

SC1007

Analysis of Algorithms

Dr. Liu Siyuan

Email: syliu@ntu.edu.sg

Office: N4-02C-72a

Overview

Conduct complexity analysis of algorithms

- Time and space complexities
- Best-case, worst-case and average efficiencies
- Order of Growth
- Asymptotic notations
 - O notation
 - Ω notation (Omega)
 - Θ notation (Theta)
- Efficiency classes

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

1. Count the number of primitive operations in the algorithm

Time Complexity

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 (the time for each operation) are not related to the problem size

Time Complexity

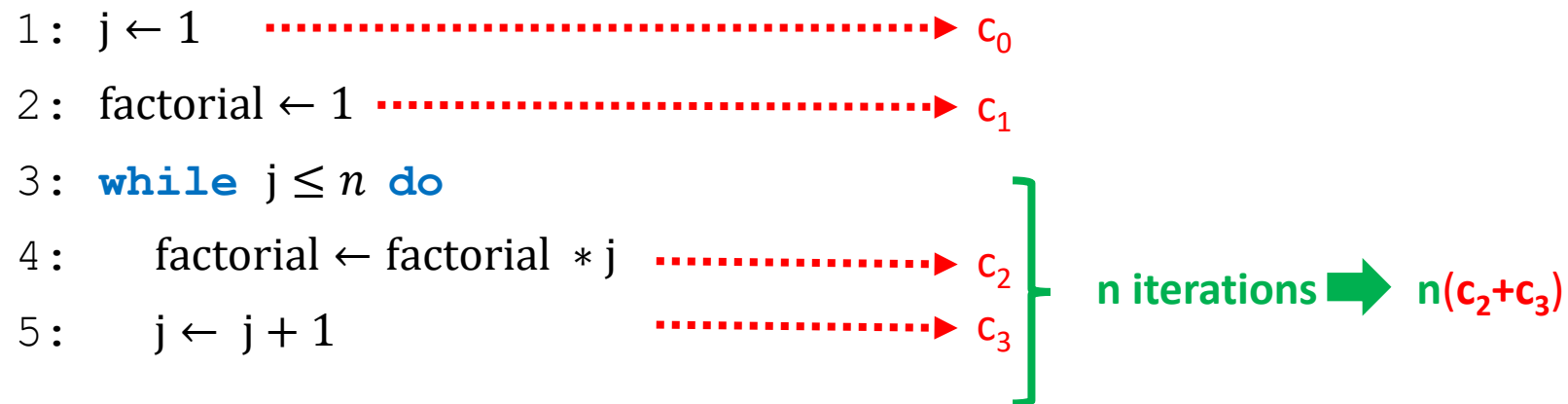
1. Count the number of **primitive operations** in the algorithm
 - i. Repetition Structure: for-loop, while-loop
 - ii. Selection Structure: if/else statement, switch-case statement
 - iii. Recursive functions
2. Express it in term of problem size (input size)

Algorithm 4 Fibonacci Sequence: A Simple Recursive Function

```
1: function Fibonacci_Recursive(n)
2: begin
3:   if  $n < 1$  then
4:     return 0
5:   if  $n == 1$  OR  $n == 2$  then
6:     return 1
7:   return Fibonacci_Recursive( $n - 1$ ) + Fibonacci_Recursive( $n - 2$ )
8: end
```

Time Complexity

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



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

The function increases linearly with n (problem size)

*Some constant time operations are ignored here.

Time Complexity

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

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

$\dots \rightarrow c_1$ } n iterations
 $n(c_1)$ } m iterations
 $m(n(c_1))$

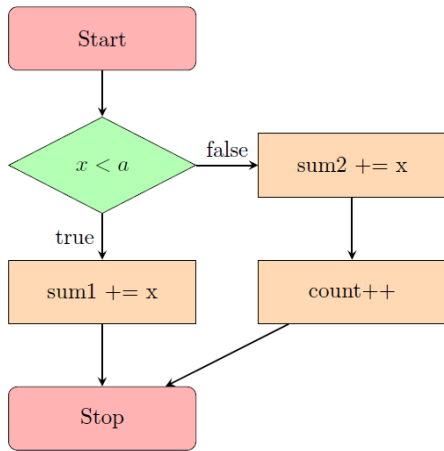
$$f(n, m) = mnc_1$$

The function increases quadratically with n if $m=n$

*Some constant time operations are ignored here.

