CX1107
Data Structures and Algorithms

Introduction to Algorithms

Dr. Loke Yuan Ren
Lecturer
yrloke@ntu.edu.sg

College of Engineering
School of Computer Science and Engineering

Overview

Conduct complexity analysis of algorithms

- Time and space complexities
- Best case, worst case and average efficiencies
- Asymptotic Notations and Efficiency Classes
  - O notation
  - Ω notation (Omega)
  - Θ notation (Theta)

Time and space complexities

- Analyze efficiency of an algorithm in two aspects
  - Time
  - Space

- Time complexity: the amount of time used by an algorithm
- Space complexity: the amount of memory units used by an algorithm

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

- Declaration: int x;
- Assignment: x =1;
- Arithmetic operations: +, -, *, /, % etc.
- Logic operations: ==, !=, >, <, &&, ||

These primitive operations take constant time to perform

Basically they are not related to the problem size

changing the input(s) does not affect the computation time

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

   i. Repetition Structure: for-loop, while-loop
   ii. Selection Structure: if/else statement, switch-case statement
   iii. Recursive functions

2. Express it in term of problem size

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop

$j \leftarrow 1 \rightarrow c_0$

$\text{factorial} \leftarrow 1 \rightarrow c_1$

while $j \leq n$ **do**

  $\text{factorial} \leftarrow \text{factorial} * j \rightarrow c_2$

  $j \leftarrow j + 1 \rightarrow c_3$

n iterations $\Rightarrow \boxed{n(c_2+c_3)}$

$f(n) = c_0+c_1+n(c_2+c_3)$

The function increases linearly with n (problem size)

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop

```
for j ← 1, m do
    for k ← 1, n do
        sum ← sum + M[ j ][ k ]
```

c_1

n iterations

$\boxed{n(c_1)}$

m iterations

$\boxed{m(n(c_1))}$

The function increases quadratically with n if m==n

*Some constant time operations are ignored here.

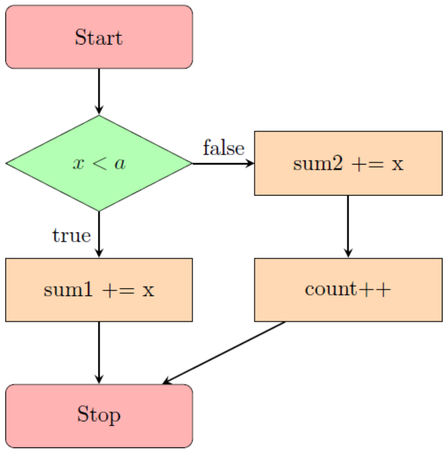
Quick Quiz 1

```
for ( i = 1; i < n; i++  )
        for ( j = 0; j < i; j++ )
                if (a[j]<=a[i])
                        r[i]++;
                else
                        r[j]++;
```

- How many (a[j]<=a[i]) are evaluated?

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement, switch-case statement




```
if(x<a)
      sum1 += x;
else {
     sum2 += x;
     count ++;
     }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis
2. Worst-case analysis
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
if(x<a)
      sum1 += x;
else {
     sum2 += x;
     count ++;
     }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis c_1
2. Worst-case analysis
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
if(x<a)
        sum1 += x;
else {
      sum2 += x;
      count ++;
      }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis
2. Worst-case analysis c_2
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
if(x<a)
      sum1 += x;
else {
     sum2 += x;
     count ++;
     }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis c_1
2. Worst-case analysis c_2
3. Average-case analysis

$$p(x < a)\ c_1 + p(x \geq a)c_2$$
$$= p(x < a)\ c_1 + (1 - p(x < a))c_2$$

Time Complexity or Time Efficiency

ii. Selection Structure: switch-case statement

`switch(choice){`	
`    case 1: compute the summation; break;`	$5n$
`    case 2: search BST; break;`	$6\log_2 n$
`    case 3: print BST; break;`	$3n$
`    case 4: search for the minimum; break;`	$4\log_2 n$
`}`	

Time Complexity

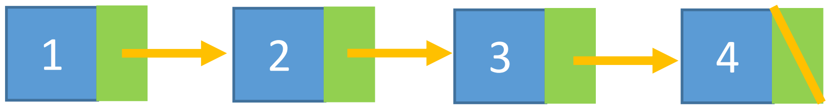
1. Best-case analysis $\dashrightarrow$ $C + 4\log_2 n$
2. Worst-case analysis $\dashrightarrow$ $C + 5n$
3. Average-case analysis $\dashrightarrow$ $C + \sum_{i=1}^{m} p(i)T_i$

Time Complexity or Time Efficiency

