1.5 DATA TYPES *VS.* DATA STRUCTURES

In a programming language, the *data type* of a variable is the set of values that the variable may assume. For example, a variable of type *Boolean* can assume either the value true or the value false, but no other value. The basic data types vary from language-to-language. The rules for constructing composite data types out of basic ones also vary from language-to-language.

An *abstract data type* is a mathematical model, together with various operations defined on the model. As we have know, we shall design algorithms in terms of ADT's, but to implement an algorithm in a given programming language we must find some way of representing the ADT's in terms of the data types and operators supported by the programming language itself. To represent the mathematical model underlying an ADT we use *data structures*, which are collections of variables, possibly of several different data types, connected in various ways.

A data type is a term which refers to the kind of data. It is a well-defined collection of data with a well-defined set of operations on it.

A data *structure* is an actual implementation of a particular abstract data type, *i.e.*, this is an extension of the concept of the data type.

Data Type = Permitted Data Values + Operations

Data Structure = Organized Data + Allowed Operations

1.6 DATA STRUCTURE OPERATIONS

The data appearing in our data structure is processed by means of certain operations. Infact the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following data structure operations play a major role in processing of data:

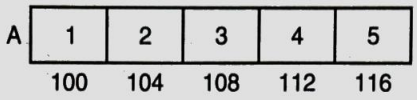
1. **Creating.** This is the first operation to create a data structure. This is just declaration and initialisation of the data structure and reserved memory locations for data elements.
2. **Inserting.** Adding new records to the structure.
3. **Deleting.** Removing a record from the structure.
4. **Updating.** It changes data values of the data structure.
5. **Traversing.** Accessing each record exactly once so that certain items in the record may be processed. (This accessing or processing is sometimes called visiting the records.)
6. **Searching.** Finding the location of the record with a given—key value, or finding the locations of all records, which satisfy one or more conditions in the data.
7. **Sorting.** Arranging the data elements in some logical order, *i.e.*, in ascending and descending order.
8. **Merging.** Combine the data elements in two different sorted sets into a single sorted set.
9. **Destroying.** This must be the last operation of the data structure and apply this operation when no longer needs of the data structure.

1.7 OVERVIEW OF VARIOUS DATA STRUCTURES

(*a*) Arrays: The simplest type of data structure is a *linear array* and most often it is the only data structure that is provided in any programming language. An array can be defined

as a collection of homogeneous elements, in the form of **index/value** pairs, stored in consecutive memory locations. An array always has a predefined size and the elements of an array are referenced by means of an index / subscript. Thus an array is a collection of variables of the same data type that share a common name. The general form of a one-dimensional array is

<type specifier> array name [size]



Memory Organisation of an Array

Advantages

- Searching is faster as elements are in continuous memory locations.
- Memory management is taken care of by the compiler itself.

Limitations

- Number of elements must be known in advance
- Inserting and Deletions are costlier as it involves shifting the rest of the elements

(*b*) Stack: *A stack is an ordered list in which items are inserted and removed at only one end called the TOP.* There are only 2 operations that are possible on a stack. They are the 'Push' and the 'Pop' operations. A Push operation inserts a value into the stack and the Pop operation retrieves the value from the stack and removes it from the stack as well. An example for a stack is a stack of plates arranged on a table. This means that the last item to be added is the first item to be removed. Hence a stack is also called as **Last-In-First-Out List or LIFO** list. A graphical representation of a stack is shown below:

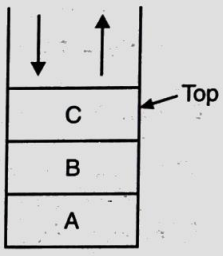


Fig. 1.3.

(*c*) Queue: A queue is an ordered list in which all insertions can take place at one end called the **rear** and all deletions take place at the other end called the front. The two operations that are possible in a queue are Insertion and Deletion. A real time example for a queue is people standing in a queue for billing on a shop. The first person in the queue will be the first person to get the service. Similarly, the first element inserted in the queue will be the first one that will be retrieved and hence a queue is also called as **First In First Out or FIFO** list.

