MODULE 4

ALGORITHM EFFICIENCY

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

MODULE 4: ALGORITHM EFFICIENCY

OBJECTIVES FOR STUDENTS

1. To analyze the number of steps of algorithms relative to the increasing of input size, *n*.

2. Find the class of complexity in big 'O' notation.

KEY CONCEPT

1.0 INTRODUCTION TO ALGORITHM

1.1 **Algorithm analysis** - Study the efficiency of algorithms when the input size grows based on the number of steps, the amount of computer time and space.

1.2 **Analysis of algorithms** is a major field that provides **tools** for evaluating the efficiency of different solutions

1.3 What is an **efficient algorithm**?
- Faster is better (Time) - How do you measure time? Wall clock? Computer clock?
- Less space demanding is better - But if you need to get data out of main memory it takes time.

1.4 **Algorithm analysis** should be independent of :
- Specific implementations and coding tricks (programming language, control statements – Pascal, C, C++, Java)
- Specific Computers (hw chip, OS, clock speed)
- Particular set of data (string, int, float)

1.5 For a particular problem size, we may be interested in:
- **Worst-case efficiency**: Longest running time for *any* input of size ***n***
  A determination of the maximum amount of time that an algorithm requires to solve problems of size ***n***.
- **Best-case efficiency**: Shortest running time for *any* input of size ***n***
  A determination of the minimum amount of time that an algorithm requires to solve problems of size ***n***.
- **Average-case efficiency**: Average running time for *all* inputs of size ***n***
  A determination of the average amount of time that an algorithm requires to solve problems of size ***n***.

1.6 The worst case is always considered as the maximum boundary for execution time or memory space for any input size. Execution time for the worst case is the complexity time.

1.7 Example of algorithm: sequential search of ***n*** elements

- **Best-case**: We get lucky and find the target in the first place we look. O**(n)** = 1
- **Worst-case**: We look at every element before finding (or not finding) the target. O**(n)** = **n**
- **Average-case**: Depends on the probability (**p**) that the target will be found. O**(n)** = **n/2**

2.0 COMPLEXITY OF ALGORITHM

2.1 Complexity time can be represented by **Big 'O' Notation** (notation that used to show the complexity time of algorithms). Big 'O' notation is denoted as : **O**(***acc***) whereby: **O** – order and ***acc*** – class of algorithm complexity.

2.2 **Big O Notation**

Notation	**Execution time / number of step**
O(1)	Constant function, independent of input size, ***n***. Example: Finding the first element of a list.
O($\log_x n$*)*	Problem complexity increases slowly as the problem size increases. Squaring the problem size only doubles the time. Characteristic: Solve a problem by splitting into constant fractions of the problem (e.g., throw away ½ at each step)
O(n)	Problem complexity increases linearly with the size of the input, ***n***. Example: counting the elements in a list.
O(n $\log_x n$*)*	Log-linear increase - Problem complexity increases a little faster than ***n***. Characteristic: Divide problem into sub problems that are solved the same way. Example: mergesort
O(n^2*)*	Quadratic increase. Problem complexity increases fairly fast, but still manageable. Characteristic: Two nested loops of size ***n***.
O(n^3*)*	Cubic increase. Practical for small input size, ***n***.
O(2^n*)*	Exponential increase - Increase too rapidly to be practical. Problem complexity increases very fast. Generally unmanageable for any meaningful ***n***. Example: Find

	all subsets of a set of ***n*** elements.

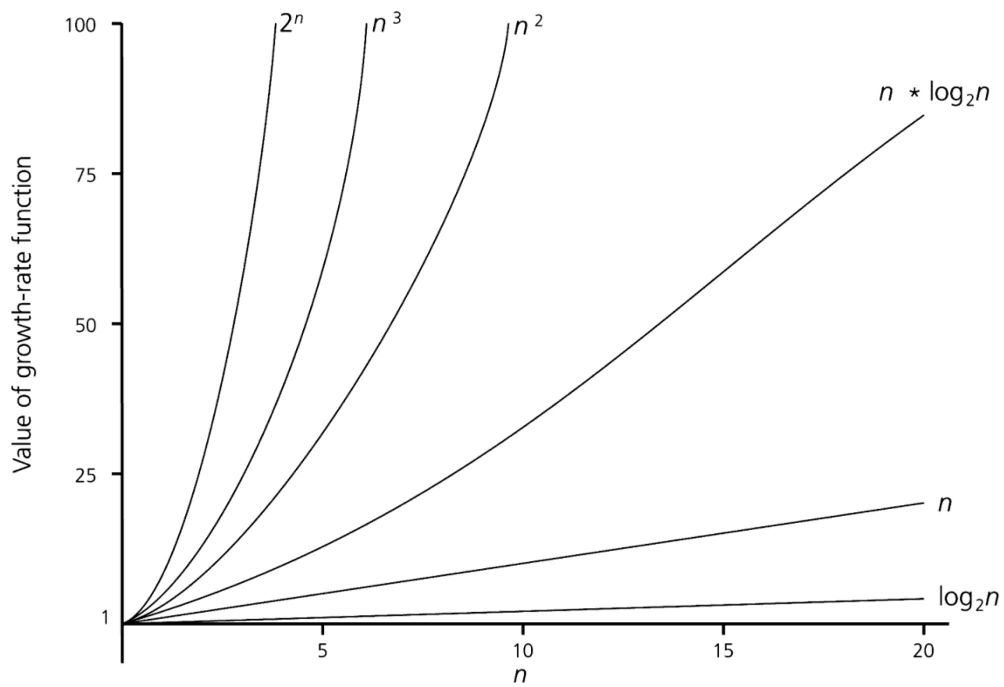
2.3 Order-of-Magnitude Analysis and Big O Notation. Figure 4.1 shows comparison of growth-rate functions in tabular and graphical form/