```
pt=head;
while (pt.key != a){
    pt = pt.next;
    if(pt == NULL) break;
}
```

c_1

c_2 (n-1) iterations



1. Best-case analysis: c_1 when **a** is the first item in the list
2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ when **a** is the last item in the list
3. Average-case analysis
   - Assumed that every item in the list has an equal probability as a search key

$$\frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1)) = \frac{1}{n}\left[nc_1 + c_2\sum_{i=1}^{n}(i-1)\right]$$

$$= c_1 + \frac{c_2}{n}\cdot\frac{n}{2}(0 + (n-1))$$

$$= c_1 + \frac{c_2(n-1)}{2}$$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of primitive operations in the algorithm
  - Primitive operations in each recursive call
  - Number of recursive calls

```
int factorial (int n)
{
    if(n==1) return 1;
    else return n*factorial(n-1);
}
```

c_2

c_1

- There are *n-1* recursive calls with the cost of c_1.
- In the last call (n==1), its cost is c_2.

$$c_1(n-1)+c_2$$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of array[0]==a in the algorithm
  - array[0]==a in each recursive call
  - Number of recursive calls: n-1

```
int count (int array[],int n, int a)
{
    if(n==1)
          if(array[0]==a)
             return 1;
          else return 0;
    if(array[0]==a)
         return 1+ count(&array[1], n-1, a);
    else
         return count (&array[1], n-1, a);
}
```

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$= 1 + 1 + W_{n-2}$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of array[0]==a in the algorithm
  - array[0]==a in each recursive call
  - Number of recursive calls: n-1

```
int count (int array[],int n, int a)
{
    if(n==1)
            if(array[0]==a)
                  return 1;
            else return 0;
    if(array[0]==a)
          return 1+ count(&array[1], n-1, a);
    else
          return count (&array[1], n-1, a);
}
```

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$= 1 + 1 + W_{n-2}$

$= 1 + 1 + 1 + W_{n-3}$

...

$= 1 + 1 + \ldots + 1 + W_1$

$= (n - 1) + W_1 = n$

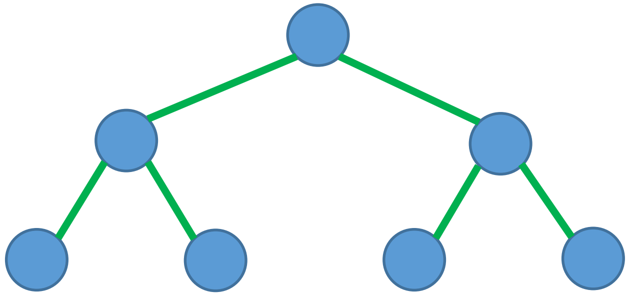
It is known as a method of backward substitutions

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of multiplication operations in the algorithm

