MODULE TITLE

BASIC CONCEPTS AND NOTATION

MODULE OVERVIEW

This module aims to enable you to understand the concepts and properties of algorithm, the problem solving process, different mathematical functions that can be used to analyze an algorithm and measuring algorithm complexity using Big – Oh Notation. Problem solving and programming exercises were included in every section in order to test your understanding of the lesson.

LEARNING OBJECTIVES

By the end of this module you should be able to:

- Define Algorithm
- Explain the process of problem solving
- Identify properties of algorithms
- Use basic mathematical functions to analyse algorithms
- Measure complexity of algorithms (Big – Oh Notation)

LEARNING CONTENTS

Unit 1: Algorithm and Properties of Algorithm

Introduction

The term algorithm is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program. It is a strictly defined finite sequence of well – defined statements that provides the solution to a problem or to a specific class of problems for any acceptable set of input values (if there are any inputs). The term "finite" means that the algorithm should reach an end point and cannot run forever.

Unit learning outcomes

By the end of this unit you should be able to:

- Define what algorithm is
- Differentiate the different properties of algorithms
- Apply these properties to design an algorithm for a specific problem

What is Algorithm?

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

The following demonstrates how algorithm is being applied in real world:

The Algorithm for Making a Cup of Tea

1. Put the teabag in a cup
2. Fill the kettle with water
3. Boil the water in the kettle
4. Pour some of the boiled water into the cup
5. Add milk to the cup
6. Add sugar to the cup
7. Stir the tea
8. Drink the tea

As you can see, there are certain steps that must be followed. These steps are in specific order, even though some of the steps could be rearranged. For example, steps 5 and 6 can be reversed. You will also notice that aside from the sequence of steps that must be performed, the entire process is finite and reach its end point which enable you to achieve your goal (in this case, making a cup of tea).

Fundamental Properties of Algorithms

- **Finiteness** - an algorithm must terminate after a finite number of steps
- **Definiteness** - ensured if every step of an algorithm is precisely defined. For example, "divide by a number x" is not sufficient. The number x must be define precisely, say a positive integer.
- **Input** - domain of the algorithm which could be zero or more quantities
- **Output** - set of one or more resulting quantities; also called the range of the algorithm
- **Effectiveness** - ensured if all the operations in the algorithm are sufficiently basic that they can, in principle, be done exactly and in finite time by a person using paper and pen

CHECK YOUR UNDERSTANDING – Consider the following Java Code (Adopted from JEDI Teacher's Manual for Data Structure and Algorithm). Analyse the code fragment below:

```
public class Minimum {
     public static void main(String[] args) {
               int a[] = { 23, 45, 71, 12, 87, 66, 20, 33, 15,
               69 };
               int min = a[0];
               for (int i = 1; i < a.length; i++) {
                    if (a[i] < min) min = a[i];
               }
               System.out.println("The minimum value is: " +
               min);
     }
}
```

What does the program do? Identify and discuss the different properties that the algorithm possess.

Unit 2: Problem Solving Process

Introduction

Programming is a problem-solving process, i.e., the problem is identified, the data to manipulate and work on is distinguished and the expected result is determined. It is implemented in a machine known as a computer and the operations provided by the machine is used to solve the given problem.

Unit learning outcomes

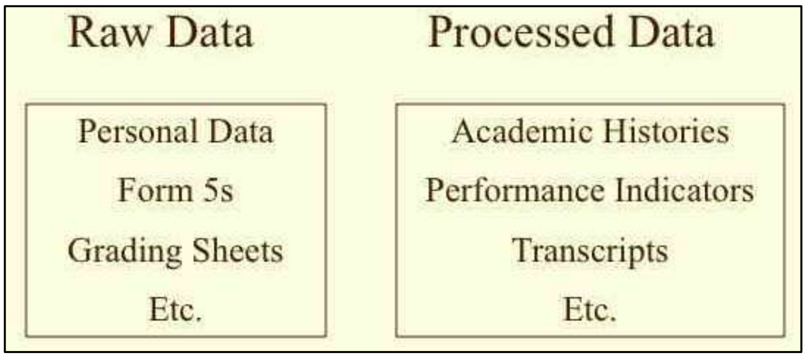
By the end of this unit you should be able to:

- Understand the problem solving process
- Design an algorithm for a specific problem

The Problem Solving Process

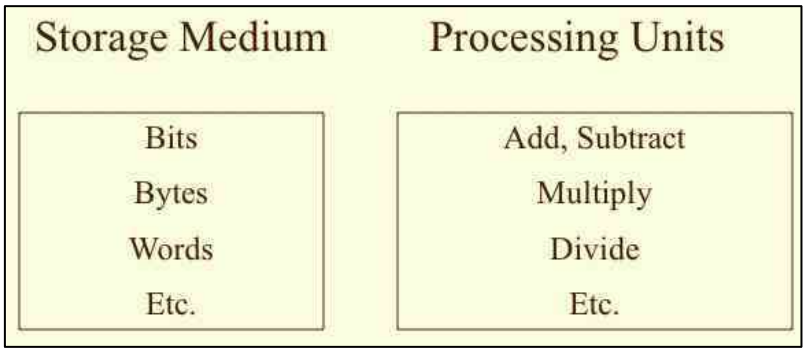
The problem solving process could be viewed in terms of domains – problem, machine and solution.

Problem domain includes the ***input*** or the raw data to process, and the ***output*** or the processed data. For instance, in sorting a set of numbers, the raw data is set of numbers in the original order and the processed data is the sorted numbers.



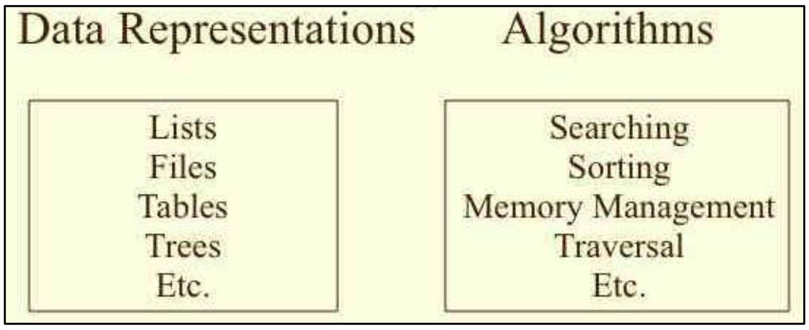
Source: JEDI Teacher's Manual for Data Structures and Algorithm

The **machine domain** consists of storage medium and processing unit. The storage medium – bits, bytes, words, etc – consists of serially arranged bits that are addressable as a unit. The processing units allow us to perform basic operations that include arithmetic, comparison and so on.



Source: JEDI Teacher's Manual for Data Structures and Algorithm

Solution domain, on the other hand, links the problem and machine domains. It is at the solution domain where structuring of higher level data structures and synthesis of algorithms are of concern.



Source: JEDI Teacher's Manual for Data Structures and Algorithm

How to Write an Algorithm

There are no well – defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code. The common constructs across programming languages such as loops and other flow control can be used to write an algorithm. Writing an algorithm is a process that is only executed after the problem domain is well – defined. That is, we should know the problem domain, for which we are designing a solution.

