Unit 2: Basics of Algorithm

Structure

2.0 Introduction

This unit is an introductory unit about the algorithm basics and complexity of the algorithm. It gives you an understanding about Algorithm structure, the format for writing the algorithm and the time of execution and space in memory during the course of execution for the algorithm which considers as the time and space complexity of the algorithm.

2.1 Objectives

Algorithm designing is an important process of solving the problem. The designing of an algorithm considers various aspects like space and time requirement for the execution of algorithm. At the end of this unit, you will be able to:

1. Understand about basic of algorithm.
2. Understand of Computation time for execution of algorithm
3. Understand about requirement of the space during the course of execution for algorithm.
4. Notation for determining the time complexity of the algorithm (Asymptotic notations)

5. Understand the analysis of algorithm with asymptotic notations

2.2 Algorithm

The algorithm provides the way for solving the given problem in a systematic way. The term algorithm refers to the sequence of instructions that must be followed to solve a problem. Alternatively, an algorithm is a logical representation of the instructions which should be executed to perform a meaningful task. There are following characteristics of the algorithm:

- Each instruction should be unique and concise
- Each instruction should be relative in nature and should not repeat infinitely
- Result should be available to the user after the algorithm terminates.

Therefore, an algorithm is any defined computational procedure, along with a specified set of allowable inputs that produce some value or set values as output. There are two basic approaches for designing the algorithm.

- Top-Down approach: In this approach we start from the main component of the program and decomposing it into sub problem or components. This process continues until all the sub modules do not solve. Top-down design method takes the form of **stepwise refinement**. In this, we start with the topmost module and incrementally add modules that it calls.
- Bottom – Up approach: In this approach of designing we start with the most basic or primitive components and proceeds to higher-level components. Bottom-up method works with **layer of abstraction**.

Here a simple example of the algorithm is presented to demonstrate the various algorithmic notations and a way to express the algorithm for solving the problem.

Example:

Algorithm Greatest:

This algorithm finds the largest algebraic element of vector A which contains N elements and places the result in MAX. I is used to subscript A.

1. [Is the vector empty?]

If N < 1

Then print('Empty Vector')

Exit

2. [Initialize]

   MAX=A[1] [We assume initially that A[1] is the greatest element]

   I=2

3. [Examine all elements of vector]

   Do while I<=N

   3.1 [Change MAX if it is smaller than the next element]

   If MAX < A[I]

   Then MAX= A[I]

   3.2 [Prepare to examine next element in vector]

   I = I+1

4. [Finished]

   Exit

2.3 Format Convention of algorithm

Hence from this example we can consider the following format conventions for writing the algorithm. These conventions are so general that these may be used for writing any algorithm.

- Name of Algorithm: Every algorithm is given an identify name
- Introductory Comment: The algorithm name is followed by a brief description of the tasks the algorithm performs.
- Steps: The algorithm is made up of a sequence of numbered steps, each beginning with a phrase enclosed in square brackets which gives an abbreviated description of that step.
- Comments: Every step of the algorithm is explained for better understanding of it. These comments are expressed in brackets. Comments specify no action and are included only for clarity.
- Statements and control Structures: It includes the various operators and looping methods those are required for logical and arithmetical operations. For example; Assignment statement, If-statement, Case statement and looping methods.

- Variable names: An entity that possesses a value and its name is chosen to reflect the meaning of the value it holds. For example The MAX is considered as the variable in our previous example of algorithm Greatest.
- Data structures: various data structures including static and dynamic structures are used for the implementation of algorithm.
- Arithmetic operations and expressions: The algorithm notation includes the standard binary and unary operators according to their standard mathematical order of precedence as follows:

	Operation	**Symbol**	**Order of Evaluation**
1.	Parentheses	()	Inner to outer
2.	Exponentiation, Unary plus, minus	∧, ++ , _ _	Right to left
3.	Multiplication, Division	*, /	Left to right
4.	Addition, Subtraction	=, -	Left to right

- Relations and Relational Operators: There are standard relational operators (<, <=, >=, ≠, =, ==) are used with their usual meaning in the implementation of algorithm. A relation evaluates to a logical expression that is, it has one of two possible values, *True* or *False*.
- Logical operations and Expressions: The algorithmic notation also includes the standard logical operators like NOT, OR & AND with their usual meaning. These may be used to connect relations to form compound relations whose only values are *True* or *False*. In order that logical expressions be clear, we consider that operators precedence is as follows:

Precedence	**Operator**
1	Parentheses
2	Arithmetic
3	Relational
4	Logical

- Input and output: The algorithm notation must include the notation for input and output. The input is obtained and placed in a variable and output is obtained by getting the value from variable.
- Functions: A function is used when we want a single value returned to the calling routine. Transfer of control and returning of the value are accomplished by 'Return (value)'.
- Procedures: A procedure is similar to a function but there is no value returned explicitly. A procedure is also invoked differently, where there are parameters, a procedure returns its results through the parameters.

All algorithms must satisfy the following criteria.

1. Input
2. Output
3. Definiteness
4. Finiteness
5. Effectiveness

The criteria 1 & 2 require that an algorithm produces one or more outputs & have zero or more input. According to criteria 3, each operation must be definite such that it must be perfectly clear what should be done. According to the 4^{th} criteria algorithm should terminate after a finite number of operations. According to 5^{th} criteria, every instruction must be very basic so that it can be carried out by a person using only pencil & paper.

After an algorithm has been designed its efficiency must be analysed. This involves determining whether the algorithm is economical in the use of computer resources, i.e. **CPU time** and **memory requirement.** The term used to refer to the memory required by an algorithm is **memory space** and the term used to refer to the computational time is the **execution time**. The importance of efficiency of an algorithm is the **correctness**. Thus, it always produces the correct result and **algorithm complexity** which considers both the difficulty of implementing an algorithm along with its efficiency.

Therefore the requirement for implementation of an algorithm with correctness considers many aspects. The fundamental question arises is that "How can we judge how useful a certain combination of data structures and algorithm is?" Of course the answer of this question depends that how can we evaluate the effort that arises from performing a computation using the certain

combination of data structures and algorithms. There may be many algorithms devised for an application and we must analyse and validate the algorithms to judge the suitable one.
Hence this effort is measured normally with following two important factors those have the direct relationship with the performance of the algorithm:

- Memory space used i.e. **Space complexity.** The space complexity of an algorithm is the amount of memory it needs to run.

- CPU time involves runtime or execution time for the program based on the algorithm i.e. **Time Complexity.** The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for.

2.4 Complexity of Algorithm

The Complexity of algorithm is considered actually as in the form of computational complexity. Computational complexity is a characterization of the time or space requirements for solving a problem by a particular algorithm. These requirements are expressed in terms of a single parameter that represents the size of the problem. For example we consider a problem of size *n.* Let the time required of a specific algorithm for solving this problem is expressed by a function:

$$f: R \longmapsto R$$

Such that *f(n)*is the largest amount of time needed by the algorithm to solve the problem of size n. The function *'f'* is usually called the time complexity function. Thus, we can say that the analysis of the algorithm requires two main considerations:

- Time Complexity
- Space Complexity

The time complexity of an algorithm is the amount of computer time that it needs to run to completion. The space of an algorithm is the amount of memory that it needs to run to completion.

2.5 Time Complexity

In order to compute the time complexity of an algorithm we consider only the frequency count of the important steps or instructions. Since these data structures are so widespread, it is important to implement them efficiently. This efficiency is measured using the following two methods:

- Asymptotic Analysis
- Big-O analysis

It is very general that the actual time (wall-clock time) of a program is affected by:

- Size of the input
- Programming language
- Programming tricks
- Compiler
- CPU Speed
- Multiprogramming level

Hence instead of wall clock time for the program if we consider the pattern of the program's behaviour as the program size increases. This is called the **Asymptotic Analysis.**

Big- O Analysis

If *f(n)* represent the computing time of some algorithm and *g(n)* represents a known standard function like *n, n^2, n log n etc* then to write: *f(n) is O g(n)* means that *f(n)* of *n* is equal to biggest order of function *g(n).* This implies only when:

$f(n) \leq Clog(n)$ for all sufficiently large integers *n*, where *C* is the constant. Thus from the above statements we can say that the computing time of an algorithm is *O(g(n)),* we mean that its execution time is no more than a constant time *g(n),* n is the parameter which characterizes the input and / or outputs. From the practical point of view, we get the Big-O notation for a function by:

1. Ignoring multiplicative constants (these are due to pesky difference in compiler, CPU, etc.)
2. Discarding the lower order terms (as *n* gets larger, the largest term has the biggest impact). Like;
    - *8410 = O(1)*

- $100n^3+nlogn+67n^7+4n = O(n^7)$

The Big-O notation helps to determine the time as well as space complexity of the algorithms. The Bog-O notation has been extremely useful to classify algorithms by their performances. Now we consider the three simple algorithms with different number of sequences or steps:

Algorithm 1:

a=a+1

Algorithm 2:

For i= 1 to n do:

a=a+1

end loop

Algorithm 3:

For i=1 to n do

For j= 1to n do

a=a+1

end loop

end loop

Now we do the analysis of these three algorithms and can see their performance with Big – O notation. In the algorithm 1 we may find that the execution statement *a=a+1* is the independent and is not constrained within any loop. Therefore, the number of times this will execute is 1. Thus, the frequency count of this algorithm is 1. Hence its **Time Complexity** is *O(1)*.

In the second algorithm, the execution statement *a=a+1* is inside the loop. The number of times it is executed is *n* as the loop runs for *n* times. The frequency count for this algorithm is *n*. Hence its **Time Complexity** is *O(n)*.

In the third algorithm, the frequency count for the execution statement *a=a+1* is n^2 as the inner loop runs *n* times, each time the outer loop runs, the outer loop also runs for n times. Hence its **Time Complexity** is $O(n^2)$.

Therefore during the analysis of algorithm we have the concern to determine the order of magnitude of an algorithm. Thus, we consider only those statements which may have the greatest frequency count.

2.6 Common Computing Times of Algorithm

The common computing times of algorithms in the order of their performance are as follows:

- *O(1)*: It means that the computing time of the algorithm is constant
- *O(log n)*: It means that the computing time of the algorithm is logarithmic
- *O(n)*: It means that the computing time of the algorithm is directly proportional to n. It is known as the linear time.
- *O(n log n)*: It means that the computing time of the algorithm is logarithmic
- $O(n^2)$: It is known as the quadratic time
- $O(n^3)$: It is known as the cubic time
- $O(2^n)$: It is known as the exponential time. Generally the algorithm with exponential time has no practical use.

There are different types of time complexities which can analyse for an algorithm:

- Best case time complexity: The best case complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size *'n'*. The running time of many algorithm varies not only for the input of different size but for the different inputs of the same size.
- Average case time complexity: The time that an algorithm will require to execute input data of size *'n'* is known as average case time complexity. We can say that the value that is obtained by averaging the running time of an algorithm for all possible inputs of size *'n'* can determine average-case time complexity.
- Worst case time complexity: The worst time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size *'n'*. The worst case complexity is useful for a number of reasons. After knowing the worst case time complexity, we can guarantee that the algorithm will never take more than this time.

Hence the computation of exact time taken by the algorithm for its execution is very difficult. Thus, the work done by an algorithm for the execution of the input of size *'n'* defines the time analysis as function *f(n)* of the input data items.

2.7 Example and analysis

Example: This example exhibits the analysis of linear search algorithm complexity.
Consider the algorithm to search vector (array) V of size N for the location containing value X.
Algorithm SEARCH [Given a vector V containing N elements, this algorithm searches V for the value of a given X. FOUND is a Boolean variable. I and LOCATION are integer variables.]

1. [Search for the location of value X in vector V]

   FOUND = false

   I=1

   Do while ((I<=N) && (FOUND ==false))

   If V[I] ==X

   Then FOUND==true

   LOCATION = I

   EXIT

   Else I = I + 1

   Print("Value of", X,"NOT FOUND")

2. [Finished]

   Exit

Analysis: A reasonable active operation in the algorithm is the comparison between values of *V* and *X*. However, a problem arises in counting the number of active operations executed and the answer depends on the index of the location containing *X*. The **best case** is when *X* is equal to *V [1]* since only one comparison is used. The **worst case** is when *X* is equal to *V [N]* and *N* comparisons are used. Now to obtain the time of execution for the average case we need to know the probability distribution for the value *X* in the vector, i.e. the probability of X occurring in each location. If we assume the vector is not sorted, it is reasonable to assume that *X* is equally likely to be in each of the locations. But *X* might not be in the list at all. Let *q* be the probability that *X* is in the list. Then using the above assumption, we have;