```
preorder (simple_t* tree)
{
    if(tree != NULL){
        tree->item *= 10;
        preorder (tree->left);
        preorder (tree->right);
    }
}
```



Geometric Series:

$$\begin{aligned} S_n &= a + ar + ar^2 + \ldots + ar^{n-1} \\ rS_n &= ar + ar^2 + \ldots + ar^{n-1} + ar^n \\ (1-r)S_n &= a - ar^n \\ S_n &= \frac{a(1-r^n)}{1-r} \end{aligned}$$

$W_0 = 0$

$W_1 = 1$

$W_2 = 1 + W_1 + W_1 = 3$

$W_3 = 1 + W_2 + W_2$

$= 1 + 2\,(1 + W_1 + W_1)$

$= 1 + 2\,(1 + 2)$

$= 1 + 2 + 4 = 7$

$W_{k-1} = 1 + 2 \cdot W_{k-2}$

$= 1 + 2 + 4 + 8 + \ldots + 2^{k-2}$

$W_k = 1 + 2 \cdot W_{k-1}$

$= \frac{1-2^k}{1-2} = 2^k - 1$

Prove the hypothesis can be done by mathematical induction

It is known as a **method of forward substitutions**

Quick Quiz 2

```
void Reverse(char *str, int start, int end )
{       char    tmp;
        if (end > start ) {
                Reverse(str,start+1,end-1);
                tmp = *(str+start);
                *(str+start) = *(str+end);
                *(str+end) =tmp;
        }
        return;
}
```

- How many swap operations will be executed?

Warm-up Question

Find a recurrence equation for the number of multiplications as a function of N. N is a power of two; that is $N = 2^K$ for some integer K.

```
int power2 (int X, int N)
{
    int HALF, HALFPOWER;

    if ( N == 1) return X;
    else{
        HALF = N/2;
        HALFPOWER = power2(X, HALF);
        if (2*HALF == N) // if N is even
            return HALFPOWER * HALFPOWER;
        else return HALFPOWER * HALFPOWER * X;
    }
}
```

```
int power2 (int X, int N)
{
    int HALF, HALFPOWER;

