IT623 Algorithms & Data Structures

Asymptotic Notation $\boldsymbol{\Theta}, \boldsymbol{O}, \boldsymbol{\Omega}$

Analysis of Algorithm for Time and Space

- To analyze an algorithm means:
  - developing a formula for predicting how fast an algorithm is, based on the size of the input (time complexity), and/or
  - developing a formula for predicting how much memory an algorithm requires, based on the size of the input (space complexity)
- Usually **time** is our biggest concern
  - Most algorithms require a fixed amount of space **and/or**
  - Memory is not as expensive as time

Problem Size Matters in Calculating Time or Space Complexity

- Time and Space complexity will depend on the Problem Size, why?
  - If we are searching an array, the “size” of the input array could decide who long it will take to go through the entire array
  - If we are merging two arrays, the “size” could be the sum of the two array sizes
  - If we are computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- We choose the “size” to be a parameter that determines the actual time (or space) required
  - It is *usually* obvious what this parameter is
  - Sometimes we need two or more parameters

Characteristic Operation

- In computing time complexity, one good approach is to count characteristic operations
  - What a “characteristic operation” is depends on the particular problem
  - If searching, it might be comparing two values
  - If sorting an array, it might be:
    - comparing two values
    - swapping the contents of two array locations
    - both of the above
  - Sometimes we just look at **how many times the innermost loop is executed**

How many times innermost loop will execute?

```
Algorithm Array_Sum(A):
    for i := 0 to n-1  do
        sum := sum + A[i][j]
    return sum;
```

```
Algorithm Array2D_Sum(A):
    for i := 0 to n-1  do
        for j := 0 to n-1 do
            sum := sum + A[i][j]
    return sum;
```

Sequential Search Analysis

Input: An array **A** storing **n** items, target **x**

Output: true if x is in A, false otherwise

Algorithm Sequential_Search(A, x):

```
for i := 0 to n-1  do
    if (A[i] = x)
        return true;
    return false;
```

- Identify “Characteristic Operations” in this code?
- How many times will the loop actually execute?
  - that depends on how many times comparison operations are require to execute

Exact vs Simplified Analysis

- It is sometimes possible, in assembly language, to compute exact time and space requirements
  - We know exactly how many bytes and how many cycles each machine instruction takes
  - For a problem with a known sequence of steps (factorial, Fibonacci), we can determine how many instructions of each type are required
- However, often the exact sequence of steps cannot be known in advance
  - The steps required to sort an array depend on the actual numbers in the array (which we do not know in advance)

Exact vs Simplified Analysis

- In a higher-level language (such as Java), we do not know how long each operation takes
  - Which is faster, x < 10 or x <= 9 ?
  - We don’t know exactly what the compiler does with this
  - The compiler almost certainly optimizes the test anyway (replacing the slower version with the faster one)
- In a higher-level language we cannot do an exact analysis
  - Our timing analyses will use major oversimplifications
  - Nevertheless, we can get some very useful results

Average, Best or Worst Case

- Usually we would like to find the average time to perform an algorithm
- However,
  - Sometimes the “average” isn’t well defined
    - Example: Sorting an “average” array
      - Time typically depends on how out of order the array is
      - How out of order is the “average” unsorted array?
  - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the worst (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The best (fastest) case is seldom of interest

Sequential Search Analysis – Best and Worst Cases

Input: An array **A** storing n items, target **x**

Output: true if x is in A, false otherwise

```
Algorithm Sequential_Search(A, x):
    for i := 0 to n-1  do
        if (A[i] = x)
            return true;
        return false;
```

- Value of x is found at the first index in an array i.e. A[0] → Best Case, required only 1 comparison/iteration

- Value of x is found at the last index in an array i.e. A[n-1] → Worst Case, require n comparison/iteration

- Worse Case Time Complexity: **k*n + c**
  - N is Problem Size i.e. Array Size
  - k time taken for loop execution (increment i, check if i < N-1) and comparison
  - c is time taken for loop init and return
- Best Case Time Complexity: **k*1 + c**
  - This is nothing but a special case of worst case equation

More Examples

Algorithm Is_Array_Sorted(A):

```
for i := 0 to n-1  do
    if A[i] > A[i+1]
        return false
return true;
```

- Best Case → ____________________?
- Worst Case → ____________________?