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A comparison of growth-rate functions: (a) in tabular form

(b)



A comparison of growth-rate functions: (b) in graphical form

Figure 4.1 A comparison of growth-rate functions

- Order of increasing complexity - $\boldsymbol{O(1) < O(\log_x n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)}$

Notation	***n = 8***	***n = 16***	***n = 32***
$O(\log_2 n)$	*3*	*4*	*5*
$O(n)$	*8*	*16*	*32*
$O(n \log_2 n)$	*24*	*64*	*160*
$O(n^2)$	*64*	*256*	*1024*
$O(n^3)$	*512*	*4096*	*32768*
$O(2^n)$	*256*	*65536*	*4294967296*

- Example of algorithm (only for **cout** operation):

Notation	**Code**
$O(1)$	**int counter = 1; cout << "Arahan cout kali ke " << counter << "\n";**
$O(\log_x n)$	**int counter = 1; int i = 0; for (i = x; i <= n; i = i * x) // *x must be > than 1* { cout << "Arahan cout kali ke " << counter << "\n"; counter++; }**
$O(n)$	**int counter = 1; int i = 0; for (i = 1; i <= n; i++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; }**
$O(n \log_x n)$	**int counter = 1; int i = 0; int j = 1; for (i = x; i <= n; i = i * x) // *x must be > than 1* { while (j <= n) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; j++; } }**
$O(n^2)$	**int counter = 1; int i = 0; int j = 0; for (i = 1; i <= n; i++) {**

	`` for (j = 1; j <= n; j++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; } } ```
$O(n^3)$	``` int counter = 1; int i = 0; int j = 0; int k = 0; for (i = 1; i <= n; i++) { for (j = 1; j <= n; j++) { for (j = 1; j <= n; j++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; } } } ```
$O(2^n)$	``` int counter = 1; int i = 1; int j = 1; while (i <= n) { j = j * 2; i++; } for (i = 1; i <= j; i++) { cout << "Arahan cout kali ke " << counter << "\n"; counter++; } ```

2.4 The complexity time of algorithm can be determined theoretically – by calculation or practically – by experiment or implementation.

2.5 Determine the complexity time of algorithm - practically
- Implement the algorithms in any programming language and run the programs
- Depend on the compiler, computer, data input and programming style.

2.6 Determine the complexity time of algorithm - theoretically
- The complexity time is related to the number of steps / operations.
- Complexity time can be determined by
  - Count the number of steps and then find the class of complexity, or
  - Find the complexity time for each step and then count the total.

2.7 The following algorithm is categorized as ***O(n)***.

```
int counter = 1;
int i = 0;
for (i = 1; i <= n; i++)
{
   cout << "Arahan cout kali ke " << counter << "\n";
   counter++;
}
```

Num	statements
1	int counter = 1;
2	int i = 0;
3	i = 1
4	i <= n
5	i++
6	cout << "Arahan cout kali ke " << counter << "\n"
7	counter++

- Statement 3, 4 and 5 are the loop's control and can be assumed as one statement.

Num	Statements
1	int counter = 1;
2	int i = 0;
3	i = 1; i <= n; i++
6	cout << "Arahan cout kali ke " << counter << "\n"
7	counter++

- Statement 3, 6 and 7 are in the repetition structure.
- It can be expressed by summation series.

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n) = n$$

where ***f(i)*** – statement executed in the loop

- Example, if ***n*** = 5, ***i*** = 1.

$$\sum_{i=1}^{5} f(i) = f(1) + f(2) + f(3) + f(4) + f(5) = 5$$

The statement that represented by **f(i)** will be repeated 5 times.

- Example, if ***n*** = 5, ***i*** = 3

$$\sum_{i=3}^{5} f(i) = f(3) + f(4) + f(5) = 3$$

The statement that represented by **f(i)** will be repeated 3 times.

- Example: if ***n*** = 1, ***i*** = 1

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

The statement that represented by **f(i)** will be executed only once.