    if ( N == 1) return X;
    else{
        HALF = N/2;
        HALFPOWER = power2(X, HALF);
        if (2*HALF == N) // if N is even
            return HALFPOWER * HALFPOWER;
        else return HALFPOWER * HALFPOWER * X;
    }
}
```

Note: $n=2^k$

power2:

$$T_1 = 0$$
$$T_n = 2 + T_{n/2}$$

Therefore, $T_n = 2+T_{n/2}$

$= 2+2+T_{n/2^2}$

…

$= 2+2+\ldots+T_{n/2^k}$ with k 2's

$= 2k$

$= 2\log_2(n)$

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (μsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10						
100						
10^4						
10^6						

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013					
10^4	.13					
10^6	13					

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

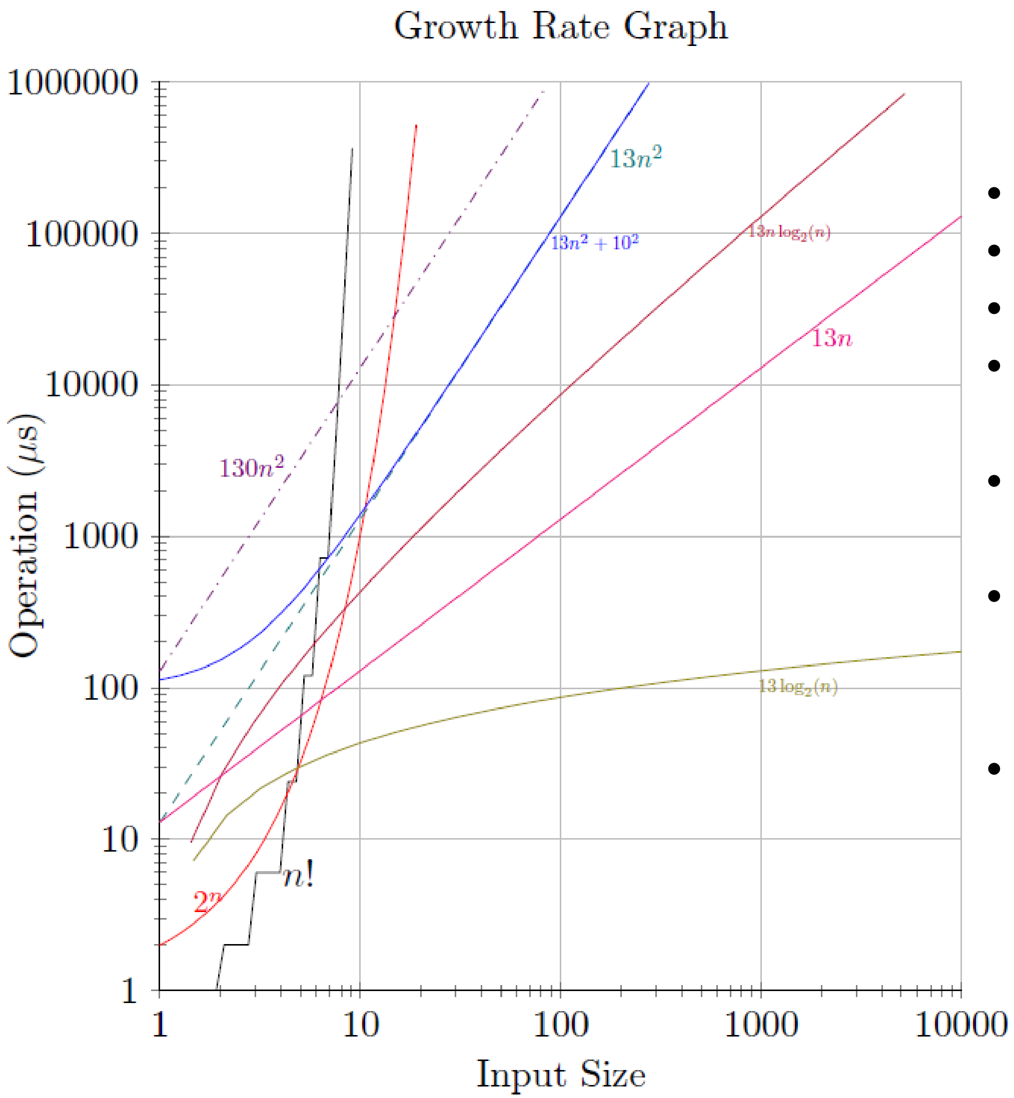
Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086				
10^4	.13	.173				
10^6	13	259				

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n
Problem size (n)						
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	4×10^{16}years
10^4	.13	.173	22 mins	3.61hrs	22mins	
10^6	13	259	150 days	1505 days	150days	

Order of Growth



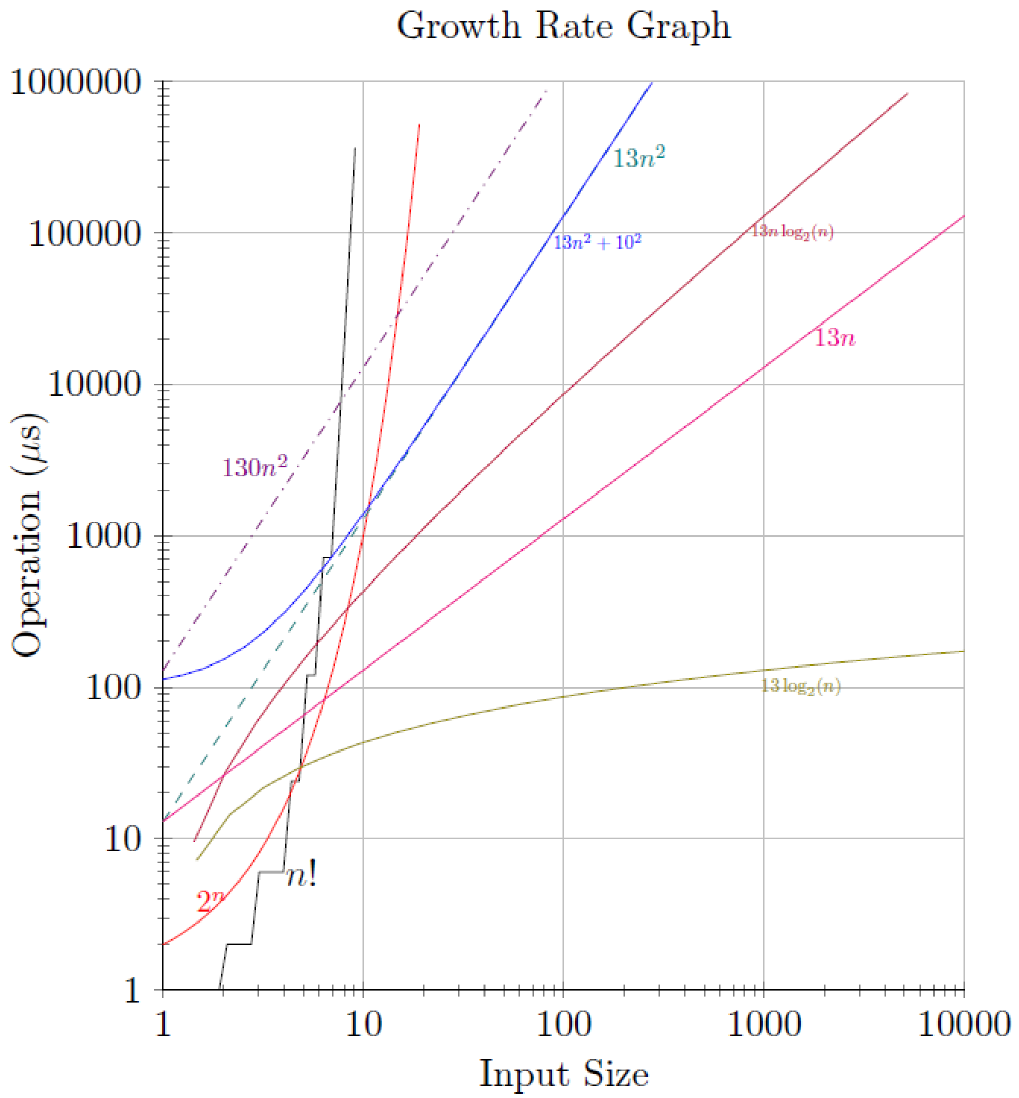
- $n!$ is the fastest growth
- 2^n is the second
- 13n is linear
- $13\log_2 n$ is the slowest

- 10^2 can be ignored when n is large

- $13n^2$ VS $(13n^2 + 10^2)$
  - As $n \to \infty$, no difference

- $13n^2$ and $130n^2$ have similar growth.
  - $130n^2$ just slightly faster

Order of Growth

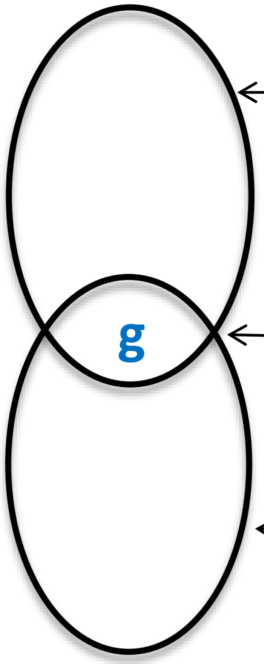


- $13n^2$ VS $(13n^2 + 10^2)$
  - As $n \to \infty$, no difference
- $13n^2$ and $130n^2$ have similar growth.
  - $130n^2$ slightly faster

- If we can't count the exact number of operations, how?
- Does it really important to represent complexity exactly?

Asymptotic Notations

- Big-Oh (**O**) , Big-Omega (**Ω**) and Big-Theta (**Θ**) are asymptotic (set) notations used for describing the order of growth of a given function.