Algorithm Swap(x, y):

```
temp := x
x := y
y := temp
```

- Best Case → ____________________?
- Worst Case → ____________________?

More Examples

```
Algorithm Array2D_Sum(A):
    for i := 0 to n-1  do
        for j := 0 to n-1 do
            sum := sum + A[i][j]
    return sum;
```

```
Algorithm Array2D_Sum():
    for i := 0 to n-1  do
        for j := 0 to n-1 do
            A[i][j] := i*j
    for i := 0 to n-1  do
        for j := 0 to n-1 do
            sum := sum + A[i][j]
    return sum;
```

Algorithm Analysis

- Represent Algorithm Analysis (i.e. Runtime) as function of input n i.e. f(n) e.g. For Sequential Search f(n) = k*n + c
- what we care about: orders of growth
  - Ex: $0.5n^2$ and $700n^2$ are no different!
  - when input size doubles, the running time quadruples
- constant factors do NOT matter

Constant Factors are Ignored, Why?

- Algo1: $f1(n) = 5n^2$
- Algo2: $f2(n) = n^2$
- $f1(n)/f2(n) = 5n^2/ n^2 = 5$
- If we double the data size i.e. $n = 2n$
- $f1(2n)/f2(2n) = 5(2n^2)/ (2n^2) = 10n^2/ 2n^2 = 5$

- Algo: $f(n) = kn^2+c$
- As $n \rightarrow \infty$, c becomes less and less dominating

- Therefore, n^2 is the dominating factor.

Only Highest Order Term Matters

- We don't need to study algorithms when we want to sort two elements, because different algorithms make no difference
- we care about algorithm performance when the input size *n* is very large
  - Ex: n^2 and $n^2 + n + 2$ are no different, because when n is really large, $n + 2$ is negligible compared to n^2
- only the highest-order term matters

Only Highest Order Term Matters

- $f1(n) = n^2$, $f2(n) = n^2+n+10$
  - With n=10
  - 10) = (100) and
  - f2(10) = (100 + 10 + 10) = 120
  - Difference of 16.67%

  - With n = 50000
  - f1(50000) = (2,500,000,000) and
  - f2(50000) = (2,500,000,000 + 50000 + 10) = 2,500,050,010 ~ 2,500,000,000
  - Difference of 0.002%

What is the time complexity ?

Algorithm Dummy(A):

```
for i := 1 to n step 2*i do
    // do something
```

Algorithm Dummy1(A):

```
for i := n to 1 step –i/2 do
    // do something
```

What is the time complexity?

Algorithm Algo1():

```
for i := 0 to n-1  do
  // do something
for j := 0 to m-1 do
  // do something
```

Algorithm Algo2():

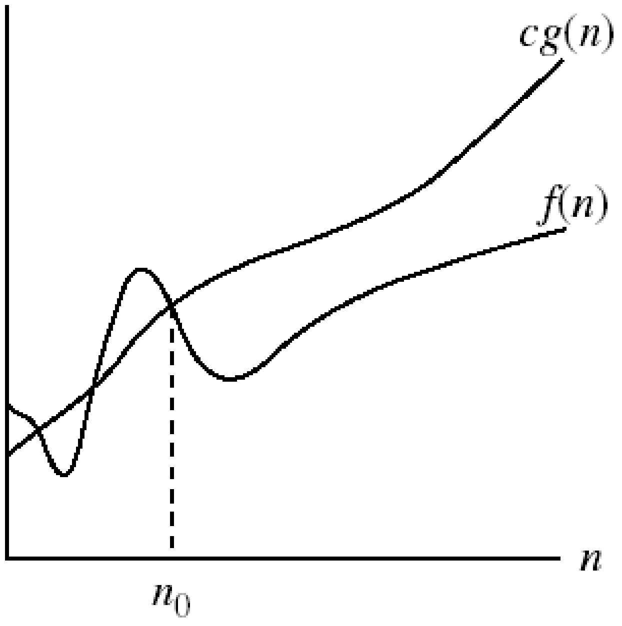
```
for i := 0 to n-1  do
  for j := 1 to n step j*2 do
    sum := sum + A[i][j]
return sum;
```

Big O notation – Simplifying f(n)