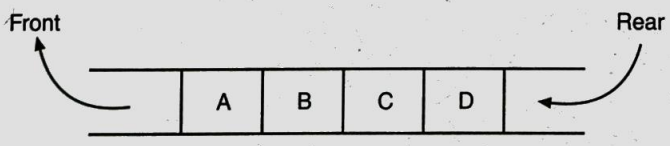


Fig. 1.4 Graphical Representation of a Queue.

The most common occurrence of a queue in computer applications is for scheduling of print jobs. For example, if several people are giving print requests, all the request get queued up in the printer and is processed on first come first serve basis.

(*d*) Linked Lists: In real time systems most often the number of elements will not be known in advance. The major drawback of an array is that the number of elements must be known in advance. Hence an alternative approach was required. This give rise to the concept called linked lists. A linear list is a linear collection of data elements called nodes, where the linear order is given by means of pointers. The key here is that every node will have two

parts: first part contains the information/data and the second part contains the link/address of the next node in the list. Memory is allocated for every node when it is actually required and will be freed when not needed.

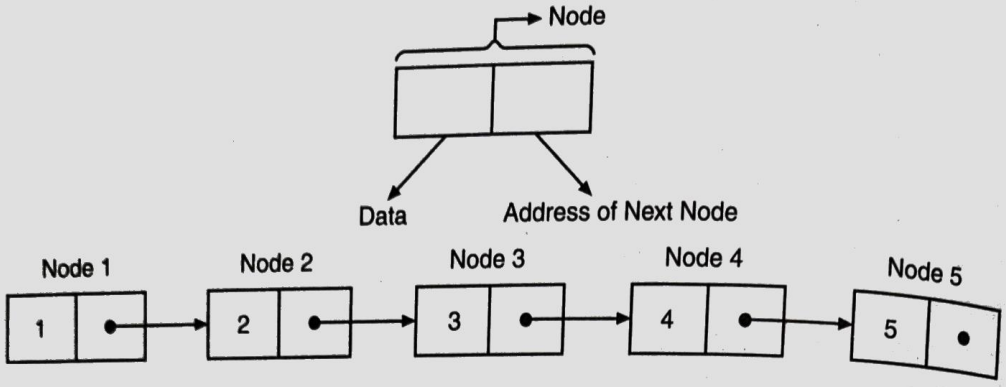


Fig. 1.5.

A linked list is used typically in computer applications for memory allocations. The system keeps tracks of all the memory available in the system with the help of a list. This area of memory is called as the memory pool. As and when the user requests for memory the system allocates memory to the user from this available pool. Once allocated these blocks will be deleted from the available list. Once the user frees the memory again the memory will be added to the list. In general linked lists are used in almost all places whenever a collection of elements are required and when the number of elements in the collection is not known in advance.

(e) Tree: A tree is a non-linear data structure. This structure is mainly used to represent data containing a hierarchical relationship between elements like family tree, organisation chart etc.

A tree is a finite set of one or more nodes such that:

(*i*) There is a specially designated node called the **root.**

(*ii*) The remaining nodes are partitioned into $n >= 0$ disjoint sets T_1, ..., T_n where each of these sets is a tree. T_1,..., T_n are called *sub-trees* of the root.

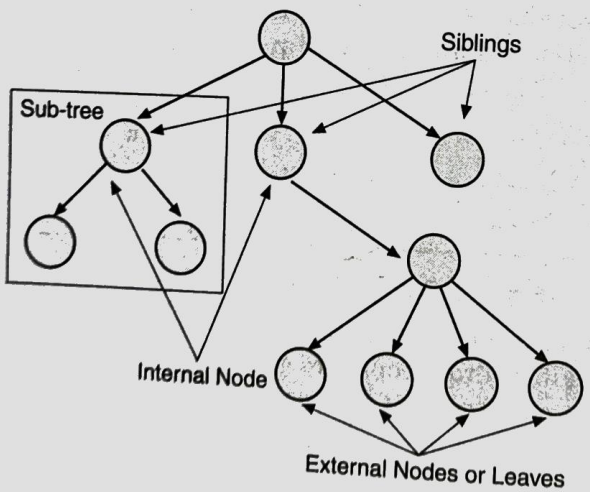


Fig. 1.6.