Time Complexity

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



```
1: if (x < a)
2:     sum1 += x;
3: else {
4:     sum2 += x;
5:     count++;
6: }
```

When $x < a$, only one primitive operation is executed
When $x \geq a$, 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

ii. Selection Structure: if/else statement

```
1: if (x < a)
2:     sum1 += x;
3: else {
4:     sum2 += x;
5:     count++;
6: }
```

When $x < a$, only one primitive operation is executed

When $x \geq a$, 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

ii. Selection Structure: if/else statement

```
1: if (x < a)
2:     sum1 += x;
3: else {
4:     sum2 += x;
5:     count++;
6: }
```

When $x < a$, only one primitive operation is executed

When $x \geq a$, 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

ii. Selection Structure: if/else statement

```
1: if (x < a)
2:     sum1 += x;
3: else {
4:     sum2 += x;
5:     count++;
6: }
```

When $x < a$, only one primitive operation is executed

When $x \geq a$, 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

$$\begin{aligned} & p(x < a) c_1 + p(x \geq a) c_2 \\ &= p(x < a) c_1 + (1 - p(x < a)) c_2 \end{aligned}$$

Time Complexity

ii. Selection Structure: switch-case statement

```
1: switch(choice) {  
2:     case 1: compute the summation; break; .....►  $T_1 = 5n$   
3:     case 2: search BST; break; .....►  $T_2 = 6\log_2 n$   
4:     case 3: print BST; break; .....►  $T_3 = 3n$   
5:     case 4: search for the minimum; break; .....►  $T_4 = 4\log_2 n$   
6: }
```

Time Complexity


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

Time Complexity

iii. Recursive functions

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

```
def factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n - 1)
```



- $n-1$ recursive calls with the cost of c_1 .
- The cost of the last call ($n==1$) is c_2 .
- Thus, $f(n) = c_1(n - 1) + c_2$
- It is a linear function

```
1: j ← 1  
2: factorial ← 1  
3: while j ≤ n do  
4:     factorial ← factorial * j  
5:     j ← j + 1
```

Time Complexity

iii. Recursive functions

- Count the **number of array[0]==a** in the algorithm, i.e., focusing on the important operations
 - array[0]==a in each recursive call
 - Number of recursive calls: n-1

```
def count(array, n, 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)
```

$$W_1 = 1$$

$$\begin{aligned} W_n &= 1 + W_{n-1} \\ &= 1 + 1 + W_{n-2} \end{aligned}$$

Time Complexity

iii. Recursive functions

- Count the **number of array[0]==a** in the algorithm, i.e., focusing on the important operations
 - array[0]==a in each recursive call
 - Number of recursive calls: n-1

```
def count(array, n, 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)
```

$$\begin{aligned}W_1 &= 1 \\W_n &= 1 + W_{n-1} \\&= 1 + 1 + W_{n-2} \\&= 1 + 1 + 1 + W_{n-3} \\&\dots \\&= \underbrace{1 + 1 + \dots + 1}_{n-1} + W_1 \\&= (n - 1) + W_1 = n\end{aligned}$$

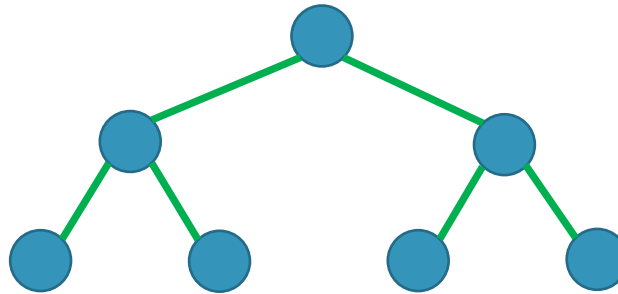
It is known as a **method of backward substitutions**