- Throwing out the constants is one of *two* things we do in analysis of algorithms
  - By throwing out constants, we simplify $12n^2 + 35$ to just n^2
- Our timing formula is a polynomial, and may have terms of various orders (constant, linear, quadratic, cubic, etc.)
  - We usually discard all but the *highest-order* term
    - We simplify $n^2 + 3n + 5$ to just n^2
- We call this a Big O notation

Asymptotic Notation

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



f(n) ∊ O(g(n))

Usually written as

f(n) = O(g(n)) or

f(n) is O(g(n))

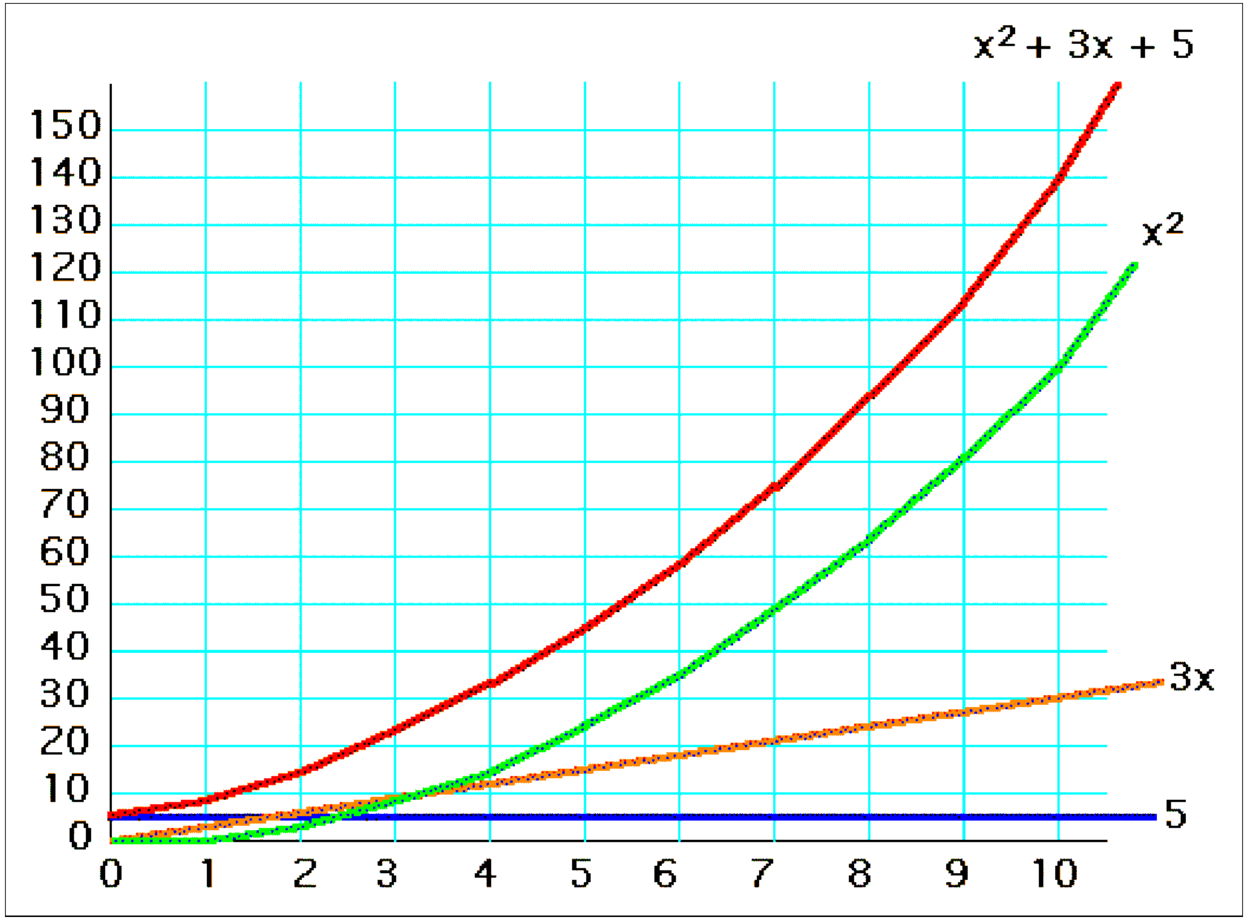
$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Can we Justify Asymptotic Notation?

