Chapter 1

Introduction to Data Structures and Algorithms

1.1 INTRODUCTION TO DATA REPRESENTATION

A computer is a machine that manipulates data. The prime aim of data structures includes the study of how data is organized in a computer, how it can be manipulated, how it can be retrieved, and how it can be utilized.

The basic unit of memory is the bit whose value is either 1 or 0. A single bit can represent the boolean state of result, fail or pass. If we need to represent more than two states in the information then it would require more than 1-bit to represent the information. The data may be numerical data or character strings data or mixed of both.

A high-level programming language defined several data types to represent all types of data in the memory. The memory of a computer is represented by collection of memory addresses. This address is usually numeric and represented in hexadecimal numbers, but for simplicity we shall represent memory addresses in decimal numbers. An address is often called a location, and the contents of the location are the values of data. A location content may be 1-byte, 2-byte, 4-byte or 8-byte long. Every computer has a set of “native” data types and implementation of their operations such as arithmetic operations, addition, subtraction and relational operations. For example, addition of two binary integers is implemented using full adder or half adder logic circuit.

Data Type

A data type is a collection of values (e.g. numeric, character strings) and a set of operations on those values. This mathematical construct may be implemented through hardware or software. But first of all this mathematical construct must be stored in memory locations and then manipulate the required operations and place the result in some memory locations. We can define data type as

Data Type = Permitted Data Values + Operations

The computer implement operations on binary values. Example is addition of two binary numbers. We can use more effective software implementation on numeric and character strings values through high-level languages such as ‘C’.

‘C’ language supports different types of data, each of which may be represented differently within the computer’s memory. Basic or automatic or primitive data types are listed below and their typical memory requirements are also given. Note that the memory requirements for each data type may vary from one C compiler to another.
### Data Type Description Memory Requirements

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Memory Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer number</td>
<td>2-byte</td>
</tr>
<tr>
<td>char</td>
<td>single character</td>
<td>1-byte</td>
</tr>
<tr>
<td>float</td>
<td>floating-point number (i.e., a number containing a decimal point and/or an exponent)</td>
<td>4-byte</td>
</tr>
<tr>
<td>double</td>
<td>double-precision floating-point number (i.e., exponent which may be larger in magnitude)</td>
<td>8-byte</td>
</tr>
</tbody>
</table>

Other data types can be built from one or more primitive data types. The **built-in** data types are enumerated, array, pointer, structure and union. The new structured data types can be built from these primitives and built-in data types are linked lists, stacks, queues, trees and graphs and many more. Sometime data types are called data structures.

Consider the following ‘C’ language declaration

```c
int a, b, c;
float x, y, z;
```

This declaration reserved six memory locations for six different values. These six locations may be referenced by the identifiers `a`, `b`, `c`, `x`, `y`, and `z`. The contents of the locations reserved for `a`, `b`, and `c` will be interpreted as integers, whereas the contents of the locations reserved for `x`, `y`, and `z` will be interpreted as floating-point numbers. The values of these identifiers are within the allowed range.

Now consider the addition operation on these values declared as

```c
c = a + b;
z = x + y;
```

The compiler that is responsible for translating ‘C’ programs into machine language will translate the “+” operator in the first statement into integer addition, and place the result into identifier `c`.

Similarly the “+” operator in the second statement into floating-point addition and place the result into identifier `z`.

The data types and their operations defined in ‘C’ language are useful in problem solving. Each programming language has some limitation in representing their values. For example in ‘C’ primitive data types can only take the range of values according to their memory requirements. It is impossible in ‘C’ to represent arbitrarily large integers on a computer, since the size of such a machine’s memory is finite.

The data types and their operations are implemented through hardware and software data structures. The efficiency of implementation is measured in terms of time and memory space requirements. If a particular application has heavy data processing task, then the speed at which those operations can be performed will be the major factor to evaluate performance. Similarly, if application uses large numbers of structures then the memory space to represent these data structures is the major factor in the efficiency criteria.

### 1.2 REVIEW OF DATA STRUCTURES—ARRAY, POINTER, STRUCTURE, LISTS, TREES AND GRAPHS

The data structure is a collection of data elements organized in a specified manner in computer’s memory (or sometimes on a disk) and there exists an efficient method to store and retrieve individual data elements. Algorithms manipulate the data in these structures in various ways, such as inserting a new data element, searching for a particular element, or sorting the elements. Generally, the data element is of specific data types. We may, therefore, say data are represented that
Data Structure = Organized Data + Allowed Operations
This is an extension of the concept of data type.
Designing and using data structures is an important programming skill. The data structures can be classified in three ways.

Data Structures

- Linear (Array)
- Homogenous (Array)
- Non-linear (Tree)
- Non-homogenous (Structure/Record)

1. **Linear and Non-linear data structures**: In linear data structures the data elements are arranged in a linear sequence like in an array, data processed one by one sequentially. Linear data structures contain following types of data structures:
   - Arrays
   - Linked lists
   - Stacks
   - Queues

   In non-linear data structures, the data elements are not in sequence that means insertion and deletion are not possible in a linear manner. Non-linear data structures contain following types of data structures:
   - Tree
   - Graph

2. **Homogenous and Non-homogenous data structures**: In homogenous data structures the data elements are of same type like an array. In non-homogenous data structures, the data elements may not be of same type like structure in ‘C’.

