NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE


**CX1107 Data Structures and Algorithms** **2020/21 Semester 2**

# 7: Analysis of Algorithms

*School of Computer Science and Engineering* *Nanyang Technological University*

## 7.1 Asymptotic Algorithm Analysis 渐近的

In algorithm analysis, we always interest to understand how the resources (e.g. **time** and **memory space**) used by an algorithm when the input size increase.
**Asymptotic Analysis**, a.k.a. **Asymptotics**: Study of functions of a parameter, $N$, as $N$ becomes larger and larger without bound.

For example, $f(n) = n^3 + 2n$. As $n$ becomes very large, the term $2n$ can be negligible. In this case, $f(n)$ is asymptotically equivalent to $n^3$.

## 7.2 Time and Space Complexity

There are two kinds of efficiency of algorithms: **time efficiency** and **space efficiency**.
**Time efficiency**, a.k.a time complexity indicates the amount of time used by an algorithm. We need to **count the number of primitive operations** in the algorithm and express it in terms of problem size (e.g. a function of $n$). We also need to understand how the parameters affect the performance of the algorithm. It is related to algorithmic aspects.
**Space efficiency**, a.k.a space complexity indicates the amount of memory units used by an algorithm. We need to understand how the data (inputs and some intermediate results) are storaged. It is related to data structures in the algorithm.

- Primitive operations: declaration (e.g. int x), assignment (e.g. x=1), arithmetic (+, -, *, /, %) and logic (==, ! =, <, >, &&, ||) operations
  These basic steps are usually performed in **constant time**
  → take constant time to employ same operation time)
- Repetition Structure: for-loop, while-loop
- Selection Structure: if/else statement, switch-case statement
- Recursive Function.

Determine time complexity

### 7.2.1 Example 1: while loop

**Algorithm 1** 'while' Loop

```
1: j ← 0                                   ▷ Constant time: c0
2: while j <= n do                         ▷ Run n iterations
3:     factorial ← factorial * j           ▷ c1
4:     j ← j + 1                           ▷ c2
                                           ▷ Time complexity = c1 n + c2 n
```

In this example, $c_0$, $c_1$ and $c_2$ are constant time for respective primitive operations. The time complexity of this 'while' algorithm is a function of $n$. The function increases **linearly** with $n$.

### 7.2.2 Example 2: nested 'for' loop

**Algorithm 2** Nested 'for' Loop

```
1: for j ← 1, m do                          ▷ Run m iterations
2:     for k ← 1, n do                      ▷ Run n iterations
3:         sum ← sum + M[j][k]              ▷ c1
                                            ▷ c2 additional cost of outer loop
                                            ▷ Time complexity = m(c2 + c1n)
```

The time complexity of this nested loop example is $m(c_2 + c_1 n)$. There are many additional costs in a loop. Not only $j++$, branch back to the first line of the loop and check the conditional statement require to take time. These operations are usually constant time. When m=n, the time complexity can be simplified to a funciton of $n^2$. The number of operations increases **quadratically** with $n$.

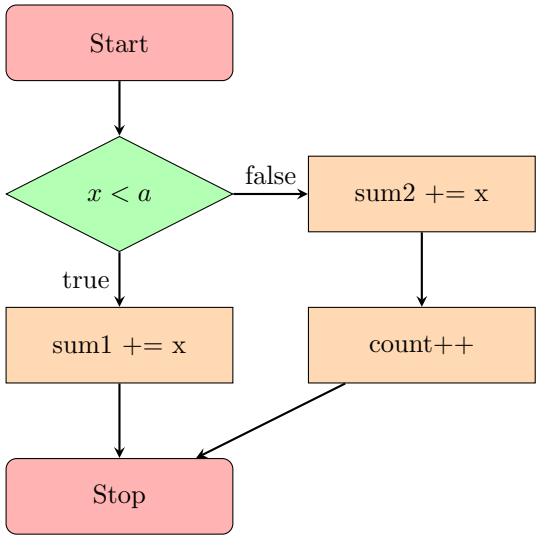
## 7.3 Different Cases of Complexity Analysis

When the algorithm involves selection structure, not all operations will be executed every time. Given different inputs, different operation blocks will be selected. Hence, the number of operations will be different. In algorithm analysis, we need to further consider the following three cases:

- Best-case analysis: The minimum number of primitive operations performed by the algorithm on any input of size $n$. It is known as **best-case time complexity**
- Worst-case analysis:The maximum number of primitive operations performed by the algorithm on any input of size $n$. It is known as **worst-case time complexity**
- Average-case analysis: The average number of primitive operations performed by the algorithm on all inputs of size $n$. It is known as **average time complexity**

### 7.3.1 Example 3: 'if-else' selection structure

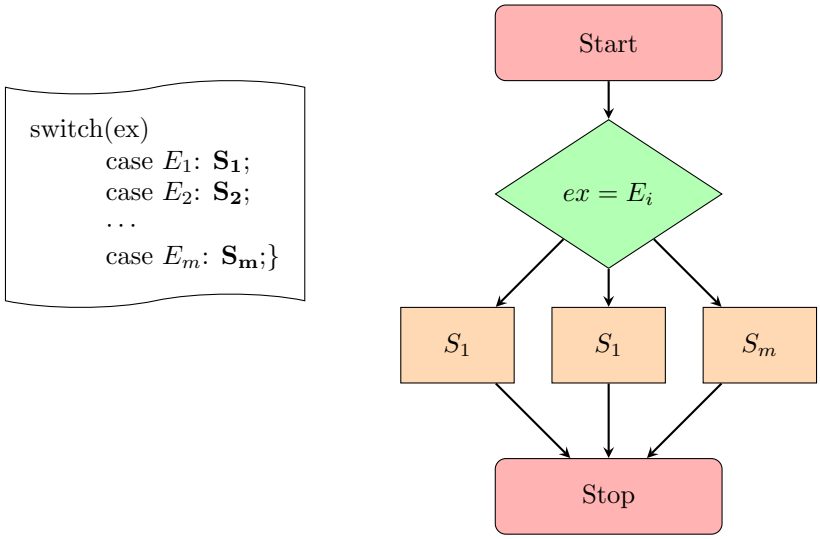
```
if(x < a)
        sum1 += x;
else{
        sum2+=x;
        count++;
        }
```



Let $c_1$ be the cost of $x < a$ branch, $c_2$ be the cost of its *else* case and $\mathbf{p}(<)$ be the probability that $x < a$ is true

- Best-case analysis: $c_1$
- Worst-case analysis: $c_2$
- Average-case analysis: $\mathbf{p}(<) * c_1 + (1 - \mathbf{p}(<)) * c_2$

### 7.3.2 Example 4: 'switch-case' multiple selection structure




Let **C** be the time complexity of the switch-case multiple selection structure, $\mathbf{T_i}$ be the time complexity of each block case. The three complexity analyses of the algorithm are:

- Best-case analysis: $C + T_{min}$
- Worst-case analysis: $C + T_{max}$
- Average-case analysis: $C + \sum_{i=1}^{m} \mathbf{p}(i)T_i$

```
switch (choice){
    case 1: computer the sum; break;     // 5n instructions
    case 2: search BST; break;           // 6lgn instructions
    case 3: print BST; break;            // 3n instructions
    case 4: search for the minimum; break; //4lgn instructions
}
```

Listing 1: switch-case Statement: The cost above are for examples only

- Best-case analysis: $C + T_{min} = C + 4\log_2 n$
- Worst-case analysis: $C + T_{max} = C + 5n$
- Average-case analysis: $C + \sum_{i=1}^{m} \mathbf{p}(i)T_i$
  Let's assume that the probabilities of cases are 0.1, 0.4, 0.3 and 0.2 respectively. Then we obtain $C + 1.4n + 3.2\log_2 n$