Time Complexity

iii. Recursive functions

- Count the **number of multiplication operations** in the algorithm

```
def preorder(tree):  
    if tree is not None:  
        tree.item *= 10  
        preorder(tree.left)  
        preorder(tree.right)
```



Geometric Series:

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

Prove the hypothesis can be done by mathematical induction
It is known as a **method of forward substitutions**

$$W_0 = 0$$

$$W_1 = 1$$

$$W_2 = 1 + W_1 + W_1 = 3$$

$$\begin{aligned} W_3 &= 1 + W_2 + W_2 \\ &= 1 + 2(1 + W_1 + W_1) \\ &= 1 + 2(1 + 2) \\ &= 1 + 2 + 4 = 7 \end{aligned}$$

$$\begin{aligned} W_{k-1} &= 1 + 2 \cdot W_{k-2} \\ &= 1 + 2 + 4 + 8 + \dots + 2^{k-2} \end{aligned}$$

$$\begin{aligned} W_k &= 1 + 2 \cdot W_{k-1} = 1 + 2 + 4 + 8 + \dots + 2^{k-1} \\ &= \frac{1-2^k}{1-2} = 2^k - 1 \end{aligned}$$

Series

- Geometric Series

$$G_n = \frac{a(1 - r^n)}{1 - r}$$

- Arithmetic Series

$$A_n = \frac{n}{2} [2a + (n - 1)d] = \frac{n}{2} [a_0 + a_{n-1}]$$

- Arithmetico-geometric Series

$$\sum_{t=1}^k t 2^{t-1} = 2^k (k - 1) + 1$$

- Faulhaber's Formula for the sum of the p-th powers of the first n positive integers

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^n k^3 = \frac{n^2(n+1)^2}{4}$$

*Derivation is in note section 0.7.4.1

Example

```
for i in range(1, n+1):  
    M[i] = 0  
    for j in range(i, 0, -1):  
        for k in range(i, 0, -1):  
            M[i] += A[j] * B[k]
```

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

- In each inner loop, both j and k are assigned by value of i.
- Inner loops takes i^2 multiplications
- The overall number of multiplications is

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \sum_{i=1}^n i^2$$

Summary

- Count the number of important primitive operations
- Express as a function of input size
- Repetition structure
- Selection structure: Best-case, worst-case and average efficiencies
- Recursive calls: backward substitution, forward substitution
- Series

Order of Growth

Considering These Algorithms

Algorithm 2 Summing Arithmetic Sequence

```
1: function Method_Two(n)
2: begin
3:  $sum \leftarrow n * (1 + n) / 2$  .....  $c_1$ 
4: end
```

$$f(n) = c_1$$

Algorithm 1 Summing Arithmetic Sequence

```
1: function Method_One(n)
2: begin
3:  $sum \leftarrow 0$ 
4: for  $i = 1$  to  $n$  do
5:    $sum \leftarrow sum + i$  .....  $c_2$ 
6: end
```

$$f(n) = c_2 \times n$$

Considering These Algorithms

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j in range(0, n - i - 1):  
            if arr[j] > arr[j + 1]:  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

$$f(n) = 1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2}$$

$$f(n) = 2f\left(\frac{n}{2}\right) + (n - 1)$$
$$f(n) \approx n \log_2 n$$

```
def merge_sort(arr):  
    if len(arr) <= 1:  
        return arr  
  
    mid = len(arr) // 2  
    left_half = merge_sort(arr[:mid])  
    right_half = merge_sort(arr[mid:])  
    return merge(left_half, right_half)  
  
def merge(left, right):  
    sorted_array = []  
    i = j = 0  
  
    while i < len(left) and j < len(right):  
        if left[i] < right[j]:  
            sorted_array.append(left[i])  
            i += 1  
        else:  
            sorted_array.append(right[j])  
            j += 1  
  
    sorted_array.extend(left[i:])  
    sorted_array.extend(right[j:])  
    return sorted_array
```

