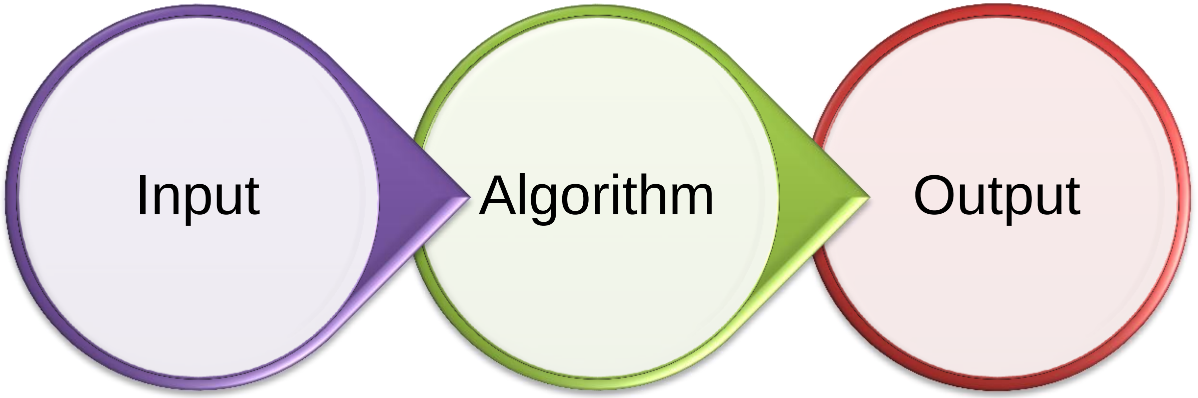
Elementary maths for GMT

Algorithm analysis

Part I

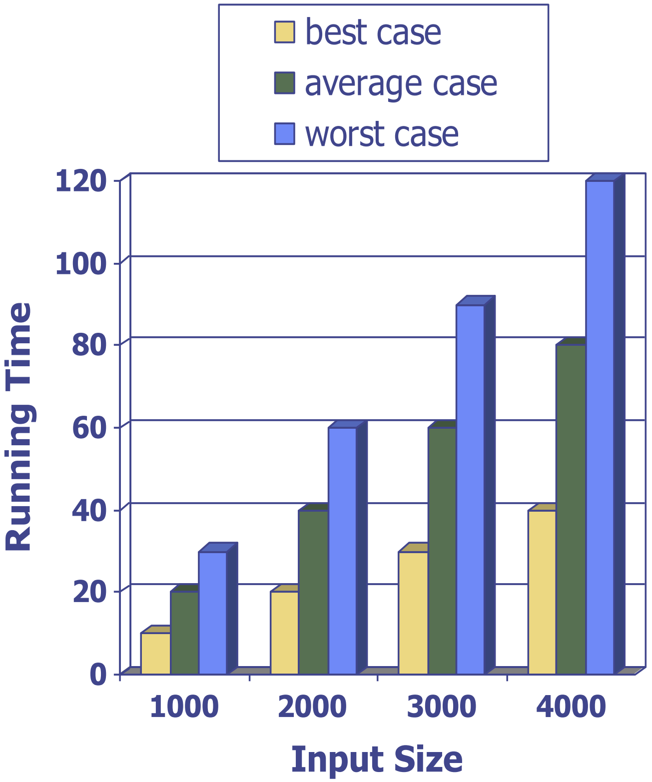
Algorithms

- An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time
- Most algorithms transform input objects into output objects



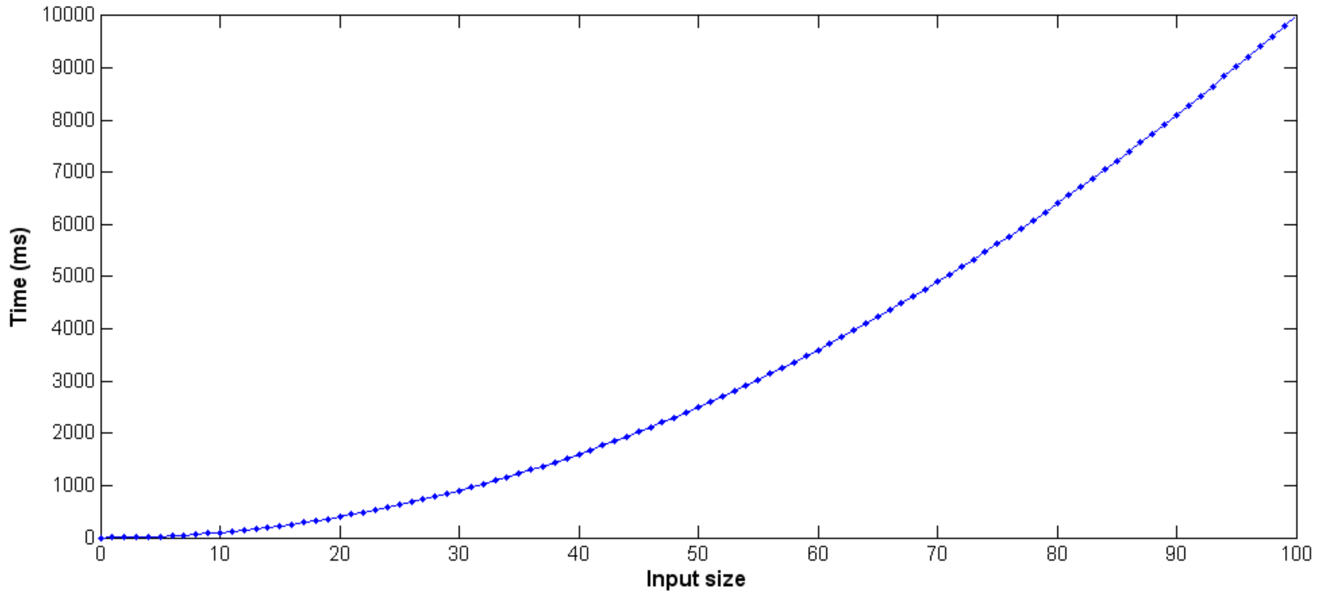
Running time

- The running time of an algorithm typically grows with the input size
- Average case time is often difficult to determine mathematically
- To define the running time, we focus on the worst case scenario
  - Easier to determine
  - Crucial and relevant to applications such as games, finance, robotics etc.