(f) Graphs: Graph is a general tree with no parent child relationship. Graphs have many applications in computer science and other fields of science. In general, graphs represent

a relatively less restrictive relationship between the data items. A graph G may be defined as a finite set V of vertices and a set E of edges (pair of connected vertices). The notation used is as follows: Graph G = (V, E). Consider the graph of Fig. 1.7.

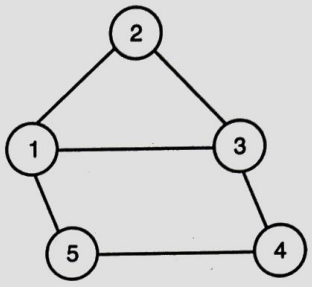


Fig. 1.7.

The set of vertices for the graph is V = {1, 2, 3, 4, 5}.

The set of edges for the graph is E = {(1,2), (1,5), (1,3), (5,4), (4,3), (2,3) }.

Graphs are used in a wide variety of applications. A graph can be used by airlines for maintaining the flight information such as the various flights, their routes and the distance between the places, etc. With the help of such a graph one will be easily able to find out whether there is any flight for a particular destination and if there are several routes to reach that destination which one is having the optimum distance or cost, etc.

1.8 WHAT IS AN ALGORITHM?

Once a data structure for a particular application is chosen, an algorithm must be developed that manipulates the related data items stored in it. Such an algorithm should have the following features.

1. It should be free of ambiguity.
2. It should be concise.
3. It should be efficient.

An *algorithm* is a precise plan for performing a sequence of actions to achieve the intended purpose. Each action is drawn from a well-understood selection of actions on data.

Definition: An *algorithm* is a step-by-step finite sequence of instructions, to solve a well-defined computational problem. That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem. In addition every algorithm must satisfy the following criteria:

(*i*) **Input:** There are zero or more quantities which are externally supplied;

(*ii*) **Output :** At least one quantity is produced;

(*iii*) **Definiteness :** Each instruction must be clear and unambiguous;

(*iv*) **Finiteness:** If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

(*v*) **Effectiveness:** Every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite as in (*iii*), but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy condition (*iv*). One important example of such a program for a computer is its operating system which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

Therefore, *an algorithm can be defined as a sequence of definite and effective instructions, while terminates with the production of correct output from the given input.*

1.9 DESIRABLE ATTRIBUTES FOR ALGORITHMS

Although an algorithm may satisfy the criteria of the previous section by being precise, unambiguous, deterministic, and finite, it still may not be suitable for use as a computer

solution to a problem. The basic desirable attributes that an algorithm should meet are explained below.

Generality: An algorithm should solve a class of problems, not just one problem. For example, an algorithm to calculate the average of four values is not as generally useful as one that calculates the average of an arbitrary number of values.

Good Structure: This attribute applies to the construction of the algorithm. A well-structured algorithm should be created using good building blocks that make it easy to:

- Explain,
- Understand,
- Test, and
- Modify it.

The blocks, from which the algorithm is constructed, should be interconnected in such a way that one of them can be easily replaced with a better version to improve the whole algorithm, without having to rebuild it.

Efficiency: An algorithm's speed of operation is often an important property, as is its size or compactness. Initially, these attributes are not important concerns. First, we must create well-structured algorithms that carry out the desired task under all conditions. Then, and only then, do we improve them with efficiency as an objective.