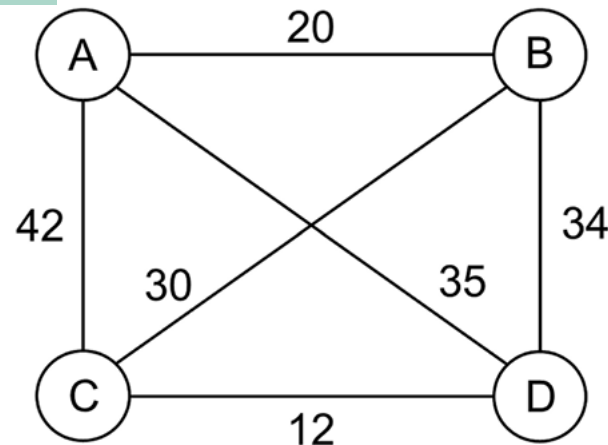
Considering These Algorithms

Algorithm 4 Fibonacci Sequence: A Simple Recursive Function

```
1: function Fibonacci_Recursive(n)
2: begin
3:   if n < 1 then
4:     return 0
5:   if n == 1 OR n == 2 then
6:     return 1
7:   return Fibonacci_Recursive(n - 1) + Fibonacci_Recursive(n - 2)
8: end
```

$$f(n) = 1 + f(n - 1) + f(n - 2)$$
$$f(n) \approx 2^n$$

- TSP: Given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin city.



Brute force approach:

- Generate all the cities permutations: $n!$
- Compute the sum of the distance for each permutation: n
- In total: $f(n) = n \times n!$

Order of Growth

Order of growth: A set of functions whose asymptotic growth behaviour is considered equivalent. The order of growth ignores the constant factor needed for fixed operations and focuses instead on the operations that increase proportional to input size

Algorithm	1	2	3	4	5	6	7
Operation (μsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n	$n!$

Problem size (n)

10							
100							
10^4							
10^6							

Order of Growth

Order of growth: A set of functions whose asymptotic growth behaviour is considered equivalent. The order of growth ignores the constant factor needed for fixed operations and focuses instead on the operations that increase proportional to input size

Algorithm	1	2	3	4	5	6	7
Operation (μsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n	$n!$

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024	3.6288
100	.0013						
10^4	.13						
10^6	13						

Order of Growth

Order of growth: A set of functions whose asymptotic growth behaviour is considered equivalent. The order of growth ignores the constant factor needed for fixed operations and focuses instead on the operations that increase proportional to input size

Algorithm	1	2	3	4	5	6	7
Operation (μsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n	$n!$

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024	3.6288
100	.0013	.0086					
10^4	.13	.173					
10^6	13	259					

Order of Growth

Order of growth: A set of functions whose asymptotic growth behaviour is considered equivalent. The order of growth ignores the constant factor needed for fixed operations and focuses instead on the operations that increase proportional to input size