Experimental studies

- Write a program implementing your algorithm
- Run the program with inputs of varying size and composition
- Use function like `clock()` 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
- Results may not be indicative of the running time on other inputs not included in the experiments
- 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, denoted *n*
- Takes into account all possible inputs
- Allows us to evaluate the cost of an algorithm independently from the hardware/software environment

Pseudo-code

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

Pseudo-code example

- How to compute the max value in an array of integers

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

        currentMax ← A [0]
        for i ← 1 to n – 1 do
                if A[i] > currentMax  then
                        currentMax ← A[i]
        return currentMax
```

Pseudo-code details

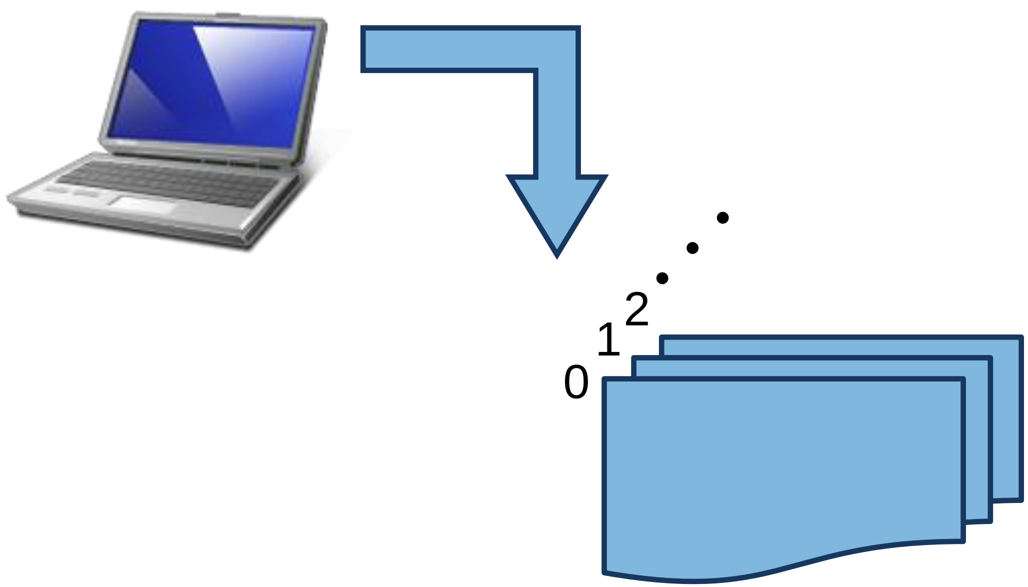
- Control flow
  - if ... then ... [else ...]
  - while ... do ...
  - repeat ... until ...
  - for ... do ...
  - Indentation replaces braces
- Method declaration
  - Algorithm *method* (*arg* [, *arg* ...])

    Input ...

    Output ...
- Method call
  - *var.method* (*arg* [, *arg*...])