### 7.3.3 Example 5:

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

Listing 2: Searching item in a linked list

Let $c_1$ be the cost of checking the first node and $c_2$ be the cost of each iteration. Assuming the item, **a**, is always in the list, we have:

- Best-case analysis: $c_1$ when **a** is the first item in the list
- Worst-case analysis: $c_1 + c_2(n-1)$ when **a** is the last item in the list
- Average-case analysis: assuming the probability to search for any item is equal, $\frac{1}{n}$

$$\frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1)) = \frac{1}{n}[nc_1 + c_2\sum_{i=1}^{n-1}(i-1)]$$
$$= c_1 + \frac{c_2}{n}\frac{(n-1)(1+(n-1))}{2}$$
$$= c_1 + \frac{c_2(n-1)}{2}$$

## 7.4 Time Complexity of Recursive Functions

To obtain the time complexity of a recursive function, we need to determine:

- number of primitive operations for each recursive call
- number of recursive calls

The following example is the recursive version of Algorithm 1.

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

Let the cost of each recursive call when $n > 1$ be $c_1$ and when $n = 1$ be $c_2$.
The total number of recursive calls is $n-1$ (factorial(n-1), factorial(n-2), ..., factorial(2), factorial(1)).
Time complexity is

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

.

In the next example, the algorithm is counting the number of item, **a** in the array.

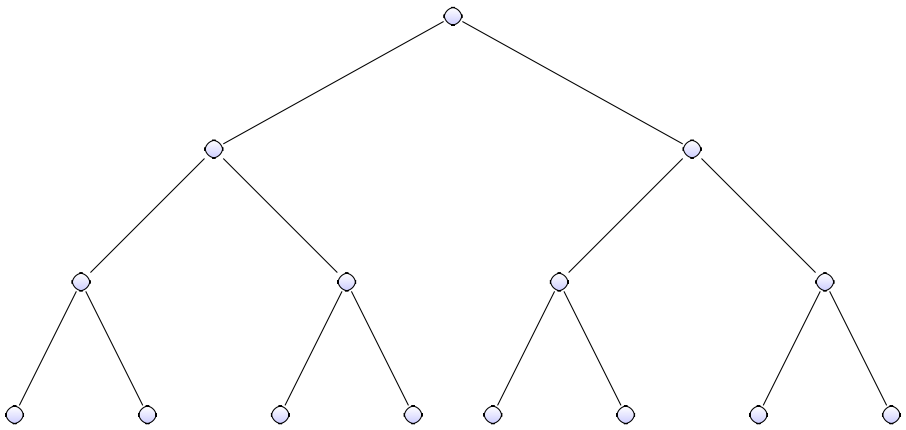
```
int count (int array[],int n, int a)
{
    if(n==1)
```

```
4           if(array[0]==a)
5              return 1;
6           else return 0;
7     if(array[0]==a)
8           return 1+ count(&array[1], n-1, a);
9     else
10          return count (&array[1], n-1, a);
11 }
```

Here we only determine the number of comparisons (array[0]==a).

$$
\begin{aligned}
W_1 &= 1 \\
W_n &= 1 + W_{n-1} \\
&= 1 + 1 + W_{n-2} \\
&= 1 + 1 + 1 + W_{n-3} \\
\ldots &= \ldots \\
&= 1 + 1 + \ldots + 1 + W_1 \\
&= n
\end{aligned}
$$

The total number of recursive call is $n-1$. We have done $n-1$ comparison. When $n = 1$, we do 1 comparison. Total number of comparison is $n$.
This is a method of backward substitutions.
When there are multiple recursive calls, the analysis becomes a bit complex. See the following binary tree example:



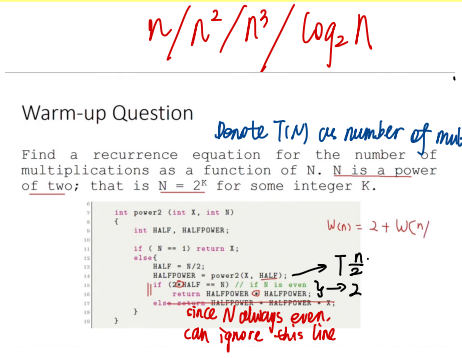
```
preorder (simple_t* tree)
{
   if(tree != NULL){
      tree->item *= 10;
      preorder (tree->left);
      preorder (tree->right);
   }
}
```

When the tree is empty (NULL), there is no recursive call and the function will simply return back to the caller. Let us assume that it is a complete binary tree. The number of nodes is $2^k - 1$ where $k$ is the depth of the binary tree.

$$
\begin{aligned}
W_0 &= 0 \\
W_1 &= 1 \\
W_2 &= 1 + W_1 + W_1 = 1 + 2 = 3 \\
W_3 &= 1 + W_2 + W_2 = 1 + 2(1 + 2) = 1 + 2 + 4 = 7 \\
&\ldots = \ldots \\
W_{k-1} &= 1 + 2 * W_{k-2} = 1 + 2 + 4 + 8 + \ldots + 2^{k-2} \\
W_k &= 2^k - 1
\end{aligned}
$$

We can easily observe that it is a geometric series. Since the function above is visiting every node of the binary tree, the number of recursive calls is the number of nodes, $2^k - 1$.
This is a method of forward substitutions.

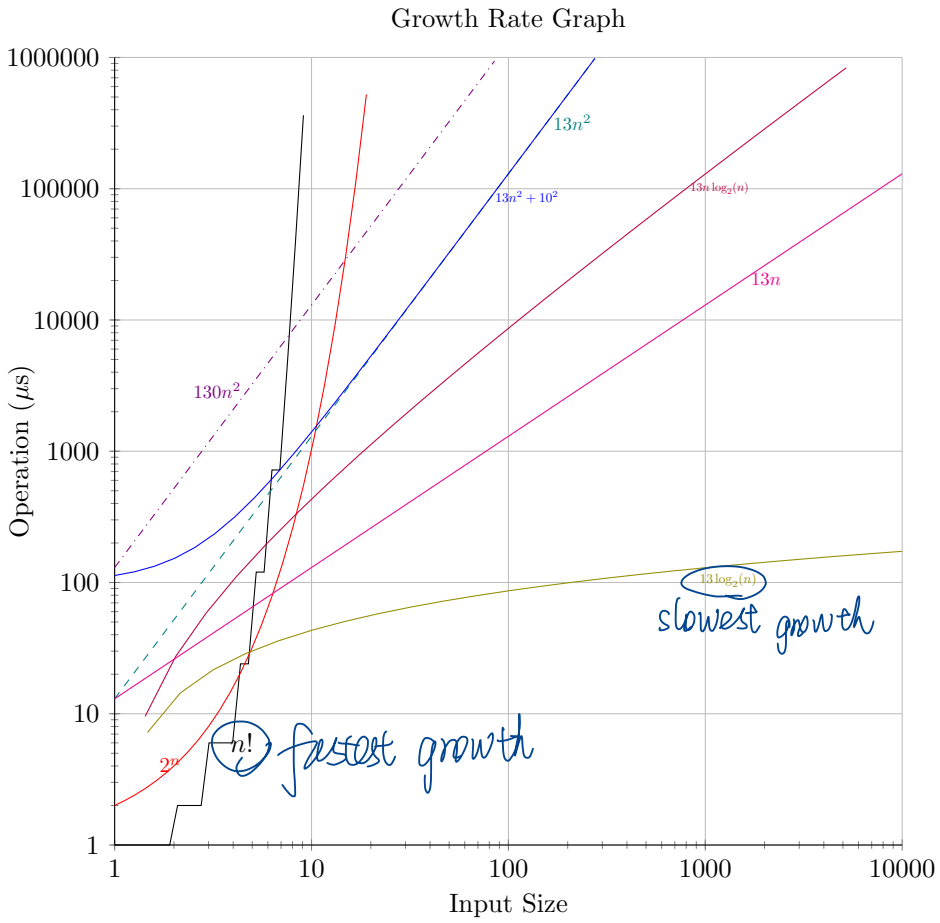
When we analyse the time complexity of an algorithm, we concern on its complexity when the problem size is large. When the problem size is large, some terms in the complexity may not be important. Thus, we are not really interested at its exact time complexity. We just would like to know its order of growth in practice.