$f \in \Omega(g)$ Set of functions that grow at higher or same rate as **g**

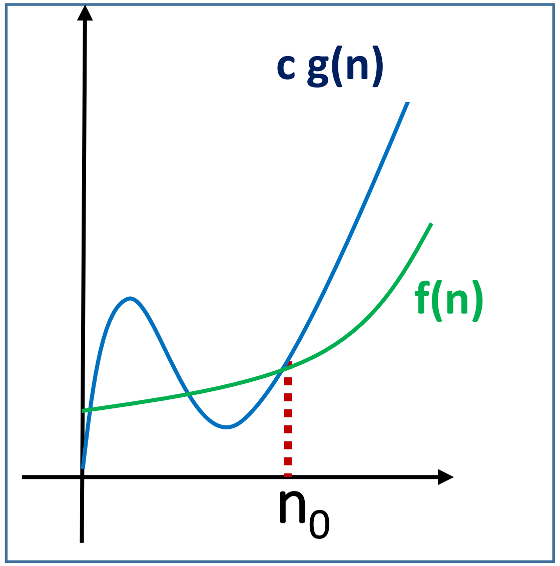
$f \in \Theta(g)$ Set of functions that grow at same rate as **g**

$f \in O(g)$ Set of functions that grow at lower or same rate as **g**

Big-Oh Notation (O)

Definition 3.1 *$\mathcal{O}$-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\mathcal{O}(g(n))$, denoted $f(n) \in \mathcal{O}(g(n))$, if $f(n)$ is **bounded above** by some constant multiple of $g(n)$ for all large n, i.e., the set of functions can be defined as*

$$\mathcal{O}(g(n)) = \{f(n) : \exists \text{positive constants, } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \quad \forall n \ge n_0\}$$

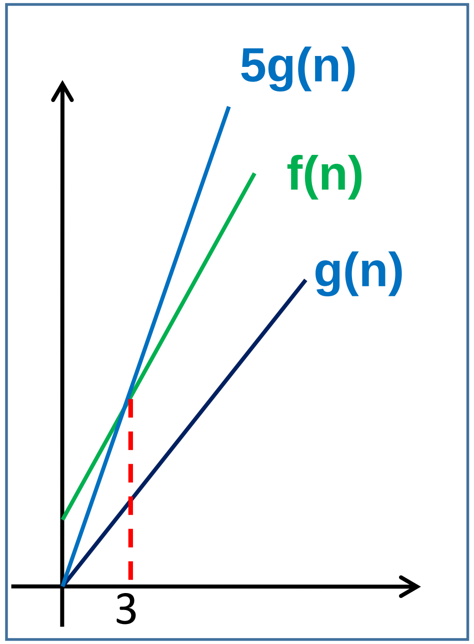


f(n) = 4n + 3 and g(n) = n

Let c = 5, n0 = 3

$$f(n) = 4n + 3$$
$$4n + 3 \le 5n \qquad \forall n \ge 3$$
$$f(n) \le 5g(n) \qquad \forall n \ge 3$$

$$\Rightarrow \quad f(n) = O(g(n)) \qquad i.e.\ 4n + 3 \in O(n)$$



Big-Oh Notation (O)

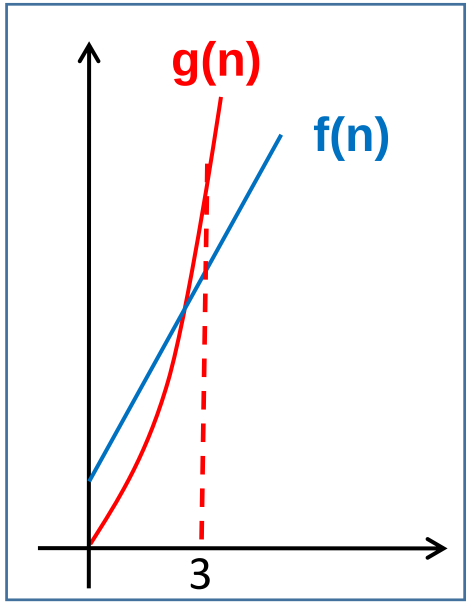
Definition 3.1 $\mathcal{O}$*-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\mathcal{O}(g(n))$, denoted $f(n) \in \mathcal{O}(g(n))$, if $f(n)$ is* ***bounded above*** *by some constant multiple of $g(n)$ for all large n, i.e., the set of functions can be defined as*

$$\mathcal{O}(g(n)) = \{f(n) : \exists \text{positive constants, } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \quad \forall n \ge n_0\}$$

f(n) = 4n + 3 and g(n) = n^3

Let c = 1, n0 = 3

$$f(n) = 4n + 3$$
$$4n + 3 \le n^3 \qquad \forall n \ge 3$$
$$f(n) \le 5g(n) \quad \forall n \ge 3$$
$$\Rightarrow \quad f(n) = O(g(n)) \qquad i.e. \, 4n + 3 \in O(n^3)$$



If $f(n) = O(g(n))$, we say

$g(n)$ is asymptotic upper bound of $f(n)$

Big-Oh Notation (O) – Alternative definition

Definition 3.2 *$\mathcal{O}$-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) \in \mathcal{O}(g(n))$ or $f(n) = \mathcal{O}(g(n))$.*