- Consider $f(n) = n^2 + 3n + 5$ as n varies:

$n = 0$	$n^2 = 0$	$3n = 0$	$f(n) = 5$
$n = 10$	$n^2 = 100$	$3n = 30$	$f(n) = 135$
$n = 100$	$n^2 = 10000$	$3n = 300$	$f(n) = 10{,}305$
$n = 1000$	$n^2 = 1000000$	$3n = 3000$	$f(n) = 1{,}003{,}005$
$n = 10{,}000$	$n^2 = 10^8$	$3n = 3*10^4$	$f(n) = 100{,}030{,}005$
$n = 100{,}000$	$n^2 = 10^{10}$	$3n = 3*10^5$	$f(n) = 10{,}000{,}300{,}005$

$f(n) = x^2 + 3x + 5$, for $x=1..10$



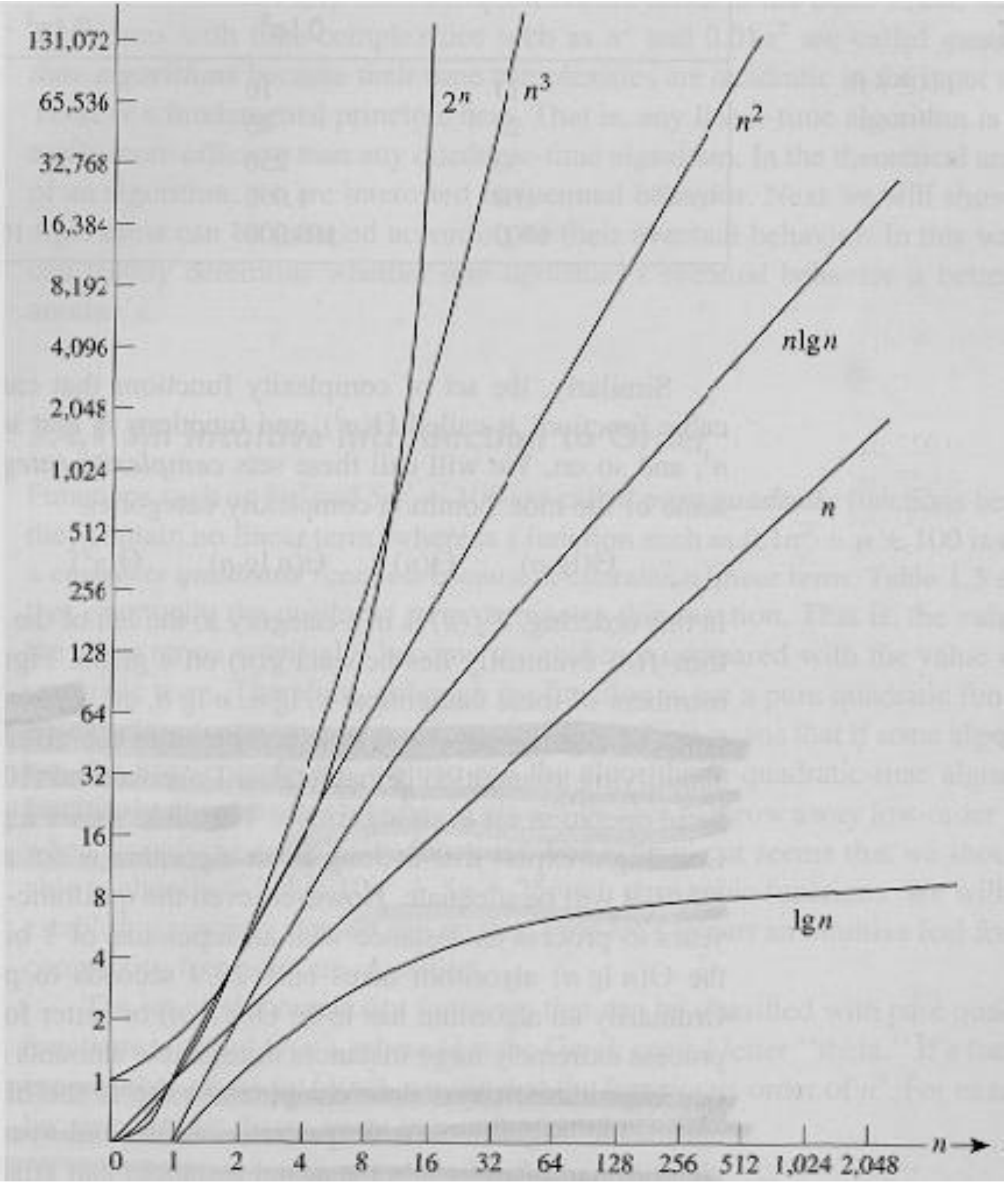
Common time complexities

BETTER

• $O(1)$	constant time
• $O(\log n)$	log time
• $O(n)$	linear time
• $O(n \log n)$	log linear time
• $O(n^2)$	quadratic time
• $O(n^3)$	cubic time
• $O(n^k)$	polynomial time
• $O(2^n)$	exponential time

WORSE

Common Time Complexities



Big O Visual - *O(g(n))* is the set of functions with smaller or same order of growth as *g(n)*

O(N)
3N+100
100N
0.5N-20

O(NlogN)
20NlogN+3N+100
NlogN+100N
100NlogN-0.5N-20

$O(N^2)$
$5N^2 + 3N + 100$
$15N^2 + 100$
$N^2 + 100N\log N - 20$

O(1)
100
5
0.5

$O(N^3)$
$5N^3 + 3N + 100$
$23N^3 + 5N^2 + 100N$
$N^3 + 100N\log N - 20$

Algorithm Examples for Time Complexity

Big O Notation	Name	Example(s)