## 7.5 Order of Growth

We are interested at growth rate of time complexity. To analysis the efficiency of an algorithm, we would like to know its running time for large input sizes. Thus, the constants are not significant and the multipliers are not important for relative growth rate.

| Algorithm | linear | linearithmic | quadratic 1 | quadratic 2 | quadratic 3 | exponential |
|---|---|---|---|---|---|---|
| Input / Operation(s) | $13n$ | $13n \log_2 n$ | $13n^2$ | $130n^2$ | $13n^2 + 10^2$ | $2^n$ |
| 10 | 0.00013 | 0.00043 | 0.0013 | 0.013 | 0.0014 | 0.001024 |
| 100 | 0.0013 | 0.0086 | 0.13 | 1.3 | 0.1301 | $4 \times 10^{16}$ years |
| $10^4$ | 0.13 | 1.73 | 22mins | 3.61hrs | 22mins | |
| $10^6$ | 13 | 259 | 150days | 1505days | 150days | |




From the growth rate functions above, we can observe the following characteristic of functions

1. The factorial of n ($n!$) is the fastest growth when $n > 10$.
2. When n is large enough, the growth rate is in the following order

$$\text{constant} < \log_{10}(n) < \log_2(n) < n < nlog_2(n) < n^2 < n^3 < 10n^3 < 2^n < n! < n^n$$

3. When n is large enough, the constant, $10^2$, can be ignored
4. $13n^2$ and $130n^2$ are almost parallel when n is large. It implies that both have similar growth rate but $130n^2$ is slightly faster.

### 7.5.1 Faster Computer Versus Faster Algorithm

Can we simply use faster computer to resolve the 'difficult' computation problems? The answer is NO.
To illustrate this problem, let us compare an old computer with a 10× faster new computer. The old computer executes 10k basic operations per hour and the new computer can execute 100k operations per hour. The following table shows the problem size can be solved by the old computer ($n$) and the new computer ($n'$) in an hour.

| $f(n)$ | $n$ | $n'$ | Change | $\frac{n'}{n}$ |
|---|---|---|---|---|
| $10n$ | 1000 | 10k | $n' = 10n$ | 10 |
| $20n$ | 500 | 5k | $n' = 10n$ | 10 |
| $5n \log n$ | 250 | 1842 | $3.16n < n' < 10n$ | 7.37 |
| $2n^2$ | 70 | 223 | $n' = 3.16n$ | 3.16 |
| $2^n$ | 13 | 16 | $n' = n + 3$ | 1.23 |

For linear complexity problem, the improvement is $10\times$. For harder problems with faster-growing function, the improvement is poorer than the linear problems. The exponential complexity problem is hardly improved by using $10\times$ faster computer. We only manage to increase the number of data size from 13 to 16.

Compare $n^2$ algorithm with $n \log n$ algorithm When data size, $n = 1024$,

- $n^2$ algorithm takes $1024 \times 1024 = 1,048,576$ primitive steps
- $n \log n$ algorithm takes $1024 \times \log 1024 = 10,240$ primitive steps

The improvement from the $n^2$ to the $n \log_2 n$ is a factor of **100**.
When data size increases to $n = 2048$,

- $n^2$ algorithm takes $2048 \times 2048 = 4,194,304$ primitive steps
- $n \log n$ algorithm takes $2048 \times \log 2048 = 22,528$ primitive steps

The improvement from the $n^2$ to the $n \log_2 n$ is a factor of **200**.

Moore's Law asserts that the number of transistors on a microchip doubles every two years, though the cost of computers is halved. In other words, we can expect that the speed and capability of our computers will increase every couple of years, and we will pay less for them. However, we can observe that the computers is getting closer to the physical limits of Moore's Law. It is hardly to make the clock speed beyond 5GHz. We are trying to use parallel process to improve the performance. Moreover, It is always good that you have a more efficient algorithm.

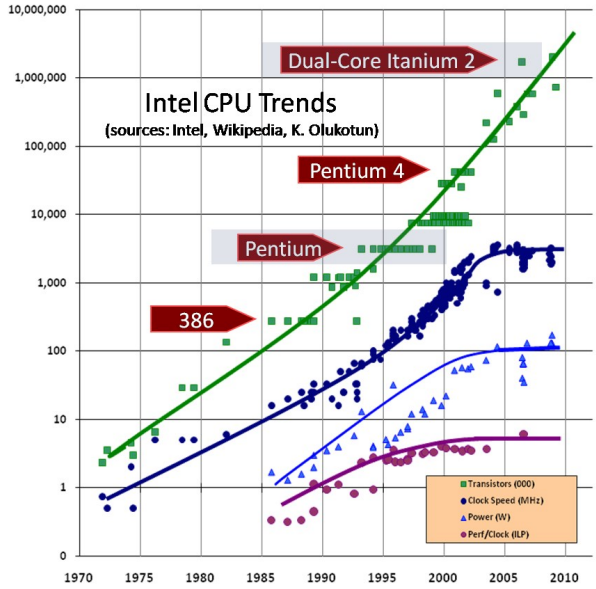



Figure 7.1: Moore's Law: https://cs.stackexchange.com/questions/27875/moores-law-and-clock-speed

## 7.6 Asymptotic Notation

General

When we consider the order of growth and efficiency of an algorithm, three asymptotic notations: $\Omega$ (big-Omega), $\Theta$ (big-Theta), $\mathcal{O}$ (big-Oh) are used.

最常用(基本想要知道最坏情况) (Know bound above. ⇒ Don't know the best case)

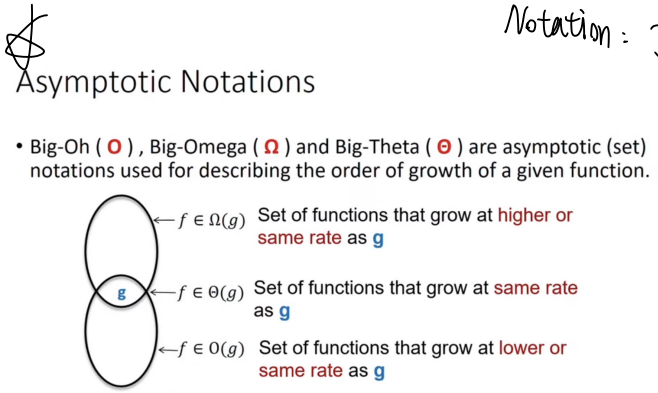
### 7.6.1 Big-Oh Notation: $\mathcal{O}$

(还需要 generally describe function 的时候常用)

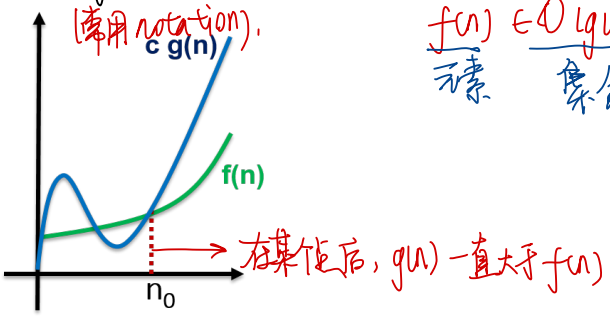
**Definition 7.1** *$\mathcal{O}$-notation: Let $f$ and $g$ be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\mathcal{O}(g(n))$, denoted $f(n) \in \mathcal{O}(g(n))$, if $f(n)$ is **bounded above** by some constant multiple of $g(n)$ for all large $n$, i.e., the set of functions can be defined as*

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

Notation: $f(n) = O(g(n))$ (实际notation should be $f(n) \in O(g(n))$ 集合 表示)




(常用 notation).