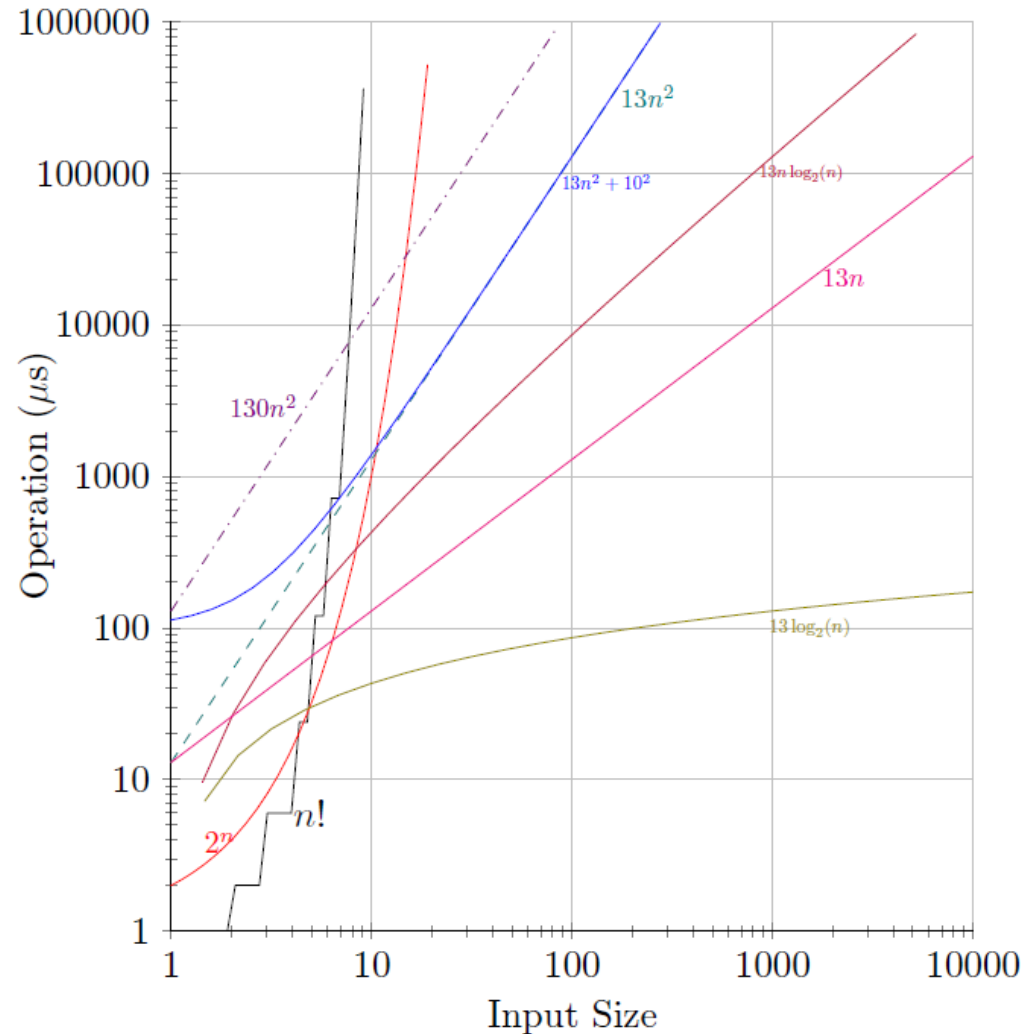
Algorithm	1	2	3	4	5	6	7
Operation (μsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n	$n!$

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024	3.6288
100	.0013	.0086	.13	1.3	.1301	4×10^{16} years	2.96×10^{144} years
10^4	.13	.173	22 mins	3.61hrs	22mins		
10^6	13	259	150 days	1505 days	150days		

Order of Growth

Growth Rate Graph



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

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 *O*-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ and $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$, $f(n)$ 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