- Return value
  - return *expression*
- Expressions
  - ← assignment (like = in Java/C#/C++)
  - = Equality testing (like == in Java/C#/C++)
  - Superscripts (e.g. n^2) and other mathematical formatting allowed

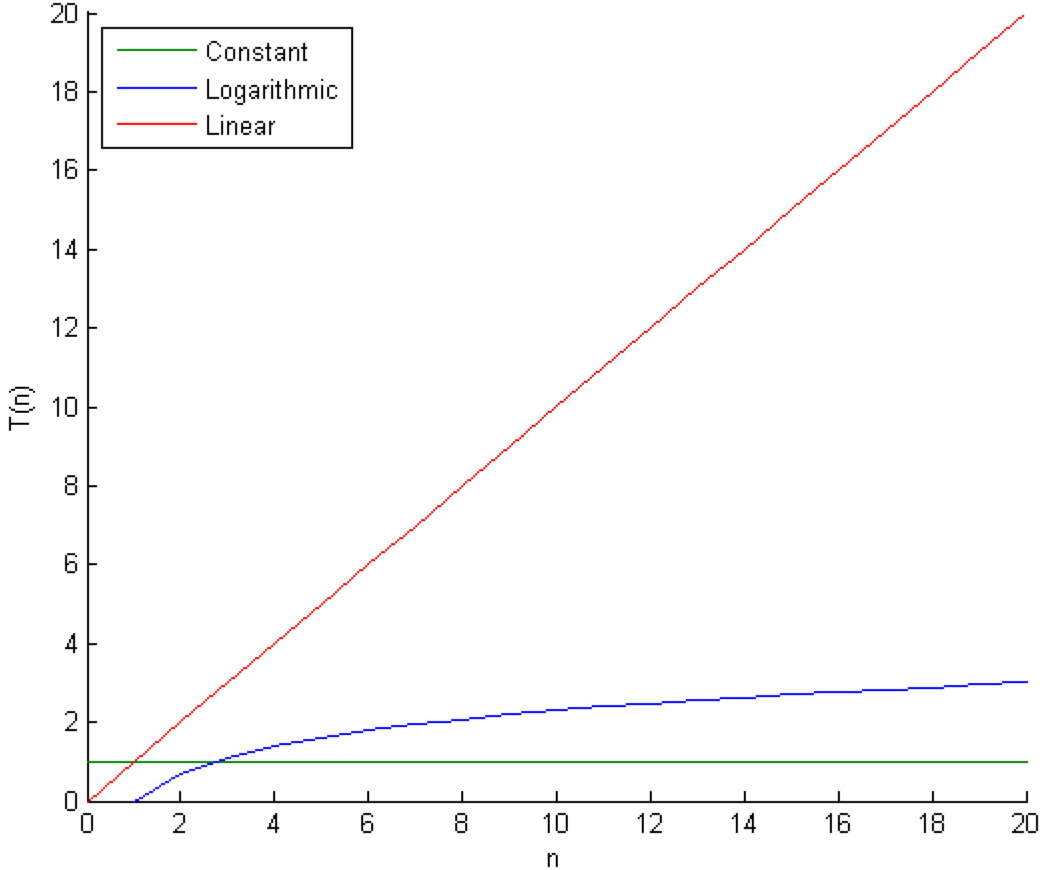
Random Access Machine (RAM)

- A CPU that "executes" the pseudo-code
- A potential 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



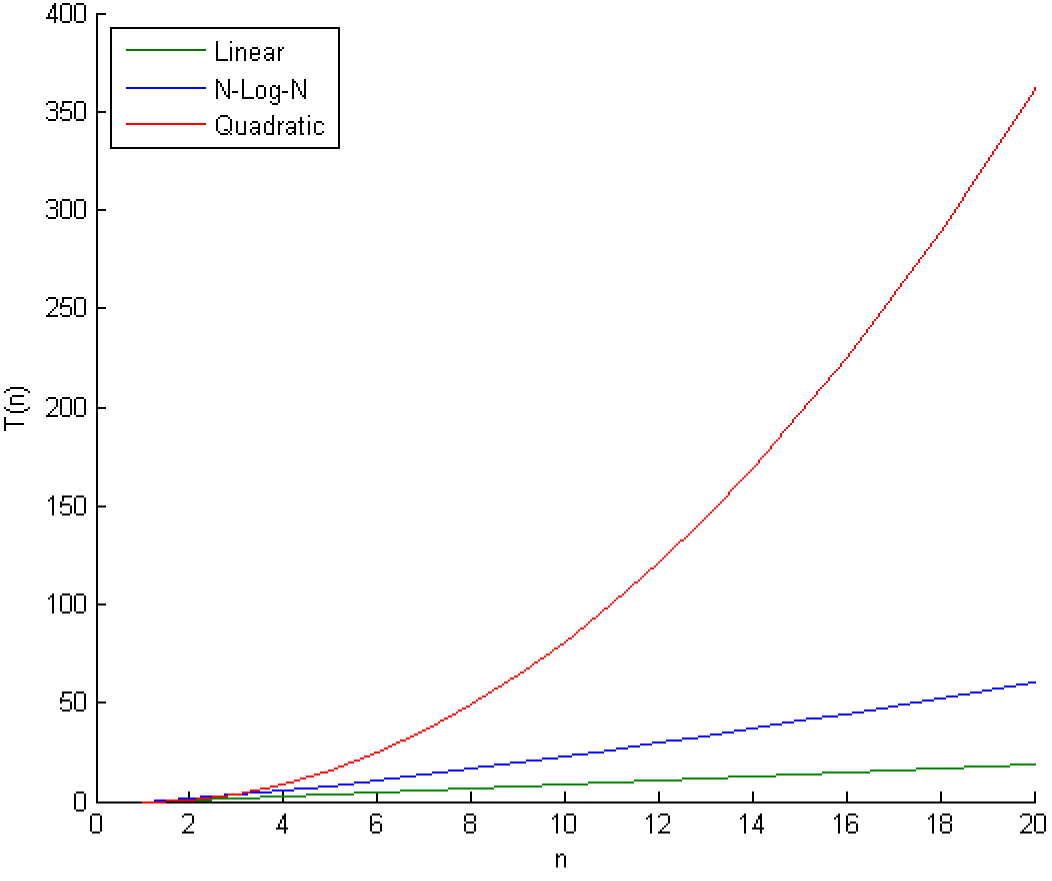
Important functions

- Eight 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
  - Factorial ≈ n!