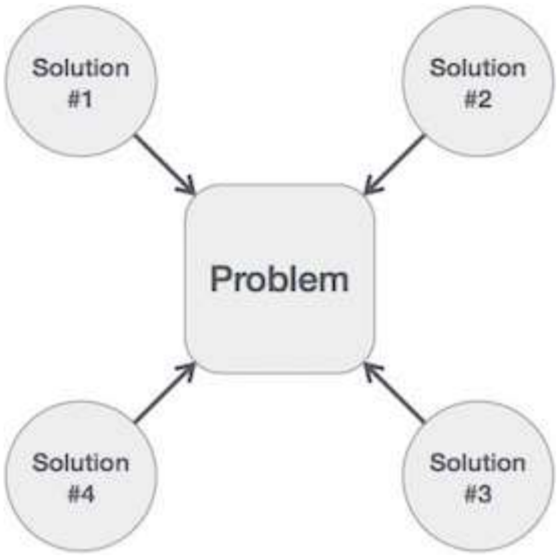
EXAMPLE: **Problem – Design an algorithm to add two numbers and display the result**
Step 1 – START **Step 2** – Declare three integers **a, b** & **c** **Step 3** – Define values of **a** & **b** **Step 4** – Add values of **a** & **b**

```
Step 5 – Store output of Step 4 to c
Step 6 – Print c
Step 7 - STOP
```

Algorithms tell the programmers how to code the program. Alternately, the algorithm can be represented using a flow chart (which was discussed during your CC 102 class) or using a pseudocode.

```
Step 1 − START ADD
Step 2 − get values of a & b
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP
```

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways. Hence, many solution algorithms can be derived for a given problem.



Source: https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm

CHECK YOUR UNDERSTANDING – Design an algorithm for the following using Pseudocode and Flowcharts

1. Show the algorithm for computing the velocity of the car. Having the following inputs:

   D1 – the starting point

   D2 – the ending point

   T – Time interval to reach D2

   V = (D2 – D1) / T

2. Show the algorithm for computing the density of a given square. Use the following as your inputs: M – Mass of the object; V – Volume of the square (S^6); D = M / V - Density of the object
3. Compute the greatest common divisor of two nonnegative integers' p and q as follows: If q is 0, the answer is p. If not, divide p by q and take the remainder r. The answer is the greatest common divisor of q and r.

Unit 3: Mathematical Functions

Introduction

Mathematical functions are useful in the creation and analysis of algorithms. In this section, some of the most basic and most commonly used functions with their properties are listed.

Unit learning outcomes

By the end of this unit you should be able to:

- Understand the mathematical functions in the creation and analysis of algorithms
- Solve problems using the different mathematical functions
- Discuss the different identities in relation to mathematical functions

The Mathematical Functions

- **Floor of x** ($\lfloor x \rfloor$) - greatest integer less than or equal to x, x is any real number
  - e.g.:
    - $\lfloor 3.14 \rfloor = 3$
    - $\lfloor 1/2 \rfloor = 0$
    - $\lfloor -1/2 \rfloor = -1$
- **Ceiling of x** ($\lceil x \rceil$) - smallest integer greater than or equal to x, where x is any real number
  - e.g.:
    - $\lceil \mathbf{3.14} \rceil = \mathbf{4}$
    - $\lceil \mathbf{1/2} \rceil = \mathbf{1}$
    - $\lceil \mathbf{-1/2} \rceil = \mathbf{0}$
- **Modulo** - given any two real numbers x and y,
  - x mod y = x if y = 0
  - = x - y * $\lfloor x / y \rfloor$ if y <> 0
  - e.g.:
    - 10 mod 3 = 1
    - 24 mod 8 = 0
    - -5 mod 7 = 2

Identities

The following are the identities related to the mathematical functions defined above

- $\lceil x \rceil = \lfloor x \rfloor$ if and only if x is an integer
- $\lceil x \rceil = \lfloor x \rfloor + 1$ if and only if x is not an integer
- $\lfloor -x \rfloor = -\lceil x \rceil$
- $\lfloor x \rfloor + \lfloor y \rfloor <= \lfloor x + y \rfloor$
- $x = \lfloor x \rfloor + x \bmod 1$
- $z (x \bmod y) = zx \bmod zy$

CHECK YOUR UNDERSTANDING – Problem Solving: Floor, Ceiling and Modulo Functions. Compute for the resulting value:

a. $\lfloor -5.3 \rfloor$
b. $\lfloor 6.14 \rfloor$
c. 8 mod 7
d. 3 mod –4
e. –5 mod 2
f. 10 mod 11
g. $\lceil (15 \bmod -9) + \lfloor 4.3 \rfloor \rceil$

Unit 4: Complexity of Algorithm

Introduction

Several algorithms could be created to solve a single problem. These algorithms may vary in the way they get, process and output data. Hence, they could have significant difference in terms of performance and space utilization. It is important to know how to analyse the algorithms, and knowing how to measure the efficiency of algorithms helps a lot in the analysis process.

