CX2001 Algorithms	3: Analysis of Algorithms	**2020/21 Semester 1**
School of Computer Science and Engineering		*Nanyang Technological University*

3.1 Asymptotic Algorithm Analysis

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.

3.2 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**
- Repetition Structure: for-loop, while-loop
- Selection Structure: if/else statement, switch-case statement

3.2.1 Example 1: while loop

Algorithm 1 'while' Loop

$j \leftarrow 0$	▷ Constant time: c_0
while $j <= n$ **do**	▷ Run n iterations
$factorial \leftarrow factorial * j$	▷ c_1
$j \leftarrow j + 1$	▷ c_2
	▷ Time complexity = $c_1 n + c_2 n$

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.

3.2.2 Example 2: nested 'for' loop

Algorithm 2 Nested 'for' Loop

for $j \leftarrow 1, m$ **do**	▷ Run m iterations
for $k \leftarrow 1, n$ **do**	▷ Run n iterations
$sum \leftarrow sum + M[j][k]$	▷ c_1
	▷ c_2 additional cost of outer loop
	▷ Time complexity $= m(c_2 + c_1 n)$

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.

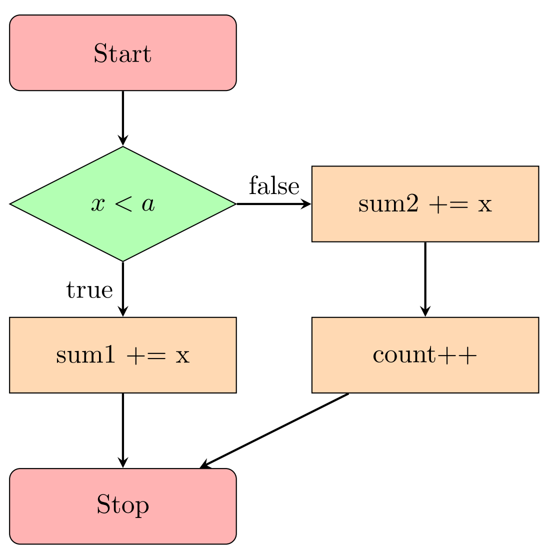
3.3 Different Cases of Complexity Analysis

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**

3.3.1 Example 3: 'if-else' selection structure

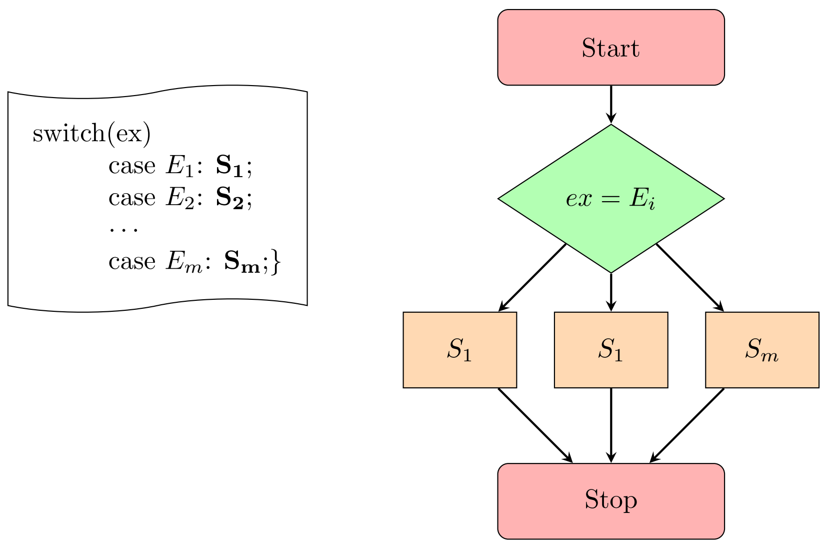
```
if(x < a)
        sum1 += x;
else{
        sum2+=x;
        count++;
        }
```



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: $\mathbf{p}(<) * c_1 + (1 - \mathbf{p}(<)) * c_2$

3.3.2 Example 4: 'switch-case' multiple selection structure



Let $\mathbf{C}$ 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 $C + 1.4n + 3.2\log_2 n$

3.3.3 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, $\frac{1}{n}$

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

3.4 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.