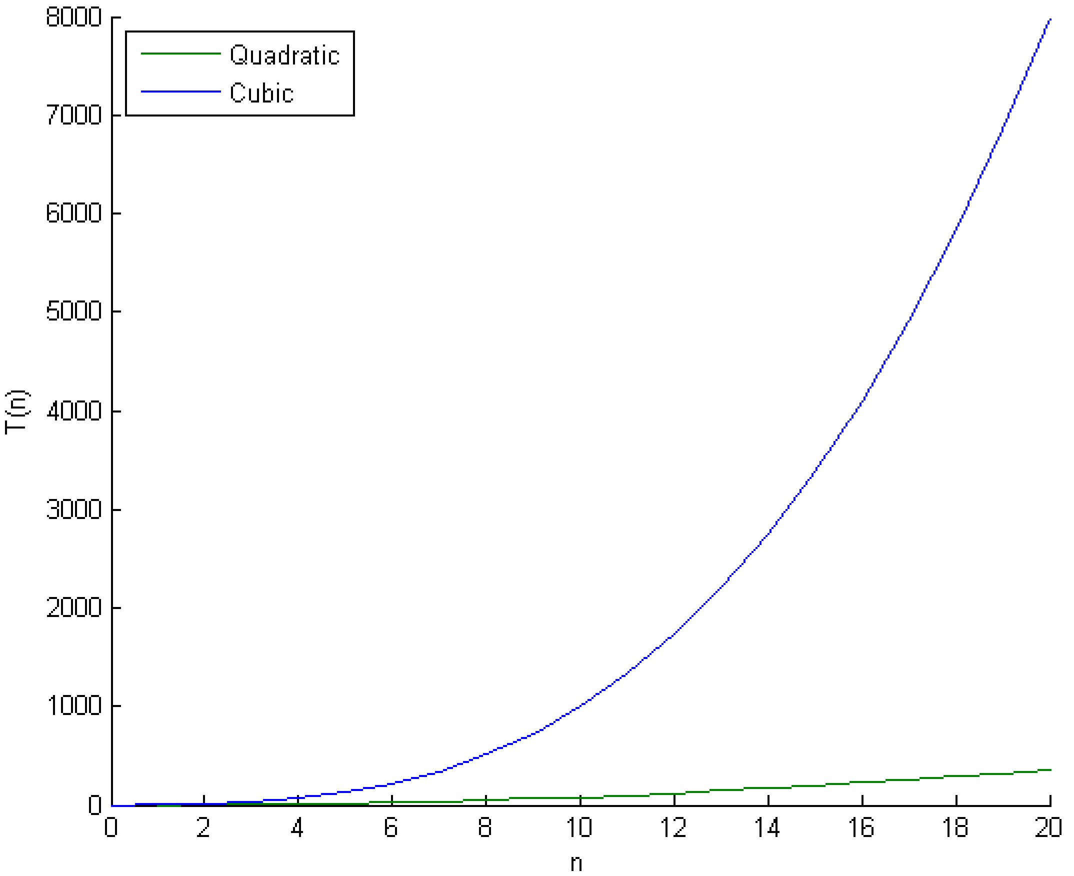
Important functions

- Eight 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
  - Factorial ≈ n!

