UNIT 3 COMPLEXITY ANALYSIS OF SIMPLE ALGORITHMS

Structure **Page Nos.**

3.0 INTRODUCTION

Computational complexity describes the amount of processsing time required by an algorithm to give the desired result.Generally we consider the worst-case time complexity (big O notation) which is the maximum amount of time required to execute an algorithm for inputs of a given size. Whereas average-case complexity, which is the average of the time taken on inputs of a given size is less common.In the previous unit we introduced a concept of efficiency of an algorithm and discussed three asymptotic notations which are formal methods for analyzing algorithm efficiency in terms of time and space complexities. The complexity analysis of algorithm helps to understand the behavior of the algorithm and compare it with other algorithms for a large input size. The structure of the unit is as follows: section 3.3makessimple statements, a brief review of asymptotic notations followed by analysis of simple constructs such as looping statement, conditional statement, and etc.In the subsequent sections we describe general rules for analysis of algorithms and illustration with several examples.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- Algorithm to add n cube numbers
- An algorithm to evaluate polynomial by Horner's rule
- Analysis of Matrix Multiplication algorithm
- Define big oh, big omega and big theta notation
- Exponent evaluation in logarithmic complexity
- Linear search and its complexityanalysis
- Basic sorting algorithm and their analysis

3.2 A BRIEF REVIEW OF ASYMPTOTIC NOTATIONS

The complexity analysis of algorithm is required to measure the time and space required to run an algorithm. In this unit we focus on only the time required to execute an algorithm. Let us quickly review some asymptotic notations(Please refer to the previous unit for detailed discussion)

The central idea of these notations is to compare the relative rate of growth of functions.

Assume T(n) and f(n) are two functions

(i) $T(n) = O(f(n))$ if there are two positive constants C and n_0 such that$T(n) \leq Cf(n)$where n $\geq n_0$

(ii) $T(n) = \Omega(f(n))$ if there are two positive constants C and n_0 such that $T(n) \geq C\Omega$ f(n) where n $\geq n_0$

(iii) $T(n) = \theta(f(n))$ if and only if $T(n) = O(f(n))andT(n) = \Omega(f(n))$

I Assume $T(n) = 1000nandf(n) = n^3$. A function 1000 n is larger than n^3 for small value of n, but n^3 will grow at faster rate if the value n become large. Therefore n^3 is a larger function. The definition of $T(n) = O(f(n))$says thatT(n) will grow slower or equal to C.f(n) after the point where n $\geq n_0$.

The second definition, $T(n) = \Omega(f(n))$ says that the growth rate of T(n) is faster than or equal to (≥) f(n).

3.3 ANALYSIS OF SIMPLE CONSTRUCTS OR CONSTANT TIME

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) 'if (i) statements are simple statement like assignment, increment or decrement operation and declaration statement and (ii) there is no recursion, loops or call to any other function.

Ex. $int\ x;$

$x = x + 5$

$x = x - 5$

2) O(n): This is running time of a single looping statement which includes comparing time, increment or decrement by some constant value looping statement.

```
// Here c is a positive integer constant
  for (i = 1; i<= n; i += c) {
     // simple statement(s)
```

```
}

for (int i = n; i> 0; i -= c) {
    // simple statement(s)
}
```

3) O(n^c): This is a running time of nested loops. Time complexity of nested loops is equal to the number of times the innermost statements is executed. For example, the following sample loops have O(n^2) time complexity

```
for (int i = 1; i<=n; i += c) {
    for (int j = 1; j <=n; j += c) {
      // some simple statements
    }
}
```

Most of the simple sorting algorithms

have O(n^2) time complexity in the worst case.

4) O(logn) If the loop index in any code fragment is divided or multiplied by a constant value, the time complexity of the code fragment is O(logn)

```
for (int i = 1; i<=n; i *= c ) {
    // some simple statements
}
for (int i = n; i> 0; i /= c) {
    // simple statements
}
```

5) Time complexities of consecutive loops

If the code fragment is having more than one loop, time complexity of the fragment is sum of time complexities of the individual loops.

```
for (int i = 1; i<=m; i ++ c) {
// simple statements taking θ(1)
}
for (int f = 1; i<=n; if += X) {
    // simple statements of θ(1)
}
```

Time complexity of above fragment is O(m) + O(n) which is O(m+n)

If one looping statement is consecutive if-else statement

6)Consecutive if else statements

The running time of the if-else statement is just running time of the testing the condition plus the larger of the running of times statement1 or statement2

code fragment of
if – else is

```
if (condition)
    statement 1
else
    statement 2
```

3.4 ANALYSIS OF SIMPLE ALGORITHMS

In this section we will illustration analysis of simple algorithmsto simplify the complexity analysis we will apply the following general rules.

- By default it is big oh running time.
- No consideration of low-order terms.
- No consideration of constant value.

3.4.1 A Summation Algorithm

The following is a simple program to calculate $\sum_{i=1}^{n} i^3$

int sum of n cube (int n)

{

1. int i, tempresult;
2. tempresult =0;
3. for (i=1 ; I <=n; i++)
4. tempresult = tempresult + i * i * i
5. return tempresult;

}

Line#1- 2 units of time required for declaration.

Line# 2- 1 unit of time required for assignment operation.

Line# 3- The **for** loop has several unit costs: initializingi, cost for testingi<=n

(n+1 unit cost) and cost of incrementing i(1 unit of cost) Total cost is 2n +2

Line# 4- 2units of time for multiplication, 1 unit for addition and one unit of time for assignment operation in one cycle. Therefore the total cost of this line is 4n

Line# 5- It will take 1 unit of time. Overall cost will be = 6n+6 which is written as O(n).

3.4.2 Polynomial Evaluation

A polynomial is an expression that contains more than two terms. A term comprises of a coefficient and an exponent.
Example: $P(x) = 15x^4+7x^2+9x+7P(x)=14x^4+17x^3-12x^2+13x+16$

A polynomial may be represented in form of array or structure. A structure representation of a polynomial contains two parts – (i) coefficient and (ii) the corresponding exponent. The following is the structure definition of a polynomial:

```
Struct polynomial{
        int coefficient;
        int exponent;
};
```

How to evaluate the polynomial? It can be evaluated through brute force method and Horner's method. Let us try to understand through an example

Considerthepolynomial

Supposethatexponentiationisimplemented through multiplications. The processes of evaluation through both the methods are shown below:

Bruteforcemethod:

$P(x)=15*x*x*x*x+17*x*x*x-12*x*x+13*x+16$

Horner'smethod:

$P(x)=(((15*x+17)*x-12)*x+13)*x+16$
Please observe the basic operations are: multiplication, additionand subtraction. Since thenumberofadditionsandsubtractionsarethesameinboth the solutions, we will consider the number of multiplications only in worst case analysis of both the methods.
[Thegeneralformofa polynomialofdegree n,andexpressourresultintermsofn We'lllookattheworstcase(maximumnumber ofmultiplications) togetanupper boundonthework]

Now consider the generalformofapolynomialofdegreenis

$P(x)=a_nx^n+a_{n-1}x^{n-1}+.....+a_1x^1+a_0x^0$

where $a_0,a_1,.....a_{n-1},a_n$ are coefficients and $x^0, X^1, \ldots X^{n-1}, X^n$ are related exponentsand we have to evaluate polynomial at a specific value for x.

(i) AnalysisofBruteForce Method

A brute force approach to evaluate a polynomial is to evaluate all terms one by one. First calculatex^n, multiply the value with the related coefficienta_n ,repeat the same steps for other terms and return the sum

$P(x)=a_n{*}x{*}x{*} \ldots{*}x{*}x + a_{n-1}{*}x{*}x{*} \ldots {*}x{*}x + a_{n-2}{*}x{*}x{*} \ldots{*}x{*}x + \cdots$

$+a_2{*}x{*}x{*}+a_1{*}x+a_0$

In the first term, it will take n multiplications, in the second term n-1 multiplications, in the third term it takes n-2multiplications….. In the last two terms: $a_2{*}x{*}x{*}$and $a_1{*}x$it takes 2 multiplications and 1 multiplicationaccordingly.

Numberofmultiplicationsneededintheworstcaseis

$T(n)=n+(n-1)+(n-2)+\ldots+3+2+1$

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

(ii) AnalysisofHorner'sMethod

$P(x)=(\ldots(((a_n{*}x+a_{n-1}){*}x+a_{n-2}){*}x+\ldots+a_2){*}x+a_1){*}x+a_0$

Inthefirstterm,ittakesone multiplication,inthesecondtermone multiplication,inthethirdtermit takesone multiplication …. . Similarly in all other terms it will take one multiplication.

AnalysisofHorner'sMethod

Numberofmultiplicationsneeded intheworstcaseis:

$T(n) = \sum_{i=1}^{n} 1 = n$

$T(n)=n$

Consider only the first term of a polynomial of degree ***n***: $a_n x^n$to appreciate the efficiency of Horner's rule .Just computing this single term by the brute-force algorithm would require ***n*** multiplications, whereas Horner's rule requires only one multiplication in every term.

(iii) Pseudo code for polynomial evaluation using Horner method, Horner(a,n,x)

//In this poly is an array of n elements which are coefficient of polynomial of degree n

1. Assign value of poly p[n]= coefficient of nth term in thepolynomial
2. set i=n-1
4. compute p = p * x + poly[i];
5.i=i-1
6. if i is greater than or equal to 0 Go to step4.

7. final polynomial value at x isp.

StepII.AlgorithmtoevaluatepolynomialatagivenpointxusingHorner'srule:

Input: An array A[0..n] of coefficient of a polynomial of degree n and a point x

Output: The value of polynomial at given pointx

```
Evaluate_Horner(a,n,x)
{
p = A[n];
for (i = n-1; i≤0;i--)
        p = p * x + A[i];
return p;
}
```

For Example: $p(x)=3x^2+5x+6$ using Horner's rule can be simplified as

follows

At x=3,
p(x) = (3x+5)x+6
p(2)=(9+5).3+6
= (14).3+6
=42+6
=48

Complexity Analysis

Polynomial of degree n using Horner's rule is evaluated as below:
Initial assignment, p = a[n]
After the first iteration p = $xa_n + a_{n-1}$
After the second iteration, p = $x(xa_n + a_{n-1}) + a_{n-2}$
$= x^2a_n + x\,a_{n-1} + a_{n-2}$

Every subsequent iteration uses the result of previous iteration i.e next iteration multiplies the previous value of p then adds the next coefficient, i.e.

$p = x(x^2a_n + x\,a_{n-1} + a_{n-2}) + a_{n-3}$

$= x^3a_n + x^2a_{n-1} + x\,a_{n-2} + a_{n-3}$etc.

Thus, after n iterations, $p = x^n a_n + x^{n-1} a_{n-1} + \ldots + a_0$, which is the required correct value.

In above function

First step is one initial assignment that takes constant time i.eO(1).

For loop in the algorithm runs for n iterations, where each iteration cost O(1) as it includes one multiplication, one addition and one assignment which takes constant time.

Hence total time complexity of the algorithm will be O(n) for a polynomial of degree n.

☞Check Your Progress 1

1. Define polynomial. Write the expression of polynomial of degree.

……

……

……

2 Evaluate $p(x)= 3x^4+2x^3-5x+7$ at x=2 usingHorner's rule. Show step wise iterations.

……

……

……

3. Write basic algorithm to evaluate a polynomial and find its complexity. Also compare its complexity with complexity of Horner's algorithm.

……

……

……

MATRIX (N X N)MULTIPLICATION

Matrix is very important tool in expressing and discussing problems which arise from real life cases. By managing the data in matrix form it will be easy to manipulate and obtain more information. One of the basic operations on matrices is multiplication.

In this section matrix multiplication problem is explained in two steps as we have discussed GCD and Horner's Rule in previous section. In the first step we will brief pseudo code and in the second step the algorithm for the matrix multiplication will be discussed. This algorithm can be easily coded into any programming language.
Further explanation of the algorithm is supported through an example.

Let us define problem of matrix multiplication formally, then we will discuss how to multiply two square matrix of order n x n and find its time complexity. Multiply two matrices A and B of order nxn each and store the result in matrix C of order nxn.

A square matrix of order nxn is an arrangement of set of elements in n rows and n columns.

Let us take an example of a matrix of order 3 x 3 which is represented as

$$A= \begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}_{3 \times 3}$$

This matrix A has 3 rows and 3 columns.

Step I : Pseudo code: For Matrix multiplication problem where we will multiply two matrices A and B of order 3x3 each and store the result in matrix C of order 3x3.

1. Multiply first row first element of first matrix with first column first elementof secondmatrix.
2. Similarly perform this multiplication for first row of first matrix and firstcolumn of second matrix. Now take the sum of thesevalues.
3. The sum obtained will be first element of product matrixC
4. Similarly Compute all remaining element of product matrix

C.

i.e $c_{11}= a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31}$

C= A x B

Step II : Algorithm for multiplying two square matrix of order n x n and find the product matrix of order n x n

Input: Two n x n matrices A and B
Output: One n x n matrix C = A x B

```
Matrix_Multiply(A,B,C,n)
{

for i = 0 to n-1 //outermost loop
      for j = 0 to n-1
      {
      C[i][j]=0              //assignment statement
                  for k = 0 to n-1 // innermostloop
                  C[i][j] = C[i][j] + A[i][k] * B[k][j]
      }
}
```

For Example matrix A (3 x 3) , B(3 x3)

$$A = \begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{matrix}$$

$$B = \begin{matrix} 1 & 1 & 1 \\ 2 & 3 & 2 \\ 3 & 2 & 1 \end{matrix}$$

To compute product matrix C= A x B

$$\begin{matrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{matrix} = \begin{matrix} 1x1+2x2+3x3 & 1x1+2x3+3x21x1+2x2+3x1 \\ 2x1+3x2+4x3 & 2x1+3x3+4x22x1+3x2+4x1 \\ 4x1+5x2+6x3 & 4x1+5x3+6x24x1+5x2+6x1 \end{matrix}$$

$$= \begin{matrix} 14 & 13 & 8 \\ 20 & 19 & 12 \end{matrix}$$

32 31 20

Complexity Analysis

First step is, for loop that will be executed n number of times i.e it will take O(n) time. The second nested for loop will also run for n number of time and will take O(n) time.

Assignment statement inside second for loop will take constant time i.eO(1) as it includes only one assignment.

The third for loop i.e innermost nested loop will also run for n number of times and will take O(n) time . Assignment statement inside third for loop will cost O(1) as it includes one multiplication, one addition and one assignment which takes constant time.

Hence, total time complexity of the algorithm will be $O(n^3)$ for matrix multiplication of order nxn.

☞Check Your Progress 2

1. Writeaprogramin‘C’tofindmultiplicationoftwomatricesA[3x3]andB[3x3].

……

……

……

3.4.3 EXPONENTEVALUATION

Exponent evaluation is the most important operation. It has applications in cryptography and encryption methods, The **exponent** tells us how many times to multiply the base by itself.Raising$a$tothe powerof$n$isexpressedasmultiplicationby$a$donen-1 times:$a^n=a\cdot a\cdot\ldots\cdot a$.However, thismethod is notpracticalforlarge$a$or$n$.Therefore we will apply the binary exponentiation method .Theideaofbinary exponentiations,to split the exponent usingthebinaryrepresentation and then do multiplication workof.Let’swrite$n$isbase2, for example:

$4^{11}=4^{10112}=4^{8}.4^{2}\ \cdot 4^{1}$

Sincethenumber$n$has exactly$\lfloor\log_2 n\rfloor+1$digitsinbase2,weonlyneedtoperform $O(\log n)$multiplications,ifweknowthepowers$a^1,a^2,a^4,a^8,\ldots,a^{\lfloor\log n\rfloor}$.

The following example illustrates the intermediate steps in binary exponentiation. Every subsequent multiplication is just the square of the previous multiplication

$4^1=4$

$4^2=(4^1)^2=4^2=16$

$4^4=(4^2)^2=16^2=256$

$4^8=(4^4)^2=256^2=65{,}536$

Therefore the final answer for 4^{11}, we only need to multiply three of them (skipping

4^4becausethe correspondingbitin$n$is set to zero):$4^{11}=65{,}536\cdot16\cdot4=4{,}194{,}304$.

Thetimecomplexityofthis algorithmis(log$n$):tocompute$a^{\log n}$powerof$a$andthentodoalmostlog$n$multiplicationstoget thefinalresult fromthem.

Computing x^n at some point x = a i.e a^n tends to brute force multiplication of a by itself n-1 times. So. To reduce the number of multiplication binary exponentiation methods to compute x^n will be discussed. Processing of binary string for exponent n to compute x^n can be done by following methods:

- left to right binary exponentiation
- right to left binary exponentiation

Left to right binary exponentiation

In this method exponent n is represented in binary string. This will be processed from left to right for exponent computation x^n at x=a i.ea^n .First we will discuss itspseudo code followed byalgorithm.

Step I : Pseudo code to compute a^n by left to right binary exponentiation method

// An array A of size s with binary string equal to exponent n, where s is length of binary string n

1. Set result=a
2. seti=s-2
3. compute result = result *result
4. if A[i] = 1 then compute result = result *a
5. i=i-1 and if i is less than equal to 0 then go to step4.
6. return computed value asresult.

Step II : Algorithm to compute a^n by left to right binary exponentiation method is as follows:

Input: a^n and binary string of length s for exponent n as an array A[s]

Output: Final value of a^n.

1. result=a
2. for i=s-2 to0
3. result = result *result
4. if A[i]= 1then
5. result= result *a
6. return result (i.e a^n)

Let us take an example to illustrate the above algorithm to compute a^{17}
In this exponent n=17 which is equivalent to binary string 10001
Step by step illustration of the left to right binary exponentiation algorithm for a^{17} :
s=5

result=a

Iteration 1:

i=3

result=a *a= a^2
A[3] ≠ 1

Iteration 2:

i=2
result= a^2 * a^2 = a^4
A[2] ≠ 1

Iteration 3:

i=1
result= a^4 * a^4 = a^8
A[1] ≠ 1

Iteration 4:

i=0
result= a^8 * a^8 = a^{16}
A[0] = 1
result = a^{16} * a = a^{17}

return a^{17}

In this example total number of multiplication is 5 instead of 16 multiplications in brute force algorithm i.e n-1

Complexity analysis: This algorithm performs either one multiplication or two multiplications in each iteration of a for loop in line no. 2 of the algorithm.

Hence

Total number of multiplications in the algorithm for computing a^n will be in the range of $s-1 \leq f(n) \leq 2(s-1)$ where s is length of the binary string equivalent to exponent n and f is function that represent number of multiplication in terms of exponent n. So complexity of the algorithm will be $O(\log_2 n)$ As n can be representation in binary by using maximum of s bits i.e $n=2^s$ which further implies $s= O(\log_2 n)$

Right to left binaryexponentiation

In right to left binary exponentiation to compute a^n , processing of bits will start from least significant bit to most significant bit.

Step I : Pseudo code to compute a^n by right to left binary exponentiation method

// An array A of size s with binary string equal to exponent n, where s is length of binary string n

1. Set x =a
2. if A[0]= 1 then set result=a
3. else set result=1
4. Initializei=1
5. compute x = x *x
6. if A[i] = 1 then compute result = result *x
7. Increment i by 1 as i=i+1 and if i is less than equal to s-1 then go to step4.

8. return computed value asresult.

Step II : Algorithm to compute a^n by right to left binary exponentiation method algorithm is as follows:

Input: a^n and binary string of length s for exponent n as an array

A[s] Output: Final value of a^n.

```
1.  x=a
2.  if A[0]=1then
3.       result =a
4.  else
5.       result=1
6.  for i= 1 tos-1
7.      x= x * x
8.      ifA[i]=1
9.           result= result *x
10. return result (i.e a^n)
```

Let us take an example to illustrate the above algorithm to compute a^{17}In this exponent n=17 which is equivalent to binary string 10001

Step by step illustration of the right to left binary exponentiation algorithm for a^{17} :
s=5, the length of binary string of 1's and 0's for exponent n

Since A[0] =1 , result=a

Iteration 1

i=1
x=a *a=
a^2
A[1] ≠
1

Iteration 2

i=2
x= $a^2 * a^2 = a^4$
A[2] ≠ 1

Iteration 3

i=3
x= $a^4 * a^4 = a^8$
A[3] ≠ 1

Iteration 4

i=4
x= $a^8 * a^8 = a^{16}$
A[4] = 1
result = result * x = $a * a^{16} = a^{17}$

return a^{17}

In this example total number of multiplication is 5 instead of 16 multiplications in brute force algorithm i.e n-1

Complexity analysis: This algorithm performs either one multiplication or two multiplications ineachiteration of for loop as shown in line no. 6.
Hence

Total number of multiplications in the algorithm for computing a^n will be in the range of $s-1 \leq f(n) \leq 2(s-1)$ where s is length of the binary string equivalent to exponent n and f is function that represent number of multiplication in terms of exponent n. So complexity of the algorithm will be $O(\log_2 n)$ As n can be representation in binary by using maximum of s bits i.e $n=2^s$ which further implies $s= O(\log_2 n)$

From the above discussion we can conclude that the complexity for left to right binary exponentiation and right to left binary exponentiation is logarithmic in terms of exponent n.

☞Check Your Progress 3

1. Compute a^{283} using left to right and right to left binary exponentiation.

……

……

……

Linear Search

Linear search or sequential search is a very simple search algorithm which is used to search for a particular element in an array. Unlike binary search algorithm, in linear search algorithm,elements are not arranged in a particular order. In this type of search, an element is searched in an array sequentially one by one.If a match is found then that particular item is returned and the search process stops. Otherwise, the search continues till the end of an array. The following steps are used in the linear search algorithm:

Linear_ Search(A[ ], X)
Step 1: Initialize i to 1
Step 2: if i exceeds the end of an array then print “element not found” and Exit
Step 3: if A[i] = X then Print “Element X Found at index i in the array”and Exit
Step 4: Increment i and go to Step 2

We are given with a list of items. The following table shows a data set for linear search:

7	17	3	9	25	18

In the above table of data set, start at the first item/element in the list and compared with the key. If the key is not at the first position, then we move

from the current item to next item in the list sequentially until we either find what we are looking for or run out of items i.e the whole list of items is exhausted. If we run out of items or the list is exhausted, we can conclude that the item we were searching from the list is not present.

The key to be searched=25 from the given data set

In the given data set key 25 is compared with first element i.e7 , they are not equal then move to next element in the list and key is again compared with 17 , key 25 is not equal to 17. Like this key is compared with element in the list till either element is found in the list or not found till end of the list. In this case key element is found in the list and search is successful.

Let us write the algorithm for the linear search process first and then analyze its complexity.

```
// a is the list of n elements, key is an element to be searched in the list function
linear_search(a,n,key)
{
        found=false // found is a boolean variable which will store either true or
false
        for(i=0;i<n;i++)
        {
                if (a[i]==key)
                        found = true
                        break;
        }
        if (i==n)
                found =
false
        return found
}
```

For the complexity analysis of this algorithm, we will discuss the following cases:

a. best case timeanalysis
b. worst-case timeanalysis
c. average case timeanalysis

To analyze searching algorithms, we need to decide on a basic unit of computation. This is the common step that must be repeated in order to solve the problem. For searching, comparison operation is the key operation in the algorithm so it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. If the item is not in the list, the only way to know it is to compare it against every item present.

Best Case: The best case - we will find the key in the first place we look, at the

beginning of the list i.e the first comparison returns a match or return found as true. In this case we only require a single comparison and complexity will be O(1).

Worst Case: In worst case either we will find the key at the end of the list or we may not find the key until the very last comparison i.e*nth* comparison. Since the search requires n comparisons in the worst case, complexity will be O(n).

Average Case: On average, we will find the key about halfway into the list; that is, we will compare against *n*/2 data items. However, that as *n* gets larger, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the linear search, is *O*(*n*). The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility.

If the performance of the system is crucial, i.e. it's part of a life-critical system, and then we must use the worst case in our design calculations and complexity analysis as it tends to the best guaranteed performance.

The following table summarizes the above discussed results.

Case	Best Case	Worst Case	Average Case
item is present	O(1)	O(n)	O(n/2) = O(n)
item is not present	O(n)	O(n)	O(n)

However, we will generally be most interested in the worst-case time calculations as worst-case times can lead to guaranteed performance predictions.

Most of the times an algorithm run for the longest period of time as defined in worst case. Information provide by best case is not very useful. In average case, it is difficult to determine probability of occurrence of input data set. Worst case provides an upper bound on performance i.e the algorithm will never take more time than computed in worse case. So, the worst-case time analysis is easier to compute and is useful than average time case.

3.4.4 SORTING

Sorting is the process of arranging a collection of data into either ascending or descending order. Generally the output is arranged in sorted order so that it can be easily interpreted. Sometimes sorting at the initial stages increases the performances of an algorithm while solving a problem.

Sorting techniques are broadly classified into two categories:

- Internal Sort: - Internal sorts are the sorting algorithms in which the complete data set to be sorted is available in the computer's main memory.

- External Sort: - External sorting techniques are used when the collection of complete data cannot reside in the main memory but must reside in secondary storage for example on a disk.

In this section we will discuss only internal sorting algorithms. Some of the internal sorting algorithms are bubble sort, insertion sort and selection sort. For any sorting algorithm important factors that contribute to measure their efficiency are the size of the data set and the method/operation to move the different elements around or exchange the elements. So counting the number of comparisons and the number of exchanges made by an algorithm provides useful performance measures. When sorting large set of data, the number of exchanges made may be the principal performance criterion, since exchanging two records will involve a lot of time.

Bubble Sort

It is the simplest sorting algorithm in which each pair of adjacent elements is compared and exchanged if they are not in order. This algorithm is not recommended for use for a bigger size array because its average and worst case complexity are of $O(n^2)$ where **n** is the number of elements in an array.This algorithm is known as **bubble sort**, because the largest element in the given unsorted array, bubbles up towards the last place in every cycle/pass .

. A list of numbers is given as input that needs to be sorted. Let us explain the process of sorting via bubble sort with the help of following Tables:

First Pass

23	18	15	37	8	11
18	23	15	37	8	11
18	15	23	37	8	11
18	15	23	37	8	11
18	15	23	8	37	11
18	15	23	8	11	**37**

Second Pass

18	15	23	8	11	**37**
15	18	23	8	11	**37**
15	18	23	8	11	**37**
15	18	8	23	11	**37**
15	18	8	11	**23**	**37**

15	18	8	11	**23**	**37**
15	18	8	11	**23**	**37**
15	8	18	11	**23**	**37**
15	8	11	**18**	**23**	**37**

Third Pass

15	8	11	**18**	**23**	**37**

8	15	11	**18**	**23**	**37**
8	11	**15**	**18**	**23**	**37**
Fourth Pass					
8	11	**15**	**18**	**23**	**37**
8	**11**	**15**	**18**	**23**	**37**
Fifth Pass					
8	**11**	**15**	**18**	**23**	**37**

In this the given list is divided into two sub list sorted and unsorted. The largest element is bubbled from the unsorted list to the sorted sub list. After each iteration/pass size of unsorted keep on decreasing and size of sorted sub list gets on increasing till all element of the list comes in the sorted list. With the list of n elements, n-1 pass/iteration are required to sort. Let us discuss the result of iteration shown in above tables.

In pass 1, first and second element of the data set i.e 23 and 18 are compared and as 23 is greater than 18 so they are swapped. Then second and third element will be compared i.e 23 and 15, again 23 is greater than 15 so swapped. Now 23 and 37 is compared and 23 is less than 37 so no swapping take place. Then 37 and 8 is compared and 37 is greater than 8 so swapping take place. At the end 37 is compared with 11 and again swapped. As a result largest element of the given data set i.e 37 is bubbled at the last position in the array. At each pass the largest element among the remaining elements in the unsorted array bubbles up towards the sorted part of the array as shown in the table above. This process will continue till n-1 passes.
The first version of the algorithm for above sorting method is as below:
Bubble Sort Algorithm- Version1

```
   // A is the list of n elements to be
   sorted function bubblesort (A,n)
{

        int i,j
          for ( i= 1 to n-1
            for (j = 0 to n-2
                        {
                                if (A[j]>A[j+1])
                                {
                        // swapping of two adjacent elements of an array A
                        exchange (A[j], A[j+1])

                                }
                         }
        }
```

Let us do complexity analysis of bubble sort algorithm:

We will perform the **worst case analysis** of the algorithm. Assume that in the worst case scenario the exchange operation inside the loop will take constant amount of time (O(1)). There are two loops in the algorithm. Both the outer and inner loops will execute n-1 times. Therefore the total running time T(n)

will be:

T(n) = (n-1)(n-1)* C(constant time required for simple statements like exchange)

T(n) = Cn^2 -2Cn + 1

If the worst time (big oh) complexity of the algorithm is expressed in polynomial expression, we consider(i) the highest order in the expression and (ii) do not consider the constant value in the final value. Therefore

T(n)= O(n^2)

Let us reanalyze the above algorithm to improve the running time algorithm further. From the example it is visible that the Bubble sort algorithm divides the array into unsorted and sorted sub-arrays. The inner loop rescans the sorted sub- array in each cycle, although there will not be any exchange of adjacent elements. The modified version (version 2) of the algorithm overcomes this problem:

Version -2

```
function Bubble Sort(A,n)

{

int i,j
        for ( i= 1 to n-1)
          for (j = 0 to n-i-1)
                    {
                              if(A[j]>A[j+1])
                              {
                    // swapping of two adjacent elements of an array A
                    exchange(A[j], A[j+1])

                              }
                     }
        }
```

There will be no change in the number of iterations i.e. n-2 iterations in the first pass, but in the second pass it will be n-3, in the third pass it will be n-4 iterations and so on. In this case too, the complexity remains to be O(n^2) but the number of exchange operations will be less. This requires further improvement of the algorithm.

In some cases there is no need of running n-1 passes in the outer loop. The array might be sorted in less than that. The following is the modified version (Version -3) of the algorithm

```
function Bubble Sort(A,n)

{

int i,j
        for ( i= 1 to n-1)
{
        flag = 0;
          for (j = 0 to n-i-1)
                {
                        if(A[j]>A[j+1])
                        {
                // swapping of two adjacent elements of an array A
                exchange( A[j], A[j+1])
                  flag = 1;
                        }
        if (flag = = 0)
exit;
}
        }
```

In case there is no swapping, flag will remain set to 0 and the algorithm will stop running.

Time Complexity

In the modified algorithm, the inner loop will execute at least once to verify that the array is sorted but not (n-i-1) times.Therefore the time complexity will be:

$T(n) = C^* (n-1)$

$= O(n)$

3.5 SUMMARY

In this unit after making a brief review of asymptotic notations, complexity analysis of simple algorithms inillustrated simple summation, matrix multiplication, polynomial evaluation, searching and sorting. Horner's rule is discussed to evaluate the polynomial and its complexity is O(n). Basic matrix multiplication is explained for finding product of two matrices of order nxn with time complexity in the order of $O(n^3)$. For exponent evaluation both approaches i.e left to right binary exponentiation and right to left binary

exponentiation is illustrated. Time complexity of these algorithms to compute x^n is O(log n).

Different versions of bubble sort algorithm are presented and its performance analysis is done at the end.

3.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1. A polynomial is an expression that contains more than two terms. A term comprises of a coefficient and an exponent.
   Example: $P(x) = 15x^3+7x^2+9x+7$

   general form of a polynomial of degree n is

   $P(x)=a_nx^n+a_{n-1}x^{n-1}+.....+a_1x^1+a_0x^0$

2. Show the steps of Horner's rule for $p(x)= 3x_4+2x_3-5x+7$ at x=2
   poly=0 , array a[5]={7,-5,0,2,3}

   Iteration 1,
   poly = x * 0 + a[4] = 3

   Iteration 2,
   poly = x * 3 + a[3] = 2 * 3 + 2 = 6 +2 = 8

   Iteration 3,
   poly = x * 8 + a[2]
   = 2 * 8 + 0 = 16 + 0 = 16

   Iteration 4,
   poly = x * 16 + a[1]
   = 2 * 16 + (-5) = 32 -5 = 27

   Iteration 5,
   poly = x * 27 + a[0]
   = 2 * 27 + 7 = 54 + 7 = 61

3. **A basic (general) algorithm:**

   /* a is an array with polynomial coefficient, n is degree of polynomial, x is the point at which polynomial will be evaluated */

   function(a[n], n, x)

   {

   poly = 0;

   for (i=0; i<= n; i++)

   {

```
                result =1;
                for (j=0; j<i; j++)

                {
                        result= result * x;
                }

                poly= poly + result *a[i];

        }
        return poly.
}
```

Time Complexity of above basic algorithm is O(n^2) where n is the degree of the polynomial. Time complexity of the Horner's rule algorithm is O(n) for a polynomial of degree n. Basic algorithm is inefficient algorithm in comparison to Horner's rule method for evaluating a polynomial.

Check Your Progress 2

1. C program to find two matrices A[3x3] and B[3x3]

```
#include<stdio.h>
int main()

{
int a[3][3],b[3][3],c[3][3],i,j,k,sum=0;

printf("\nEnter the First matrix->");
for(i=0;i<3;i++)
        for(j=0;j<3;j++)
                scanf("%d",&a[i][j]);
printf("\nEnter the Second matrix->");
for(i=0;i<3;i++)
        for(j=0;j<3;j++)
                scanf("%d",&b[i][j]);
printf("\nThe First matrix is\n");
for(i=0;i<3;i++)
{
        printf("\n");
        for(j=0;j<3;j++)
        {
                printf("%d\t",a[i][j]);
        }
}
printf("\nThe Second matrix is\n");
for(i=0;i<3;i++)


{
        printf("\n");
        for (j=0;j<3;j++)
                printf("%d\t",b[i][j]);
}
```

```
for(i=0;i<3;i++)
        for(j=0;j<3;j++)
                c[i][j]=0;

for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
        {
        sum=0;
                for(k=0;k<3;k++)
                        sum=sum+a[i][k]*b[k][j];
                c[i][j]=sum;
        }
}
printf("\nThe multiplication of two matrix is\n");
for(i=0;i<3;i++)
{
        printf("\n");
        for(j=0;j<3;j++)
                printf("%d\t",c[i][j]);
}
return 0;
}
```

Check Your Progress 3

1. Left to right binary exponentiation for a^{283} is as follows:
n=283, binary equivalent to binary string 100011011, s=9 (length of binary string)
result = a

Iteration no.	i	Bit	result
1	7	0	a2
2	6	0	a4
3	5	0	a8
4	4	1	$(a_8)_2$*a= a_{17}
5	3	1	$(a_{17})_2$*a= a_{35}
6	2	0	$(a_{35})_2$= a_{70}
7	1	1	$(a_{70})_2$*a= a_{141}
8	0	1	$(a_{141})_2$*a= a_{283}

Right to left binary exponentiation for $a^{283}$is as follows: n=283, binary equivalent to binary string 100011011, s=9 (length of binary string)
result = a (since A[0]=1)

Iteration no.	i	Bit	x	result
1	1	1	a2	a *a_2 = a_3
2	2	0	a4	a3
3	3	1	a8	a3 * a8= a11
4	4	1	(a8)2	a16 * a11= a27
5	5	0	(a16)2	(a27)
6	6	0	(a32)2	a27
7	7	0	(a64)2	a27
8	8	1	(a128)2	(a256)* a27= a283