O(1)	Constant	# Odd or Even number, # Swapping two numbers
O(log n)	Logarithmic	# Finding element on sorted array with **binary search**
O(n)	Linear	# Find max element in unsorted array, # Duplicate elements in array with Hash Map
O(n log n)	Linearithmic	# Sorting elements in array with **merge sort**
$O(n^2)$	Quadratic	# Duplicate elements in array **(naïve)**, # Sorting array with **bubble sort**
$O(n^3)$	Cubic	# 3 variables equation solver
$O(2^n)$	Exponential	# Find all subsets
O(n!)	Factorial	# Find all permutations of a given set/string

Examples:

- $2n^2 = O(n^3)$: $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$
- $n^2 = O(n^2)$: $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2 + 1000n = O(n^2)$: $1000n^2 + 1000n \leq 1000n^2 + n^2 = 1001n^2 \Rightarrow c = 1001$ and $n_0 = 1000$
- $n = O(n^2)$: $n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

Exercise

1. $n^2 + 3n + 5 = O(n^2)$: Find c and n_0

[redacted]

- Which of the below Big O notation is correct.

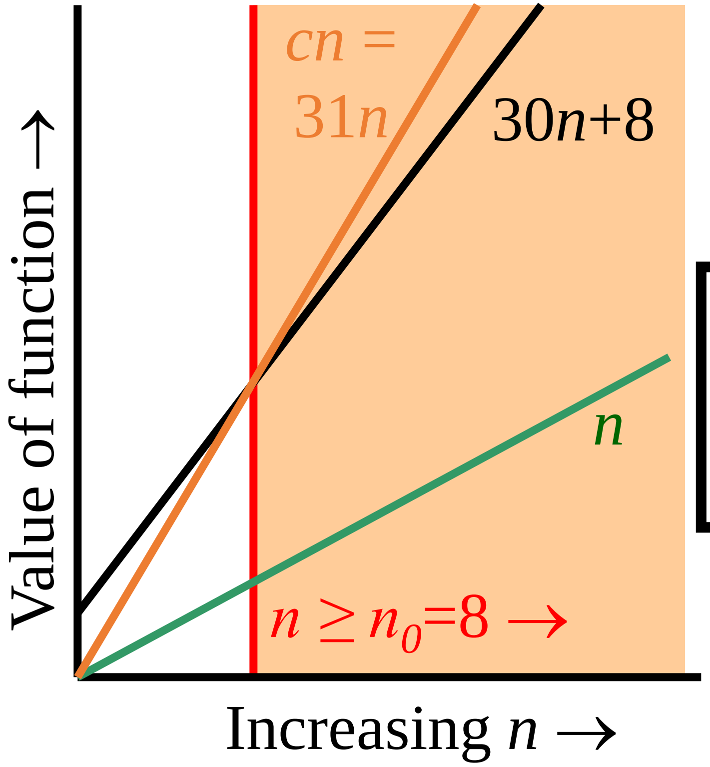
1. $n^4 + 100n^2 + 10n + 50 = O(n^3)$
2. $10n^3 + 2n^2 = O(n^3)$
3. $n^3 - n^2 = O(n^4)$
4. $nlogn + n^2 = O(nlogn)$
5. $10 = O(n)$
6. $1273 = O(1)$

