Design & Analysis of Algorithms

Lecture 1

ALGORITHMS

- An **algorithm** is a set of steps of operations to solve a problem performing **calculation**, **data processing**, and **automated reasoning tasks**.

- An **algorithm** is an efficient method that can be expressed within finite amount of **time** and **space**.

- An **algorithm** is the best way to represent the solution of a particular problem in a very simple and efficient way.

- If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming languages**.

Algorithm Design

- The important aspects of **algorithm design** include creating an **efficient algorithm** to **solve a problem** in an **efficient way** using **minimum time** and **space**.

- To solve a problem, different approaches can be followed.
- Some of them can be efficient with respect to **time consumption**, whereas other approaches may be **memory efficient.**
- However, one has to keep in mind that **both** time consumption and memory usage **cannot be optimized simultaneously**.

- If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Problem Development Steps

- The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm

- Designing an Algorithm
- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

Characteristics of Algorithms

- The main characteristics of algorithms are as follows –

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs

- Algorithms are well-ordered with unambiguous operations.
- Algorithms halt in a finite amount of time.
- Algorithms should not run for infinity, i.e., an algorithm must end at some point

Pseudocode

- Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

Difference between Algorithm and Pseudocode

- An algorithm is simply a solution to a problem.

- An algorithm presents the solution to a problem as a well defined set of steps or instructions.

- Pseudo-code is a general way of describing an algorithm.
- Pseudo-code does not use the syntax of a specific programming language, therefore cannot be executed on a computer.
- But it closely resembles the structure of a programming language and contains roughly the same level of detail.

- An **algorithm** is a well defined sequence of steps that provides a solution for a given problem, while a **pseudocode** is one of the methods that can be used to represent an algorithm.

- While algorithms can be written in natural language, pseudocode is written in a format that is closely related to high level programming language structures.

- But pseudocode does not use specific programming language syntax and therefore could be understood by programmers who are familiar with different programming languages.

- Additionally, transforming an algorithm presented in pseudocode to programming code could be much easier than converting an algorithm written in natural language.

How to write an Algorithm?

1. Keep in mind that algorithm is a step-by-step process.
2. Depending upon programming language, include syntax where necessary.
3. Begin.
4. Include variables and their usage.
5. If they are any loops, try to give sub number lists.
6. Try to give go back to step number if loop or condition fails.

Examples Of Algorithms & Pseudocodes

1. An algorithm to add two numbers entered by user

- Step 1: Start
- Step 2: Declare variables num1, num2 and sum.
- Step 3: Read values num1 and num2.
- Step 4: Add num1 and num2 and assign the result to sum. sum←num1+num2
- Step 5: Display sum
- Step 6: Stop

A Pseudocode to add two numbers entered by user

/ x and y will store the input and sum will store the result */*

```
SumOf2Num()
Begin
        Read: x, y;
        Set sum = x + y;
        Print: sum;
End
```

- From the data structure point of view, following are some important categories of algorithms –

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

- Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.

- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

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.

- Since all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

- We write algorithms in a step-by-step manner, but it is not always the case.
- Algorithm writing is a process and is executed after the problem domain is well-defined.
- That is, we should know the problem domain, for which we are designing a solution.

Example

- Design an algorithm to add two numbers and display the result.

