Objectives

At the end of the class, students are expected to be able to do the following:

• Know how to measure algorithm efficiency.
• Know the meaning of big O notation.
Introduction

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**.
Analysis of algorithms

- Is a major field that provides tools for evaluating the efficiency of different solutions

**What is an efficient algorithm?**

- Faster is better (Time)
  - How do you measure time? Wall clock? Computer clock?
- Less space demanding is better (Space)
  - But if you need to get data out of main memory it takes time
Analysis of algorithms

• 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)

But size of data should matter
Analysis of algorithms

• Three possible states in algorithm analysis:
  - best case
  - average case
  - worst case

• The worst case is always considered $\rightarrow$ the maximum boundary for execution time or memory space for any input size.

• Execution time for the worst case $\rightarrow$ complexity time
Worse Case/Best Case/Average Case

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 $n$
Examples of the 3 cases

Algorithm: sequential search of $n$ elements

- **Best-case:** Find the target in the first place the element set. $C(n) = 1$
- **Worst-case:** Find or cannot find the target after compare every element with the target value. $C(n) = n$
- **Average-case:** Depends on the probability ($p$) that the target will be found. $C(n) = n/2$
Big O Notation

- Complexity time can be represented by **Big ‘O’ notation**.
- Big ‘O’ notation is denoted as \( O(f(n)) \)
  - \( O \) – “on the order of”
  - \( f(n) \)- algorithm’s growth-rate function that may consist of \( 1, \log n, n, n \log n, n^2, \ldots \)
- An algorithm requires time proportional to \( f(n) \). \( O(f(n)) \) means order of \( f(n) \).
Big $O$ Notation

- Notation that used to show the complexity time of algorithms.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Execution time / number of step</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant function, independent of input size, $n$</td>
</tr>
<tr>
<td></td>
<td>Example: Finding the first element of a list.</td>
</tr>
<tr>
<td>$O(\log_x n)$</td>
<td>Problem complexity increases slowly as the problem size increases.</td>
</tr>
<tr>
<td></td>
<td>Squaring the problem size only doubles the time.</td>
</tr>
<tr>
<td></td>
<td>Charac.: Solve a problem by splitting into constant fractions of the problem (e.g., throw away ( \frac{1}{2} ) at each step)</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Problem complexity increases linearly with the size of the input, $n$</td>
</tr>
<tr>
<td></td>
<td>Example: counting the elements in a list.</td>
</tr>
</tbody>
</table>
# Big O Notation

| $O(n \log_x n)$ | Log-linear increase - Problem complexity increases a little faster than $n$
| | Characteristic: Divide problem into subproblems 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 |
**Big O Notation**

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

<table>
<thead>
<tr>
<th>notation</th>
<th>code</th>
</tr>
</thead>
</table>
| \(O(1)\) Constant | int counter = 1;  
  `cout << "Arahan cout kali ke " << counter << "\n";`             |
| \(O(\log_x n)\) Logarithmic | 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++;  
  } |
Order of increasing complexity

Order of growth for some common function:
• $O(1) < O(\log_x n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

<table>
<thead>
<tr>
<th>Notasi</th>
<th>$n = 8$</th>
<th>$n = 16$</th>
<th>$n = 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(\log_2 n)$</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>$O(n \log_2 n)$</td>
<td>24</td>
<td>64</td>
<td>160</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>64</td>
<td>256</td>
<td>1024</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>512</td>
<td>4096</td>
<td>32768</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>256</td>
<td>65536</td>
<td>4294967296</td>
</tr>
</tbody>
</table>
Order-of-Magnitude Analysis and Big O Notation

The diagram illustrates the growth-rate function values for several functions, including $2^n$, $n^3$, $n^2$, $n \cdot \log_2 n$, $n$, and $\log_2 n$.
### Big O Notation

- Example of algorithm for common function:

| $O(n)$ Linear | int counter = 1; int i = 0;  
for (i = 1; i <= n; i++) {  
    cout << "Arahan cout kali ke " << counter << "\n";  
    counter++;  
} |
|--------------|-----------------------------------------------------------------|
| $O(n \log_x n)$ Linear Logarithmic | 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++;  
    }  
} |
Big $O$ Notation

