Big-Oh Notation

- To simplify the running time estimation, for a function *f(n),* we ignore the constants and lower order terms.

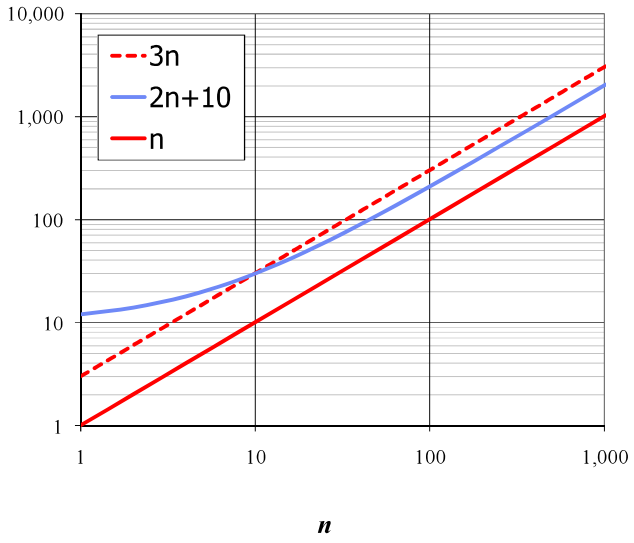
Example: $10n^3+4n^2-4n+5$ is $O(n^3)$.

Big-Oh Notation (Formal Definition)

- 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

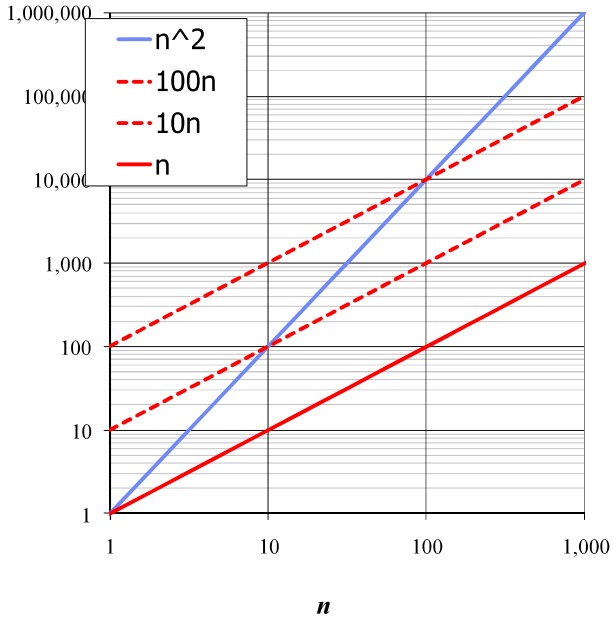
  $0 \leq f(n) \leq cg(n)$ for $n \geq n_0$

- 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
  - n^2 is $O(n^2)$.



More Big-Oh Examples

- $7n-2$

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

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ 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 $n_0 = 21$

- $3 \log n + 5$

  $3 \log n + 5$ 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 $n_0 = 2$

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

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  - Drop lower-order terms
  - Drop constant factors
- 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 $O(3n)$"

Growth Rate of Running Time

- Consider a program with time complexity $O(n^2)$.
  - for the input of size n, it takes 5 seconds.
  - If the input size is doubled (2n).
  - then it takes 20 seconds.

- Consider a program with time complexity O(n).
  - for the input of size n, it takes 5 seconds.
  - If the input size is doubled (2n).
  - then it takes 10 seconds.

- Consider a program with time complexity $O(n^3)$.
  - for the input of size n, it takes 5 seconds.
  - If the input size is doubled (2n) .
  - then it takes 40 seconds.

Asymptotic Algorithm Analysis

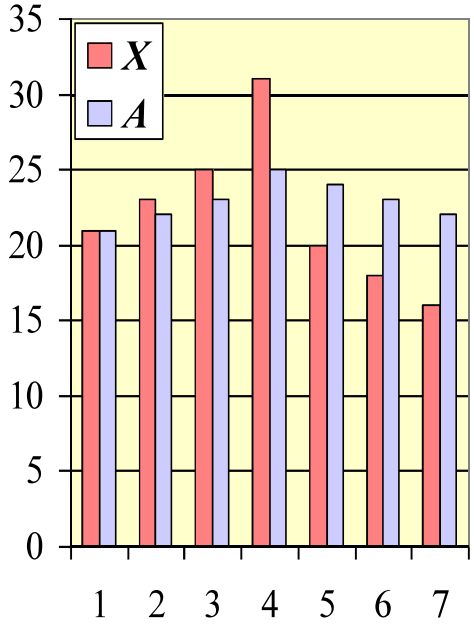
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $6n - 1$ primitive operations
  - 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.