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

$$f(n) = 4n + 3, \text{ and } g(n) = n$$

$$\text{Let } c = 5, n_0 = 3$$

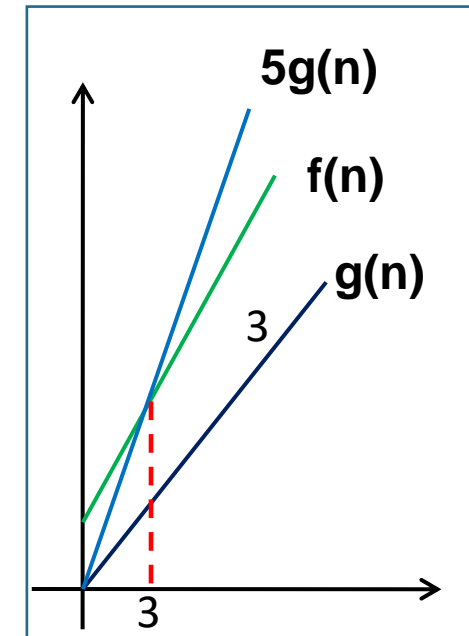
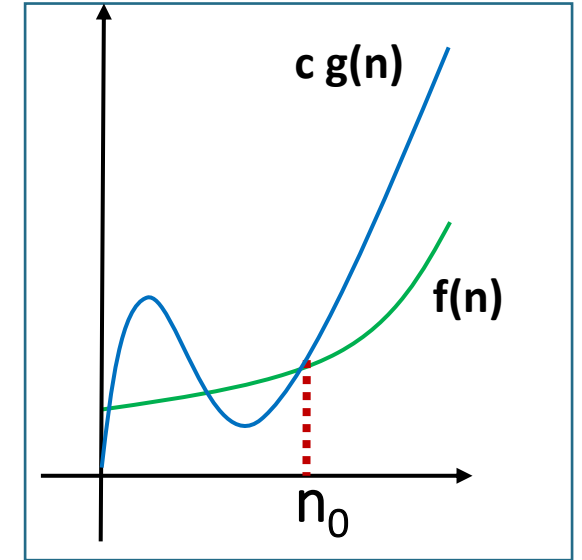
$$f(n) = 4n + 3$$

$$4n + 3 \leq 5n \quad \forall n \geq 3$$

$$f(n) \leq 5g(n) \quad \forall n \geq 3$$



$$f(n) = \mathcal{O}(g(n)) \quad \text{i.e., } 4n + 3 \in \mathcal{O}(n)$$



Big-Oh Notation (O)

Definition 3.1 *O*-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ and $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$, $f(n)$ is said to be in $O(g(n))$, denoted $f(n) \in O(g(n))$, 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

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

$$f(n) = 4n + 3 \text{ and } g(n) = n^3$$

$$\text{Let } c = 1, n_0 = 3$$

$$f(n) = 4n + 3$$

$$4n + 3 \leq n^3 \quad \forall n \geq 3$$

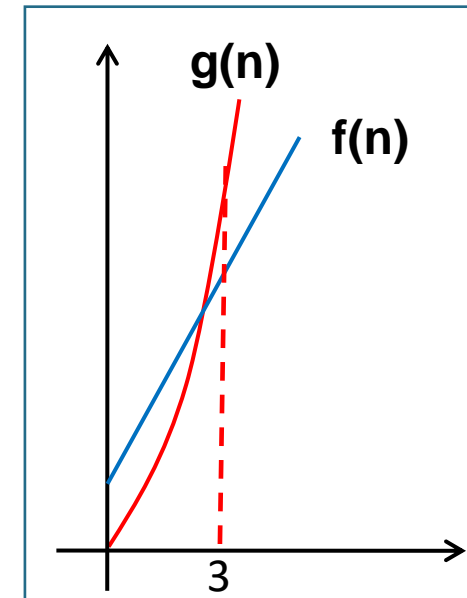
$$f(n) \leq g(n) \quad \forall n \geq 3$$



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

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

$$f(n) = 4n + 3 \text{ and } g(n) = n$$

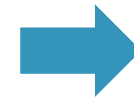
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{4n + 3}{n} = 4 < \infty$$



$$f(n) = \mathcal{O}(g(n)) \quad \text{i.e., } 4n + 3 \in \mathcal{O}(n)$$

$$f(n) = 4n + 3 \text{ and } g(n) = n^3$$

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



$$f(n) = \mathcal{O}(g(n)) \quad \text{i.e., } 4n + 3 \in \mathcal{O}(n^3)$$

Big-Omega Notation (Ω)

Definition 3.3 Ω -notation: Let f and g be two functions such that $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ and $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$, $f(n)$ 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 $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ and $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$, if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$, then $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 4 > 0$$