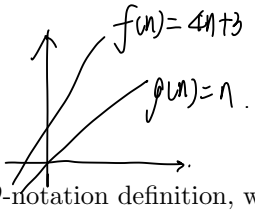



存某值后，g(n) 一直大于 f(n)

Figure 7.2: Big-Oh Notation

次数相同时，f和g的关系同 Ω and O (但是后来找的 notation).

example: $f(n) = n$, $g(n) = n^3$. $\therefore f \in O(g) \;/\; g \in \Omega(f)$




**Example 1 (a):**
Given that $f(n) = 4n + 3$ and $g(n) = n$,
If let $c = 5$ and $n_0 = 3$, then based on $\mathcal{O}$-notation definition, we have

$$f(n) \leq cg(n) \quad \forall n \geq n_0$$
$$f(n) \leq 5g(n) \quad \forall n \geq 3$$

$\therefore f(n) = \mathcal{O}(g(n))$ or $4n + 3 \in \mathcal{O}(n)$

**Example 2 (a):**
Given that $f(n) = 4n + 3$ and $g(n) = n^3$,
If let $c = 1$ and $n_0 = 3$, then based on $\mathcal{O}$-notation definition, we have

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

$\therefore f(n) = \mathcal{O}(g(n))$ or $4n + 3 \in \mathcal{O}(n^3)$

The following alternative definition of $\mathcal{O}$-notation can help you to find the complexity class of the given function easily via their limit

**Definition 7.2** *$\mathcal{O}$-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 $f(n) = \mathcal{O}(g(n))$.*

$\therefore = C$ eq

**Example 1 (b) :**
Given that $f(n) = 4n + 3$ and $g(n) = n$,

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

$\therefore f(n) = \mathcal{O}(g(n))$ or $4n + 3 \in \mathcal{O}(n)$

**Example 2 (b):**
Given that $f(n) = 4n + 3$ and $g(n) = n^3$,

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

$\therefore f(n) = \mathcal{O}(g(n))$ or $4n + 3 \in \mathcal{O}(n^3)$

In certain cases, we may need to use L'Hôpital's Rule **Example 3 :**
Given that $f(n) = 4n + 3$ and $g(n) = e^n$,

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

Apply L'Hôpital's Rule to find $\lim_{n\to\infty} \frac{4n+3}{e^n}$

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

$\therefore f(n) = \mathcal{O}(g(n))$ or $4n+3 \in \mathcal{O}(e^n)$

### 7.6.2 Big-Omega Notation: $\Omega$

**Definition 7.3** *$\Omega$-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*

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

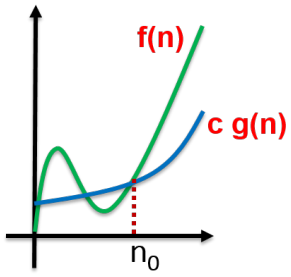



Figure 7.3: Big-Omega Notation

The following alternative Definition of $\Omega$-notation can help you to find the complexity class of the given function easily via their limit

**Definition 7.4** *$\Omega$-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 $f(n) = \Omega(g(n))$.*

$\therefore$ By definition, $= C \in \Omega$

**Example 1 (a):**
Given that $f(n) = 4n+3$ and $g(n) = 5n$,
Let $c = \frac{1}{5}$, $n_0 = 0$
Then

$$f(n) \geq \frac{1}{5} g(n)$$
$$4n+3 \geq n \quad \forall n \geq 0$$

$\therefore f(n) = \Omega(g(n))$ or $4n+3 \in \Omega(n)$

**Example 1 (b):**
Given that $f(n) = 4n + 3$ and $g(n) = 5n$,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3}{5n}$$
$$= \frac{4}{5} > 0$$

$\therefore f(n) = \Omega(g(n))$ or $4n + 3 \in \Omega(n)$

**Example 2:**
Given that $f(n) = n^3 + 2n$ and $g(n) = 5n$,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3+2n}{5n}$$
$$= \infty > 0$$

$\therefore f(n) = \Omega(g(n))$ or $4n + 3 \in \Omega(5n)$

### 7.6.3 Big-Theta Notation: $\Theta$

常用于 describe worst case (very sure tight bound) 不适合描述整个函数

**Definition 7.5** *$\Theta$-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*

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

The following alternative Definition of $\Omega$-notation can help you to find the complexity class of the given function easily via their limit

**Definition 7.6** *$\Theta$-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 $f(n) = \Theta(g(n))$.*

**Example 1 (a):**
Given that $f(n) = 2n^2 + 7$ and $g(n) = 7n^2 + n$,
Using alternative definition above,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2n^2+7}{7n^2+n}$$
$$= \frac{2}{7}$$

$\therefore f(n) = \Theta(g(n))$ or $4n + 3 \in \Theta(n)$

### 7.6.4 Summary of Asymptotic Notation

| $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ | $f(n) \in \mathcal{O}(g(n))$ | $f(n) \in \Omega(g(n))$ | $f(n) \in \Theta(g(n))$ |
|---|---|---|---|
| 0 | ✓ | | |
| $0 < c < \infty$ | ✓ | ✓ | ✓ |
| $\infty$ | | ✓ | |

The $\mathcal{O}$, $\Omega$ and $\Theta$ notations are used in studying the asymptotic efficiency of an algorithm.

- If $f(n) = \mathcal{O}(g(n))$, it implies that $g(n)$ is asymptotic upper bound of $f(n)$
- If $f(n) = \Omega(g(n))$, it implies that $g(n)$ is asymptotic lower bound of $f(n)$
- If $f(n) = \Theta(g(n))$, it implies that $g(n)$ is asymptotic tight bound of $f(n)$

In practice, $\mathcal{O}$-notation is the most useful notation.

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

### 7.6.5 How to determine the big-Oh notation from an algorithm?

1. Count primitive operations to derive complexity function $f(n)$ (in terms of problem size)
2. Discard constant terms and multipliers in $f(n)$
3. Determine dominant term in $f(n)$
4. Dominant term = big-Oh notation for $f(n)$ (= big-Oh notation for algorithm)

The dominant term can refer 7.6.7.