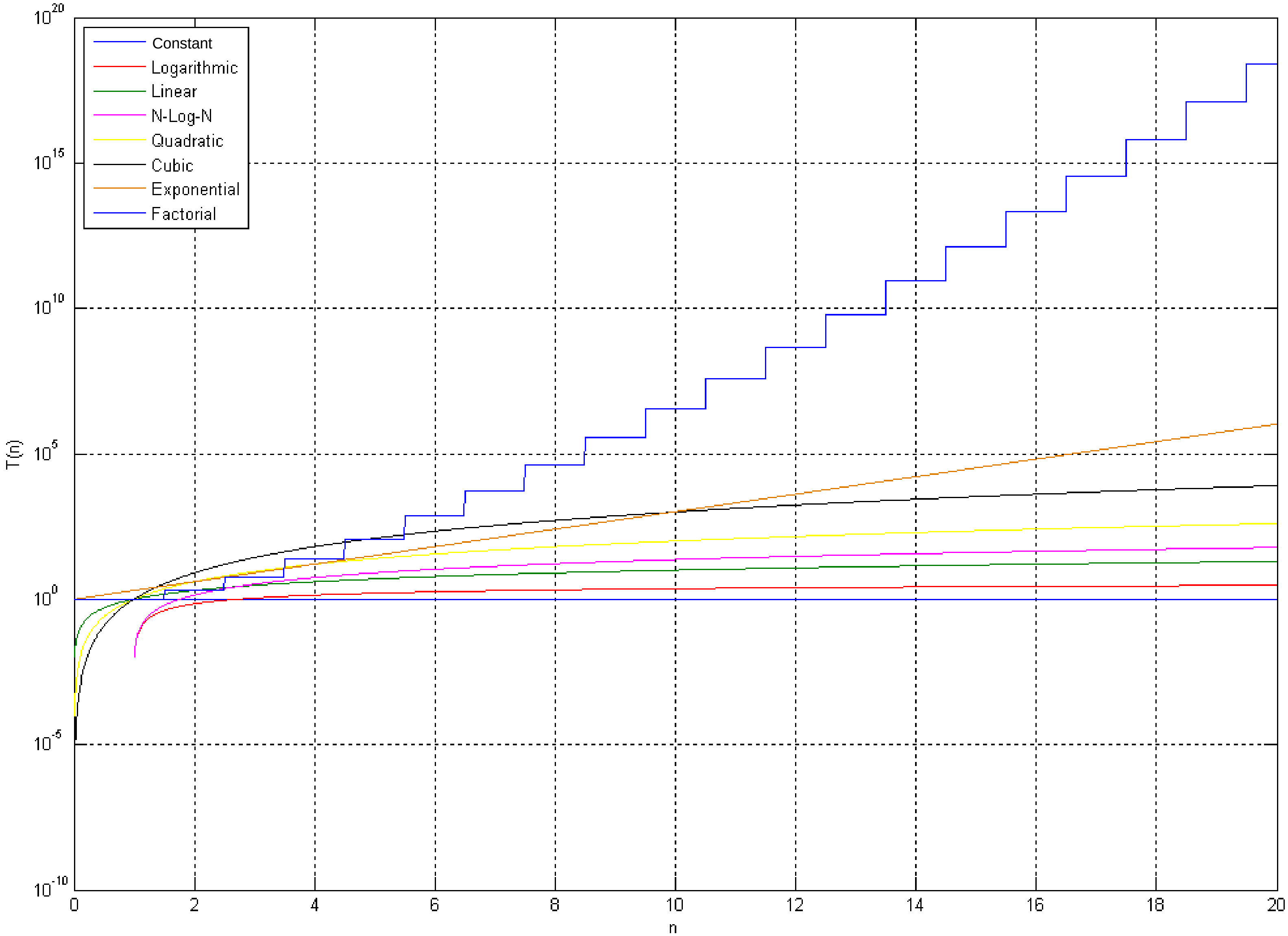


Important functions

- Eight 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
  - Factorial ≈ n!



Important functions



Necessary math

- Summations
- Logarithms and exponentials
  - Properties of logarithms

    $^{b}\log(xy) = {}^{b}\log x + {}^{b}\log y$

    $^{b}\log(x/y) = {}^{b}\log x - {}^{b}\log y$

    $^{b}\log x^{a} = a\,{}^{b}\log x$

    $^{b}\log a = {}^{x}\log a \,/\, {}^{x}\log b$
  - Properties of exponentials

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

    $a^{bc} = \left(a^{b}\right)^{c}$

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

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

    $b^{c} = a^{c\,{}^{a}\log b}$

Primitive operations

- Basic computations performed by an algorithm
- Identifiable in pseudo-code