Probability *X* is in location = *q/N;*

Probability *X* is not in the vector = *1-q;*

The average time is given by:

$T(avg) = \sum_{\sin S}$ *(Probability of situation s) * (time for situation s)* [where *S* is the set of all possible situations where the *X* can be found]

$$T(avg) = \sum_{s=1}^{N} \frac{q}{N} * s + (1-q) * N = \frac{q}{N} \sum_{s=1}^{N} s + (1-q) * N = q * \frac{(N+1)}{2} + (1-q) * N$$

Thus if *q=1*, then: $T(avg) = \frac{(N+1)}{2}$

And if *q=1/2*, then: $T(avg) = \frac{(N+1)}{4} + \frac{N}{2} \approx \frac{3N}{4}$

So in either case the time is proportional to N.

Thus we obtain the time complexity for three cases as:

Best - case time for the linear search is *O (1)*

Worst-case time for the linear search is *O (N)*

Average-case time for the linear search is *O (N)*

Space Complexity

The space needed by the program is the sum of the following components:

- **Fixed space requirement:** This includes the instruction space, for simple variables, fixed size structured variables and constants.
- **Variable space requirement:** This consists of space needed by structured variables whose size depends on particular instance of variables.

2.8 Summary

This unit described the algorithm and its analysis in very concise manner. The algorithm provides the way for solving the given problem in a systematic way. The following points are described in this unit:

- The term algorithm refers to the sequence of instructions that must be followed to solve a problem.
- An algorithm is a logical representation of the instructions which should be executed to perform a meaningful task.

- Analysis of the algorithm is done after determining the running time of an algorithm based on the number of basic operations it performs.
- There are two basic approaches for designing the algorithm i.e. Top-down approach and Bottom –Up approach.
- The running time varies depending upon the order in which input data is supplied to it.
- Analysis of an algorithm is done on the following basis:
    - Best case time complexity
    - Worst case time complexity
    - Average case time complexity
- Comparison of algorithm is done on the basis of the programming efforts for a program and on the basis of time and space requirements for the program.
- Big 'O' notation is extremely useful for classifying algorithms by their performances.
- Examples and analysis for computing the time complexity of the algorithm is explained.

Bibliography

Horowitz, E., S. Sahni: "Fundamental of computer Algorithms", Computer Science Press, 1978

J. P. Tremblay, P. G. Sorenson "An Introduction to Data Structures with Applications", Tata McGraw-Hill, 1984

M. Allen Weiss: "Data structures and Problem solving using C++", Pearson Addison Wesley, 2003

Ulrich Klehmet: "Introduction to Data Structures and Algorithms", URL: http://www7 . Informatik.uni-erlangen.de/~klehmet/teaching/SoSem/dsa

Markus Blaner: "Introduction to Algorithms and Data Structures", Saarland University, 2011

V. Abo. Hopcropft, Ullaman, "data Structure and Algorithms", I.P.E.

Seymour Lipschutz, "Data Structure", Schaum's outline Series.

Self Evaluation

1. Algorithm must be:
    a. Efficient
    b. Concise and compact
    c. Free of ambiguity
    d. None of these
2. In top-down approach:
    a. A problem is subdivided into sub problems.
    b. A problem is tackled from beginning to end in one go.
    c. Sub-problems are solved first; these all solutions to sub-problems are put to solve the main problem.
    d. None of these.
3. Which one of the following is better computing time?
    a. O (N) b. O (2^N)
    c. O($\log_2$ N) d. None of the above
4. Define algorithm and design an algorithm to find out the total number of even and odd numbers in a list of 100 numbers.
5. Explain different ways of analyzing algorithm.
6. What is time and space complexity for the algorithm?
7. What is Big-O method for algorithm analysis?
8. Determine the complexity of the algorithm with Big - O notation for the following statement:

```
for i = 1 to n
for j = 1 to n
for k = i to n
        a = a*2;
        b = b+1
end loop
end loop
end loop
```