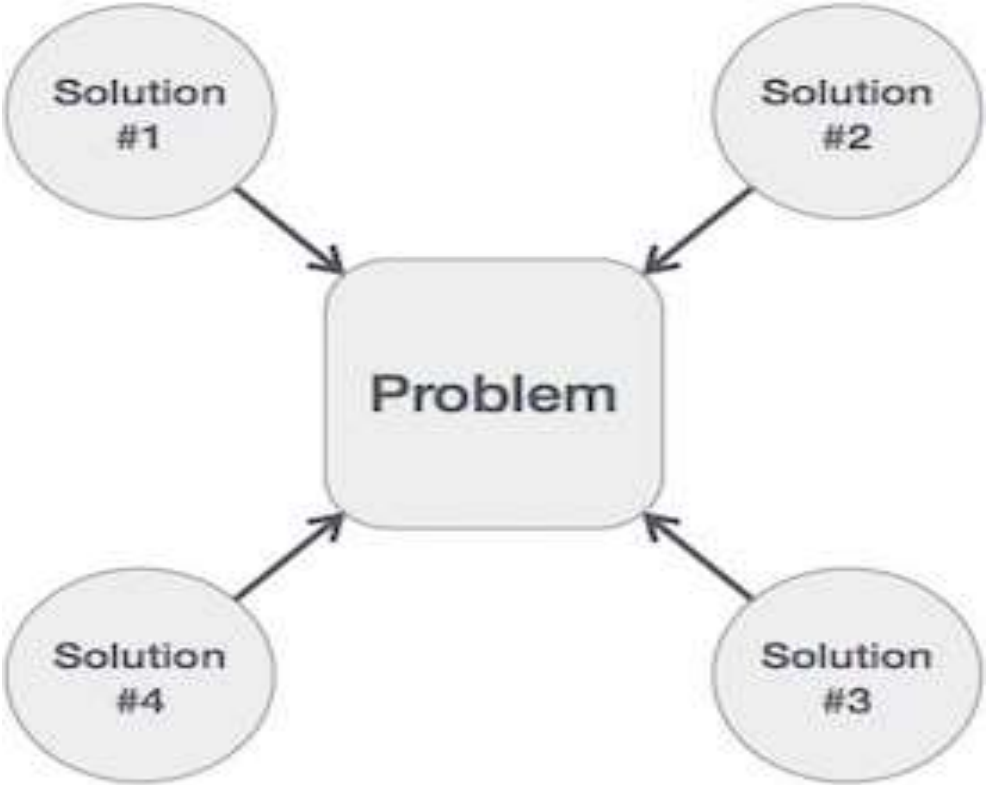
```
step 1 - START

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. Alternatively, the algorithm can be written as –