- 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 pseudo-code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm ***arrayMax*(*A*, *n*)**	
	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n-1)$
$currentMax \leftarrow A[i]$	$2(n-1)$
{ increment counter i }	$2(n-1)$
return $currentMax$	1
	Total: $8n - 3$

Estimating running time

- The algorithm `arrayMax` executes $8n - 3$ primitive operations in the worst case
- If we define
  - ***a*** as the time for the fastest primitive operation
  - ***b*** as the time for the slowest primitive operation
  - ***T(n)*** as the worst-case time of `arrayMax`
- Then, $a(8n - 3) \leq T(n) \leq b(8n - 3)$
- Hence, the running time ***T(n)*** is bounded by two linear functions

Growth rate of running time

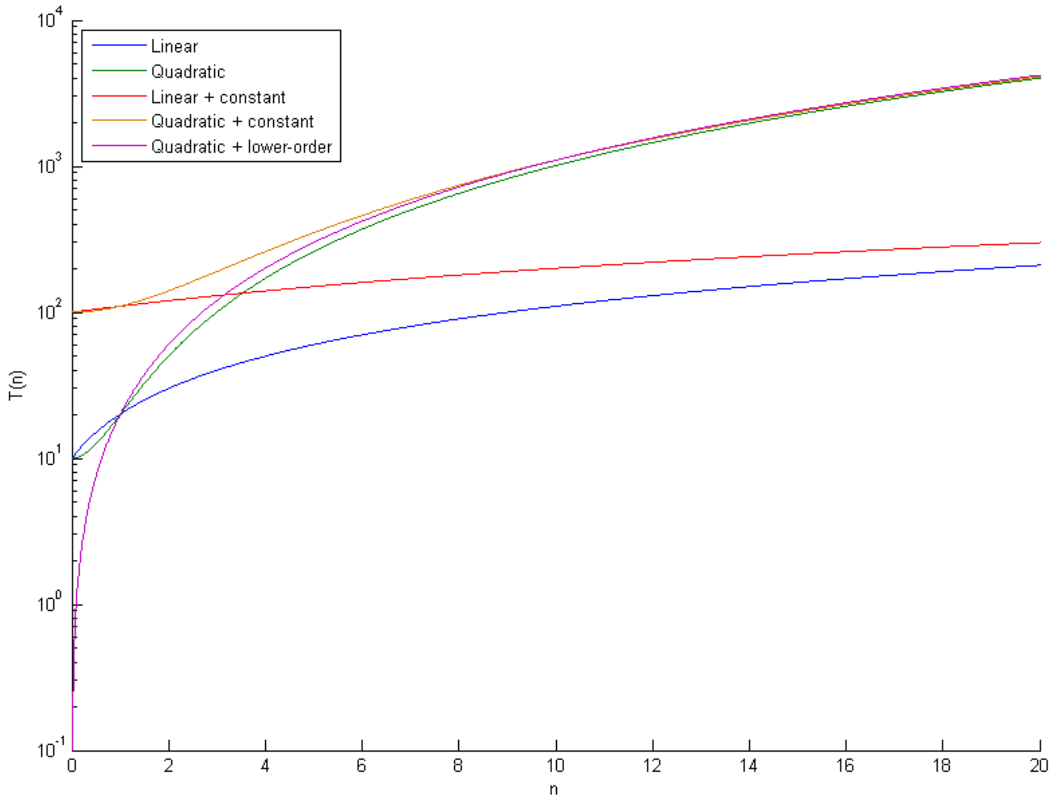
- Changing the hardware / software environment
  - affects ***T(n)*** by a constant factor, but
  - does not alter the *growth rate* of ***T(n)***