Ease of Use: This property describes the convenience and ease with which users can apply the algorithm to their data. Sometimes what makes an algorithm easy to understand for the user also makes it difficult to design for the designer (and vice-versa).

Elegance: This property is difficult to define, but it appears to be connected with the qualities of harmony, balance, economy of structure and contrast, whose contribution to beauty in the arts is prized. Also, as with the arts, we seem to be able to "know beauty when we see it". In algorithms, sometimes the term *beauty* is used for this attribute of an algorithm.

Other desirable properties, such as ***robustness*** (resistance to failure when presented with invalid data) and ***economy*** (cost-effectiveness), will only be touched upon in this chapter. Not because these properties are unimportant, but because the basic attributes of an algorithm should be understood first.

1.10 ALGORITHM DESIGN TECHNIQUES

There are two approaches for algorithm design; they are **top-down and bottom-up** algorithm design.

(a) Top-down Algorithm Design: The principle of top-down design states that a program should be divided into a main module and its related modules. Each module should also be divided into sub-modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary processes that are basically understood and cannot be further sub-divided.

Top-down algorithm design is a technique for organising and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

In C, the idea of top-down design is done using functions. A C program is made of one or more functions, one and only one of which must be named *main*. The execution of the program always starts and ends with *main*, but it can call other functions to do special tasks.

(*b*) Bottom-up algorithm design: Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and builds them into more complex structures, ending at the top.

The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the *main* function is tested.

For a given problem, there are many ways to design algorithms for it, *e.g.*, insertion sort is an *incremental approach,* Merge sort is a *divide and conquer approach.* The following is a list of several popular design approaches.

1. Incremental approach
2. Divide and Conquer approach
3. Greedy approach
4. Dynamic programming approach
5. Backtracking approach
6. Branch and bound approach
7. Randomised approach

Incremental Appraoch

— Insertion sort uses an incremental approach.

— Having sorted the sub-array [1 ... j-1], insert the single element $A[j]$ into its proper place, yielding the sorted sub-array A[1...j].

Divide and Conquer Approach

- Divide the original problem into a set of sub-problems
- Solve every sub-problem individually, recursively
- Combine the solutions of the sub-problems (top level) into a solution of the whole original problem.

Greedy Approach

Greedy algorithms seek to optimise a function by making choices (greedy criterion) which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

Dynamic Approach

Dynamic programming is a technique for efficiently computing recurrences by storing partial results. It is a method of solving problems exhibiting the properties of overlapping sub-problems and optimal sub-structure that takes much less time than naive methods.

Branch-and-Bound

In a branch and bound algorithm a given sub-problem, which cannot be bounded, has to be divided into at least two new restricted sub-problems. Branch and bound algorithms are

methods for global optimisation in non-convex problems. Branch and bound algorithms can be (and often are) slow, however, in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the methods converge with much less effort.

Randomised Algorithms

A randomised algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

Backtracking Algorithms

Backtracking algorithms try each possibility until they find the right one. It is a depth-first search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alterative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. It there are no more choice points, the search fails.

1.11 ANALYSIS OF ALGORITHM

(UPTU 2009-10)

Why is it necessary to analyse an algorithm? After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analysing the algorithm. Thus, an algorithm analysis measures the *efficiency* of the algorithm. The ***efficiency of an algorithm*** can be checked by:

- Correctness of an algorithm
- Implementation of an algorithm
- Simplicity of an algorithm
- Execution time and memory requirements of an algorithm

The algorithm can be analysed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyse the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem.

1.11.1 Types of Analysis

1. Worst case running time
2. Average case running time
3. Best case running time

Worst Case Running Time

(UPTU 2007)

- The behaviour of the algorithm with respect to the worst possible case of the input instance.
- The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the item does not occur in data.
- There is no need to make an educated guess about the running time.

Average Case Running Time

- The expected behaviour when the input is randomly drawn from a given distribution.
- The average-case running time of an algorithm is an estimate of the running time for an "average" input.
- Computation of average-case running time entails "knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences".
- Often it is assumed that all inputs of a given size are equally likely.