Unit learning outcomes

By the end of this unit you should be able to:

- Discuss the different methods on measuring algorithm efficiency
- Understand the impact of Time Efficiency and Space Utilization to the performance of the algorithm
- Apply Big Oh Notation in determining algorithm complexity

Measuring Algorithm Efficiency

Algorithm efficiency is measured in two criteria: space utilization and time efficiency.

- **Time complexity (Time Efficiency)** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
  - **Execution time** - amount of time spent in executing instructions of a given algorithm. Notation: T(n). Several factors that affect the execution time include:

- ▪ input size
- ▪ Instruction type
- ▪ machine speed
- ▪ quality of source code of the algorithm implementation
- ▪ quality of the machine code generated from the source code by the compiler

- **Space complexity (Space Utilization)** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
  - Time and space complexity depends on lots of things like **hardware**, **operating system**, **processors**, etc.

There are several methods on how to measure the efficiency of an algorithm. Some of these methods include Empirical Method, Theoretical Method, Input and the Process.

- **The Empirical Method**
  - **Def.**: A method of analyzing an algorithm by determining:
    - The number of times each program line runs
    - The amount of time required to execute each line
    - The number of times each procedures were called
    - The amount of time each calls were executed
    - Total space allocated to run the program
  - A program can be written using some programming language and execute in a machine.
  - Measure the elapsed time from the start of execution to completion.
  - The amount of memory utilized can also be monitored.
  - The analysis is a results are based on actual experiences with the program's execution.
  - External Factors affecting the performance of the algorithm
    - Programming language employed
    - Computer processor
    - Available memory
    - Users concurrently using the computer

- **The Theoretical Method**
  - **Def.**: The analysis is based solely on the way algorithms solve problems.
  - The algorithm is language and machine independent.
  - Runtime and space consumptions are not measured in milliseconds, minutes, kilobytes and gigabytes.
  - Algorithms are compared based on which runs faster or which takes less space when implemented in the same programming language and runtime environment.

- **The Input**
  - Algorithms should be given the same input to compare them fairly.
  - Algorithms may be compared when they are given inputs of the same size to keep the efficiency analysis manageable but still fair.
  - Factors to consider when analysing algorithms using Input:

    - **Input Size**

        - The size of the input is a factor in the time and space needed by an algorithm to execute.
            - If the input to an algorithm is a number, then the number of bits to store the number, the magnitude of that number, or the number itself can be the size of the input.
            - If the input is a string, the input size could be the number of symbols in that string.
            - If the input is a two – dimensional array with m rows and n columns with k integers per cell, then the input size could be m x n x k.
        - The more inputs there are, the more time and space algorithms take to process.
        - Algorithm's performance is put to the test when the input size is large enough.
        - Inefficiency is magnified by the size of the input.
        - As the input size increases, algorithms tend to exhibit performance patterns.
        - When input size becomes sufficiently large, some algorithms will always be more efficient than others.

    - **Best, Worst and Average Input Cases**
        - An algorithm may have different runtimes for inputs of the same sizes.
        - Consider a problem of finding a key in a list of n values.
        - The algorithm halts the moment a key matches the value in a list or did not match any of these values. Matching begins from left to right of the key.
            - Best Case: it will run the fasters when the first value equals the key.
            - Worst Case: When the key matches to the rightmost value in the list or the key does not match any value.