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 $(i + 1)$ elements of X:

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

- 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.

```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #operations
    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
```

Arithmetic Progression

- The running time of ***prefixAverages1*** is $\boldsymbol{O}(1 + 2 + \ldots + \boldsymbol{n})$
- The sum of the first $\boldsymbol{n}$ integers is $\boldsymbol{n}(\boldsymbol{n} + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in T(n)=$\boldsymbol{O}(\boldsymbol{n}^2)$ time

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum.

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X          #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
```

- Algorithm ***prefixAverages2*** runs in T(n)=$O(n)$ time

Exercise: Give a big-Oh characterization

```
Algorithm Ex1(A, n)
    Input an array X of n integers
    Output the sum of the elements in A

    s ← A[0]
    for i ← 0 to n – 1 do
            s ← s + A[i]

    return s
```

Exercise: Give a big-Oh characterization

```
Algorithm Ex2(A, n)
    Input an array X of n integers
    Output the sum of the elements at even cells in A

    s ← A[0]
    for i ← 2 to n – 1 by increments of 2 do
            s ← s + A[i]
    return s
```

Exercise: Give a big-Oh characterization

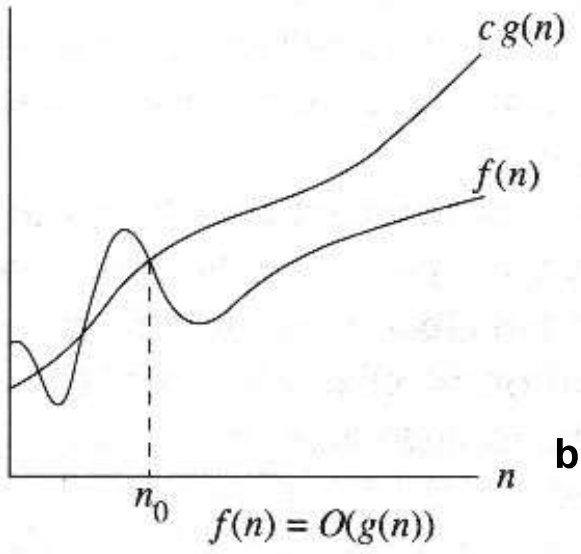
```
Algorithm Ex3(A, n)
    Input an array X of n integers
    Output the sum of the prefix sums  A
    s ← 0
    for i ← 0 to n − 1 do
            s ← s + A[0]
            for j← 1 to i do
                    s ← s + A[j]
    return s
```

Asymptotic Notation

- Convenient for describing worst-case running time function T(n)

- 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 is asymptotically strictly greater than g(n)

big-oh (O)-notation



big-Oh (O)**-notation:**

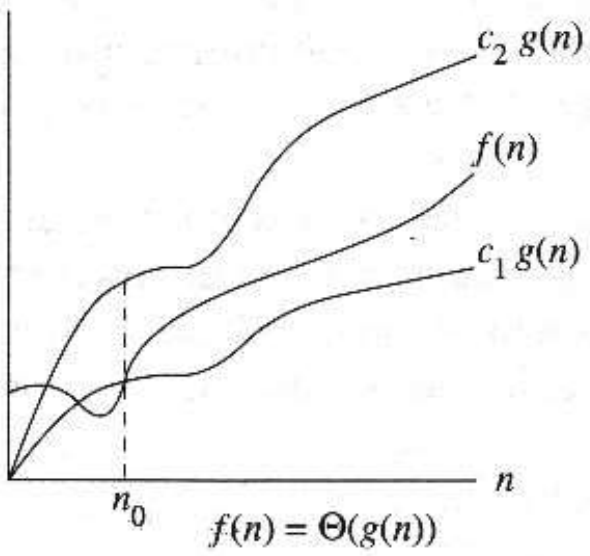
f(n) is O(g(n)) if there is a constant c > 0 and an integer constant n_0>=1 such that 0=<f(n) =< c g(n) for n >= n_0

- a upper bound that is asymptotically tight.

big-Theta (Θ)-notation

Θ-notation

- asymptotic tight bound
- $\Theta(g(n))$ = { f(n) : there exists positive constants c_1, c_2,
  - and n_0 ,s.t. $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$
- $\Theta(g(n))$ is a set
- $f(n) = \Theta(g(n))$
  - is an abuse of '='
  - really
    - f(n) is a member of $\Theta(g(n))$
    - $f(n) \in \Theta(g(n))$
- asymptotically nonnegative
  - f(n) is nonnegative for large enough n



Example

Show that:

$$\tfrac{1}{2}n^2 - 3n = \Theta(n^2)$$

- determine c1, c2 and n0 s.t.

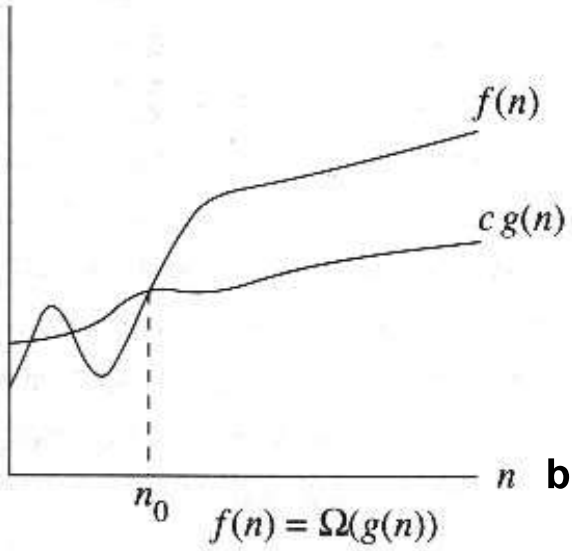
$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

- for an $n \geq n0$.
- dividing all by n^2:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- this holds for
  - $c2 \geq 1/2$, so choose c2 = 1/2
  - $n \geq 7$, so choose n0 = 7
  - $c1 \leq 1/14$, so choose c1 = 1/14
  - i.e., some choice exists.