- The linear growth rate of the running time ***T(n)*** is an intrinsic property of the algorithm `arrayMax`

Constant factors

- The growth rate is not affected by
  - constant factors
  - lower-order terms

- Examples
  - $10n + 10$ is a linear function
    - what if we replace $+ 10$ by $+ 10^2$?
  - $10n^2 + 10$ is a quadratic function
    - what if we replace $+ 10$ by $+ 10^2$?
    - what if we replace $+ 10$ by $+10n$?

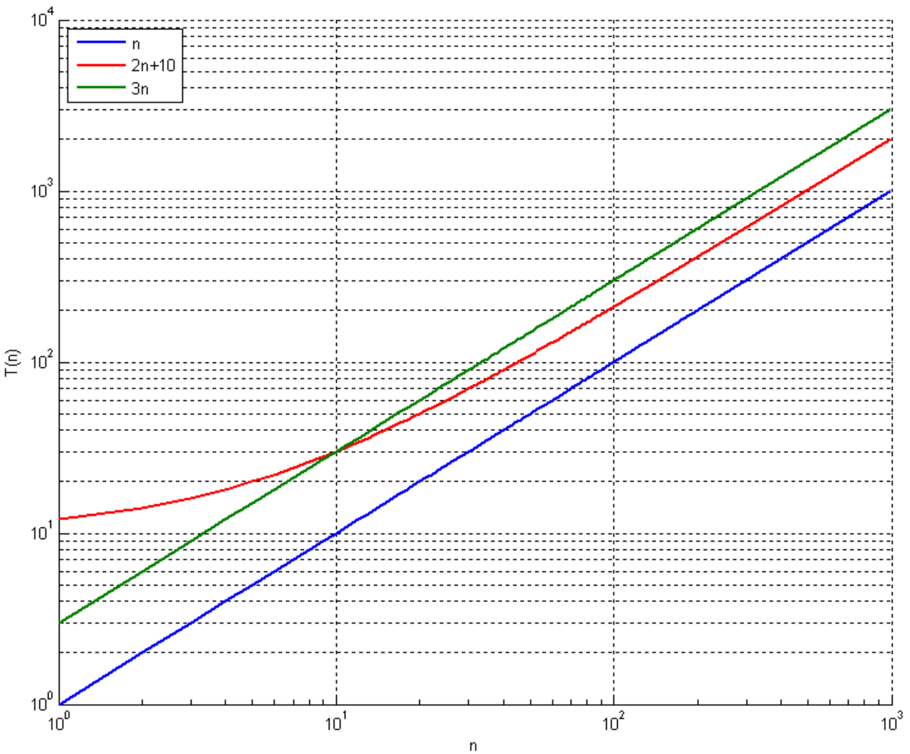


The Big-Oh

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a positive constant c and an integer constant $n_0 \geq 1$ such that

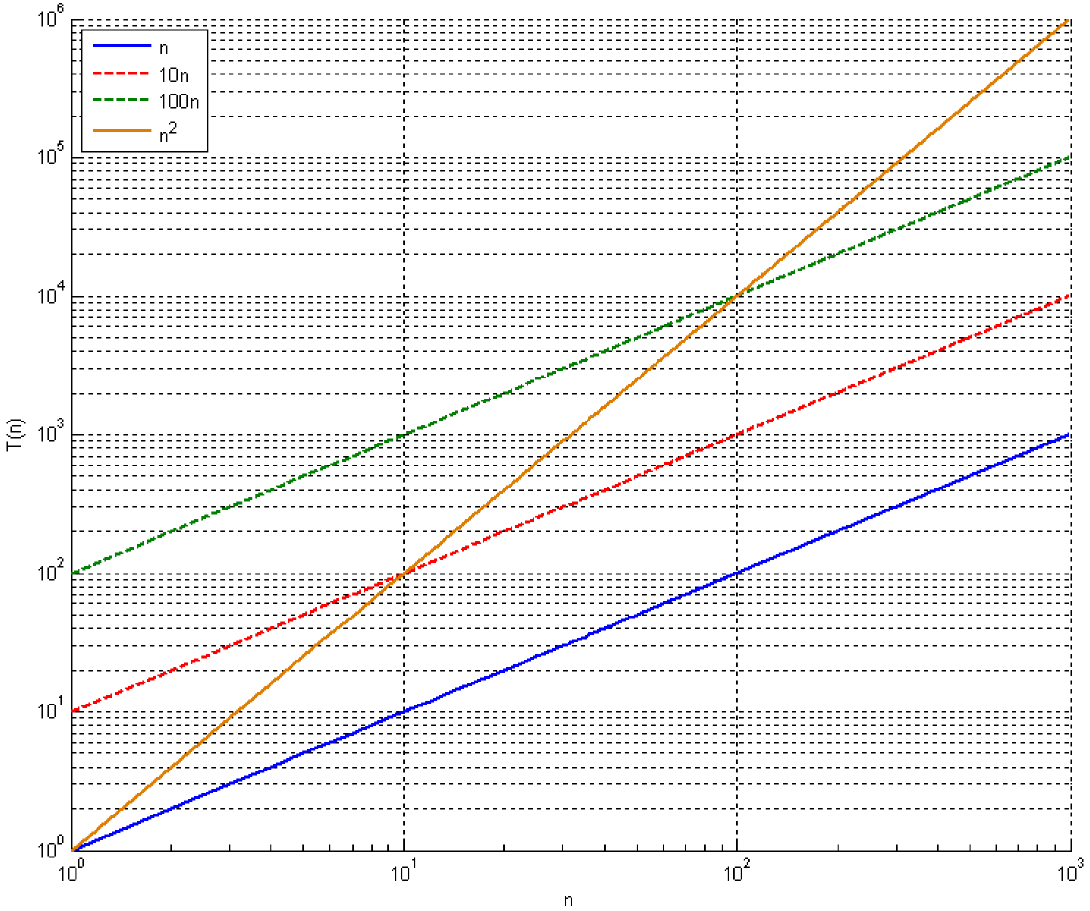
$$f(n) \leq cg(n) \text{ for all } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq c\,n$
  - $(c - 2)\,n \geq 10$
  - $n \geq 10 \;/\; (c - 2)$
  - Pick $c = 3$ and $n_0 = 10$ (or $c = 12$ and $n_0 = 1$, ...)



