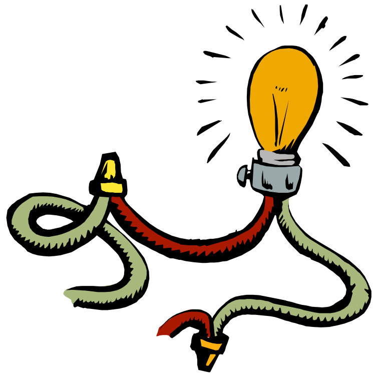
Little-oh

- $f(n)$ is $o(g(n))$ if $\forall c > 0$, $\exists n_0 \geq 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

Little-omega

- $f(n)$ is $\omega(g(n))$ if $\forall c > 0$, $\exists n_0 \geq 0$ such that $cg(n) \leq f(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation



Big-Oh

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

Big-Omega

- *f(n)* is Ω(g(*n*)) if *f(n)* is asymptotically **greater than or equal** to *g(n)*

Big-Theta

- *f(n)* is Θ(g(*n*)) if *f(n)* is asymptotically **equal** to *g(n)*

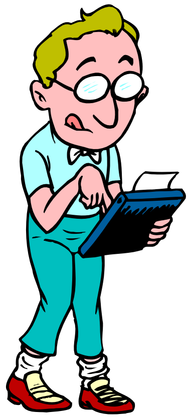
little-oh

- *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 *f(n)* is asymptotically **strictly greater** than *g(n)*

Math you need to review



- Logarithms and Exponents

$$\log_b a = c \quad \text{if} \quad a = b^c$$

properties of logarithms:

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

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

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

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

properties of exponentials:

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

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

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

$b = a^{\log_a b}$

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