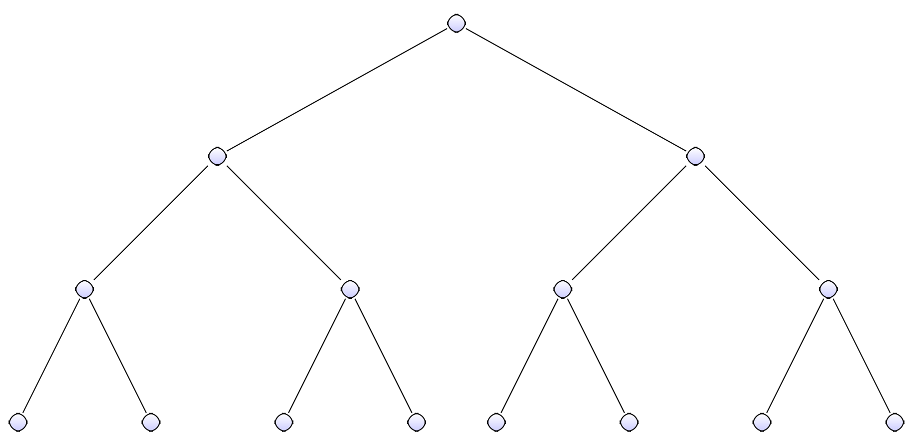
```
int count (int array[],int n, int a)
{
    if(n==1)
```

```
        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);
}
```

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.

3.5 Second Order Linear Recurrences with Constant Coefficients

Fibonacci sequence which has been briefly discussed in the first lecture is a typical linear recurrence example. Let Fibonacci numbers denote as F_n. Each number is the sum of the two preceding ones starting from 0 and 1. It is defined as

$$F_0 = 0, F_1 = 1$$

and

$$F_n = F_{n-1} + F_{n-2}$$

for $n > 1$.