More Examples

- Show that $30n + 8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n + 8 \leq cn$, $\forall n \geq n_0$.
    - Let c=31, n_0 = 8. Then
      $cn = 31n = 30n + n \geq 30n + 8$, so $30n + 8 \leq cn$

Big-O example, graphically

- Note $30n + 8$ isn't less than n *anywhere* (n>0).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of n=8</u>.



$$30n+8 \in O(n)$$

No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \le 100n + n = 101n \le 101n^2$

    for all $n \ge 5$

    $n_0 = 5$ and $c = 101$ is a solution
  - $100n + 5 \le 100n + 5n = 105n \le 105n^2$

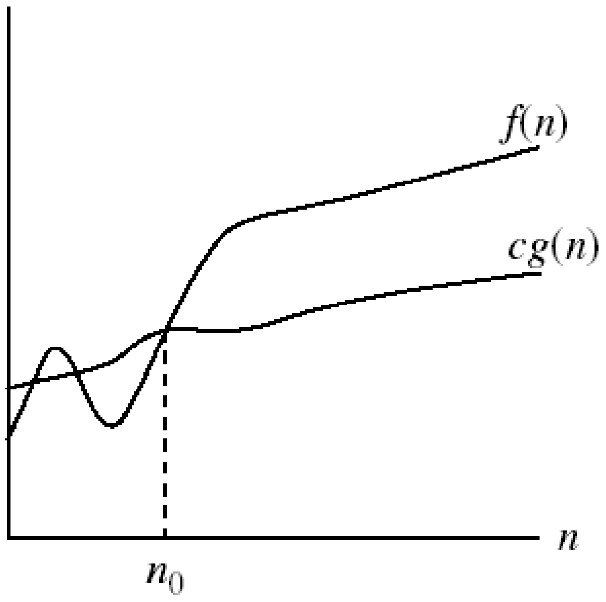
    for all $n \ge 1$

    $n_0 = 1$ and $c = 105$ is also a solution

Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

Big Ω (Omega)

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\} .$$



$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

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

Examples

- $5n^2 = \Omega(n)$

$\exists$ c, n_0 such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

- $100n + 5 \neq \Omega(n^2)$

$\exists$ c, n_0 such that: $0 \leq cn^2 \leq 100n + 5$

$100n + 5 \leq 100n + 5n$ ($\forall$ $n \geq 1$) $= 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

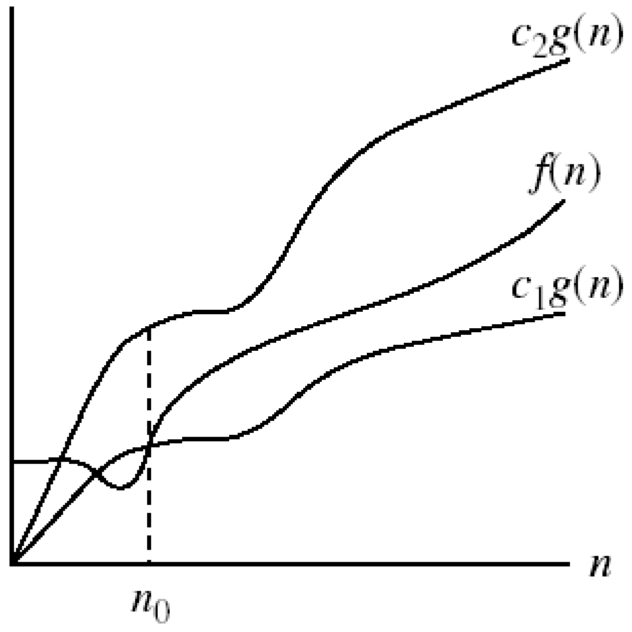
Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

