CHAPTER

1

BASICS OF ALGORITHM

1.0 INTRODUCTION

Suppose we have to find the greatest common divisor (GCD) of two positive integers A and B. Euclid[2] gave the following procedure to compute it.

```
Step 0: Input A and B.
Step 1: If A and B are equal, then either is the GCD.
Step 2: If A is greater than B, replace A by the difference of
        A and B, otherwise replace B by the difference of B and A.
Step 3: Go to Step 1.
```

Suppose $A = 45$ and $B = 15$. The sequence of computations is:

	A	B
Step 0	45	15
Step 1	30	15
Step 2	15	15
Step 3	GCD = 15	

The above is a well-ordered sequence of precise arithmetic actions. It requires finite input. Each action takes finite time, the output is finite and the actions are guaranteed to terminate. The above is an example of an algorithm. The above algorithm requires time logarithmic with respect to the value of the numbers and is linear with respect to the input length.

[2] The great Greek mathematician gave the first non-trivial algorithm sometime between 400 and 300 B.C.

An algorithm is a procedure that can be executed on a Turing machine. An algorithm has the following characteristics:

(1) It is a finite sequence of instructions to achieve some particular output.
(2) It requires finite inputs.
(3) It produces finite output.
(4) It terminates after a finite time.

The word algorithm comes from the name of a Persian mathematician, A.J.M ibn Musa al Khowarizmi. In computer science 'algorithm' refers to a method that can be used by a computer for the solution of a problem.

Algorithms can be broadly classified into deterministic and non-deterministic algorithms.

Deterministic algorithm: The output for the same input is always the same irrespective of place and time.

Non-deterministic algorithm: Different outputs can be obtained with the same set of inputs while using the same algorithm. Action corresponding to one or more steps of the algorithm is dependent on a number of alternatives. It is basically a definitional device for capturing the notion of verifiability.

There are a number of standard methods for designing an algorithm. The most important among them are:

(1) Divide-and-conquer method
(2) Greedy method
(3) Dynamic programming method
(4) Branch-and-bound method
(5) Approximation methods, etc.

In addition, we have the brute-force method based on a problem's statement and definitions of the concepts involved. This method is the easiest to apply. Sometimes, this strategy is quite useful but in most of the cases, it leads to inefficient algorithms.

The quality of an algorithm is decided by two parameters: how much extra space it requires other than the input, and the time it requires for execution for a given input of size n. An algorithm is generally considered to be a good one if its time complexity is of the form of a polynomial in 'n', where n is the size of input (size of input is defined differently for different cases). We analyze algorithms in terms of their time complexities.

For analysis of time complexity, we consider the performance of the algorithm in the following cases:

(1) *Best case:* Inputs are provided in such a way that the minimum time is required to process them.

(2) *Average case:* Average behaviour of the algorithm is studied for various kinds of inputs. Let $Prob(I)$ be the probability of occurrence of problem instance I. Let $t(I)$ be the time required for the processing of this instance. The average time complexity of the algorithm is then the sum of the product $Prob(I)*t(I)$ for all possible problem instances of the given size. $Prob(I)$ is determined from experience or from simplifying

assumption like that all inputs of size n are equally likely to occur. If finding $Prob(I)$ is complicated, the computation of average behaviour is difficult.

(3) *Worst case:* Inputs are given in such a way that maximum time is required to process them. If $t(I)$ is the time for processing of the problem instance I of size n, the worst case complexity for problems of size n is the maximum of all such $t(I)$.

A single operation can be more time consuming in some situations, though the total time for a sequence of n such operations is always better than the worst-case time of that operation multiplied by n. The high cost of such a worst-case occurrence over the entire sequence can be amortized in a way similar to the way a business house amortizes the cost of an expensive product over the years of the product's production cycle.

Let us consider the problem of searching an element in a given array of N elements. In the best case, we will make one comparison, in the worst case we will make $N - 1$ comparisons, in the average case we will make $N/2$ comparisons under the assumption that the element we are looking for is equally likely to be present at any of the N positions in the array.

We can have the following situations with algorithms:

(1) Different algorithms for the same problem may take different amounts of time. For example, the problem of searching can be solved by both linear search algorithm and binary search algorithm. But the binary search is much faster than the linear search.

(2) Some algorithms require different amounts of time for different inputs. As an example, consider bubble sort that has different time complexities for the best, the average and the worst-case inputs.

In algorithms, we may have to follow a set of instructions repeatedly. For implementing this, we have two approaches: (a) recursion and (b) iteration. For programming purpose, they are different. However, for analysis purpose, they are the same except that the analysis is more straightforward and easier in the iterative case than in the recursive case. We will give two examples to show both the iterative and recursive implementations.

EXAMPLE 1.1: Fibonacci series: $F_0 = F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$, $i \geq 2$.

(a) Recursion

```
Fib(n) /* n terms */
   If (n = 0 or n = 1) Then Return(1);
   Else
   Return (Fib(n - 1) + Fib(n - 2));
   Endif
End Fib
```

(b) Iteration:

```
Fib(n)
   If (n = 0 or n = 1) Then Return(1);
   F0 = 1;
   F1 = 1;
```

```
    For i = 2 to n Step 1 Do
      F = F0 + F1;
      F0 = F1;
      F1 = F;
    Endfor
    Return(F)
End Fib
```

We can see that it is much easier to count the number of steps (operations that can be assumed to take constant/unit amount of time) in the iterative case than in the recursive case.

Similarly, for the next example, we find that the same is true.

EXAMPLE 1.2: Factorial of a number.

(a) Iterative

```
Fac(n)
If (n < 0) Then Return('error');
  Else If (n < 2) Then Return(1);
    Else
      PROD = 1;
For i = n down to 1 step 1 Do
  PROD = PROD * i;
Endfor
Return (PROD);
END Fac
```

(b) Recursive

```
Fac(n)
If (n < 0) Then Return('error');
  Else if (n < 2) Then Return(1);
    Else
      Return (n * Fac(n - 1));
END Fac
```

We now give some examples of algorithms with different forms of time complexities.

EXAMPLE 1.3: Trace of a matrix (sum of diagonal elements).

```
Procedure Trace (A,n)
  Array A(n)
  T = 0;
  For i = 1 to n in Step 1 Do
    T = T + A(i,i);    /* A is n × n matrix */
  Endfor
End Trace
```