The Big-Oh

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



More Big-Oh examples

- $7n - 2$ is $O(n)$
  - We need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c \cdot n$ for $n \geq n_0$
  - This is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - We need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  - This is true for $c = 4$ and $n_0 = 21$

- $3\log n + 5$ is $O(\log n)$
  - We need $c > 0$ and $n_0 \geq 1$ such that $3\log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  - This is true for $c = 8$ and $n_0 = 10$

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** $O(f(n))$
$\mathrm{g}(n)$ 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 $O(n^d)$, so:
  - Drop the lower-order terms
  - Drop the constant factors

- Use the smallest possible class of functions
  - We say that "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class
  - We say that "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic algorithm analysis

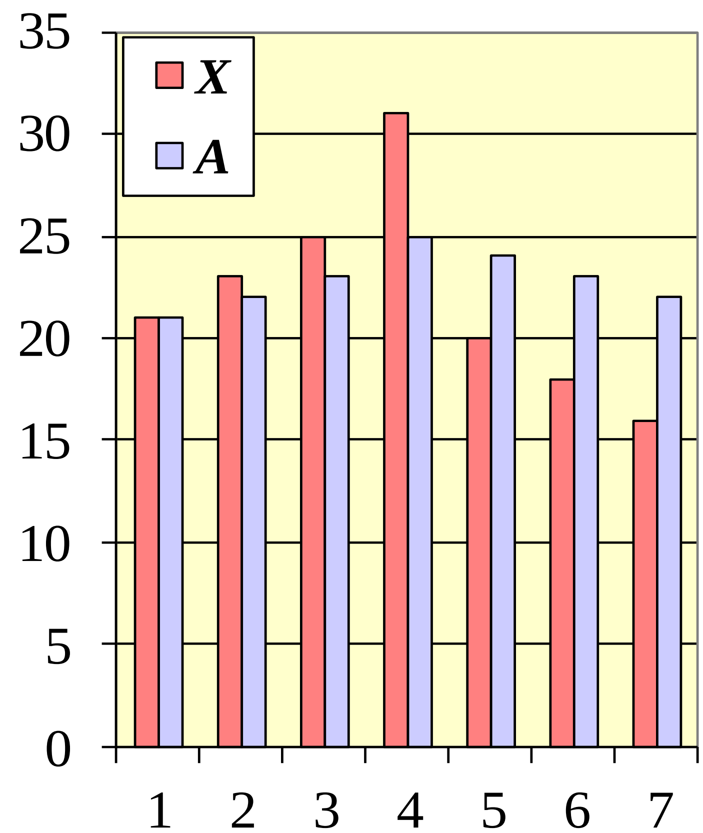
- The asymptotic analysis of an algorithm determines the running time in Big-Oh notation
- To perform the asymptotic analysis
  - We find the worst case number of primitive operations executed as a function of the input size
  - We express this function with Big-Oh notation
- Example
  - We determine that algorithm `arrayMax` executes at most $8n - 3$ 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 primitives 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 the average of the first *(i+1)* elements of ***X***:

$$A[i] = \frac{X[0] + X[1] + \cdots + X[i]}{i+1}$$

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



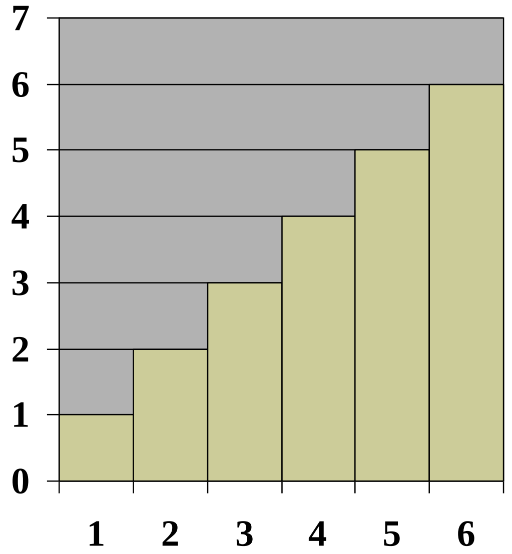
Prefix averages – Quadratic example

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

Algorithm	# operations (after drop)
Algorithm *prefixAveragesQuad*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	
$A \leftarrow$ 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 example

- Arithmetic progression
  - The running time of `prefixAveragesQuad` is $O(1+2+\cdots+n)$
  - The sum of the first n integers is $n(n+1)/2$
    - There is a simple visual proof of this fact
  - Thus, the algorithm `prefixAveragesQuad` runs in $O(n^2)$ time
    - recall that lower-order terms can be disregarded (n / 2)



Prefix averages – Linear example

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

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

- Algorithm `prefixAveragesLinear` runs in $O(n)$ time

Relatives of Big-Oh

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

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

- 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

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

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 $\boldsymbol{\Omega}(g(n))$ 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 $g(n)$
- In Big-Omega and Big-Theta notation we also omit constants and lower-order terms

Examples of relatives of Big-Oh

- $5n^2$ is $\mathbf{\Omega}(n^2)$
  - $f(n)$ is $\mathbf{\Omega}(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
  - True for $c = 5$ and $n_0 = 1$
- $5n^2$ is $\mathbf{\Omega}(n)$
  - $f(n)$ is $\mathbf{\Omega}(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
  - True for $c = 1$ and $n_0 = 1$
- $5n^2$ is $\Theta(n^2)$
  - $f(n)$ is $\Theta(g(n))$ if it is $\mathbf{\Omega}(n^2)$ 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 $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
  - True for $c = 5$ and $n_0 = 1$