### 7.6.6 Asymptotic Notation in Equations and Its Simplification

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**:

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
- $T(n) = T(\frac{n}{2} + \Theta(n)$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

Some simplification rules for asymptotic analysis:

Simplification rules

1. If $f(n) = \mathcal{O}(cg(n))$ for any constant $c > 0$, then $f(n) = \mathcal{O}(g(n))$ (常数可消)
2. If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$. (传递性)
   e.g. $f(n) = 2n$, $g(n) = n^2$, $h(n) = n^3$
   $\Rightarrow f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$
   $\therefore 2n = \mathcal{O}(n^3)$
3. If $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$, then $f_1(n) + f_2(n) = \mathcal{O}(\max(g_1(n), g_2(n)))$
   e.g. $5n + 3\lg n = \mathcal{O}(n)$ (最差情况考虑)
4. If $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$, then $f_1(n)f_2(n) = \mathcal{O}(g_1(n)g_2(n))$
   e.g. $f_1(n) = 3n^2$, $f_2(n) = \lg n$, $f_1(n) = \mathcal{O}(n^2)$, $f_2(n) = \mathcal{O}(\lg n)$,
   then $3n^2 \lg n = \mathcal{O}(n^2 \lg n)$ (乘法规律)

   Some properties of $\mathcal{O}$, $\Omega$ and $\Theta$:

   - $\mathcal{O}$, $\Omega$ and $\Theta$ are **Reflexive**: → 自我传递
     - $f(n) = \mathcal{O}(f(n))$
     - $f(n) = \Omega(f(n))$
     - $f(n) = \Theta(f(n))$
   - $\mathcal{O}$, $\Omega$ and $\Theta$ are **Transitive**:
     - If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$
     - If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$
     - If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$
   - $\Theta$ is **Symmetric**:
     - If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$ because it is tight bounded.

If $O(f(n)) = O(g(n))$ Cannot compare
If $\Theta(f(n)) = \Theta(g(n))$ can compare

### 7.6.7 Common Complexity Classes

based on Big-O, not case analysis

| Order of Growth | Class | Example |
|---|---|---|
| 1 | constant | Finding midpoint of an array |
| $\log n$ | 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 |

1. Constant order: the running time is independent to the problem size, $n$. It denotes as $f(n) \in \mathcal{O}(1)$
   **Example:** $sum = \frac{n}{2}(n+1)$;
   We can count in the statement, 1 addition, 1 multiplication, 1 division and 1 assignment. There are 4 operations, which is independent of n. We have $f(n) = 4$, which means $f(n)$ is big Oh of 4, i.e. $\mathcal{O}(4)$. Formally, if you wish to verify that 4 is in $\mathcal{O}(1)$, you can pick $c = 4$ and any $n_o = 0$, such that $\forall n \geq 0$, $f(n)$, which is $\leq 4 \times 1$. As such, we can deduce that $f(n) \in \mathcal{O}(1)$.

2. Logarithmic order: $f(n) \in \mathcal{O}(\log n)$. $\log n$ grows slower than $n$ which means the running time of $f(n)$ increases slower than its problem size n.
**Example:**

update statement

```
for (i=n; i>=1; i/=2)
    sum++;
```

It is noted that this example let $i = n$ and $i$ is reduced to $\frac{n}{2}$, then $\frac{n}{4}$ until it reaches 1. Details will be discussed in the tutorial but you can see that it will take $\lfloor \log_2 n \rfloor + 1$ iteration.
$\therefore f(n) \in \mathcal{O}(\log n)$

**Note 1**: Prove that Growth rate of $\log n$ is slower than $n^\varepsilon$ for all $\varepsilon > 0$

$$\begin{aligned}\lim_{n\to\infty} \frac{\log n}{n^\varepsilon} &= \lim_{n\to\infty} \frac{\frac{1}{\ln 10}\cdot\frac{1}{n}}{\varepsilon n^{\varepsilon-1}} \\ &= \lim_{n\to\infty} \frac{c}{\varepsilon n^\varepsilon} \\ &= 0\end{aligned}$$

**Note 2**: Base of log is convertible with a constant multiplier. Thus it is not important. $\log_b n = \frac{\log_c n}{\log_c b}$ where $\log_c b$ is a constant

3. Linear Order: $f(n) \in \mathcal{O}(n)$

```
for (i=1; i<=n; j++)
    sum++;
```

The number of iterations is $n$. $\therefore f(n) \in \mathcal{O}(n)$

4. Linearithmic Order:$f(n) \in \mathcal{O}(n \log n)$. It is commonly seen in algorithms that break a problem into sub-problems, solve them independently and combine the solutions. e.g. merge sort.
For example, consider this set of recurrent equation, that represents the time complexity function of a recursive algorithm.
$W(2) = 1$
$W(n) = 2W(\frac{n}{2}) + n - 1$
After you solve this recurrent equation, you will obtain a complexity class of $n \log n$. Please try to derive it out.

5. Polynomial Order:$f(n) \in \mathcal{O}(n^p)$ for $\exists p \in \mathbb{N}$
**Example:**

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        for (k=1; k<=n; k++)
            M[i][j] = A[i][k]*B[k][j];
```

Consider this piece of codes above, consisting of a triple nested for loop, the number of primitive operations is proportional to $n^3$, where $n$ is the problem size.
$\therefore f(n) \in \mathcal{O}(n^3)$

6. Exponential Order:$f(n) \in \mathcal{O}(a^n)$ for $\exists a \in \mathbb{N}$ usually it is not practical for normal use especially when the problem size is large. **Example:**Print all subsets of a set of n elements
$\therefore f(n) \in \mathcal{O}(2^n)$

Example 6: 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);
  }
}
```

## 7.7 Space Complexity (Not so important nowadays)

For space complexity, we count the number of basic storage units in an algorithm. We first determine the number entities in problem (or problem size, $n$). Instead of count the number of primitive operations, we concern about the storage usage of the algorithm. The storage units can be integer (**int**), floating point number (**float**), or character (**char**).
**Example**:

1. The space complexity of an array of $n$ integers is $\mathbf{\Theta(n)}$.
2. A matrix used for storing edge information of a graph, i.e. $G[x][y] = 1$ if there exists an edge from $x$ to $y$. The space complexity of a graph with $n$ vertices is $\mathbf{\Theta(n^2)}$.

**Spaece/ Time Tradeoff Principle** It is important to note that there is typically a tradeoff between space complexity and time complexity. In other words, the reduction in time complexity can be achieved by sacrificing space complexity and vice versa. We shall see some examples of this in the later part of this course.

Additional Materials

# The Growth of Functions and Big-O Notation

## Introduction

Note: "big-O" notation is a generic term that includes the symbols O, $\Omega$, $\Theta$, o, $\omega$.

Big-O notation allows us to describe the aymptotic growth of a function $f(n)$ without concern for i) constant multiplicative factors, and ii) lower-order additive terms. By *asymptotic growth* we mean the growth of the function as input variable $n$ gets arbitrarily large. As an example, consider the following code.

```
int sum = 0;

for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        sum += (i+j)/6;
```

How much CPU time $T(n)$ is required to execute this code as a function of program variable $n$? Of course, the answer will depend on factors that may be beyond our control and understanding, including the language compiler being used, the hardware of the computer executing the program, the computer's operating system, and the nature and quantity of other programs being run on the computer. However, we can say that the outer `for` loop iterates $n$ times and, for each of those iterations, the inner loop will also iterate $n$ times for a total of $n^2$ iterations. Moreover, for each iteration will require approximately a constant number $c$ of clock cycles to execture, where the number of clock cycles varies with each system. So rather than say "$T(n)$ is approximately $cn^2$ nanoseconds, for some constant $c$ that will vary from person to person", we instead use big-O notation and write $T(n) = \Theta(n^2)$ nanoseconds. This conveys that the elapsed CPU time will grow quadratically with respect to the program variable $n$.