f(n) = 4n + 3 and g(n) = n

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3}{n} = 4 < \infty$$

$$f(n) = O(g(n)) \quad i.e. 4n+3 \in O(n)$$

f(n) = 4n + 3 and g(n) = n^3

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3}{n^3} = 0 < \infty$$

$$f(n) = O(g(n)) \quad i.e. 4n+3 \in O(n^3)$$

Big-Omega Notation (Ω)

Definition 3.3 *Ω-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\Omega(g(n))$, denoted $f(n) \in \Omega(g(n))$, if $f(n)$ is* ***bounded below*** *by some constant multiple of $g(n)$ for all large n, i.e., the set of functions can be defined as*

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

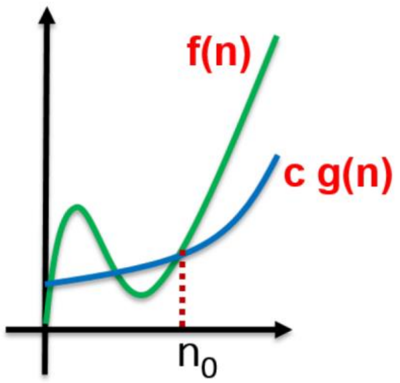
Definition 3.4 *Ω-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, then $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$.*

f(n) = 4n + 3 and g(n) = 5n

Let c=1/5, n_0 =0

f(n) ≥ (1/5)g(n)

4n+3 ≥ (1/5)5n for all n≥0



If $f(n) = \Omega(g(n))$, we say

$g(n)$ is asymptotic lower bound of $f(n)$

Big-Theta Notation (Θ)

Definition 3.5 Θ-*notation: Let* f *and* g *be two functions such that* $f(n) : \mathbb{N} \to \mathbb{R}^+$ *and* $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ *is said to be in* $\Theta(g(n))$, *denoted* $f(n) \in \Theta(g(n))$, *if* $f(n)$ *is* ***bounded both above and below*** *by some constant multiples of* $g(n)$ *for all large* n, *i.e., the set of functions can be defined as*

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

Definition 3.6 Θ-*notation: Let* f *and* g *be two functions such that* $f(n) : \mathbb{N} \to \mathbb{R}^+$ *and* $g(n) : \mathbb{N} \to \mathbb{R}^+$, *if* $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ *where* $0 < c < \infty$, *then* $f(n) \in \Theta(g(n))$ *or* $f(n) = \Theta(g(n))$.

If $f(n) = \Theta(g(n))$, we say