$\Rightarrow$ contradiction: n cannot be smaller than a constant

- $n = \Omega(2n)$
- $n^3 = \Omega(n^2)$
- $n = \Omega(\log n)$

Big Θ (theta)

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

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

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \quad \forall n \geq 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \; (\forall n \geq 2) = \frac{1}{4} n^2 \quad \Rightarrow \quad c_1 = \frac{1}{4}$
- $n \neq \Theta(n^2)$: $c_1 n^2 \leq n \leq c_2 n^2$

$\Rightarrow$ only holds for: $n \leq 1/c_1$

More Examples

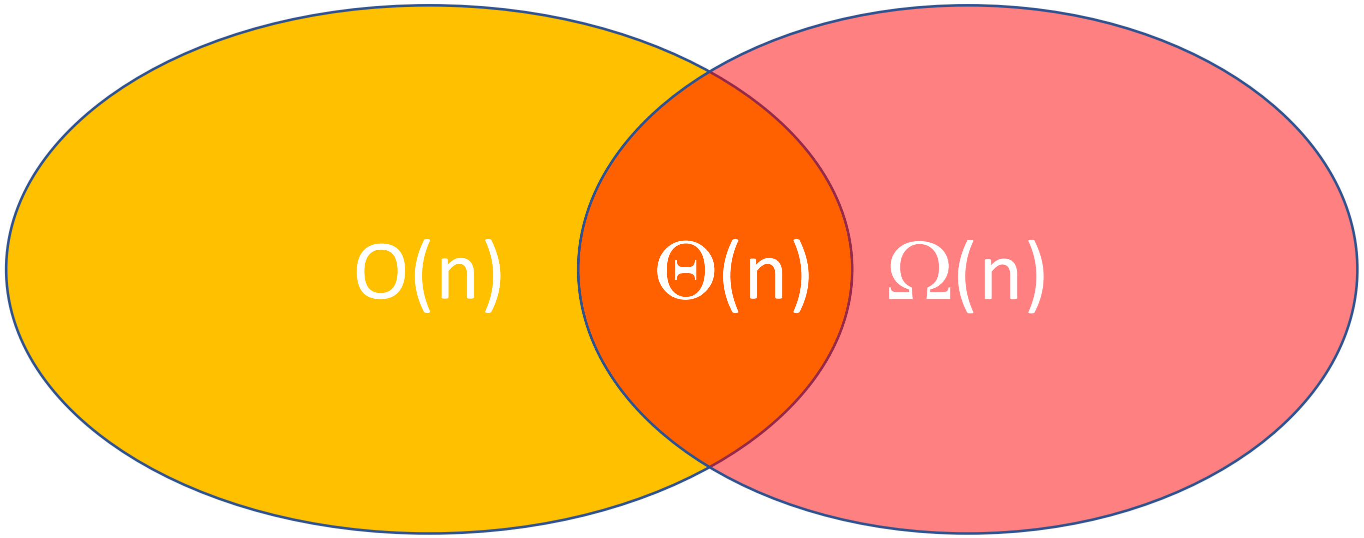
- $6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

  $\Rightarrow$ only holds for: $n \leq c_2/6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

  $\Rightarrow c_2 \geq n/\log n,\ \forall\, n \geq n_0$ – impossible

Asymptotic Notation Sets



For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct.

- $f(n) = \log n^2$; $g(n) = \log n + 5$ -> $f(n) = \Theta\ (g(n))$
- $f(n) = n$; $g(n) = \log n^2$ -> $f(n) = \Omega(g(n))$
- $f(n) = \log \log n$; $g(n) = \log n$ -> $f(n) = O(g(n))$
- $f(n) = n$; $g(n) = \log^2 n$ -> $f(n) = \Omega(g(n))$
- $f(n) = n \log n + n$; $g(n) = \log n$ -> $f(n) = \Omega(g(n))$
- $f(n) = 10$; $g(n) = \log 10$ -> $f(n) = \Theta(g(n))$
- $f(n) = 2^n$; $g(n) = 10n^2$ -> $f(n) = \Omega(g(n))$
- $f(n) = 2^n$; $g(n) = 3^n$ -> $f(n) = O(g(n))$

Properties

- *Theorem:*

  $$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$