The following table shows the most common kinds of growth that are used within big-O notation.

| **Function** | **Type of Growth** |
|---|---|
| $1$ | constant growth |
| $\log n$ | logarithmic growth |
| $\log^k n$, for some integer $k \geq 1$ | polylogarithmic growth |
| $n^k$ for some positve $k < 1$ | sublinear growth |
| $n$ | linear growth |
| $n \log n$ | log-linear growth |
| $n \log^k n$, for some integer $k \geq 1$ | polylog-linear growth |
| $n^j \log^k n$, for some integers $j, k \geq 1$ | polylog-polynomial growth |
| $n^2$ | quadratic growth |
| $n^3$ | cubic growth |
| $n^k$ for some integer $k \geq 1$ | polynomial growth |
| $2^{\log^c n}$, for some $c > 1$ | quasi-polynomial growth |
| $a^n$ for some $a > 1$ | exponential growth |

Generally speaking, two functions $f(n)$ and $g(n)$ are said to have the same asymptotic growth provided their growths differ by some positive constant factor $c > 0$. In this case we may say $f(n) = \Theta(g(n))$ or, equivalently, $g(n) = \Theta(f(n))$.

**Big-$\Theta$ Notation.** We say that $f(n) = \Theta(g(n))$ provided there is a constant $c > 0$ for which

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

Note that this definition is narrower than what is stated in most textbooks, but it will suffice for this course.

**Example 1.** Prove that if $f(n) = 3n^2 + 6n + 7$, then $f(n) = \Theta(n^2)$. In other words, $f(n)$ has quadratic growth.

**Example 2.** Suppose $a > 1$ and $b \neq 0$ are constants, with $|b| < a$. Prove that $a^n + b^n = \Theta(a^n)$.

**Example 3.** Verify that $f(n)$ from Example 1 does not satisfy $f(n) = \Theta(n^3)$.

# Little-o and Little-$\omega$

**Little-o and Little-$\omega$ Notation.** Given two functions $f(n)$ and $g(n)$. $f(n) = \mathrm{o}(g(n))$ iff

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Also, $f(n) = \omega(g(n))$ iff

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

**Example 4.** From Example 3, we see that $3n^2 + 6n + 7 = o(n^3)$, and hence $n^3 = \omega(n^2)$.

**Theorem 1.** The following are all true statements.

1. $1 = o(\log n)$
2. $\log n = o(n^\epsilon)$ for any $\epsilon > 0$
3. $\log^k n = o(n^\epsilon)$ for any $k > 0$ and $\epsilon > 0$
4. $n^a = o(n^b)$ if $a < b$, and $n^a = \Theta(n^b)$ if $a = b$.
5. $n^k = o(2^{\log^c n})$, for all $k > 0$ and $c > 1$.
6. $2^{\log^c n} = o(a^n)$ for all $a, c > 1$.
7. For nonnegative functions $f(n)$ and $g(n)$, if

   $$\lim_{n\to\infty} \frac{f(n)}{\max(f,g)(n)} \text{ and } \lim_{n\to\infty} \frac{g(n)}{\max(f,g)(n)}$$

   both exist, then $(f + g)(n) = \Theta(\max(f, g)(n))$.
8. For $f(n) = \Theta(h(n))$ and $g(n) = \Theta(k(n))$, then $f(n)g(n) = \Theta((hk)(n))$.

**Example 5.** Use the results of Theorems 1 to determine the growth of $(fg)(n)$, where $f(n) = n\log(n^4+1) + n(\log n)^2$ and $g(n) = n^2 + 2n + 3$.

**L'Hospital's Rule.** Suppose $f(n)$ and $g(n)$ are both differentiable functions with either

1. $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, or
2. $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = 0$.

Then

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

**Example 6.** Prove Theorem 1.2 using L'Hospital's Rule.

# Big-O and Big-$\Omega$

Suppose we have a function $f(n)$ defined as follows.

$$f(n) = \begin{cases} 2n & \text{if n is even} \\ 3n^2 & \text{if n is odd} \end{cases}$$

What is the growth of $f(n)$? Unfortunately, we can neither say that $f(n)$ has linear growth, nor can we say it has quadratic growth. This is because neither of the limits

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

and

$$\lim_{n\to\infty} \frac{f(n)}{n^2}$$

exist, since infinitely often $f(n)$ jumps from being quadratic to linear, back to quadratic, back to linear, etc.. The best we can do is provide a lower and upper bound for $f(n)$.

**Big-O** $f(n) = \mathrm{O}(g(n))$ iff $f(n)$ does not grow any faster than $g(n)$. In other words, $f(n) \leq Cg(n)$ for some constant $C > 0$.

**Big-$\Omega$** $f(n) = \Omega(g(n))$ iff $f(n)$ grows at least as fast as $g(n)$. In other words, $f(n) \geq Cg(n)$ for some constant $C > 0$.

Using these definitions, we see that $f(n) = \mathrm{O}(n^2)$ and $f(n) = \Omega(n)$.

**Example 7.** Suppose function $f(n)$ defined as follows.

$$f(n) = \begin{cases} 2n^{2.1}\log^3 & \text{if } n \bmod 3 = 0 \\ 10n^2\log^{50} n & \text{if } n \bmod 3 = 1 \\ 6n^2\log^{30} n^{80} & \text{if } n \bmod 3 = 2 \end{cases}$$

Provide a big-O upper bound and big-$\Omega$ lower bound for $f(n)$.

**Example 8.** Let $T(n)$ denote the CPU time needed to execute the following code as a function of program variable $n$.

```
//Linear Search for x in array a
Boolean linear_search(int a[ ], int n, int x)
{
    int i;

    for(i=0; i < n; i++)
        if(a[i] == x)
            return true; //found x

    return false; //x is not a member of a[]
}
```

Provide a big-O upper bound and big-$\Omega$ lower bound for $T(n)$.

**Theorem 2 (Log Ratio Test).** Suppose $f$ and $g$ are continuous functions over the interval $[1, \infty)$, and

$$\lim_{n\to\infty} \log(\frac{f(n)}{g(n)}) = \lim_{n\to\infty} \log(f(n)) - \log(g(n)) = L.$$

Then

1. If $L = \infty$ then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$
2. If $L = -\infty$ then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$
3. If $L \in (-\infty, \infty)$ is a constant then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 2^L.$$

**Example 9.** Use the log-ratio test to determine show that $f(n) = n^{\log n}$ has quasi-polynomial growth.

**Example 9 Solution.**

Consider the problem of determining the big-O growth of the series

$$\sum_{i=1}^{\infty} f(i),$$

where $f$ is some nonnegative integer function. In other words, we want to determine the growth of the function $S(n) = f(1) + f(2) + \cdots + f(n)$ which is called the $n$ th **partial sum** of the series.

**Example 10.** Determine the growth of $S(n)$ in case i) $f(i) = i$, ii) $f(i) = i^2$, and iii) $f(i) = i^3$.

**Example 10 Solution.**

Unfortunately, the $n$ th partial sum function $S(n)$ for many important series cannot be expressed using a formula. However, the next result shows how we may still determine the big-O growth of $S(n)$, which quite often is our main interest.

**Integral Theorem.** Let $f(x) > 0$ be an increasing or decreasing Riemann-integrable function over the interval $[1, \infty)$. Then

$$\sum_{i=1}^{n} f(i) = \Theta(\int_1^n f(x)dx),$$

if $f$ is decreasing. Moreover, the same is true if $f$ is increasing, provided $f(n) = \mathrm{O}(\int_1^n f(x)dx)$.