- **The Process**
    - The input is not under the control of the algorithm designer. The algorithm should is supposed to work for all valid inputs.
    - Process is what the designer can control.
    - One way to analyze algorithms is to find an upper bound to the time and space required by an algorithm.
        - The space required is the total memory that needs to be allocated.
        - This includes the input, constants and variables at runtime.

The Big Oh Notation

- **T(n) grows at a rate proportional to n and thus T(n) is said to have** *"order of magnitude n" denoted by the O-notation: T(n) = O(n)*
- Formal definition: g(n) = O(f(n)) if there exists two constants c and n0 such that | g(n) | <= c * | f(n) | for all n >= n0.

- The following are examples of computing times in algorithm analysis. To make the difference clearer, let's compare based on the execution time where n = 1000000 and time = 1msec:

Big-Oh	***Description***	***Algorithm***	***Running Time***	***Sample Code Implementation***
$O(1)$	Constant			return n * (n + 1) / 2

$O(\log_2 n)$	Logarithmic	Binary Search	19.93 microseconds	While *n* > 1
				count ← count + 1
				n ← n / 2

$O(n)$	Linear	Sequential Search	1.00 seconds	For i ← 1 to n sum ← sum + i
$O(n \log_2 n)$		Heapsort	19.93 seconds	
$O(n^2)$	Quadratic	Insertion Sort	11.57 days	For *i* ← 1 to *n* For *j* ← *i* to *n* sum ← sum + *j*
$O(n^3)$	Cubic	Floyd's Algorithm	317.10 centuries	
$O(2^n)$	Exponential		Eternity	

Operations on the O-Notation:

- **Rule for Sums**
  - Suppose that $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$.
  - Then, $t(n) = T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

- **Rule for Products**
  - Suppose that $T1(n) = O(f(n))$ and $T2(n) = O(g(n))$.
  - Then, $T(n) = T1(n) * T2(n) = O(f(n) * g(n))$.
  - **For example, consider the algorithm below:**

```
for(int i=1; i<n-1; i++){
     for(int i=1; i<=n; i++){
          steps taking O(1) time
     }
}
```

  - Since the steps in the inner loop will take

    n + n-1 + n-2 + ... + 2 + 1 times,

  - then

    $n(n+1)/2$
    $= n^2/2 + n/2$
    $= O(n^2)$

  - **Consider the code snippet below:**

```
for (i=1; i <= n, i++)
for (j=1; j <= n, j++)
     // steps which take O(1) time
```

- Since the steps in the inner loop will take n + n-1 + n-2 + ... + 2 + 1 times,
  then the running time is

$$n(n+1) / 2 = n^2 / 2 + n / 2$$
$$= O(n^2)$$

Analysis of Algorithms

- **Example 1: Minimum Revisited**

```
1. public class Minimum {
2.
3. public static void main(String[] args) {
4.    int a[] = { 23, 45, 71, 12, 87, 66, 20, 33, 15, 69
};
5.    int min = a[0];
6.    for (int i = 1; i < a.length; i++) {
7.         if (a[i] < min) min = a[i];
8.    }
9.    System.out.println("The minimum value is: " + min);
10. }
11.}
```

In the algorithm, the declarations of *a* and *min* will take constant time each. The constant time if-statement in the "**for**" loop will be executed **n** times, where n is the number of elements in the array a. The last line will also execute in constant time.

Line	# Times Executed
4	1
5	1
6	n+1
7	n
9	1

Using the rule for sums, we have:

T(n) = 2n +4 = O(n)

Since $g(n) <= c\ f(n)$ for $n >= n_0$, then

$2n + 4 <= cn$

$$\frac{2n + 4}{n} <= c$$

2   + 4/n <= c

Thus c = 3 and n_0 = 4.

Therefore, the minimum algorithm is in O(n).

- **Example 2: Linear Search Algorithm**

```
1 found = false;
2 loc = 1;
3 while ((loc <= n) && (!found)){
4    if (item == a[loc]
5    else loc = loc + 1;
6 }
```

STATEMENT	# of times executed
1	1
2	1
3	n + 1
4	n
5	n

T(n) = 3n + 3 so that T(n) = O(n)

Since g(n) <= c f(n) for n >= n 0, then

3n + 3 <= c n

(3n + 3)/n <= c = 3 + 3 / n <= c

Thus c = 4 and n0 = 3.