- Example of algorithm for common function:

| $O(n^2)$ Quadratic | 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++;  
|                   |     }  
|                   | } |
Big $O$ Notation

- Example of algorithm for common function:

```cpp
O(n^3) Cubic

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++;
        }
    }
}
```
Big $O$ Notation

- Example of algorithm for common function:

| $O(2^n)$ Exponential | 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++;  
| | }  

Determine the complexity time of algorithm

Can be determined

• theoretically – by calculation
• practically – by experiment or implementation
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.
Determine the complexity time of algorithm - theoretically

- The **complexity time** is related to the number of steps /operations.
- Complexity time can be determined by
  1. Count the **number of steps** and then find the class of complexity.
  2. Find the **complexity time** for each steps and then count the total.
Determine the number of steps

• The following algorithm is categorized as O(n).

```c++
int counter = 1;
int i = 0;
for (i = 1; i <= n; i++) {
    cout << "Arahan cout kali ke ";
    cout << counter << "\n";
    counter++;
}
```
Determine the number of steps

<table>
<thead>
<tr>
<th>Num</th>
<th>statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int counter = 1;</td>
</tr>
<tr>
<td>2</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>3</td>
<td>i = 1</td>
</tr>
<tr>
<td>4</td>
<td>i &lt;= n</td>
</tr>
<tr>
<td>5</td>
<td>i++</td>
</tr>
<tr>
<td>6</td>
<td>cout &lt;&lt; &quot;Arahan cout kali ke &quot; &lt;&lt; counter &lt;&lt; &quot;\n&quot;</td>
</tr>
<tr>
<td>7</td>
<td>counter++</td>
</tr>
</tbody>
</table>
Determine the number of steps

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

<table>
<thead>
<tr>
<th>Num</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int counter = 1;</td>
</tr>
<tr>
<td>2</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>3</td>
<td>i = 1; i &lt;= n; i++;</td>
</tr>
<tr>
<td>6</td>
<td>cout &lt;&lt; &quot;Arahan cout kali ke &quot; &lt;&lt; counter &lt;&lt; &quot;\n&quot;</td>
</tr>
<tr>
<td>7</td>
<td>counter++;</td>
</tr>
</tbody>
</table>
Determine the number of steps-
summation series

• statement 3, 6 & 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
Determine the number of steps-
summation series

• 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
Determine the number of steps-
summation series

• 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 time
Determine the number of steps-
summation series

• 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.
## Determine the number of steps

<table>
<thead>
<tr>
<th>statements</th>
<th>Number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>int counter = 1;</td>
<td>$\sum_{i=1}^{1} f(i) = 1$</td>
</tr>
<tr>
<td>int i = 0;</td>
<td>$\sum_{i=1}^{1} f(i) = 1$</td>
</tr>
<tr>
<td>i = 1; i = n; i++</td>
<td>$\sum_{i=1}^{n} f(i) = n$</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;Arahan cout kali ke &quot; &lt;&lt; counter &lt;&lt; &quot;\n&quot;</td>
<td>$\sum_{i=1}^{n} f(i) . \sum_{i=1}^{1} f(i) = n . 1 = n$</td>
</tr>
<tr>
<td>counter++</td>
<td>$\sum_{i=1}^{n} f(i) . \sum_{i=1}^{1} f(i) = n . 1 = n$</td>
</tr>
</tbody>
</table>
Determine the number of steps

• Total steps:

$$1 + 1 + n + n + n = 2 + 3n$$

Consider the largest factor.

• Algorithm complexity can be categorized as $O(n)$
Determine the number of steps

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>void sample4 ( )</td>
<td>0</td>
</tr>
<tr>
<td>{</td>
<td>0</td>
</tr>
<tr>
<td>for (int a=2; a&lt;=n; a++)</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; “Contoh kira langkah“;</td>
<td>n-2+1=n-1</td>
</tr>
<tr>
<td>}</td>
<td>(n-1).1=n-1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

| Total Steps      | 2(n-1)          |

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>void sample5 ( )</td>
<td>0</td>
</tr>
<tr>
<td>{</td>
<td>0</td>
</tr>
<tr>
<td>for (int a=1; a&lt;=n-1; a++)</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; “Contoh kira langkah“;</td>
<td>n-1-1+1=n-1</td>
</tr>
<tr>
<td>}</td>
<td>(n-1).1=n-1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

| Total steps      | 2(n-1)          |
### Determine the number of steps

**Continued......**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>void sample7 ( )</td>
<td>0</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>for (int a=1; a&lt;=n; a++)</td>
<td></td>
</tr>
<tr>
<td>for (int b=1; b&lt;=a; b++)</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; “Contoh kira langkah“;</td>
<td>n-1+1=n</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Total steps</td>
<td>2n+n^2</td>
</tr>
</tbody>
</table>

- Total steps = 2n+n^2
- Complexity Time = O (n^2)
Determine the number of steps

Continued...