**Proof of Integral Theorem.** We prove the case when $f$ is decreasing. The case when $f$ is increasing is left as an exercise. The quantity $\int_1^n f(x)dx$ represents the area under the curve of $f(x)$ from 1 to $n$. Moreover, for $i = 1, \ldots, n-1$, the rectangle $R_i$ whose base is positioned from $x = i$ to $x = i + 1$, and whose height is $f(i + 1)$ lies under the graph. Therefore,

$$\sum_{i=1}^{n-1} \mathrm{Area}(R_i) = \sum_{i=2}^{n} f(i) \leq \int_1^n f(x)dx.$$

Adding $f(1)$ to both sides of the last inequality gives

$$\sum_{i=1}^{n} f(i) \leq \int_1^n f(x)dx + f(1).$$

Now, choosing $C > 0$ so that $f(1) = C \int_1^n f(x)dx$ gives

$$\sum_{i=1}^{n} f(i) \leq (1 + C) \int_1^n f(x)dx,$$

which proves $\sum_{i=1}^{n} f(i) = \mathrm{O}(\int_1^n f(x)dx)$.

Now, for $i = 1, \ldots, n-1$, consider the rectangle $R_i'$ whose base is positioned from $x = i$ to $x = i+1$, and whose height is $f(i)$. This rectangle covers all the area under the graph of $f$ from $x = i$ to $x = i + 1$. Therefore,

$$\sum_{i=1}^{n-1} \mathrm{Area}(R_i') = \sum_{i=1}^{n-1} f(i) \geq \int_1^n f(x)dx.$$

Now adding $f(n)$ to the left side of the last inequality gives

$$\sum_{i=1}^{n} f(i) \geq \int_1^n f(x)dx,$$

which proves $\sum_{i=1}^{n} f(i) = \Omega(\int_1^n f(x)dx)$.

Therefore,

$$\sum_{i=1}^{n} f(i) = \Theta(\int_1^n f(x)dx).$$

**Example 11.** Determine the big-O growth of the series

$$\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}.$$

**Example 11 Solution.**

# Exercises

1. An algorithm takes 0.5 seconds to run on an input of size 100. How long will it take to run on an input of size 1000 if the algorithm has a running time that is linear? quadratic? cubic?

2. An algorithm is to be implemented and run on a processor that can execute a single instruction in an average of $10^{-9}$ seconds. What is the largest problem size that can be solved in one hour by the algorithm on this processor if the number of steps needed to execute the algorithm is $n$? $n^2$?, $n^3$? $1.3^n$? Assume $n$ is the input size.

3. Suppose that the Insertion Sort sorting algorithm has a running time of $T(n) = 8n^2$, while the Counting Sort algorithm has a running time of $T(n) = 64n$. Find the largest positive input size for which Insertion Sort runs at least as fast as Counting Sort.

4. If you were given a full week to run your algorithm, which has running time $T(n) = 5 \cdot 10^{-9}(n^3)$ seconds, what would be the largest input size that could be used, and for which your algorithm would terminate after one week? Explain and show work.

5. Use big-O notation to state the growth of $f(n) = n + n \log n^2$. Defend your answer.

6. Which function grows faster: $f(n) = n \log^2 n$ or $g(n) = n^{0.3} \log^{36} n$.

7. Prove that $f(n) = \mathrm{O}(g(n))$ if and only if $g(n) = \Omega(f(n))$.

8. Prove that $(n + a)^b = \Theta(n^b)$, for all real $a$ and $b > 0$.

9. Prove that if $\text{limit}_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = \mathrm{O}(g(n))$, but $g(n) \neq \mathrm{O}(f(n))$.

10. Prove or disprove: $2^{n+1} = \Theta(2^n)$.

11. Prove or disprove: $2^{2n} = \Theta(2^n)$.

12. Use big-O notation to state the growth of the expression

$$\log^{50}(n)n^2 + \log(n^4)n^{2.1} + 1000n^2 + 100000000n.$$

13. Prove or disprove: if $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$.

14. If $g(n) = o(f(n))$, then prove that $f(n) + g(n) = \Theta(f(n))$.

15. Use L'Hospital's rule to prove that $a^n = \omega(n^k)$, for every real $a > 1$ and integer $k \geq 1$.

16. Prove that $\log_a n = \Theta(\log_b n)$ for all $a, b > 0$.

17. Suppose function $f(n)$ defined as follows.

$$f(n) = \begin{cases} 4n^{2.3} \log^3 n & \text{if } n \bmod 3 = 0 \\ 2^{\log^{2.2} n} & \text{if } n \bmod 3 = 1 \\ 10n^{2.2} \log^{35} n & \text{if } n \bmod 3 = 2 \end{cases}$$

Provide a big-O upper bound and big-$\Omega$ lower bound for $f(n)$.

18. Let $T(n)$ denote the CPU time needed to execute the following code as a function of program variable $n$. Hint: assume $x$ is some integer input that is capable of assuming any integer value.

```
    for(i=0; i < n; i++)
    {
        if(x % 2 == 0)
        {
            for(j=0; j < n; j++)
                sum++;
        }
        else
            sum += x;
    }
}
```

    Provide a big-O upper bound and big-$\Omega$ lower bound for $T(n)$.

19. Prove that $f(n) = 2^{\sqrt{2\log n}}$ is $\omega(g(n))$, for any poly-log function $g(n)$, but is $o(g(n))$ for any sub-linear function $g(n)$. Explain and show work.

20. Determine the big-O growth of the function $f(n) = (\log n)^{\log n}$. Explain and show work.

21. Determine the big-O growth of the function $f(n) = (\sqrt{2})^{\log n}$. Explain and show work.

22. Determine the big-O growth of the function $f(n) = n^{1/\log n}$.

23. Use the integral theorem to establish that $1^k + 2^k + \cdots + n^k = \Theta(n^{k+1})$, where $k \geq 1$ is an integer constant.

24. Use the Integeral Theorem to prove that $\log 1 + \log 2 + \cdots + \log n = \Theta(n \log n)$.

25. Show that $\log(n!) = \Theta(n \log n)$.

26. Prove that $n! = \omega(2^n)$.

27. Determine the big-O relationship between $2^{2^n}$ and $n!$.

28. Determine the big-O growth of the function $f(n) = (\log n)!$.

29. Determine the big-O growth of $n + n/2 + n/3 + \cdots + 1$.

30. Suppose we want to prove that statement $P(n)$ is true for every $n \geq 1$. The **principle of mathematical induction** can establish this using the following two steps. **Basis step**: show $P(1)$ is true. **Inductive step**: assume $P(n)$ is true for *some* $n \geq 1$, and show that this implies $P(n+1)$ is true. The assumption in Step ii is called the **inductive assumption**. Use mathematical induction to prove that

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

31. Use mathematical induction to prove that $1 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

32. Use mathematical induction to prove that, for all integers $k \geq 1$ and some $\epsilon > 0$, $\log^k n = o(n^\epsilon)$.

# Exercise Hints and Solutions

1. The input size has grown by a factor of 10. For linear time, the running time will also grow by a factor of 10. For quadratic, it will grow by a factor of $10^2 = 100$, to 50 seconds. For cubic, it will grow by a factor of $10^3 = 1000$, to 500 seconds. For log linear, we can first solve for the running time cofficient $C$ (i.e. $T(n) = Cn\log(n)$). This yields $C = 0.5/(100\log(100)$. Then use this value to compute $T(1000)$.

2. The elapsed time will be $10^{-9}T(n)$ seconds which must not exceed 3600. Thus we must have $T(n) \leq (3600)(10^9)$, which implies $n = \lfloor T^{-1}((3600)(10^9))\rfloor$. For example, if $T(n) = n^2$, then $T^{-1}(n) = \sqrt{n}$. In this case $n = \lfloor\sqrt{(3600)(10^9)}\rfloor = 1897366$. In the case when $T(n) = n\log n$, use a graphing calculator to determine the value of $n$ for which $T(n)$ is approximately $(3600)(10^9)$. We must do this because there is no formula for computing $T^{-1}$.