3. **Static and Dynamic data structures**: Static structures are ones whose sizes and structures associated memory location are fixed at compile time e.g. array. Dynamic structures are ones that expand or shrink as required during the program execution and their associated memory location change, e.g. linked list.

**Data Structure Operations**

Algorithms manipulate the data that is represented by different data structures using various operations. The following data structure operations play major role in the processing of data:

- **Creating**: This is the first operation to create a data structure. This is just declaration and initialization of the data structure and reserved memory locations for data elements.
- **Inserting**: Adding a new data element to the data structure.
- **Updating**: It changes data values of the data structure.
- **Deleting**: Remove a data element from the data structure.
- **Traversing**: The most frequently used operation associated with data structure is to access data elements within a data structure. This form of operation is called traversing the data structure or visiting these elements once.
- **Searching**: To find the location of the data element with a given value, find the locations of all elements which satisfy one or more conditions in the data structure.
- **Sorting**: Arranging the data elements in some logical order e.g. in ascending or descending order of students’ name.
Merging. Combine the data elements in two different sorted sets into a single sorted set.

Destroying. This must be the last operation of the data structure and apply this operation when no longer needs of the data structure.

Here, we are exploring some real world applications to understand requirements of data structure operations.

Consider AddressBook, which holds a student’s name, address and phone number. Choose the appropriate data structures to organized data of AddressBook and then create these structures. The creating of data structure operation stores some initial data in computer’s memory. The following operation may require in the AddressBook:

- To insert new students’ name, address and phone number in the AddressBook
- To delete old students’ name, address and phone number in the AddressBook
- To update existing students’ phone number
- To traverse all students’ address for mailing
- To search particular phone number of the student
- To arrange AddressBook by student’s name in ascending order

In a similar manner airlines/trains reservation system that stores passenger and flight/train information may require several data structures and their operations.

The complete details of these operations are given with each and every data structures in subsequent chapters.

1.3 WHAT IS AN ALGORITHM?

Definition
An algorithm is a finite set of instructions that takes some raw data as input and transforms it into refined data. An algorithm is a tool for solving a well-specified computational problem. Every algorithm must satisfy the following criteria:

- **Input:** In every algorithm, there must be zero or more data that are externally supplied.
- **Output:** At least one data is produced.
- **Definiteness:** Each instruction must be clear and unambiguous (i.e., clearly one meaning).
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm will terminate after a finite number of steps.
- **Effectiveness:** Every instruction must be feasible and provides some basis towards the solution.

An algorithm is composed of finite set of steps, each may require one or more operations. An algorithm requires the following key points in reference to data:

- Memory requirement for data.
- Language for describing data manipulation.
- The type of operations, which describes what kinds of refined data can be produced from raw data.
- Data structures for representing data.

In fact, a data structure and an algorithm should be thought of as a unit, neither one making sense without the other.

**Algorithm Specification**
An algorithm can be described in many ways. A natural language such as English can be used to write an algorithm but generally algorithms are written in English-like pseudocode that resembles with high level languages such as ‘C’ and pascal. An algorithm does not follow exactly any programming language, so that
it does not require any translator. Therefore, we can’t run algorithm with the help of compiler/interpreter, just we can dry run to check the results. The following format conventions are used to write an algorithm:

1. **Name of algorithm:** Every algorithm is given an identifying name written in capital letters.

2. **Introductory comment:** The algorithm name is followed by a brief description of the tasks the algorithm performs and any assumptions that have been made. The description gives the name and types of the variables used in the algorithm. Comment begins with // and continues until end of line. Comments specify no action and are included only for clarity.

3. **Steps:** Each algorithm is made of a sequence of numbered steps and each step has an ordered sequence of statements, which describe the tasks to be performed. The statements in each step are executed in a left-to-right order.

4. **Data type:** Data types are assumed simple such as integer, real, char, boolean and other data structures such as array, pointer, structure are used. Array i^{th} element can be described as A[i] and (i, j) element can be described as A[i, j]. Structure data type can be formed as

   ```
   node = record
   { 
     data type-1 identifier 1;
     ...
     data type-n identifier n;
     node *link;
   }
   ```

   link is a pointer to the record type node.

5. **Assignment statement:** The assignment statement is indicated by placing equal (=) between the variable (in left hand side) and expression or value(s) (in right hand side). For example, addition of a and b is assigned in variable a.

   ```
   a = a + b
   ```

6. **Expression:** There are three expressions: arithmetic, relational and logical. The arithmetic expression used arithmetic operator such as /, *, +, –, relational expression used relational operator such as <, <=, >, >=, <> = and logical expression used logical operator such as not, or, and. There are two boolean values, true or false.

7. **If statement:** If statement has one of the following two forms:

   (a) if condition then statement(s)

   (b) if condition then statement(s)

   else statement(s)

   if the condition is true, statement(s) followed then are to be executed, otherwise if condition is false, statement(s) followed else are to be executed. Here we can use multiple statements, if required.

8. **Case statement:** In general case statement has the form:

   Select case (expression)
case value 1: statement (s)
case value 2: statement (s)
.
.
.
case value n: statement (s)
default:

the expression is evaluated then a branch is made to the appropriate case. If the value of the expression does not match that of any case, then a branch is made to the default case.

9. **Looping statements:** These statements are used when there is a need of some statements to be executed number of times. These statements also called iterative control statements.

(a) while (condition) do

    { 
    statement(s) 
    }

As long the condition is true, the statement(s) within while loop are executed. When the condition tests to false then while loop is exited.

(b) for variable = start value to final value step increment value do

    { 
    statement(s) 
    }

This for loop is executed from start value to final value with addition of increment value in start value. In the absence of step clause, assume increment value 1.

(c) repeat

    { 
    statement(s) 
    } until (condition)

This repeat loop is executed until the condition is false, and exited when condition becomes true.

10. **Input and output:** For input of data it used read(variable name) statement and for output of data it used write(variable name) statement. If more than one input or output data then we can use comma as separator among the variable names.

11. **Goto statement:** The goto statement causes unconditional transfer of control to the step referenced. Thus statement goto step N will cause transfer of control to step N.

12. **End statement:** The end statement is used to terminate an algorithm. It is usually the last step and algorithm name is written after the end.

13. **Functions:** A function is used to return single value to the calling function. Transfer of control and returning of the value are accomplished by return(value) statement. A function begins as follows: function function_name (parameters list).

1.4 **DESIGNING ALGORITHMS**

To solve any problem on computer, first of all we have to design efficient algorithm of any computational problems. An algorithm may be an existing procedure, routine, method, process, function or recipe. But we can devise in new one that is more appropriate towards the solution of the problem in respect of memory
and execution time it required. The complete program can be smoothly and perfectly written in the supporting programming language with the help of the algorithm. When designing an algorithm, consider the following key points:

1. Machines that hold data.
2. Language for describing data manipulations.
3. Functions that describe what kinds of refined (e.g. output) data can be produced from raw (e.g. input) data.
4. Structures for representing data.

Effectiveness and efficiency of an algorithm also depends upon the data structures that it can use.

Now we are discussing some common problems and try to learn how to write an algorithm.

**Example 1.** Consider a problem to find the maximum of n numbers and its location in the vector, n>0.

For the simplicity of the algorithm, consider n integer numbers are stored in a vector A (i.e., single dimensional array).

For example n=10 and integer values of A[ ] = { 12, 43, 9, 3, 46, 25, 20, 15, 32, 39 } and large is a variable which holds always maximum number and j points to the location of maximum value element.

Initially assume that the value of large =A[1] has maximum value (i.e., 12). Compare the subsequent elements of vector A[i] with large and if the value of element is greater than large then assign those values to variable large and location to variable j. This method can be best described by below table.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>&lt;12&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>&lt;46&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

j→1 2 2 2 2 5 5 5 5 5 5

large value is enclosed in angle bracket. The maximum value is 46 and 5th element of vector A stores it. Algorithm MAX(A, n, j)

```java
// Set j so that A[j] is the maximum in A[1:n], and here i is integer, and n>0
step 1:  large = A[1]; j = 1;
step 2:  for i = 2 to n do
        {
            if A[i] > large
              then  large = A[i]; j = i;
        }
end MAX
```

Algorithm 1.1 finding the maximum of n numbers
An algorithm can contain three kinds of variables: local, global and formal parameters. A local variable is one, which is declared in the current function (e.g. i is local variable). A global variable is one which has already been declared, as local to a function which contains the current function (e.g. large is global variable). Formal parameters contained in parameter list following the name of the algorithm. At execution time formal parameter are replaced by the actual parameters. In the above example, A, n, j are the formal parameters.

**Example 2. Euclid’s algorithm:** A process for finding the greatest common divisor (GCD) of two numbers. This algorithm is for computing greatest common divisor of two non-negative integers. The essential step which guarantees the validity of this process consists of showing that the greatest common divisor of a and b (a > b ≥ 0) is equal to a if b is zero and is equal to the greatest common divisor of b and remainder of a divided by b if b is non-zero.

For example, to find the greatest common divisor of two non-negative integers 22 and 8 can be computed as below:

\[
gcd(22, 8) \quad \text{where } a=22, b=8 \text{ and } b \text{ is non-zero, so find remainder } r= a \mod b \text{ and assign } a=b, \text{ and } b=r \text{ and compute again.}
\]

\[
gcd(8, 6) \quad \text{where } a=8, b=6 \text{ and } b \text{ is non-zero so find remainder } r= a \mod b \text{ and assign } a=b, \text{ and } b=r \text{ and compute again.}
\]

\[
gcd(6, 2) \quad \text{where } a=6, b=2 \text{ and } b \text{ is non-zero so find remainder } r= a \mod b \text{ and assign } a=b, \text{ and } b=r \text{ and compute again.}
\]

\[
gcd(2, 0) \quad \text{where } a=2, b=0 \text{ and } b \text{ is zero so greatest common divisor is value of } a \text{ that is 2.}
\]

The algorithm can be easily designed on this method.

```
Algorithm GCD(a, b)
// Assume a > b ≥ 0
step 1 : if b = 0
    then return(a);
step 2 : else
    // r is the remainder
    r = a mod b;
step 3 : set a = b; b = r; and go to step1
end GCD
```

Algorithm 1.2 Finding the greatest common divisor

**Example 3.** To design an algorithm for matrix multiplication of two m x n and n x p matrix A and B to form m x p matrix C.

The matrix multiplication of two matrices A and B is possible only when no. of column of matrix A must be equal to no. of row of matrix B.

That means

\[
\text{Matrix C} = \text{Matrix A} * \text{Matrix B}
\]

\[
(\text{of order m x p}) \quad (\text{of order m x n}) \quad (\text{of order n x p})
\]

\[
m\text{- rows, p- columns} \quad m\text{- rows, n- columns} \quad n\text{- rows, p- columns}
\]

The matrix multiplication can be carried by the following formula:
\[ j = n \]
\[ C_{ik} = \sum_{j=1}^{A_{ij} * B_{jk}} \text{for } 1 \leq i \leq m, \text{ and } 1 \leq k \leq p \]

where

- \( A_{ij} \) = Elements of Matrix A
- \( B_{jk} \) = Elements of Matrix B
- \( C_{ik} \) = Elements of resultant Matrix C

Consider an example to multiplication of matrix \( A_{2,3} \) and matrix \( B_{3,2} \). Here the value of \( m = 2, n = 3 \) and \( p = 2 \), the resultant matrix \( C \) must be order of \( 2 \times 2 \) and matrices data are stored in two dimensional arrays.

\[
C_{1,1} = A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1} + A_{1,3} * B_{3,1} \\
C_{1,2} = A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2} + A_{1,3} * B_{3,2} \\
C_{2,1} = A_{2,1} * B_{1,1} + A_{2,2} * B_{2,1} + A_{2,3} * B_{3,1} \\
C_{2,2} = A_{2,1} * B_{1,2} + A_{2,2} * B_{2,2} + A_{2,3} * B_{3,2}
\]

Algorithm MAT\_MULTIPLY

// Matrix A order is \( m \times n \) and Matrix B order is \( n \times p \)

**Step 1**: if number of columns of matrix A = number of rows of matrix B

for \( i = 1 \) to \( m \) do

for \( k = 1 \) to \( p \) do

\[
C[i, k] = 0; \\
f for \( j = 1 \) to \( n \) do \\
C[i, k] = C[i, k] + A[i, j] * B[j, k];
\]

**Step 2**: else

write(“ Matrix multiplication is not possible and exit”);

end MAT\_MULTIPLY

1.5 ANALYZING ALGORITHMS

Why is it necessary to analyze an algorithm? An algorithm analysis measures the efficiency of the algorithm. The efficiency of an algorithm can be checked by:

1. The correctness of an algorithm.
2. The implementation of an algorithm.
3. The simplicity of an algorithm.
4. The execution time and memory space requirements of an algorithm.
5. The new ways of doing the same task even faster.

The analysis emphasizes timing analysis and then, to a lesser extent, spaces analysis. It is very difficult to calculate how fast a problem can be solved. It requires an algorithm execution machine. An algorithm can be executed only of formal models of machines e.g. Turing Machine and Random Access Machine. An algorithm analysis does not depend on computers and programming languages but the program that is coded using the algorithm depends on both. Therefore, programs written in different programming languages using the same algorithm and same computer have different time and space requirements.
In an algorithm analysis, instructions of an algorithm are assumed to be executed one at a time, and so major time of an algorithm depends upon the number of operations it requires.

Given an algorithm to be analyzed, the first task is to determine which operations are involved and what their space and timing requirements. These operations may be arithmetic, comparisons, and assigning values to variables and executing function calls. For the simplicity of analysis, consider that all operations take a fixed amount of time and so their time is bounded by a constant.

An algorithm analysis must cover the sufficient number of data sets that cause the algorithm to exhibit all possible patterns of behaviour.

This requires each statement frequency counts (i.e., the number of times the statement will be executed) and the time for one execution. The total time required to execute that algorithm is the sum of product of these two numbers. But the execution time for each statement depends on both the computer and the programming language.

Here we are determining only frequency counts of each statement of the following codes:

\[ x = x + y \]  
\[ \text{for } i = 1 \text{ to } n \text{ do} \]
\[ x = x + y \]  
\[ \text{for } j = 1 \text{ to } n \text{ do} \]
\[ x = x + y \]

(a)  (b)  (c)

For each code a statement \( x = x + y \) is executed. In code (a), the frequency count is 1. For code (b) the count is \( n \) and for code (c) the count is \( n^2 \). These are in increasing order of magnitude. The order of magnitude of an algorithm refers to the sum of the frequency counts of all statements of an algorithm. There are several kinds of mathematical notation to represent order of magnitude. One of these is the \( \mathcal{O} \) (big oh) notation.

The time needed by an algorithm is expressed as a function of the size of a problem is called the time complexity of the algorithm. If an algorithm processes input of size \( n \) in time \( cn^2 \) for some constant \( c \), the time complexity of the algorithm is \( \mathcal{O}(n^2) \), read as ‘order of \( n^2 \)’. The time complexity of an algorithm can be defined in terms of two functions. The function \( f(n) \) computes the actual time to execute an algorithm and function \( g(n) \) represents the time required by the algorithm.

The notation \( f(n) = \mathcal{O}(g(n)) \) (read as \( f(n) \) equals big-oh of \( g(n) \)) has a precise mathematical definition, \( f(n) = \mathcal{O}(g(n)) \) if and only if there exist two constants \( c \) and \( n_0 \) such that \( |f(n)| \leq cg(n) \) for all \( n \geq n_0 \).

The \( \mathcal{O}(g(n)) \) means that if an algorithm is run on same computer on same type of data but for increasing values of \( n \), the resulting times will always be less than some constant time \( g(n) \).

The computing time \( \mathcal{O}(1) \) is called constant, \( \mathcal{O}(\log_2 n) \) is logarithmic, \( \mathcal{O}(n) \) is called linear, \( \mathcal{O}(n^2) \) is called quadratic, \( \mathcal{O}(n^3) \) is called cubic, and \( \mathcal{O}(2^n) \), \( \mathcal{O}(n!) \) and \( \mathcal{O}(n^n) \) are called exponential. If an algorithm takes \( \mathcal{O}(\log_2 n) \) time, it is faster for sufficiently large \( n \), than if it had taken \( \mathcal{O}(n) \) time. The order of computing times for algorithms are \( \mathcal{O}(1) \leq \mathcal{O}(\log_2 n) \leq \mathcal{O}(n) \leq \mathcal{O}(n \log_2 n) \leq \mathcal{O}(n^2) \leq \mathcal{O}(n^3) \leq \mathcal{O}(2^n) \leq \mathcal{O}(n!) \leq \mathcal{O}(n^n) \) for sufficiently large \( n \).

If we have two algorithms which perform the same task, and first has a computing time \( \mathcal{O}(n) \) and the second one has \( \mathcal{O}(n \log_2 n) \), then first one is preferred to second one.

To clarify some of these ideas, let us consider an example to compute the \( n \)th Fibonacci number. The Fibonacci series starts as

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots \]

Each new number is obtained by taking the sum of the two previous numbers in the series. The first number is \( F_0 = 0 \) and second number is \( F_1 = 1 \) and in general

\[ F_n = F_{n-1} + F_{n-2} \]

The algorithm Fibonacci inputs any non-negative integer \( n \) and computes the \( n \)th number in the Fibonacci series i.e., \( F_n \).
Algorithm FIBONACCI
// compute the Fibonacci number \( F_n \)

**Step 1**: read(n)

**Step 2**: if (n ≤ 1) then
write(n) else

**Step 3**: \( f_0 = 0; f_1 = 1; \)

**Step 4**: for i = 2 to n do

\[
\begin{align*}
& f_n = f_0 + f_1; \\
& f_0 = f_1; \\
& f_1 = f_n;
\end{align*}
\]

**Step 5**: write(\( f_n \));

**Step 6**: end FIBONACCI

To analyze the time complexity of this algorithm, we need to consider two cases: (i) \( n = 0 \) or \( 1 \), and (ii) \( n > 1 \). We shall count the total number of statement executions and use this as a measure of the time complexity of the algorithm.

When \( n = 0 \) or \( 1 \), step 1, step 2 and step 6 get executed once. The step 2 has two statements. Therefore total statement count for this case is 4 (i.e., 1 + 2 + 1 respectively).

When \( n > 1 \), step 3 has two assignment statements and get executed once. For loop of the step 4 gets executed \( n \) times while the statements in the loop get executed \( n - 1 \) times each. The contributions of each step are summarized below for \( n > 1 \).

<table>
<thead>
<tr>
<th>Step</th>
<th>Frequency count by the statement</th>
<th>Total count for the step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Frequency count by the statement</td>
<td>Total count for the step</td>
</tr>
<tr>
<td>2</td>
<td>if condition always false for when ( n &gt; 1 ), Thus required ( n - 1 ) counts</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>3</td>
<td>one for each statement</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>( n ) by for loop and ( n - 1 ) by each of three statements.</td>
<td>( 4n - 3 )</td>
</tr>
<tr>
<td>5</td>
<td>1 for write statement and 1 for end of else</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hence the total count when \( n > 1 \) = \( 5n + 1 \)
And total count when \( n < = 1 \) = 2
Thus total count for the algorithm is = \( 5n + 3 \)
The order of magnitude of the algorithm can be written as \( \text{O}(n) \), ignoring the two constants 5 and 3.
The performance of the different order of algorithms, their time complexity and their function values are computed in the below table.
The rates of growth of functions are shown in Fig. 1.1. An algorithm that is exponential will work only for very small inputs. The time complexity of order $O(\log_2 n)$, $O(n)$ and $O(n \log_2 n)$ grows much more slowly than the other three. For the large data sets (i.e., value of $n$ is sufficiently large), the algorithms with a complexity greater than $O(n \log_2 n)$ are often impractical. For small $n$, the exponential function is less time consuming than $O(n^2)$ and $O(n^3)$. Up to $n=4$, $O(2^n)$ function has less or equal time required as by $O(n^2)$ and up to $n=8$, $O(2^n)$ function has less time required as by $O(n^3)$.

### 1.6 MATHEMATICAL NOTATION AND FUNCTIONS

**Summation symbol:** Here we introduce the summation symbol $\Sigma$ (Sigma). Consider a sequence of $n$-terms $a_1, a_2, \ldots, a_n$, then the sums $a_1 + a_2 + \ldots + a_n$ will be denoted as

\[ \sum_{i=1}^{n} a_i \]

Examples are sums of $n$ natural numbers, square of $n$ positive integers and many more. We can write an expression as a sum of series or sequence. We can also call summation of summation.

<table>
<thead>
<tr>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1024</td>
<td>32768</td>
<td>4,294,967,296</td>
</tr>
</tbody>
</table>

![Plot of function values](image)
Consider that a function \( f(n) \) denotes the summation of \( n \) positive integers. Here function \( f(n) \) computes the order of magnitude of an algorithm.

\[
    f(n) = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n
\]

\( 1 \leq i \leq n \)

\( = n(n+1)/2, \) and the order of complexity is \( O(n^2) \)

and in similar fashion the summation of square of \( n \) positive integers can be expressed as:

\[
    f(n) = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2
\]

\( 1 \leq i \leq n \)

\( = n(n+1)(2n+1)/6, \) and the order of complexity is \( O(n^3) \)

Now we want to prove that \( f(n) \) is true for all positive numbers \( n. \) A mathematical induction does this in following manners.

(a) Give a proof that \( f(1) \) is true.

(b) Give a proof that “If all of \( f(1), f(2), \ldots, f(n) \) are true, then \( f(n+1) \) is also true”.

This proof should be valid for any positive integer \( n. \)

**Example 1.** Let us consider an example: summation of \( n \) odd integer numbers in the following series:

\[
    1 + 3 + \ldots + (2n - 1) = n^2
\]

We can denote as sum of \( n \) odd integers

\[
    \text{Sum} = \sum_{i=1}^{n} i = n^2
\]

\( 1 \leq i \leq 2n, \) where \( i \) is odd integer

Now consider the following cases:

For \( n = 1, \) \( \text{sum} = 1 \) i.e., \( 1^2 \)

For \( n = 2, \) \( \text{sum} = 1 + 3 \) i.e., \( 2^2 \)

For \( n = 3, \) \( \text{sum} = 1 + 3 + 5 \) i.e., \( 3^2 \)

It seems that the sum of \( n \) odd numbers is \( n \)-square. We can proof it by mathematical induction.

Given a positive number \( n, \) following procedure is outlined:

(a) \( f(1) \) is true, since \( 1 = 1^2 \)

(b) If all of \( f(1), f(2), \ldots, f(n) \) are true, then in particular \( f(n) \) is true, so the series holds, addition \( 2n+1 \) to both sides we obtained:

\[
    1 + 3 + \ldots + (2n - 1) + (2n + 1) = n^2 + (2n + 1)
\]

\( = (n + 1)^2 \)

which proves that \( f(n+1) \) is also true. That means sum of \( (n+1) \) odd integers is square \( (n+1) \).

The method of proof is called a proof by mathematical induction.

The time complexity of this algorithm is \( O(n^2) \).

**Example 2.** The sum of a geometric progression series. Assume that \( x \neq 1, n \geq 0. \)

\[
    a + ax + ax^2 + \ldots + ax^n = \sum_{i=0}^{n} ax^i
\]

\( 0 \leq i \leq n \)

\( a + \sum_{i=0}^{n} ax^i \quad (1) \)

\( 1 \leq i \leq n \)

\( a + x \sum_{i=1}^{n} ax^{i-1} \quad (2) \)

\( 1 \leq i \leq n \)

\( a + x \sum_{i=1}^{n} ax^{i-1} \quad (3) \)

\( 0 \leq i \leq n -1 \)

\( a + x \sum_{i=1}^{n} ax^{i-1} - ax^{n+1} \quad (4) \)

\( 0 \leq i \leq n \)

\( a + x \sum_{i=1}^{n} ax^{i-1} - ax^{n+1} \quad (5) \)
Comparing the first relation with the fifth, we have
\[ \sum a x^i = a + x \sum a x^i - a x^{i+1} \]
0 ≤ i ≤ n 0 ≤ i ≤ n
or
\[ \sum a x^i - x \sum a x^i = a - a x^{i+1} \]
0 ≤ i ≤ n 0 ≤ i ≤ n
= (1 - x) \sum a x^i = a - a x^{i+1} (6)
0 ≤ i ≤ n
and thus we obtain the basic formula
\[ \sum a x^i = a (1 - x^{i+1})/(1 - x) \]
for x < 1 (7)
0 ≤ i ≤ n
or
\[ \sum a x^i = a (x^{i+1} - 1)/(x - 1) \]
for x > 1 (7)
0 ≤ i ≤ n

The time complexity of any algorithm whose frequency count is sum of geometric progression is order of O(x^n).

The example of geometric progression series is
\[ 2^1 + 2^2 + 2^3 + \ldots + 2^7 = 2(2^7-1)/(2-1) = 2*127 = 254 \]

Example 3. The sum of an arithmetic progression series. Assume that n ≥ 0.
\[ a + (a + b) + (a + 2b) + \ldots + (a + nb) \]
= \( \sum (a + bi) \)
0 ≤ i ≤ n
(1)
= \( \sum (a + b(n - i)) \)
0 ≤ n−i ≤ n
(2)
= \( \sum (a + bn - bi) \)
0 ≤ i ≤ n
(3)
= \( \sum (2a + bn) - n \sum (a + bi) \)
0 ≤ i ≤ n
(4)
= (n + 1)(2a + bn) - \( \sum (a + bi) \)
0 ≤ i ≤ n
(5)

Since the first sum was simply a sum of (n + 1) terms which did not depend on i. Now by equating the first and fifth expressions and dividing by 2, we obtain
\[ \sum (a + bi) = a(n + 1) + (1/2)bn(n + 1) \]
0 ≤ i ≤ n
(6)
The order of magnitude of this type of algorithms are O(n^2), by ignoring constants a, and b.

Example 4. If function \( f(n) = a_m n^m + \ldots + a_1 n + a_0 \) is a polynomial of degree m then \( f(n) = O(n^m) \).

Proof: Using the definition of function \( f(n) \) and a simple inequality
\[
|f(n)| \leq |a_m| n^m + \ldots + |a_1| n + |a_0| \\
\leq (|a_m| + |a_{m-1}|/n + \ldots + |a_1|/n^m)n^m \\
\leq (|a_m| + \ldots + |a_1|)n^m, n \geq 1
\]
Choosing \( c = |a_m| + \ldots + |a_1| \) and \( n_0 = 1 \), the function immediately follows the order \( O(n^m) \).

The above function says that if we can describe the frequency count of any algorithm by a polynomial such that \( f(n) \), then the time complexity of that algorithm is \( O(n^m) \).
Exponents and Logarithms

Let us consider a is a positive real number, if \( n \) is an integer then \( a^n \) is defined by the familiar rules:

\[
a^0 = 1, \quad a^n = a \cdot a \cdot a \ldots (n \text{ times})
\]

\[
a^{-n} = \frac{1}{a^n} \quad \text{if } n > 0,
\]

\[
a^{-n} = \frac{1}{a^{n+1}} \quad \text{if } n < 0
\]

It is easy to prove by induction that the laws of exponents are valid:

\[
a^{x+y} = a^x \cdot a^y
\]

\[
(a^x)^y = a^{xy}
\]

where \( x \) and \( y \) are integers.

Exponents include all rational numbers by defining, for any rational number \( m/n \)

\[
a^{m/n} = \sqrt[n]{a^m} = (\sqrt[n]{a})^m
\]

For example,

\[
125^{2/3} = (\sqrt[3]{125})^2 = 5^2 = 25
\]

Now consider a positive real number \( y \) is given, can we find a real number \( x \) such that

\[
y = b^x
\]

For example, what is the value of \( x \) in expression \( 25 = 5^x \), the value of \( x \) is definitely 2. But how?

The number \( x \) is calculated as logarithm of \( y \) to base \( b \), and we write this as

\[
x = \log_b y
\]

The other examples are

\[
\log_2 8 = 3, \text{ since } 2^3 = 8
\]

\[
\log_{10} 100 = 2, \text{ since } 10^2 = 100
\]

\[
\log_2 64 = 6, \text{ since } 2^6 = 64
\]

Furthermore for any base \( b \),

\[
\log_b 1 = 0, \text{ since } b^0 = 1
\]

\[
\log_b b = 1, \text{ since } b^1 = b
\]

The logarithm of a negative number and the logarithm of 0 are not defined.

We can view exponential and logarithm as inverse of each other.

\[
f(x) = b^x \quad \text{and} \quad g(x) = \log_b x
\]

The term \( \log x \) shall mean \( \log_2 x \) unless otherwise specified.

From the laws of exponents it follows that:

\[
\log_b (xy) = \log_b x + \log_b y \quad \text{if } x > 0, y > 0
\]

\[
\log_b (c^y) = y \log_b c \quad \text{if } c > 0
\]

\[
\log_b x = \log_b x / \log_b c \quad \text{if } x > 0 \quad \text{e.g } \log_2 10 = \log_{10} 10 / \log_{10} 2
\]

\[
\log_b x/y = \log_b x - \log_b y \quad \text{if } x > 0, y > 0
\]

Product: Product of the positive numbers from 1 to \( n \) is denoted by \( n! \) (read ‘n factorial’).

That is \( n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n \)

It is also to define \( 0! = 0 \)

It can be represented by product of numbers as equivalent to summation.

\[
n! = \prod_{i=1}^{n} i = 1.2.3\ldots n
\]

\[
1 \leq i \leq n
\]

For example \( 4! = 1.2.3.4 \) and \( 5! = 1.2.3.4.5 \)
1.7 ASYMPTOTIC NOTATION (O, Θ, Ω)

The notation we use to describe the asymptotic running time of an algorithm is defined in terms of function.

**O (big oh)-notation**

When we have only an asymptotic upper bound we use O (big-oh)-notation, for a given function \( g(n) \) we denoted by \( O(g(n)) \) the set of functions

\[
O(g(n)) = \{ f(n): \text{there exists positive constant } c \text{ and } n_0 \text{ such that } \ 0 \leq f(n) \leq c g(n) \text{ for all } n > n_0 \}
\]

Figure 1.2(a) depicted the asymptotic O-notation. All values n to the right of \( n_0 \), the value of the function \( f(n) \) is on or below \( g(n) \).

We use O-notation to give upper bound on a function to within a constant factor. It is used to bind worse case running time of an algorithm. The time complexity \( O(n^2) \) bound worse case for insertion sort algorithm.

Thus, we can write the computing time for the following frequency counts of algorithms as follows:

<table>
<thead>
<tr>
<th>Frequency counts (g(n))</th>
<th>Time complexity (f(n) = O(g(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10n^3</td>
<td>O(n^3)</td>
</tr>
<tr>
<td>9n^2 + 4n +1</td>
<td>O(n^2) as 9n^2 + 4n +1 \leq 10n^2 for all n \geq 3 so c=10 and n_0 =3</td>
</tr>
<tr>
<td>n(n + 1)/2</td>
<td>O(n^2) as n^2/2 + n/2 \leq n^2 for all n \geq 2 so c=1 and n_0 =2</td>
</tr>
<tr>
<td>n + log_2 n</td>
<td>O(n) as n + log_2 n \leq 2n for all n \geq 2 so c=2 and n_0 =2</td>
</tr>
<tr>
<td>4*2^n + n^2</td>
<td>O(2^n) as 4<em>2^n + n^2 \leq 5</em>2^n for all n \geq 4 so c=5 and n_0 =4</td>
</tr>
</tbody>
</table>

Here we have considered the higher order frequency counts, remaining lower-order assuming constant.

**Ω (Omega)-notation**

Ω-notation provides an asymptotic lower bound. For a given function \( g(n) \) we denoted by \( Ω(g(n)) \) the set of functions.

\[
Ω(g(n)) = \{ f(n): \text{iff there exists positive constant } c \text{ and } n0 \text{ such that } \ 0 \leq c g(n) \leq f(n) \text{ for all } n > n_0 \}
\]

Figure 1.2 (b) depicted the asymptotic Ω-notation. All values n to the right of \( n_0 \), the value of the function \( f(n) \) is on or above \( g(n) \).

Ω-notation describes a lower bound, when we use it to bound the best case running time of an algorithm. The best case running time of insertion sort is \( Ω(n) \) as all n inputs are sorted.

Thus, we can write the computing time for the following function counts of algorithms as follows:

<table>
<thead>
<tr>
<th>Function counts (g(n))</th>
<th>Upper bound time complexity (f(n) = Ω(g(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10n^3</td>
<td>Ω(n^3)</td>
</tr>
<tr>
<td>9n^2 + 4n +1</td>
<td>Ω(n^2) as 9n^2 + 4n +1 \geq 9n^2 for all n \geq 1 so c = 9 and n_0 = 1</td>
</tr>
<tr>
<td>n(n + 1)/2</td>
<td>Ω(n^2) as n^2/2 + n/2 \geq n^2/2 for all n \geq 1 so c = 1 and n_0 = 1</td>
</tr>
<tr>
<td>n + log_2 n</td>
<td>Ω(n) as n + log_2 n \geq n for all n \geq 2 so c = 1 and n_0 = 2</td>
</tr>
<tr>
<td>4*2^n + n^2</td>
<td>Ω(2^n) as 4<em>2^n + n^2 \geq 4</em>2^n for all n \geq 1 so c = 4 and n_0 = 1</td>
</tr>
</tbody>
</table>

**Θ (Theta)-notation**

Θ-notation provides an asymptotic average bound. For a given function \( g(n) \) we denoted by \( Θ(g(n)) \) the set of functions.

\[
Θ(g(n)) = \{ f(n): \text{iff there exists positive constants } c1, c2 \text{ and } n0 \text{ such that } \ 0 \leq c1 g(n) \leq f(n) \leq c2 g(n) \text{ for all } n > n_0 \}
\]
Figure 1.2(c) depicted the asymptotic $\Theta$-notation. All values $n$ to the right of $n_0$, the value of the function $f(n)$ lie at or above $c_1 g(n)$ and at or below $c_2 g(n)$. Function $g(n)$ is an asymptotically tightly bound for $f(n)$.

$\Theta$-notation can be expressed as between worse case $O$-notation and best case $\Omega$-notation. For insertion sort the complexity may be between $\Omega(n)$ as lower bound to $O(n^2)$ upper bound.

Thus, we can write the computing time for the following function counts of algorithms as follows:

- **Function counts** ($g(n)$)
- **average bound time complexity** ($f(n) = \Theta(g(n))$)
- **$3n + 2$**
  - $\Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$
- **$9n^2 + 4n + 1$**
  - $\Omega(n^2)$ as $9n^2 + 4n + 1 \geq 9n^2$ for all $n \geq 1$ and $9n^2 + 4n + 1 \leq 10n^2$ for all $n \geq 1$ so $c_1 = 9$, $c_2 = 10$ and $n_0 = 1$
- **$n + \log_2 n$**
  - $\Omega(n)$ as $n + \log_2 n \geq n$ for all $n \geq 2$ and $n + \log_2 n \leq 2n$ for all $n \geq 3$ so $c_1 = 1$, $c_2 = 2$ and $n_0 = 3$
- **$4*2^n + n^2$**
  - $\Omega(2^n)$ as $4*2^n + n^2 \geq 4*2^n$ for all $n \geq 1$ and $4*2^n + n^2 \leq 5*2^n$ for all $n \geq 1$ so $c_1 = 4$, $c_2 = 5$ and $n_0 = 1$

**Figure 1.2 Asymptotic notation**

### 1.8 PERFORMANCE MEASUREMENT

Performance measurement is concerned with obtaining the memory (or space) and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as computer on which the algorithm is run.

The run-time complexity of a sequential search, for size of $n$ is $O(n)$ is plotted in Fig. 1.3. So we expect a plot of the times to be straight line.

**Figure 1.3 Time complexity graph for sequential search**