The sequence is $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

The algorithm of finding the n^{th} Fibonacci number is

```
int fibonacci (int n){
    if( n < 1) return 0;
    if(n == 1 || n == 2) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

It is a second order recurrence because the new term is obtained by adding two previous terms.

3.5.1 Approximation Solution

Here, we would like to analyse its number of function calls. In this example, neither forward nor backward substitutions can solve the time complexity of the algorithm exactly. Here, we approximate that the number of the recursive calls is the same for fibonacci (n-1) and fibonacci (n-2).

$$\begin{aligned}
W(1) &= W(2) = 1 \\
W(n) &= W(n-1) + W(n-2) + 1 \\
&\approx 2 \cdot W(n-1) + 1 \\
&= 2 \cdot (2 \cdot W(n-2) + 1) + 1 \\
&= 2 \cdot 2 \cdot (2 \cdot W(n-3) + 1) + 2 + 1 \\
\ldots &= \ldots \\
&= 2^{n-3}(2 \cdot W(n-(n-2)) + 1) + 2^{n-4} + 2^{n-5} + \ldots + 2^2 + 2 + 1 \\
&= 2^{n-1} - 1
\end{aligned}$$

It is noted that the assumption of the number of the recursive calls is not very correct especially when n is large.

Can we obtain the exact time complexity of the algorithm?

3.5.2 The General Solution To Second-Order Linear Recurrences with Constant Coefficients

If we rearrange the recurrence above as follow

$$aW(n) + bW(n-1) + cW(n-2) = f(n)$$

where a, b and c are real numbers, $a \neq 0$. The recurrence is called second-order linear recurrence. Since $W(n)$ and $W(n-2)$ are two positions apart in the sequence, this recurrence is **second-order**. If the right-hand side of the equation is zero ($f(n) = 0$), it is a **homogeneous** recurrence. Otherwise, it is called **inhomogeneous** or **nonhomogeneous**. Let us consider the homogeneous cases where $f(n) = 0$ first. Its characteristic equation is

$$ar^2 + br + c = 0$$

The roots of the characteristic equation are

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

There are three cases and their corresponding homogeneous solutions are:

1. Two real and distinct roots: The homogeneous solution to recurrence is obtained by the formula

$$W^h(n) = \alpha r_+^n + \beta r_-^n$$

   where α and β are two arbitrary real constants.

2. A real double root: The homogeneous solution to recurrence is obtained by the formula

$$W^h(n) = \alpha r^n + \beta n r^n$$

   where α and β are two arbitrary real constants.

3. Two distinct complex roots: $r_+ = u + iv$ and $r_- = u - iv$ are two complex conjugate numbers, the homogeneous solution to recurrence is obtained by the formula

$$W^h(n) = \gamma^n[\alpha \cos n\theta + i\beta \sin n\theta]$$

where $\gamma = \sqrt{(u^2 + v^2)}$, $\theta = \arctan(\frac{v}{u})$ and α, and β are two arbitrary real constants.

For second-order inhomogeneous linear recurrences where $f(n) \neq 0$, we only consider $f(n) = kr_p^n$ cases. Then the general solution to the recurrence is

$$W(n) = W^h(n) + W^p(n)$$

where $W^h(n)$ is the homogeneous solution and $W^p(n)$ is a particular solution. The particular solution $W^p(n)$ can be obtained as follow:

Let $f(n) = kr_p^n$ and r_+ and r_- be the roots of the characteristic equation

1. If $r_p \neq r_+$, $r_p \neq r_-$, then $W^p(n) = Ar_p^n$;
2. If either $r_p = r_+$ or $r_p = r_-$, $r_+ \neq r_-$, then $W^p(n) = Anr_p^n$;
3. If $r_p = r_+ = r_-$ (A real double root case), then $W^p(n) = An^2r_p^n$;

where A is a constant to be determined.

3.5.3 Time Complexity of The Recursive Fibonacci Algorithm

To rearrange the Fibonacci recurrence equation, we obtain

$$W(n) - W(n-1) - W(n-2) = 1$$

Let us consider the homogeneous recurrence first.

$$W(n) - W(n-1) - W(n-2) = 0$$

Its corresponding characteristic equation is

$$r^2 - r - 1 = 0$$

The roots of the characteristic equation are

$$r = \frac{1 \pm \sqrt{5}}{2}$$

Since there are two real and distinct roots, its homogeneous solution is

$$W^h(n) = \alpha(\frac{1+\sqrt{5}}{2})^n + \beta(\frac{1-\sqrt{5}}{2})^n$$

Since $f(n) = 1$ in this case, the $k = 1$ and $r_p = 1$. Thus $r_p \neq r_+$ and $r_p \neq r_-$. The particular solution is

$$W^p(n) = A$$

where A is a constant. We can substitute $W^p(n)$ to the recurrence equation.

$$\begin{aligned} W^p(n) - W^p(n-1) - W^p(n-2) &= 1 \\ A - A - A &= 1 \\ A &= -1 \end{aligned}$$

Next α and β can be obtained by substituting the initial conditions, W(1) = W(2) = 1. Since