```
step 1 - START ADD

step 2 - get values of a & b

step 3 - c ← a + b

step 4 - display c

step 5 - STOP
```

- In design and analysis of algorithms, usually the second method is used to describe an algorithm.
- It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions.
- One can observe what operations are being used and how the process is flowing.

- 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.
- The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

- While writing algorithms we will use following symbol for different operations:

‘+’ for Addition

‘-’ for Subtraction

‘*’ for Multiplication

‘/’ for Division and

‘← ’ for assignment.

- For example

A ←X*3 means A will have a value of X*3.

Examples of Algorithm

Problem 1: Find the area of a Circle of radius r

- **Inputs to the algorithm:**
- Radius r of the Circle.
- **Expected output:**
- Area of the Circle

- **Algorithm:**

- Step1: Read\input the Radius r of the Circle
- Step2: Area PI*r*r // calculation of area
- Step3: Print Area

Problem2: Write an algorithm to read two numbers and find their sum.

- **Inputs to the algorithm:**

  First num1.

  Second num2.

- **Expected output:**

  Sum of the two numbers.

- **Algorithm**:

- Step1: Start
- Step2: Read\input the first num1.
- Step3: Read\input the second num2.
- Step4: Sum num1+num2 // calculation of sum
- Step5: Print Sum
- Step6: End

Problem 3: Convert temperature Fahrenheit to Celsius

- **Inputs to the algorithm:**

  Temperature in Fahrenheit

- **Expected output:**

  Temperature in Celsius

- **Algorithm:**

°F to °C	Deduct 32, then multiply by 5, then divide by 9
°C to °F	Multiply by 9, then divide by 5, then add 32

- Step1: Start
- Step 2: Read Temperature in Fahrenheit F
- Step 3: C <- 5/9*(F-32)
- Step 4: Print Temperature in Celsius: C
- Step5: End

°C x 9/5 + 32 = °F

(°F - 32) x 5/9 = °C

Type of Algorithms

- The algorithm are classified in to the three types of *control structures*.

- They are:

1. Sequence
2. Branching (Selection)
3. Loop (Repetition)

- The sequence is exemplified by sequence of statements placed one after the other – the one above or before another gets executed first.

- The *branch* refers to a binary decision based on some condition.

- If the condition is true, one of the two branches is explored; if the condition is false, the other alternative is taken.
- This is usually represented by the '**if-then**' construct in pseudo-codes and programs.
- This structure is also known as the *selection* structure.

Problem1: write algorithm to find the greater number between two numbers

- Step1: Start
- Step2: Read/input A and B
- Step3: If A greater than B then C=A
- Step4: if B greater than A then C=B
- Step5: Print C
- Step6: End

Problem2: write algorithm to find the result of equation:

$$f(x) = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

- Step1: Start
- Step2: Read/input x
- Step3: If X Less than zero then F=-X
- Step4: if X greater than or equal zero then F=X
- Step5: Print F
- Step6: End

Problem3: A algorithm to find the largest value of any three numbers.

- Step1: Start
- Step2: Read/input A,B and C
- Step3: If (A>=B) and (A>=C) then Max=A
- Step4: If (B>=A) and (B>=C) then Max=B
- Step5:If (C>=A) and (C>=B) then Max=C
- Step6: Print Max
- Step7: End

- The ***loop*** allows a statement or a sequence of statements to be repeatedly executed based on some loop condition.

- It is represented by the ‘while’ and ‘for’ constructs in most programming languages, for unbounded loops and bounded loops respectively.

- Unbounded loops refer to those whose number of iterations depends on the eventuality that the termination condition is satisfied; bounded loops refer to those whose number of iterations is known before-hand.

- A trip around the loop is known as iteration.
- One must ensure that the condition for the termination of the looping must be satisfied after some finite number of iterations, otherwise it ends up as an infinite loop, a common mistake made by inexperienced programmers.
- The loop is also known as the ***repetition* structure**.

Problem1: An algorithm to calculate even numbers between 0 and 99

1. Start
2. I ← 0
3. Write I in standard output
4. I ← I+2
5. If (I <=98) then go to line 3
6. End

Problem2: Design an algorithm which gets a natural value, n,as its input and calculates odd numbers equal or less than n. Then write them in the standard output:

1. Start
2. Read n
3. I ← 1
4. Write I
5. I ← I + 2
6. If (I <= n) then go to line 4
7. End

Problem3: Design an algorithm which generates even numbers between 1000 and 2000 and then prints them in the standard output. It should also print total sum:

1. Start
2. I ← 1000 and S ← 0
3. Write I
4. S ← S + I
5. I ← I + 2
6. If (I <= 2000) then go to line 3
else go to line 7
7. Write S
8. End

Problem4: Design an algorithm with a natural number, n, as its input which calculates the following formula and writes the result in the standard output:

$S = ½ + ¼ + \ldots + 1/n$

1. Start
2. Read n
3. I ← 2 and S ← 0
4. S= S + 1/I
5. I ← I + 2
6. If (I <= n) then go to line 4
   else write S in standard output
7. End

Algorithm Analysis

- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- **A Priori Analysis** – This is a **theoretical analysis** of an algorithm.

- Efficiency of an algorithm is measured by assuming that all other factors, for example, **processor speed**, are **constant** and have no effect on the implementation.

- **A Posterior Analysis** – This is an **empirical analysis** of an algorithm.

- The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

- We shall learn about a priori algorithm analysis.
- Algorithm analysis deals with the **execution** or **running time of various operations** involved.
- The **running time of an operation** can be defined as **the number of computer instructions executed per operation**.

- This analysis is done before we code.
- Some tools are required to do this.
- Tools include **instruction count** and **size of input**.

- In **apriori analysis**, we need to **identify the basic operations in the algorithm** and **count** the **number of times that operation is being performed.**

- For example : - in sorting algorithms, the basic operation is the comparison between two elements, and the performance is to measure the total number of comparisons required to sort an array of numbers.

- While doing an **instruction count**, we can do either a **macro** analysis or **micro** analysis.

- **Macro Analysis** – Perform the instruction count for all operations.

- **Micro Analysis** – Perform the instruction count only for dominating operations.

- **For example**, the **comparisons** are basic operations in **searching** and **sorting** algorithms, the **arithmetic** operations are basic operations in **math** algorithms, and **multiplication** and **addition** are basic operations in **matrix multiplication** algorithms.

Algorithm Complexity

- Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

- The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Space Complexity

- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.

- The space required by an algorithm is equal to the sum of the following two components –

1. A **fixed part** that is a space required to store certain data and variables, that are independent of the size of the problem.

- For example, simple variables and constants used, program size, etc.

2. A **variable part** is a space required by variables, whose size depends on the size of the problem.

- For example, dynamic memory allocation, recursion stack space, etc.

- Space complexity S(P) of any algorithm P is

$$S(P) = C + S(I),$$

where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

- Following is a simple example that tries to explain the concept –

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

- Here we have three variables A, B, and C and one constant.
- Hence S(P) = 1 + 3.
- Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

- Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion.

- Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

- For example, addition of two n-bit integers takes **n** steps.

- Consequently, the total computational time is

$$T(n) = c * n,$$

where c is the time taken for the addition of two bits.

- Here, we observe that T(n) grows linearly as the input size increases.

Algorithm & Program

Algorithm	Program
Design	Implementation
Domain Knowledge	Programmer
Any Language	Programming Language
H/W & OS Independent	H/W & OS Dependent
Analysis	Testing

Priori Analysis & Posteriori Testing

Priori	Posteriori
Algorithm	Program
Language Independent	Language Dependent
Hardware Independent	Hardware Dependent
Time & Space Function	Watch Time & Bytes