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

To get \( n \frac{(n+1)}{2} \), we used summation series as shown above:
Determine the number of steps - exercise

Count the number of steps and find the Big ‘O’ notation for the following algorithm

```c++
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++;
    }
}
```
## Determine the number of steps - solution

<table>
<thead>
<tr>
<th>statements</th>
<th>Number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>int counter = 1;</td>
<td>$\sum_{i=1}^{1} f(i) = 1$</td>
</tr>
<tr>
<td>int i = 0;</td>
<td>$\sum_{i=1}^{1} f(i) = 1$</td>
</tr>
<tr>
<td>int j = 1;</td>
<td>$\sum_{i=1}^{1} f(i) = 1$</td>
</tr>
<tr>
<td>i = 3; i &lt;= n; i = i * 3</td>
<td>$\sum_{i=3}^{n} f(i) = f(3) + f(9) + f(27) + \ldots + f(n) = \log_3 n$</td>
</tr>
<tr>
<td>j &lt;= n</td>
<td>$\sum_{i=3}^{n} f(i) \cdot \sum_{j=1}^{n} f(i) = \log_3 n \cdot n$</td>
</tr>
</tbody>
</table>
Determine the number of steps - solution

<table>
<thead>
<tr>
<th>cout &lt;&lt; &quot;Arahan cout kali ke &quot; &lt;&lt; counter &lt;&lt; &quot;\n&quot;;</th>
<th>[ \sum_{i=3}^{n} f(i) \cdot \sum_{j=1}^{n} f(i) \cdot \sum_{i=1}^{1} f(i) = \log_3 n \cdot n \cdot 1 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter++;</td>
<td>[ \sum_{i=3}^{n} f(i) \cdot \sum_{j=1}^{n} f(i) \cdot \sum_{i=1}^{1} f(i) = \log_3 n \cdot n \cdot 1 ]</td>
</tr>
<tr>
<td>j++;</td>
<td>[ \sum_{i=3}^{n} f(i) \cdot \sum_{j=1}^{n} f(i) \cdot \sum_{i=1}^{1} f(i) = \log_3 n \cdot n \cdot 1 ]</td>
</tr>
</tbody>
</table>
Determine the number of steps - solution

Total steps:

\[ 1 + 1 + \log_3 n + \log_3 n \cdot n + \log_3 n \cdot 1 + \log_3 n \cdot n \cdot 1 \]
\[ \Rightarrow 3 + \log_3 n + \log_3 n \cdot n + \log_3 n \cdot n + \log_3 n \cdot n \]
\[ \Rightarrow 3 + \log_3 n + 4n \log_3 n \]
Determine the number of steps:

**solution**

\[3 + \log_3 n + 4n \log_3 n\]

- Consider the largest factor
  \[(4n \log_3 n)\]
- 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, \[\log_3 n = \log_2 n / \log_2 3 = \log_2 n / 1.58\ldots\]
- Remove the coefficient 1/1.58..
- So we get the complexity time of the algorithm is \[O(n \log_2 n)\]
Determine the number of steps

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>No. of Steps</th>
</tr>
</thead>
</table>
| void sample8 ( )  
{  
  int n, x, i=1;  
  while (i<=n)  
  {  
    x++;  
    i++;  
  }  
}  
Number of Steps | 1 + 3n |

Consider the largest factor : 3n
and remove the coefficient : O(n)
Conclusion and Summary

- 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:
  \[ O(1) < O(\log \_x n) < O(n) < O(n \log \_2 n) < O(n^2) < O(n^3) < O(2^n) \]
- Three possible states in algorithm analysis **best case**, **average case** and **worst case**.
References