$$W^h(n) = \alpha(\frac{1+\sqrt{5}}{2})^n + \beta(\frac{1-\sqrt{5}}{2})^n$$

and

$$W^p(n) = -1$$

then

$$\begin{aligned} W(n) &= W^h(n) + W^p(n) \\ W(n) &= \frac{2}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n) - 1 \end{aligned}$$

where $\alpha = \frac{2}{\sqrt{5}}$ and $\beta = -\frac{2}{\sqrt{5}}$

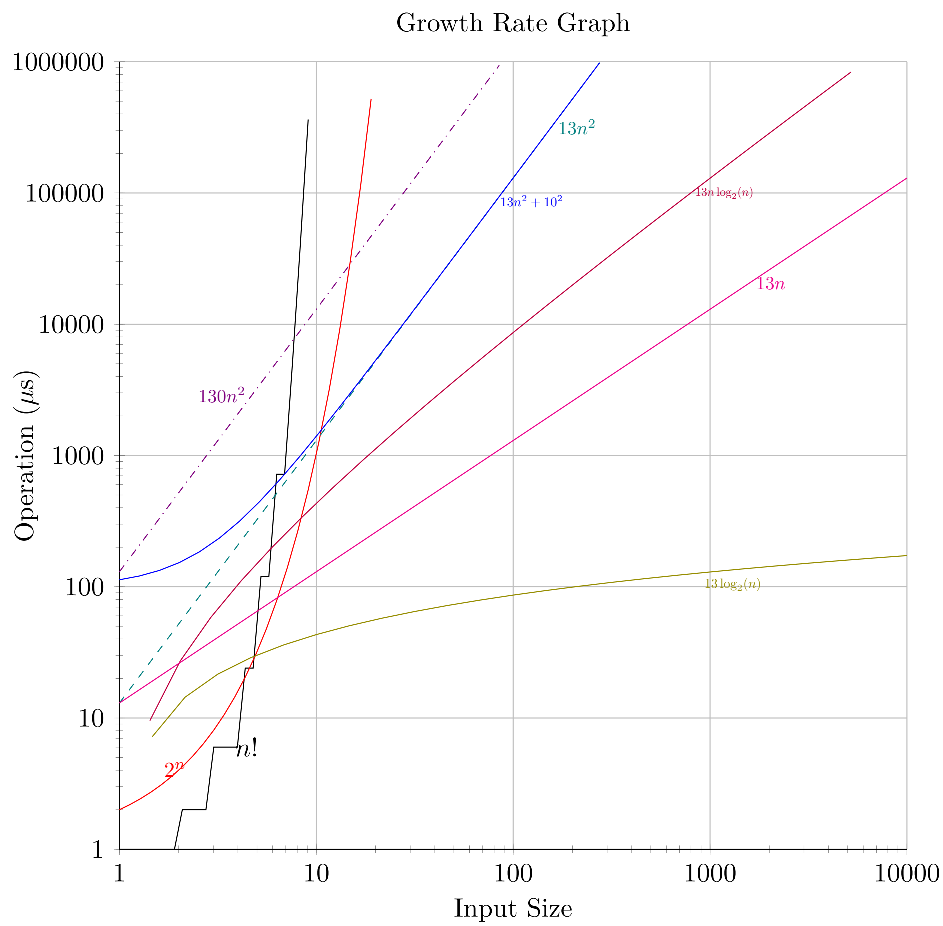
As problem size n increases, $W(n) \approx 1.618^n$.

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.

3.6 Order of Growth

We are interest 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	4×10^{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.

3.6.1 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 $10\times$ 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.

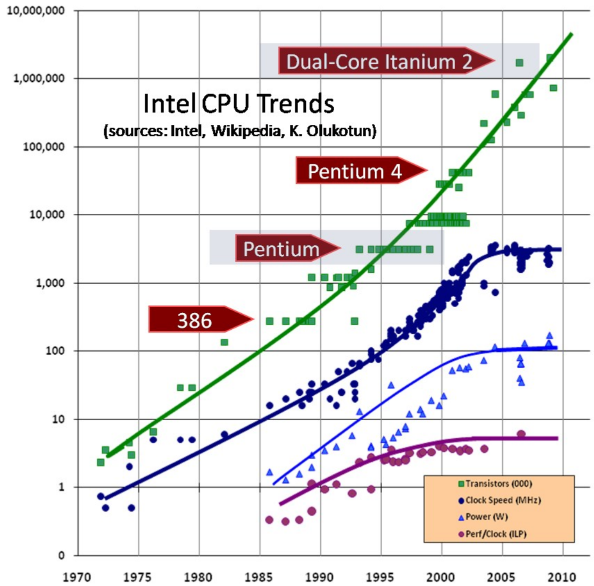


Figure 3.1: Moore's Law: https://cs.stackexchange.com/questions/27875/moores-law-and-clock-speed

3.7 Asymptotic Notation

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

3.7.1 Big-Oh Notation: $\mathcal{O}$

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*

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

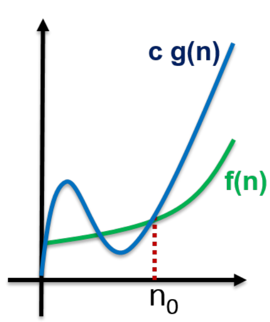


Figure 3.2: Big-Oh Notation

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

$$f(n) \le cg(n) \quad \forall n \ge n_0$$
$$f(n) \le 5g(n) \quad \forall n \ge 3$$

$\therefore f(n) = \mathcal{O}(g(n))$ 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

$$f(n) \le g(n) \forall n \ge 3$$

$\therefore f(n) = \mathcal{O}(g(n))$ 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 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 $f(n) = \mathcal{O}(g(n))$.*

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

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

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

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

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

$\therefore f(n) = \mathcal{O}(g(n))$ 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 $g(n) = e^n$,

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 $4n+3 \in \mathcal{O}(e^n)$

3.7.2 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*

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

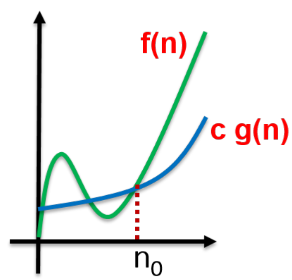


Figure 3.3: Big-Omega Notation

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

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 $f(n) = \Omega(g(n))$.*

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

$$\begin{gathered} f(n) \ge \frac{1}{5} g(n) \\ 4n+3 \ge n \quad \forall n \ge 0 \end{gathered}$$

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

3.7.3 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*

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

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

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

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 $4n + 3 \in \Theta(n)$

3.7.4 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	✓		
$0 < c < \infty$	✓	✓	✓
∞		✓	

The $\mathcal{O}$, Ω and Θ 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.

3.7.5 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 3.7.7.

3.7.6 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 simplication rules for asymptotic analysis:

1. If $f(n) = \mathcal{O}(cg(n))$ for any constant $c > 0$,
   then $f(n) = \mathcal{O}(g(n))$
2. If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$,
   then $f(n) = \mathcal{O}(h(n))$.
   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 $f_2(n) = \}_{\in}(\backslash)$,
   then $f_1(n) + f_2(n) = \mathcal{O}(\max(g_1(n), g_2(n)))$
   e.g. $5n + 3\lg n = \mathcal{O}(n)$
4. If $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \}_{\in}(\backslash)$,
   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 $3n^2 \lg n = \mathcal{O}(n^2 \lg n)$

   Some properties of $\mathcal{O}$, Ω and Θ:

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

3.7.7 Common Complexity Classes

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:

```
    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^ε 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
$\therefore f(n) \in \mathcal{O}(2^n)$

3.8 Space Complexity

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.