Statements	**Number of steps**
int counter = 1;	1
int i = 0;	1
i = 1; i = n; i++	n
cout << "Arahan cout kali ke " << counter << "\n"	n
counter++	n

Total steps:

1 + 1 + n + n + n

= 2 + 3*n*

2.8 Besides by summation series, the steps can be calculated using the following formula:
Number of steps = *b* - *a* + *1* where,
- *b* is the final conditions to control the loop,
- *a* is the initial conditions of the control loop,
- *1* is the constant at beginning of the loop.

2.9 Consider the largest factor.

- Algorithm complexity can be categorized as ***O*(*n*)**

Algorithm	**Number of Steps**
void sample4 ( ) **{** **for (int a=2; a<=n; a++)** **cout << "Contoh kira langkah ";** **}**	**0** **0** **n-2+1 = n-1** **(n-1).1 = n-1** **0**
Total steps	**2(n-1)**

Total steps = **2(*n*-1)**, Complexity Time = ***O*(*n*)**

Algorithm	**Number of steps**
void sample5 ( ) **{** **for (int a=1; a<=n-1; a++)** **cout << " Contoh kira langkah ";** **}**	**0** **0** **n-1-1+1 = n-1** **(n-1).1 = n-1** **0**
Total steps	**2(n-1)**

Total steps = **2(*n*-1)**, Complexity Time = ***O*(*n*)**

Algorithm	**Number of Steps**
void sample6 ( ) **{** **for (int a=1; a<=n; a++)** **for (int b=1; b<=n; b++)** **cout << " Contoh kira langkah ";** **}**	**0** **0** **n-1+1 = n** **n.(n-1+1) = n.n** **n.n.1 = n.n** **0**
Total steps	**$n+2n^2$**

Total Steps = $n+2n^2$, Complexity Time = $O(n^2)$

Algorithm	Number of Steps
void sample7 ( ) **{** **for (int a=1; a<=n; a++)** **for (int b=1; b<=a; b++)** **cout << " Contoh kira langkah ";** **}**	**0** **0** **n-1+1=n** **n.(n+1)/2** **n.(n+1)/2** **0**
Total steps	**2n+n²**

Total steps = $\boldsymbol{2n+n^2}$, Complexity Time = $\boldsymbol{O(n^2)}$

To get **n.(n+1)/2**, we used summation series as shown below:

$$\sum_{a=1}^{n}\sum_{b=1}^{a} = n(1+2+3+4+\ldots+n)$$

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

$$= \frac{n^2+n}{2}$$

2.10 Count the number of steps and find the Big 'O' notation for the following algorithm

```
int counter = 1;
int i = 0;
int j = 1;

for (i = 3; i <= n; i = i * 3) {
   while (j <= n) {
     cout << "Arahan cout kali ke " << counter << "\n";
     counter++;
     j++;
     }
}
```

Statements	Number of steps
int counter = 1;	$\sum_{i=1}^{1} f(i) = 1$

int i = 0;	$\sum_{i=1}^{1} f(i) = 1$
int j = 1;	$\sum_{i=1}^{1} f(i) = 1$
i = 3; i <= n; i = i * 3	$\sum_{i=3}^{1} f(i) = f(3) + f(9) + f(27) + ... + f(n) = \log_3 n$
j <= n	$\sum_{i=3}^{n} f(i) \sum_{j=1}^{n} f(j) = \log_3 n.n$
cout << "Arahan cout kali ke " **<< counter** **<< "\n";**	$\sum_{i=3}^{n} f(i).\sum_{j=1}^{n} f(j).\sum_{i}^{1} f(i) = \log_3 n.n.1$
counter++;	$\sum_{i=3}^{n} f(i).\sum_{j=1}^{n} f(j).\sum_{i}^{1} f(i) = \log_3 n.n.1$
j++;	$\sum_{i=3}^{n} f(i).\sum_{j=1}^{n} f(j).\sum_{i}^{1} f(i) = \log_3 n.n.1$

Total steps:

=> $1 + 1 + 1 + \log_3 n + \log_3 n . n + \log_3 n . n . 1 + \log_3 n . n . 1 + \log_3 n . n . 1$
=> $3 + \log_3 n + \log_3 n . n + \log_3 n . n + \log_3 n . n + \log_3 n . n$
=> $3 + \log_3 n + 4n \log_3 n$

Consider the largest factor for : $3 + \log_3 n + 4n \log_3 n$

($4n \log_3 n$)

- and remove the coefficient

  ($n \log_3 n$)

- In asymptotic classification, the base of the log can be omitted as shown in this formula:

  $\log_a n = \log_b n / \log_b a$

- Thus, $\log_3 n = \log_2 n / \log_2 3 = \log_2 n / 1.58...$
- Remove the coefficient 1/1.58..
- So we get the complexity time of the algorithm is

  $O(n \log_2 n)$

2.11 Summary on algorithm efficiency

- Algorithm analysis to study the efficiency of algorithms when the input size grow, based on the number of steps, the amount of computer time and space

- Can be done using Big O notation by using growth of function.

- Order of growth for some common function:
  O(1) < O(logxn) < O(n) < O(n log2n) < O(n2) < O(n3) < O(2n)

- Three possible states in algorithm analysis best case, average case and worst case.