3. Solve the quadratic equation $8n^2 = 64n$ to get $n = 8$. For $n \geq 9$, Counting Sort will run faster.

4. One week contains $(3600)(24)(7) = 604800$ seconds. Thus, the largest input that can be solved in one week is $n = \lfloor(((604800)(10^9)/5)^{1/3}\rfloor = 49455$. So even though you have an entire week to run the program, the largest problem that can be solved will be in the tens of thousands.

5. We have
$$f(n) = n + n\log n^2 = n + 2n\log n = \Theta(n\log n),$$
since $n = o(n\log n)$.

6. Function $f(n)$ grows faster since
$$\lim_{n\to\infty}\frac{n\log^2 n}{n^{0.3}\log^{36} n} = \lim_{n\to\infty}\frac{n^{1.7}}{\log^{34} n} = \infty$$
since powers of logarithms grow more slowly than power functions such as $n^{1.7}$.

7. Set up the inequality for big-O and divide both sides by $C$.

8. Take the limit of the ratio of the two functions, and use the fact that, for any two functions, $f^k(n)/g^k(n) = (f(n)/g(n))^k$.

9. Since
$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0,$$
we know that $f(n) \leq g(n)$ for sufficiently large $n$. Thus, $f(n) = \mathrm{O}(g(n))$.

   Now suppose it was true that $g(n) \leq Cf(n)$ for some constant $C > 0$, and $n$ sufficiently large. Then dividing both sides by $g(n)$ yields $\frac{f(n)}{g(n)} \geq 1/C$ for sufficiently large $n$. But since
$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0,$$
we know that $\frac{f(n)}{g(n)} < 1/C$, for sufficiently large $n$, which is a contradiction. Therefore, $g(n) \neq \mathrm{O}(f(n))$.

10. True, since $2^{n+1} = 2 \cdot 2^n$.

11. False. $2^{2n} = 4^n$ and
$$\lim_{n\to\infty} \frac{2^n}{4^n} = \lim_{n\to\infty} \frac{1}{2^n} = 0.$$
Now use Exercise 9.

12. $\Theta(n^{2.1} \log n)$.

13. False. Consider $f(n) = 2n$ and $g(n) = n$.

14. Use Theorem 1.7 and the fact that $g(n) < f(n)$ for $n$ sufficiently large.

15. Since the derivative (as a function of $n$) of $a^n$ equals $(\ln a)a^n$, it follows that the $k$ th derivative of $a^n$ divided by the $k$ th derivative of $n^k$ equals $\ln^k a^n/k!$, which tends to infinity. Therefore, $a^n = \omega(n^k)$.

16. By the Change of Base formula,
$$\log_a n = \frac{\log_b n}{\log_b a},$$
and so $\log_a n = C \log_b n$, where $C = 1/\log_b a$. Therefore, $\log_a n = \Theta(\log_b n)$.

17. $f(n) = \Omega(n^{2.2} \log^{35} n)$ and $f(n) = \mathrm{O}(2^{\log^{2.2} n})$.

18. $T(n) = \Omega(n)$ and $T(n) = \mathrm{O}(n^2)$

19. First use the Log-ratio test with $g(n) = \log^k n$, then use it again with $g(n) = n^k$.

20. Superpolynomial but not exponential growth. Use the Log-ratio test against an arbitrary polynomial function $n^k$. Then use it again against an arbitrary exponential function $a^n$, where $a > 1$. Or use the fact that $(\log n)^{\log n} < n^{\log n}$, and that $n^{\log n}$ was shown to *not* have exponential growth.

21. Sublinear. Use logarithm identities.

22. Constant growth. Take the logarithm of $f(n)$ and analyze its growth.

23. By the Integral Theorem,
$$\sum_{i=1}^{n} i^k = \Theta(\int_1^n x^k dx) = \Theta(\frac{x^{k+1}}{k+1}\Big|_1^n) = \Theta(n^{k+1}).$$

24. By the Integral Theorem,
$$\sum_{i=1}^{n} \ln i = \Theta(\int_1^n \ln x dx).$$
Moreover,
$$\int_1^n \ln x dx = x \ln x|_1^n - \int_1^n 1 dx =$$
$$n \ln n - n + 1 = \Theta(n \ln n).$$

25. Since $\log ab = \log a + \log b$, we have

$$\log(n!) = \log(n(n-1)(n-2)\cdots 1) = \log n + \log(n-1) + \log(n-2) + \cdots \log 1.$$

Therefore, from the previous exercise, we have $\log(n!) = \Theta(n \log n)$.

26. Use the Log-ratio test, and the fact that $\sum_{i=1}^{n} \log(i) = \Theta(n \log n)$.

27. Use the Log-ratio test to show that the first function is little-$\omega$ of the second.

28. Superpolynomial. First analyze the growth of $\log(f(n))$. Then consider the growth of $2^{\log(f(n))}$ which has the same growth as $f(n)$.

29. We have

$$n + n/2 + n/3 + \cdots + 1 = n(1 + 1/2 + 1/3 + \cdots + 1/n).$$

Therefore, by Example 3, $n + n/2 + n/3 + \cdots + 1 = \Theta(n \log n)$.

30. **Basis step** $1 = \frac{(1)(2)}{2}$.

**Inductive step** Assume $1 + 2 + \ldots n = \frac{n(n+1)}{2}$ for some $n \geq 1$. Show:

$$1 + 2 + \ldots n + (n+1) = \frac{(n+1)(n+2)}{2}.$$

$$1 + 2 + \ldots n + (n+1) = \frac{n(n+1)}{2} + (n+1) = (n+1)(\frac{n}{2} + 1) = \frac{(n+1)(n+2)}{2},$$

where the first equality is due to the inductive assumption.

31. **Basis step** $1 = \frac{(1)(2)(3)}{6}$.

**Inductive step** Assume $1^2 + 2^2 + \ldots n^2 = \frac{n(n+1)(2n+1)}{6}$ for some $n \geq 1$. Show:

$$1^2 + 2^2 + \ldots n^2 + (n+1)^2 = \frac{(n+1)(n+2)(2n+3)}{6}.$$

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

$$\frac{(n+1)}{6}(n(2n+1) + 6(n+1)) = \frac{(n+1)}{6}(2n^2 + 7n + 6) = \frac{(n+1)}{6}(n+2)(2n+3) =$$

$$\frac{(n+1)(n+2)(2n+3)}{6},$$

where the first equality is due to the inductive assumption.

32. **Basis step** $\log^1 n = o(n^\epsilon)$ by Example 4.

**Inductive step** Assume $\lim_{n\to\infty} \frac{\log^k(n)}{n^\epsilon} = 0$ for some $k \geq 1$. Show:

$$\lim_{n\to\infty} \frac{\log^{k+1}(n)}{n^\epsilon} = 0.$$

For simplicity, we assume that $\log(n)$ has base $e$, so that $(\log n)' = \frac{1}{n}$. This will not affect the big-O growth of the ratio, since, by the change of base formula,

$$\log_b a = \frac{\log_c a}{\log_c b},$$

and so logarithms with different bases only differ by a constant factor, which does not affect big-O growth. Then by L'Hospital's Rule,

$$\lim_{n\to\infty} \frac{\log^{k+1}(n)}{n^\epsilon} = \lim_{n\to\infty} \frac{(\log^{k+1}(n))'}{(n^\epsilon)'} =$$

$$\lim_{n\to\infty} \frac{\log^k(n)}{n\epsilon \cdot n^{\epsilon-1}} = \frac{1}{\epsilon} \lim_{n\to\infty} \frac{\log^k(n)}{n^\epsilon} = 0,$$

where the last equality is due to the inductive assumption.