Best Case Running Time

- The behaviour of the algorithm when input is in already in order. For example, in sorting, if elements are already sorted for a specific algorithm.
- The best case running time rarely occurs in practice comparatively with the first and second case.

The choice of a particular algorithm depends on following performance analysis and measurements:

1. Space complexity
2. Time complexity

1.11.2 Space Complexity

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion. Some of the *reasons* for studying space complexity are:

1. If the program is to run on multi-user system, it may be required to specify the amount of memory to be allocated to the program.
2. We may be interested to know in advance that whether sufficient memory is available to run the program.
3. There may be several possible solutions with different space requirements.
4. It can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

• ***Instruction Space:*** Space needed to store the executable version of the program and it is fixed.

• ***Data Space:*** Space needed to store all constants; variable values and has further two components:

(*a*) Space needed by constants and simple variables. This space is fixed.

(*b*) Space needed by fixed sized structural variables, such as arrays and structures.

(*c*) Dynamically allocated space. This space usually varies.

• ***Environment stack space:*** This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack:

(*a*) **Return address:** For example, from where it has to resume after completion of the called function.

(*b*) Values of all lead variables and the values of formal parameters in the function being invoked.

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables

and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

1.11.3 Time Complexity

(UPTU 2005, 07)

The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimising the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine-to-machine. By analysing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction.

The time complexity also depends on the amount of data inputed to an algorithm. But we can calculate the order of magnitude for the time required. That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used.

DO YOU KNOW ?

The *complexity of an algorithm* is a function $f(n)$ which measures the time and/or space used by an algorithm in terms of the input size n.

Some of the reasons for studying time complexity are:

1. We may be interested to know in advance that whether an algorithm/ program will provide a satisfactory real time response.
2. There may be several possible solutions with different time requirements.

The time complexity of a program depends on all factors that the space complexity depends on.

1.12 TIME-SPACE TRADE-OFF

In computer science, a space-time or time-memory trade-off is a way of solving a problem or calculation in less time by using more storage space (or memory), or by solving a problem in very little space by spending a long time. So if our problem is taking a long time but not much memory, a space-time trade-off would let we use more memory and solve the problem more quickly or, if it could be solved very quickly but requires more memory than, we can try to spend more time solving the problem in the limited memory.

We may sometimes seek a trade-off between space and time complexity. For example, we may have to choose a data structure that requires a lot of storage in order to reduce the computation time. Therefore, the programmer must make a judicious choice from an informed point of view. The programmer must have some verifiable basis based on which a data structure or algorithm can be selected. Complexity analysis provides such a basis. The best algorithm (or program) to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice, it is not always possible to achieve both of these objectives. One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution. Thus, we may have to sacrifice one at the cost of the other. If the space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint such as in real time

system, we have to choose a program that takes less time to complete its execution at the cost of more space.

DO YOU KNOW ?

The *time-space tradeoff* refers to a choice between algorithmic solutions of a data processing problem that allows one to decrease the running time of an algorithmic solution by increasing the space to store the data and vice versa.

Over 50 years of researches for algorithms related to different problem areas like decision-making, optimisation etc. and scientists have tried to concentrate on computational hardness of different solution methods. The study of *time-space trade-offs, i.e.*, formulae that relate the most fundamental complexity measures, time and space, was initiated by **Cobham,** who studied problems like recognising the set of palindromes on Turing machines. There are two main lines of motivation for such studies: one is the lower bound perspective, where restricting space allows you to prove general lower bounds for decision problems; the other line is the upper bound perspective where one attempts to find time efficient algorithms that are also space efficient (or vice-versa). Also, upper bounds are interesting for finding, in conjunction with lower bounds, the computational complexity of fundamental problems such as sorting. So, mainly algorithms are constrained under two resources *i.e.*, time and space. However, there are certain problems for which it is difficult or even impossible to find such a solution that is both time and space efficient. In such cases, the algorithms are written for specific resource configuration and also taking in consideration nature of input instance. Fundamentally, we measure either of the space and time complexities as a function of the size of the input instance. However, it is also likely to depend upon nature of the input. So, let us define the time and space complexity as below:

Time Complexity $T(n)$: An algorithm A for a problem P is said to have time complexity of $T(n)$ if the number of steps required to complete its run for an input of size n is always less than equal to $T(n)$.