big-Omega (Ω)-notation



big-Omega (Ω)**-notation:**

f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant n_0>=1 such that f(n) >= c g(n) >=0 for n >= n_0

• a lower bound and that is asymptotically tight.

Little-oh & Little-omeha

- **little-oh (o)-notation**
  - f(n) is o(g(n)) if, for any constan c > 0, there is an integer constant n_0 >= 0 such that 0=<f(n) < c g(n) for n >= n_0
  - a upper bound that is not asymptotically tight.

  - similar to O() but
    - $2n^2 = O(n^2)$ is tight
    - $2n = O(n^2)$ is not tight
  - main difference
    - O() - some constant c
    - o() - for all constants c
  - note
    - $2n = o(n^2)$
    - $2n^2 \neq o(n^2)$

- **little-omega (ω)-notation**
  - f(n) is ω(g(n)) if, for any constant c > 0, there is an integer constant n_0 >= 0 such that f(n) > c g(n) >=0 for n >= n_0
  - a lower bound that is not asymptotically tight.

  - similar to $\Omega()$ but
  - $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$
  - note
    - $n^2/2 = \omega(n)$
    - $n^2/2 \neq \omega(n^2)$

The family of Bachmann–Landau notations

Notation	Name	Intuition	As $n \to \infty$, eventually...	Definition
$f(n) \in O(g(n))$	Big Omicron; Big O; Big Oh	f is bounded above by g (up to constant factor) asymptotically	$\|f(n)\| \leq g(n) \cdot k$ for some k	$\exists k > 0, n_0 \; \forall n > n_0 \; \|f(n)\| \leq \|g(n) \cdot k\|$ or $\exists k > 0, n_0 \; \forall n > n_0 \; f(n) \leq g(n) \cdot k$
$f(n) \in \Omega(g(n))$ (Older math papers sometimes use this in the weaker sense that f=o(g) is false)	Big Omega	f is bounded below by g (up to constant factor) asymptotically	$\|f(n)\| \geq g(n) \cdot k$ for some k	$\exists k > 0, n_0 \; \forall n > n_0 \; g(n) \cdot k \leq \|f(n)\|$
$f(n) \in \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$\|g(n)\| \cdot k_1 \leq \|f(n)\| \leq \|g(n)\| \cdot k_2$ for some k_1, k_2	$\exists k_1, k_2 > 0, n_0 \; \forall n > n_0 \; \|g(n) \cdot k_1\| < \|f(n)\| < \|g(n) \cdot k_2\|$
$f(n) \in o(g(n))$	Small Omicron; Small O; Small Oh	f is dominated by g asymptotically	$\|f(n)\| \leq \|g(n)\| \cdot \varepsilon$ for every ε	$\forall \varepsilon > 0 \; \exists n_0 \; \forall n > n_0 \; \|f(n)\| \leq \|g(n) \cdot \varepsilon\|$
$f(n) \in \omega(g(n))$	Small Omega	f dominates g asymptotically	$\|f(n)\| \geq \|g(n)\| \cdot k$ for every k	$\forall k > 0 \; \exists n_0 \; \forall n > n_0 \; \|g(n) \cdot k\| \leq \|f(n)\|$
$f(n) \sim g(n)$	on the order of; "twiddles"	f is equal to g asymptotically	$\|f(n)/g(n) - 1\| < \varepsilon$ for every ε	$\forall \varepsilon > 0 \; \exists n_0 \; \forall n > n_0 \; \|f(n)/g(n) - 1\| < \varepsilon$

Comparison of Functions

- Transitivity:
$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n))$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n))$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n))$$

- Reflexivity:
$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose Symmetry:
$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n))$$