$g(n)$ is asymptotic tight bound of $f(n)$

Summary of Limit Definition

	f(n) ∈ O(g(n))	f(n) ∈ Ω(g(n))	f(n) ∈ Θ(g(n))
$\mathbf{0}$	✓		
$\mathbf{0 < C < \infty}$	✓	✓	✓
∞		✓	

Quiz

State whether f is $O(g)$; whether f is $\Theta(g)$; and whether f is $\Omega(g)$

$$f(n) = \log_2(n^3)$$

$$g(n) = \log_2(n)$$

Method I:

$f(n) = \log_2(n^3) = 3\log_2(n)$

$f(n) \leq 3g(n) \Rightarrow f(n) \in O(g(n))$

$f(n) \geq 3g(n) \Rightarrow f(n) \in \Omega(g(n))$

Method II:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 3$$

$$\because 0 < 3 < \infty$$

Therefore, f(n) is in $O(g(n))$, $\Omega(g(n))$ and $\Theta(g(n))$

Asymptotic Notation in Equations

When an asymptotic notation appears in an equation, we interpret it as standing for some anonymous function that we do not care to name.

Examples:

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
- $T(n) = T(n/2) + \Theta(n)$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n2)$

Simplification Rules for Asymptotic Analysis

1. If $f(n) = O\left(cg(n)\right)$ for any constant c > 0, then $f(n) = O(g(n))$
2. If $f(n) = O\left(g(n)\right)$ and $\mathrm{g}(n) = O\left(h(n)\right)$, then $f(n) = O\left(h(n)\right)$

   e.g. $f(n) = 2n$, $\mathrm{g}(n) = n^2$, $\mathrm{h}(n) = n^3$

3. If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$,

   then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

   e.g. $5\mathrm{n} + 3\log_2 n = O(n)$

4. If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$

   then $f_1(n)f_2(n) = O\left(g_1(n)g_2(n)\right)$

   e.g. $f_1(n) = 3n^2 = O(n^2)$, $f_2(n) = \log_2 n = O(\log_2 n)$

   Then $3n^2 \log_2 n = O(n^2 \log_2 n)$

Properties of Asymptotic Notation

- Reflexive of O, Ω and Θ

$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$
$$f(n) = \Theta(f(n))$$

- Symmetric of Θ

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

- Transitive of O, Ω and Θ