- The following are the general rules on determining the running time of an algorithm:
  - **FOR loops:** At most the running time of the statement inside the **"for"** loop times the number of iterations.
  - **NESTED FOR loops:** Analysis is done from the inner loop going outward. The total running time of a statement inside a group of for loops is the running time of the statement multiplied by the product of the sizes of all the for loops.
  - **CONSECUTIVE STATEMENTS:** The statement with the maximum running time.
  - **IF/ELSE:** Never more than the running time of the test plus the larger of the running times of the conditional block of statements.

CHECK YOUR UNDERSTANDING – Problem Solving

1. Prove that:
   a. 5n + 3 is O(n)
   b. 4 + 6 log n is O(log n)

c. $3n^2 + 2n + 35$ is $O(n^2)$

d. For above 3 problems, algebraic manipulation is sufficient: find a pair (c, n_0) such that $f(n) <= c * g(n)$ for all $n >= n_0$.

2. Prove that log n is O(n)

   a. Simple algebraic manipulation is not enough; need also induction proof.

3. Prove that $n^2 + 5$ is not O(n)

   a. Assume it is, and then show there will be a contradiction

LEARNING POINTS

- Algorithm is a finite set of instructions which, if followed, will accomplish a task.
- The fundamental properties of algorithm includes Finiteness, Definiteness, Input, Output and Effectiveness.
- The problem solving process could be viewed in terms of domains – problem, machine and solution.
- There are different methods on measuring the efficiency of an algorithm. These methods include Empirical Method, Theoretical Method, Input and Process.
- There are factors that need to be considered in analyzing the algorithm based on input. This includes the Input Size and the Best, Worst and Average Cases for the inputs.
- We always consider the Time and Space Complexity when analyzing an algorithm.

LEARNING ACTIVITIES

1. What is the time complexity of the algorithm with the following running times?
   a. $3n^5 + 2n^3 + 3n + 1$
   b. $n^3/2 + n^2/5 + n + 1$
   c. $n^5 + n^2 + n$
   d. $n^3 + \lg n + 34$

2. Suppose we have two parts in an algorithm, the first part takes $T(n_1)=n^3 + n + 1$ time to
   execute and the second part takes $T(n_2) = n^5 + n^2 + n$, what is the time complexity of the algorithm if part 1 and part 2 are executed one at a time?

3. Sort the following time complexities in ascending order.

$O(n \log_2 n)$	O(n2)	O(n)	$O(\log_2 n)$	$O(n^2 \log_2 n)$
O(1)	O(n3)	$O(n^n)$	$O(2^n)$	$O(\log_2 \log_2 n)$

4. What is the execution time and time complexity of the algorithm below?

```
void warshall(int A[][], int C[][], int n){
    for(int i=1; i<=n; i++)
```

```
            for(int j=1; j<=n; j++)
                 A[i][j] = C[i][j];

       for(int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                 for(int k=1; k<=n; k++)
                      if (A[i][j] == 0) A[i][j] = A[i][k] & A[k][j];
}
```

REFERENCES

- Cormen, Thomas H., et.al. Introduction to Algorithms. The MIT Press, 2009. Page 5 – 15, 43 – 64
- Goodrich, Michael T. (2011). Data Structures and Algorithms in Java. 5th ed. Page 158 – 165
- Delfinado, C. J. A. (2016). Data Structures and Algorithms. Page 3 – 7, 26 – 37
- JEDI Slide – Chapter 01 (Basic Concepts and Notation)
- JEDI Teacher's Manual for Data Structures and Algorithms
- Hackerearth website: www.hackerearth.com
- https://www.bouraspage.com/repository/algorithmic-thinking/what-is-an-algorithm
- https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm