Book

A Simplified Approach to Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala
Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar
Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors

Edition 2014

Introduction

- Among the vast applications of the computer, the major application is to manage and process large amounts of data.
- The knowledge about representing data is the fundamental to study of Computer Science.
- The primary objective behind computer programming should be to do the calculations efficiently and to store data and retrieve data in an optimistic way.
- Some structures need to be devised to store, organize, and search through **data.**
- Thus, the study of structuring the data (hence data structures) and algorithms that manipulate data is the heart of Computer Science.

Data and Information

Data
Information

- The term data has been derived from the word datum (means that is given). Data is plural of datum. The term data refers to the value or simply set of values that are raw and unorganized. Data may be simple, random, and useless until it is organized (so called processed). Data is valuable raw material which can be in different forms such as numbers, words, alphabets, etc.
- When data is processed i.e. organized, structured, and presented in a meaningful context so that it becomes useful for decision making and understanding, it becomes information. In layman language, data and information are interchangeable terms, but technically, no conclusion can be drawn from data. Information play vital role in decision making which is obtained from data. So, indirectly data is an important entity for decision making.

Data Structures

- A data structure is a way of storing the data in computer's memory so that it can be used efficiently.
- Formally, a data structure is a logical/mathematical model of organization of data. The choice of data structure begins with the choice of an Abstract Data Type (ADT) such as array.

Classification of data structures

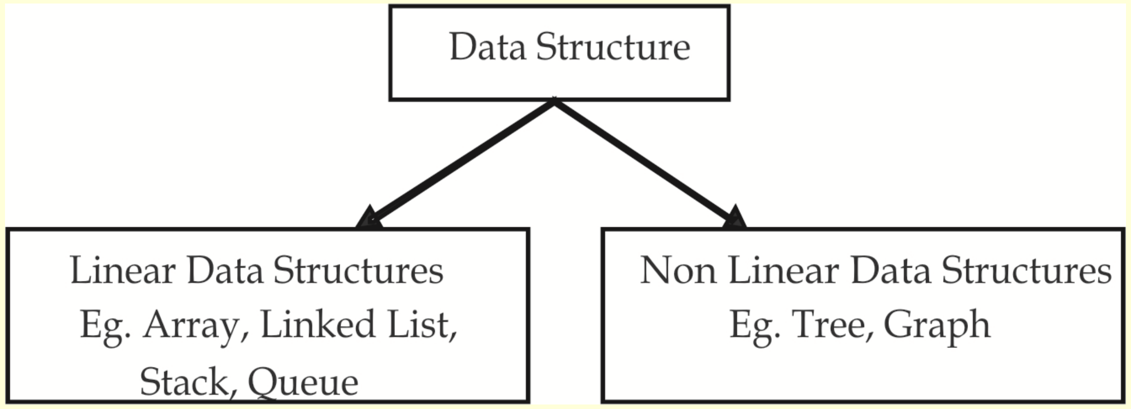
Data structures can be classified in several ways. These classifications are:

- Linear and Non-Linear Data Structures
- Static and Dynamic Data Structures
- Homogeneous and Non-Homogeneous Data Structures

Linear and Non-Linear Data Structures

Linear Data Structure: The elements in a linear data structure form a linear sequence.
Example of the linear data structures are: Arrays, Linked list, Queue, Stack etc.

Non-Linear Data Structure: The elements in a non linear data structure do not form any linear sequence.
For example, Trees and Graphs.



Static and Dynamic Data Structures

Static Data Structure: Static data structures are those whose memory occupation is fixed. The memory taken by these data structures can-not be increased or decreased at run time. Example of the static data structure is an Array. The size of an array is declared at the compile time and this size cannot be changed during the run time.

Dynamic Data Structure: Dynamic data structures are those whose memory occupation is not fixed. The memory taken by these data structures can be increased or decreased at run time. Example of the dynamic data structure is Linked List. The size of linked list can be changed during the run time.

Other data structures like stack, queue, tree, and graph can be static or dynamic depending on, whether these are implemented using an array or a linked list.

Homogeneous and Non-Homogeneous Data Structures

Homogeneous Data Structure: Homogeneous data structures are those in which data of same type can be stored.
Example of the homogeneous data structure is an Array.

Non-Homogeneous Data Structure: Non-Homogeneous data structures are those in which data of different types can be stored.
Example of the non-homogeneous data structure is linked list.

Other data structures like stack, queue, tree, and graph can be homogeneous or non-homogeneous depending on, whether these are implemented using an array or a linked list respectively.

A Note

There are two ways of storing linear data structures into the computer’s memory.

In the first way, the linear relationship between the elements of the structure is represented by sequential memory locations. For example, arrays, as elements of an array are stored in continuous/adjacent memory locations.

In second way, the linear relationship between the elements of the structure is given by using pointers (addresses) e.g. Linked List.

Concept of Data Types

A data type can be defined as set of values and set of operations which are permissible on those values.

For example, an integer data type in ‘C’ language can have range of values [-32768 to 32767] and set of operations etc.

Data Types can be classified into two broad categories:

- ❖ Primitive Data Types
- ❖ Abstract Data Types

Primitive Data Types

Primitive data types are also known as predefined or basic data types. These data types may be different for different languages.

For example, in ‘C’ language, the primitive data types for storing the integer values are int, long, short and for storing the real values are float, double and long double.

Abstract Data Types (1)

To understand the notion Abstract Data Type (ADT), it is necessary to understand the concept of Abstract and Data Type separately.

The term Abstract here stands for considering apart from the detailed specification or implementation.

Abstraction refers to the act of representing the essential features without including the details.

Data type as mentioned earlier is the set of values and set of operations that are permissible on those values.

Abstract Data Types (2)

Abstract Data Types uses the following principles:

Encapsulation: Providing the data and operations on the data in a single unit.
Abstraction: Hiding the details of implementation e.g. a class in C++ or in Java Language exhibits what it does through its methods but the detail of how methods work is hidden from the user.

The examples of abstract data types are stack, queue, tree etc. The Stack is an abstract data type as two operations push and pop are performed on the stack without knowing the detail of whether stack is implemented using an array or a linked list.

Data Structure Vs File Organization

✓ The file organization is the study of storing the data records into the files. There are various file organization techniques like sequential, random, and indexed sequential file organization. These organization techniques differ in record sequencing and retrieval methods used.

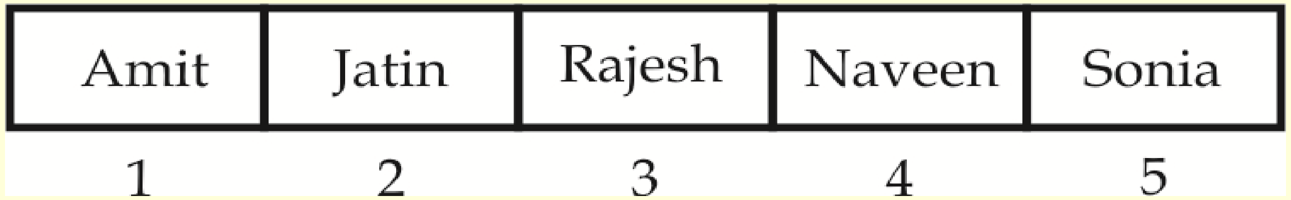
✓ Conceptually, a data structure and a file organization are same but differ in the following aspects:

- ❖ In Implementation
- ❖ In Access Methods

✓ Data structure are thought of as in main memory (RAM) and file organization as in auxiliary storage (tapes, disks)

Short Descriptions of various Data Structures

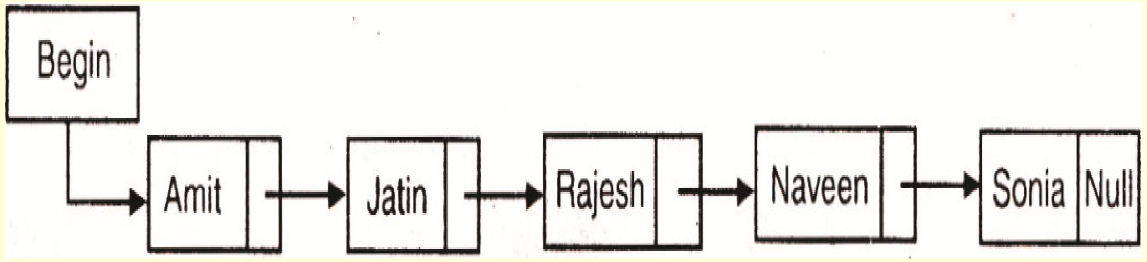
Array:



- ✓ It is the linear collection of finite number of homogeneous data elements.
- ✓ If we consider an array with elements then elements of the array will be referred using index set consisting of consecutive numbers i.e. The elements of an array can be referred by using different notations:
- ✓ When array is stored into the computer's memory, its elements occupies the consecutive memory locations.
- ✓ The total number of elements in an array is known as the size of that a array and can be calculated by using a simple formula:
- ✓ Where is the lower index of the array and is the upper index of the array.
- ✓ Programming languages also support multidimensional arrays.

Short Descriptions of various Data Structures

Linked List :



✓ A linked list refers to the linear collection of data elements in which linear order is not given by their physical placement in memory (as in case of array).
✓ In linked list, the data elements are managed by collection of nodes, where each node contains link or pointer which points to the next node in the list.
✓ The beginning of the linked list is maintained by a special pointer variable which contains the address of the first node in the list.
✓ The link part of the last node contains a special value called **Null**, which shows the end of the list.

Short Descriptions of various Data Structures

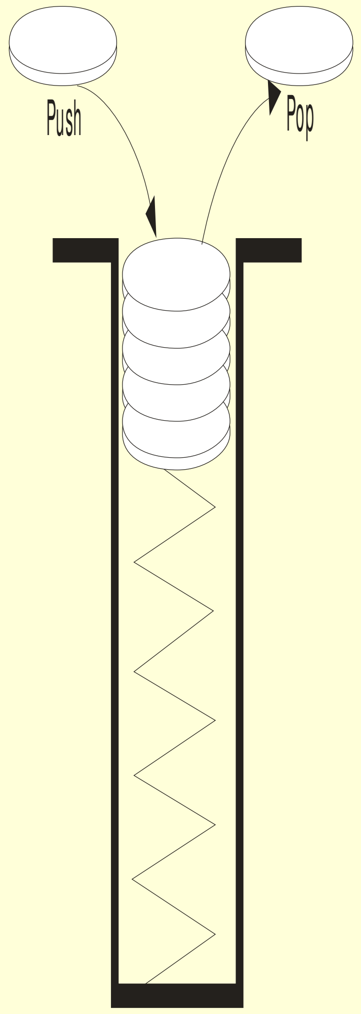
Stacks:

✓ Stack which is also popular with the name **LIFO** list (Last In First Out) is a linear collection of data elements in which insertion and deletion are restricted at only one end known as top.

✓ Every new item is inserted at the top of the stack and only item at the top can be removed from the stack.

Consider an example of stack of dish plates in which clean plates are added at the top of the stack. Plates are also removed from the top of the stack. The first plate put on the stack is the last plate to be removed from the stack.

✓ A stack is linear data structure in which elements are removed in reverse order of that in which these elements were inserted into the stack. Although stack is a restricted kind of data structure but it has many applications in the programming.



Short Descriptions of various Data Structures

Queue:

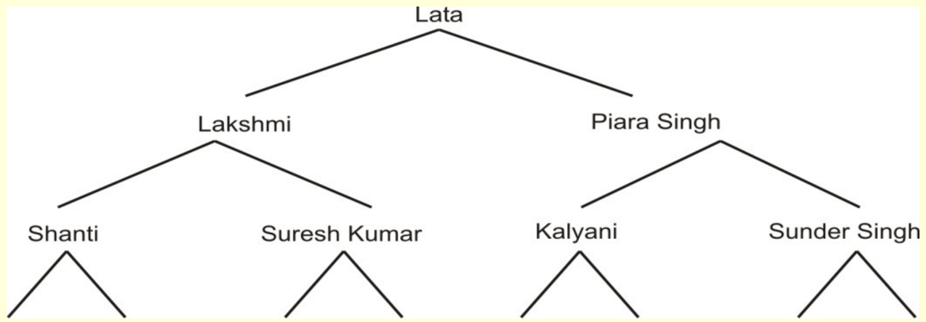
✓ Queue which is popularly known as FIFO (First In First Out) list, is linear collection of data elements in which insertion can take place at one end and deletion can take place at the other end.

✓ The end at which insertion is allowed is called **rear** of the queue and the end at which deletion takes place is called **front** of the queue.

✓ Queue operates just like the queues of real life, for example, in a queue of people standing at the ticket counter every new person joins the queue by standing at the rear of the queue and the person at the front of the queue is the first who leaves the queue after taking the ticket.

Short Descriptions of various Data Structures

Tree:



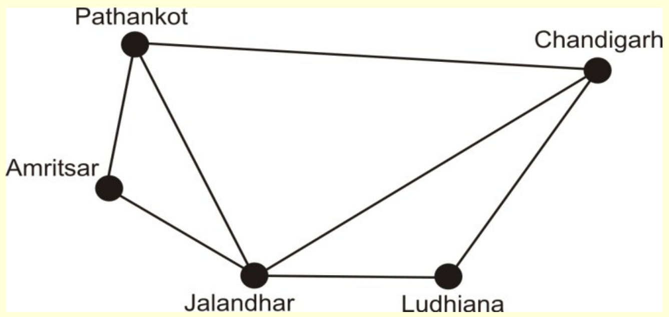
✓ Tree is a non-linear kind of data structure which is used to represent data elements having hierarchical relationship between them.

✓ Tree structure is also referred as parent child relationship. The basic difference between linear and non linear data structures is that in linear data structures, for each element there is a fixed next element but in case of non-linear data structure each element can have many different next elements.

✓ A very common example is the ancestor tree as shown in figure. This tree shows the ancestors of Lata. Her parents are Lakshmi and Piyara Singh. Lakshmi's parents are Shanti and Suresh Kumar. Piyara Singh's parents are Kalyani and Sunder Singh and so on.

Short Descriptions of various Data Structures

Graphs:



✓ Graph is a non-linear kind of data structure which is used to represent data having relationship among its elements which are not necessarily hierarchical in nature. For example, a graph can be used to represent the road map of a state in which various cities are connected with each other by different roads.

✓ A Graph can be defined as collection of nodes and edges, where edges connect various pairs of nodes. In our example of road map, cities (Amritsar, Jalandhar, Ludiana, Chandigarh, Pathankot) represents the nodes whereas roads represents the edges.

Operations on Data Structures (1)

Insertion: Adding a new data element into the data structure is known as insertion.

Deletion: Removing a data element from the data structure is known as deletion.

Traversing: Accessing each data element exactly once in order to process it is known as traversing.

Searching: Finding the position of any given data element in the data structure is known as searching.

Operations on Data Structures (2)

Besides these basic operations there are some other operations which are less frequently performed on some data structures. These operations are:

Combining two or more lists into a single list. This operation is popularly known as **merging**.

Splitting a list into two or more lists.

Copying and concatenation of lists.

Sorting the data elements of a list in ascending/descending order.

Algorithm

The computer is a manmade machine which does not have any decision making capabilities. It only follows the instructions given by its user. Instructions given to the computer to solve a particular problem are known as algorithm.

An algorithm can be defined as finite collection of well defined steps designed to solve a particular problem. An algorithm must have following characteristics:

Input: An algorithm must take some inputs that are required for the solution of problem in question.

Process: An algorithm must perform certain operations on the input data which are necessary for the solution of the problem.

Output: An algorithm should produce certain output after processing the inputs.

Finiteness: An algorithm must terminate after executing certain finite number of steps.

Effectiveness: Every step of an algorithm should play a role in the solution to the problem. Also, each step must be unambiguous, feasible and definite.

Importance of Algorithm Analysis

There are two basic requirements to solve any particular problem.

•Data
•Instructions to manipulate data i.e. algorithm

The choice of an algorithm is of great importance, which can be made by considering the following factors:

•Programming requirements of an algorithm
•Time requirement of an algorithm
•Space requirement of an algorithm

Programming Requirements of an Algorithm

An algorithm must use the features supported by the programming language in which it is to be implemented.

If we have designed an algorithm that is optimal but does not support the programming language features then it is of no use as compared to other one which may be not be optimal but supports the programming language features.

As the algorithm is of no use if it cannot be programmed and executed. Therefore, an algorithm must satisfy the programming features of the language.

Space Requirements of an Algorithm

To execute any program, space is needed for various reasons:

Space required by data: This includes space required to store variables and constants which are fixed. Sometimes space is allocated dynamically (i.e. at the runtime) and this space is not fixed.

Space required by instructions: This is the space required to store the instruction sets. This space remains unchanged as the instructions of the program do not change during run time.

Time Requirements of an Algorithm

Each algorithm takes some amount of time to execute. The study for the time requirements of an algorithm is important for various reasons:

✓Sometimes it is necessary to know in advance, the time required to execute the program to see whether it is within the acceptable limits or not.

✓There can be several different solutions for a particular problem each with different time requirement, so that one can choose the most optimal solution.

✓It is not very easy to calculate the exact time requirements for any algorithm as it depends upon various factors like machine on which algorithm is to be executed, algorithm itself and input size of the algorithm.

✓Because the processor speed in different machines may be different so, we mainly concentrate to estimate the execution time of an algorithm irrespective to the processor/machine.

Complexity of an algorithm

Complexity is the time and space requirement of the algorithm.

If time and space requirement of the algorithm is more, then complexity of the algorithm is more and if time and space requirement of the algorithm in less, complexity of that algorithm is less.

Out of the two factors time and space, the space requirement of the algorithm is not a very important factor because it is available at very low cost.

Only the time requirement of the algorithm is considered an important factor to find the complexity. Because of the importance of time in finding the complexity, it is sometimes termed as time complexity.

As the time requirement of the algorithm is dependent upon the input size irrespective of the other factors like machine/processor, time complexity is measured in terms of input size **n**. If the input size to the algorithm is more, the complexity will be more and if the input size to the algorithm is less, the complexity will be less.

Complexity of an algorithm

For example, consider an algorithm which sorts an array of size 2000, will definitely take more time than to sort an array of size 20. Thus, we express the time complexity in terms of input size **n**.

Hence, to calculate the time complexity of an algorithm, the basic approach is to count the number of times, a key operation is executed. Here, the key operation is the major operation that is executed maximum number of times in the algorithm. For example, in a searching algorithm the key operation is comparison between the elements. We only count the key operation because most of the time taken by the algorithm is consumed by key operation. The time complexity is expressed as a function of key operation performed in that algorithm.

Complexity of an algorithm

As the complexity of an algorithm is dependent upon the input size, still complexity can be divided into three types:

- Worst case complexity
- Best case complexity
- Average case complexity

For a particular input, the result can be obtained in minimum time or maximum time or average time. For instance, consider the linear search in which we find the desired element by comparing it with all the elements of the list (say **n** number of elements in the list) starting from the first element of the list.

If we get the desired element at first position then number of comparisons will be 1. So complexity is 1. If we get the desired element at the last position then number of comparisons will be **n** so complexity is **n**.

If we get the desired element at any other position then the complexity will be between **1** and **n**. So, the complexity can be different (maximum, minimum, or average) for a particular problem.

Complexity of an algorithm

Types of Complexities

Worst Case Complexity: If the running time of the algorithm is longest for all the inputs then the complexity is called worst case complexity. In this type of complexity, the key operation is executed maximum number of times. Worst case is the upper bound of complexity and in certain application domains e.g. air traffic control, medical surgery, the worst case complexity is of crucial/high importance.

Best Case Complexity: If the running time of the algorithm is shortest for all the inputs then the complexity is called best case complexity. In this type of complexity, the key operation is executed minimum number of times.

Average Case Complexity: If the running time of the algorithm falls between the worst case and the best case then the complexity is called average case complexity. Average case complexity of an algorithm is difficult to find. To calculate the average case complexity of an algorithm we have to take some assumptions.

Asymptotic Analysis and Notations

Complexity can be defined as the rate at which the storage or time requirement grows as a function of the problem size.

The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors.

We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations.

This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a proportionality approach, expressing the complexity in terms of its relationship to some known function.

The type of analysis known as asymptotic analysis is used to simplify the analysis of running time by removing the details which may be affected by hardware or compilers used.

For example, the number 999999 can be read as 1000000 or the number 10000001 can be read as 10000000.

Also as the constant 4 is dependent upon the compiler, hardware, and many other factors.

Asymptotic Analysis and Notations

To measure the increase in running time of algorithm with the increase in size of input.
Asymptotically, more efficient algorithms are best for all but small input.

To measure the complexity of an algorithm, various asymptotic notations can be used such as,

- Big O Notation
- Big Omega(Ω) Notation
- Big Theta(θ) Notation
- Little Omega Notation
- Little Theta Notation

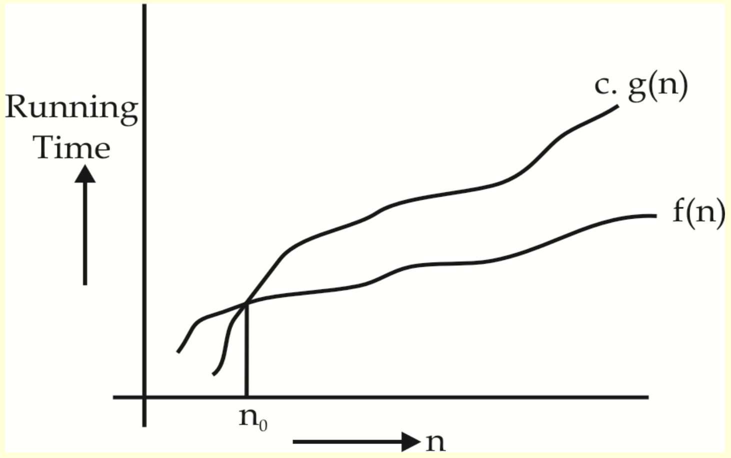
All the above notations are used to express the complexity of the algorithm.

Big O Notation

Big O notation is **upper bound** asymptotic notation. This means, a function f(x) is Big O of function g(x) and there exists the positive constants c and n_o such that Here, and are the functions of non negative integers.

$c.\ g(n) \geq f(n) \qquad \text{for all } n \geq n_o$

Here, f(x) and g(x) are the functions of non negative integers.



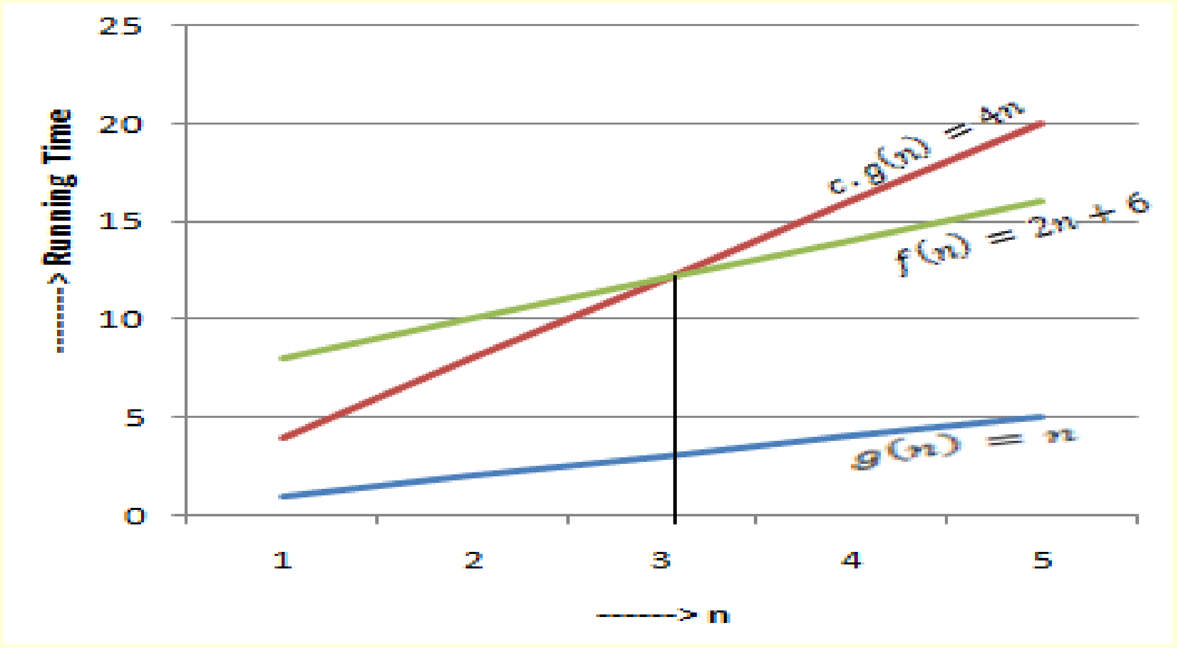
Complexity is O(g(n)).

Big O Notation - Examples

$f(n) = 2n + 6$, $g(n) = n$

Taking constants $c = 4$, and $n_o = 3$,

$f(n) \leq c.g(n)$ where $n \geq n_o$



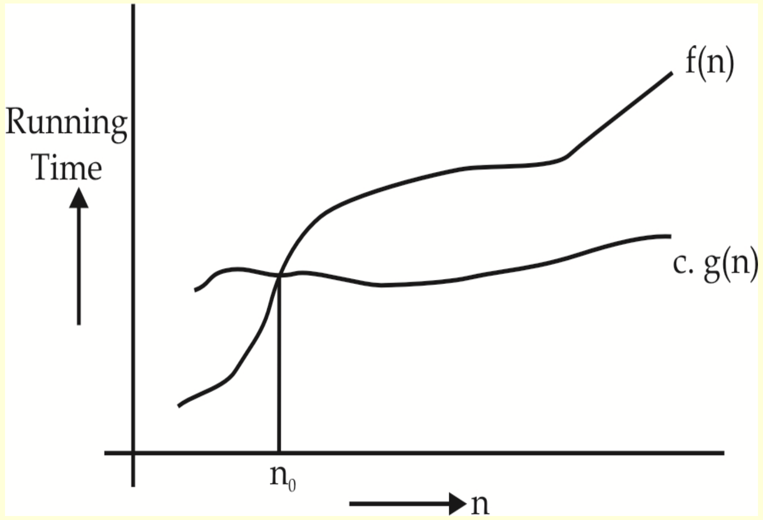
Complexity is $O(g(n) = n$.

In general, Big O notation drops all the constant and lower order terms.

Big Omega Notation

Big Omega is asymptotically **lower bound** notation. This means a function f(x) is Big Omega of function g(x) and there exists two positive constants c and n_o such that

$$cg(n) \le f(n) \qquad \text{for all } n \ge n_0$$



Big Omega is used to express the best case running time or the lower bounds of the algorithmic problems. The function g(n) is only a lower bound on f(n).

Big Omega Notation - Examples

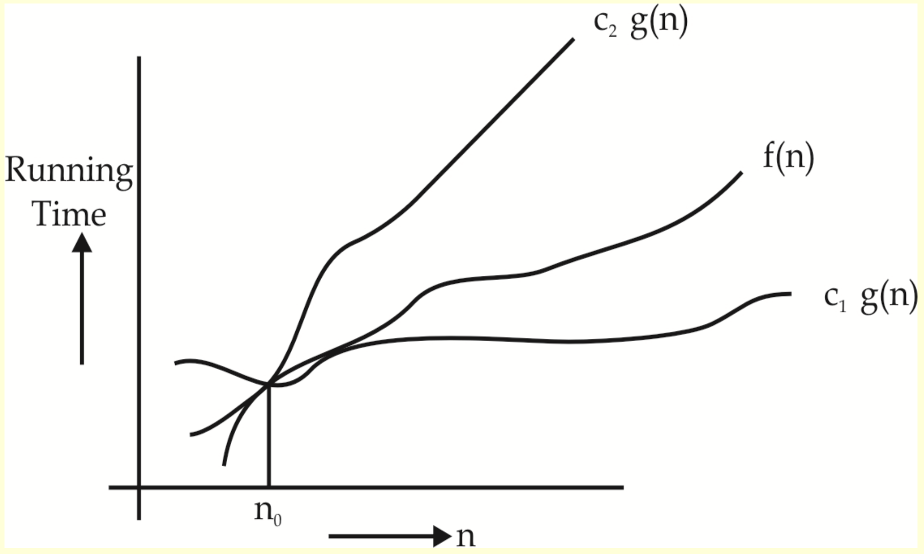
3n +2 is Ω(n) as 3n+2 ≥3n for all n ≥ 11

100n+7 is Ω(n) as 10n+7 ≥ 100n for all n ≥ 1

Big Theta Notation

Big Theta is asymptotically a **tight bound** notation. This means a function f(x) is Big Theta of function g(x) and there exists three positive constants c_1, c_2 , and n_o such that

$$c_1 g(n) \le f(n) \le c_2 g(n) \qquad for\ all\ n \ge n_0$$



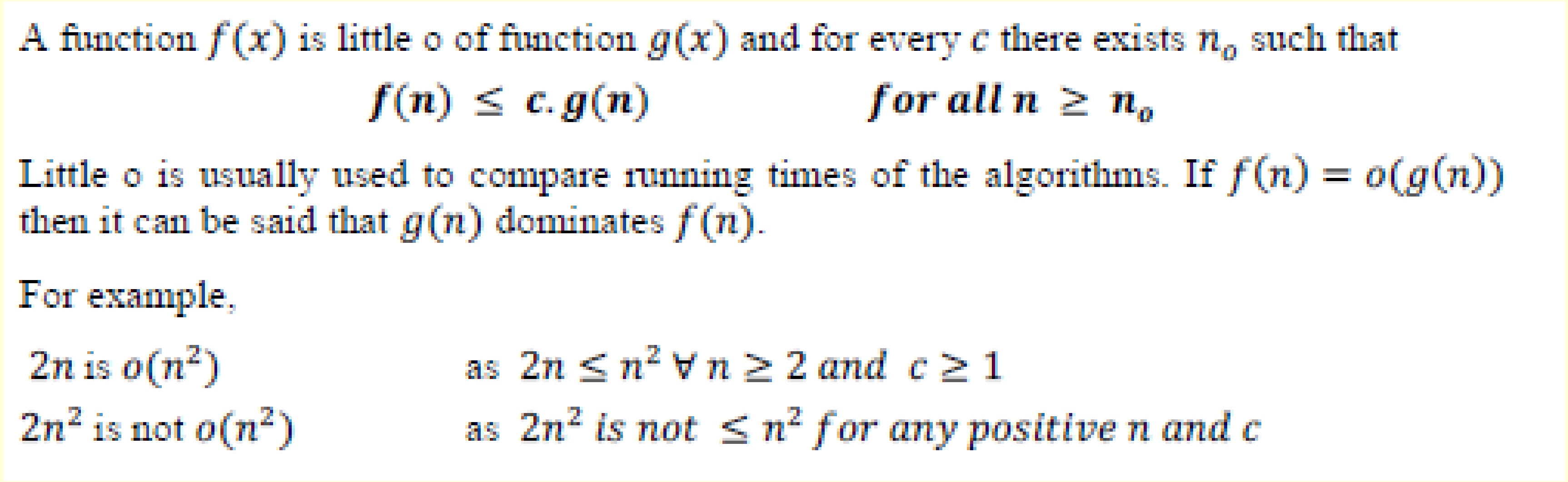
It may be noted that f(n) = φ(g(n)) iff
F(n) = O(g(n) and
F(n) = Ώ(g(n))

Little O Notation

A function f(x) is little O Notation of function g(x) and for every c there exists n_o such that

$$f(n) \leq cg(n) \qquad for\ all\ n \geq n_o$$

Little o is usually used to compare running times of the algorithms. If f(n) = o(g(n)) then it can be said that g(n) dominates f(n).



A function $f(x)$ is little o of function $g(x)$ and for every c there exists n_o such that

$$f(n) \leq c.g(n) \qquad for\ all\ n \geq n_o$$

Little o is usually used to compare running times of the algorithms. If $f(n) = o(g(n))$ then it can be said that $g(n)$ dominates $f(n)$.

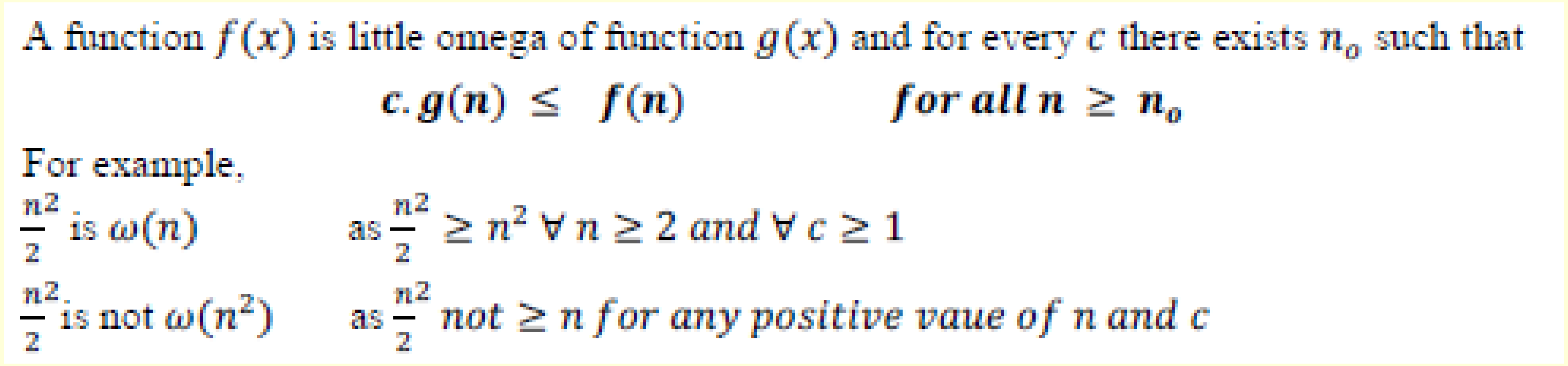
For example,

$2n$ is $o(n^2)$	as $2n \leq n^2\ \forall\ n \geq 2\ and\ \ c \geq 1$
$2n^2$ is not $o(n^2)$	as $2n^2\ is\ not\ \leq n^2\ for\ any\ positive\ n\ and\ c$

Little omega (ω) Notation

A function f(x) is little omega of function g(x) and for every c there exists n₀ such that

$$cg(n) \leq f(n) \qquad \qquad for\ all\ n \geq n_o$$



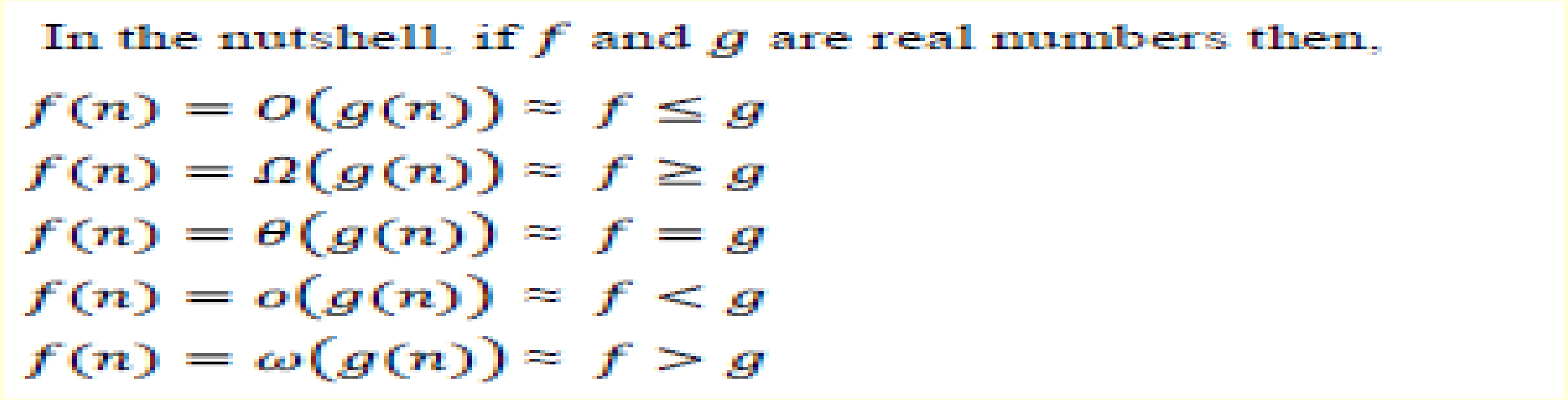
A function $f(x)$ is little omega of function $g(x)$ and for every c there exists n_o such that

$$\boldsymbol{c.g(n) \leq f(n) \qquad for\ all\ n \geq n_o}$$

For example,

$\frac{n2}{2}$ is $\omega(n)$ as $\frac{n2}{2} \geq n^2\ \forall\ n \geq 2\ and\ \forall\ c \geq 1$

$\frac{n2}{2}$ is not $\omega(n^2)$ as $\frac{n2}{2}\ not \geq n\ for\ any\ positive\ vaue\ of\ n\ and\ c$

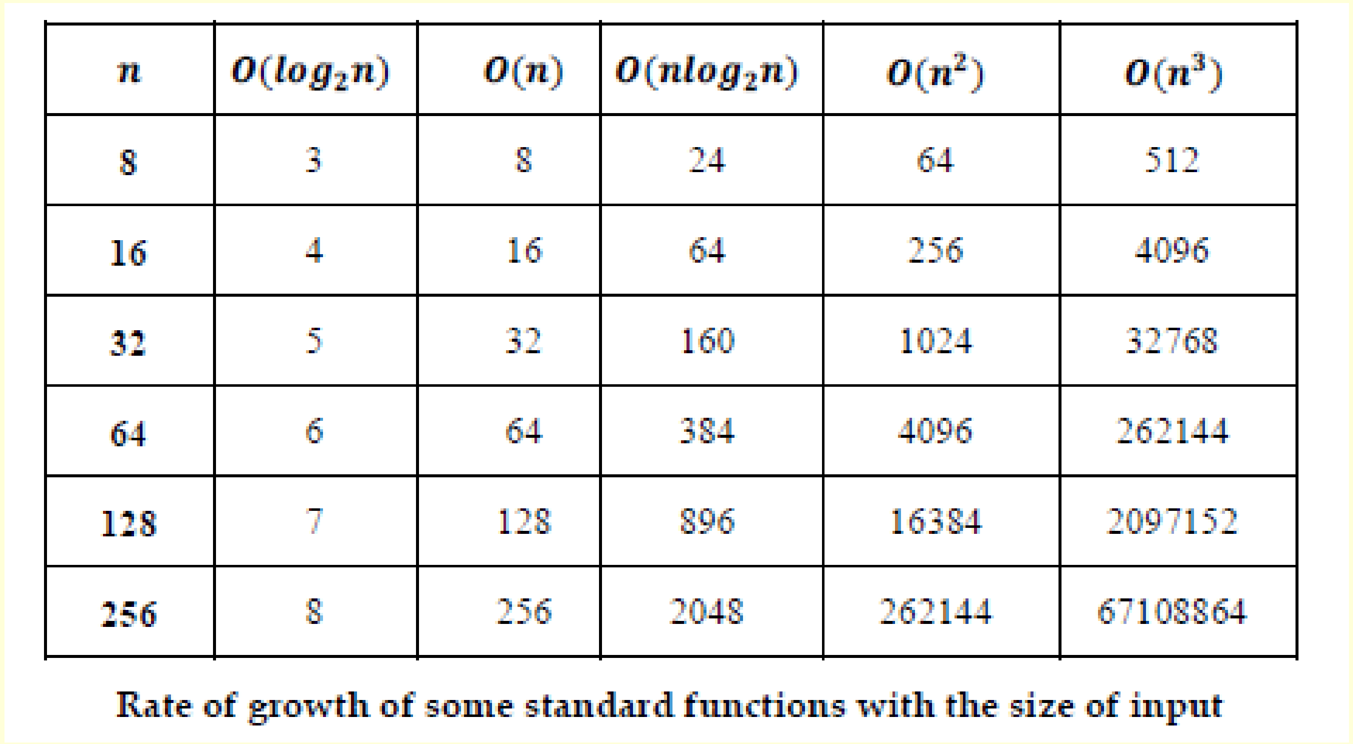


In the nutshell, if f and g are real numbers then,

$$f(n) = O(g(n)) \approx f \leq g$$
$$f(n) = \Omega(g(n)) \approx f \geq g$$
$$f(n) = \theta(g(n)) \approx f = g$$
$$f(n) = o(g(n)) \approx f < g$$
$$f(n) = \omega(g(n)) \approx f > g$$

Rate of Growth of Complexity with Input Size

It is the Rate at which the complexity increases when the input size is increased.

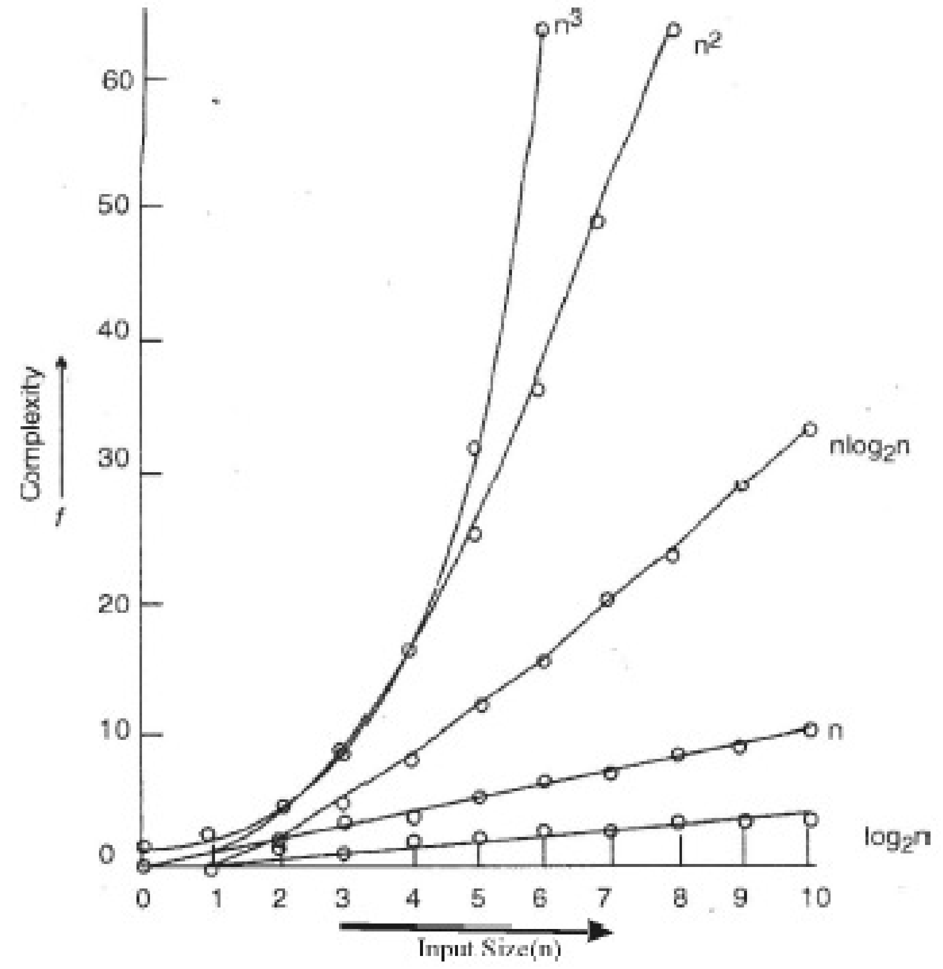


n	$O(log_2n)$	$O(n)$	$O(nlog_2n)$	$O(n^2)$	$O(n^3)$
8	3	8	24	64	512
16	4	16	64	256	4096
32	5	32	160	1024	32768
64	6	64	384	4096	262144
128	7	128	896	16384	2097152
256	8	256	2048	262144	67108864

Rate of growth of some standard functions with the size of input

Rate of Growth of Complexity with Input Size

Continued….



Rate of growth of complexity with input size

Time Space Trade-Off among Algorithms

The complexity of an algorithm can be defined as the function which gives the time and space requirements of an algorithm in terms of input size.
The best algorithm for a particular problem is that which requires minimum time to execute it by taking the minimum amount of space into the memory. But, practically both these objectives are not possible to achieve simultaneously.
As mentioned earlier, there may be various different solutions for a particular problem. One such solution may require more time but less space and another may require less time but more space. Therefore, we choose an algorithm according to our requirements and constraints. If we have time constraint i.e. we want to execute the program in less time then we have to choose an algorithm which takes less time but it may take more space. For example, in the real life applications we need to choose an algorithm that takes less time for its execution. On the other hand, if we have space constraint then we have to choose an algorithm which takes less space but it may take more execution time.
We have to choose from the two options, one with less time but more space another with more time but less space. This is what is popularly known as Time-Space tradeoff among algorithms.
Because of advancement in hardware technology and decreased cost of hardware, space is no longer a constraint. So, we can opt for algorithms with less execution time at the cost of storage space.

Time Space Trade-Off among Algorithms

Tradeoff : Situation that involves losing one quality or aspect of something in return for gaining another quality or aspect.

Time and space requirements of algorithm can not be minimized and hence time-space tradeoff.

	Situation 1	Situation 2
Time	↑	↓
Space	↓	↑