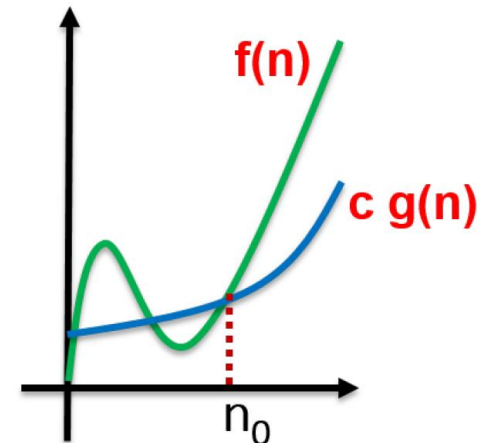
$$f(n) = 4n + 3, \text{ and } g(n) = n$$

$$\text{Let } c = 1, n_0 = 1$$

$$\begin{aligned} f(n) &\geq cg(n) \quad \forall n \geq n_0 \\ 4n + 3 &\geq n \quad \forall n \geq 1 \end{aligned}$$

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 $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ and $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$, $f(n)$ 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

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants, } c_1, c_2 \text{ and } n_0 \text{ such that } c_1g(n) \leq f(n) \leq c_2g(n) \quad \forall n \geq n_0\}$$

Definition 3.6 Θ -notation: Let f and g be two functions such that $f(n) : \mathbb{N} \rightarrow \mathbb{R}^+$ and $g(n) : \mathbb{N} \rightarrow \mathbb{R}^+$, if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ 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

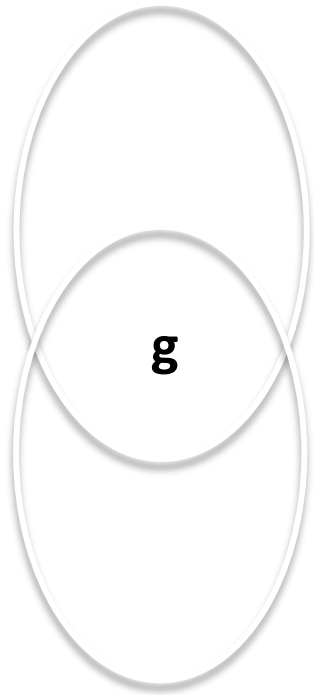
$g(n)$ is asymptotic tight bound of $f(n)$

Summary of Limit Definition

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$	$f(n) \in O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \in \Theta(g(n))$
0	✓		
$0 < C < \infty$	✓	✓	✓
∞		✓	

Asymptotic Notations

- Big-Oh (\mathcal{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**

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

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	Bubble Sort
n^3	Cubic	Matrix Inversion (Gauss-Jordan Elimination)
2^n	Exponential	Fibonacci Sequence (recursive)
$n!$	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.

To Simplify.....

- Given an algorithm
 - Derive the time complexity function f with respect to problem size n
 - Count the number of (important) primitive operations
 - Compare against g : $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$

$g(n)$
1
$\log_2 n$
n
$n \log_2 n$
n^2
n^3
2^n
$n!$

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$	$f(n) \in O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \in \Theta(g(n))$
0	✓		
$0 < c < \infty$	✓	✓	✓
∞		✓	

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:

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

Simplification Rules for Asymptotic Analysis

1. If $f(n) = O(cg(n))$ for any positive constant $c > 0$, then $f(n) = O(g(n))$
2. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
e.g., $f(n) = 2n$, $g(n) = n^2$, $h(n) = n^3$
3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
e.g., $5n + 3 \log_2 n = O(n)$
4. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$
then $f_1(n)f_2(n) = O(g_1(n)g_2(n))$
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

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

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

- **Transitive** of O , Ω and Θ

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

$$\Rightarrow f(n) = O(h(n))$$

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

$$\Rightarrow f(n) = \Omega(h(n))$$

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

$$\Rightarrow f(n) = \Theta(h(n))$$

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/time tradeoff principle

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

Summary

Conduct complexity analysis of algorithms

- Time and space complexities
- Best-case, worst-case and average efficiencies
- Order of Growth
- Asymptotic notations
 - O notation
 - Ω notation (Omega)
 - Θ notation (Theta)
- Efficiency classes