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

Common Complexity Classes

Order of Growth	Class	Example
1	Constant	Finding midpoint of an array
$\log_2 n$	Logarithmic	Binary Search
n	Linear	Linear Search
$n\log_2 n$	Linearithmic	Merge Sort
n^2	Quadratic	Insertion Sort
n^3	Cubic	Matrix Inversion (Gauss-Jordan Elimination)
2^n	Exponential	The Tower of Hanoi Problem
$n!$	Factorial	Travelling Salesman Problem

When time complexity of algorithm A grows faster than algorithm B for the same problem, we say A is inferior to B.

Towers of Hanoi

- The Tower of Hanoi consists of three rods (towers) and a number of disks of different sizes which can slide onto any rod.
    - Only one disk can be moved each time
    - The disk at the top of a rod will be removed and replaced on top of another rod or on an empty rod in each move.
    - No larger disk can be placed on top of any smaller disk.
    - Runs in exponential time i.e. $\Theta(2^n)$

Towers of Hanoi

```
void TowersOfHanoi(int n, int x, int y, int z){
// Move n disks from tower x to tower y
// Use tower z for intermediate storage
   if (n > 0) {
      TowersOfHanoi(n-1, x, z, y);
      cout << "Move  disk from  " << x << " to  " << y << endl;
      TowersOfHanoi(n-1, z, y, x);
                  }
}
```

Space Complexity

- Determine number of entities in problem (also called problem size)
- Count number of basic units in algorithm

- Basic units
- Things that can be represented in a constant amount of storage space
- E.g. integer, float and character.

Space Complexity

- Space requirements for an array of n integers - $\Theta(n)$
- If a matrix is used to store edge information of a graph,

  i.e. G[x][y] = 1 if there exists an edge from x to y,

  space requirement for a graph with n vertices is $\Theta(n^2)$

Space and time trade-offs:

- Reduction in time can be achieved by sacrificing space and vice-versa.