Space Complexity $S(n)$: An algorithm A for a problem P is said to have space complexity of S(n) if the no. of bits required to complete its run for an input of size n is always less than equal to $S(n)$.

If we consider, both time and space requirements, general format is to use the quantity **(time * space)** for a given algorithm. $T(n)*S(n)$ is therefore quite handy to estimate the overall efficiency of an algorithm in general. The amount of space can easily be estimated for most of the algorithms directly and the measurement of run-time of algorithm mathematically or manually by testing can tell us the efficiency of the algorithm. From here we can establish, the range of this $T*S$ term, that we can afford. All algorithms that lie in this range are then acceptable to us.

1.13 ASYMPTOTIC ANALYSIS

We will learn about various techniques to bind the complexity function. In fact, our aim is not to count the exact number of steps of a program or the exact amount of time required for executing an algorithm. In theoretical analysis of algorithms, it is common to estimate their complexity in *asymptotic sense, i.e.*, to estimate the complexity function for reasonably large length of input '*n*'. Big *Oh* (O) notation, omega (Ω) notation and theta (θ) notation are used for this purpose.

1.13.1 Asymptotic Notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0, 1, 2 \ldots\}$. Such

notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes.

Why are asymptotic notations important?

1. They give a simple characterisation of an algorithm's efficiency.
2. They allow the comparison of the performances of various algorithms.

(a) Big-oh Notation

We have $O(f)$, called *big-oh*, which represents the class of functions that grow no faster than f. This means that for all values of n greater than some threshold n_0, all of the functions in $O(f)$ have values that are no greater than f. The class $O(f)$ has f as an upper bound, so none of the functions in this class grow faster than f.

Formally this means that if $g(x) \in O(f)$, $g(n) = cf(n)$ for all $n = n_0$ (where c is a positive constant).

For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions.

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that:

$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

We use O-notation to give an upper bound on a function, to within a constant factor. Figure shows the intuition behind O-notation. For all values n to the right of n_0, the value of the function $f(n)$ is on or below $g(n)$.

Thus, big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.

This is the class that will be of the greatest interest to us. Considering two algorithms, we will want to know **if the function categorising the behaviour of the first is in big oh of the second. If so, we know that the second algorithm does no better than the first in solving the problem.**

- **O (1):** Pronounced "order I" and denoting a function that runs in constant time.
- **O (n):** Pronounced "order N" and denoting a function that runs in linear time.
- **O (N^2):** Pronounced "order N squared" and denoting a function that runs in quadratic time.
- **O (log N):** Pronounced "order log N and denoting a function that runs in logarithmic time.
- **O (N log N):**Pronounced "order N log N" and denoting a function that runs in time proportional to the size of the problem and the logarithmic time.
- **O(N!):** Pronounced "order N factorial" and denoting a function that runs in factorial time.

DO YOU KNOW?

Asymptotic means a line that tends to converge to a curve, which may or may not-eventually touch the curve. It is a line that stays within bounds.

Properties of the Big-oh Notation

1. Constant factors may be ignored such as $3n^2$ and $50n^2$ are both $O(n^2)$.
2. Higher powers of n grow faster than lower powers.
3. The growth rate of the sum of terms is the growth rate of its fastest growing term, *e.g.*, $5n^3+2n$ is $O(n^3)$.

4. If f is a polynomial of degree d, then f is $O(n^d)$.
5. Exponential functions grow faster than powers.
6. Logarithms grow more slowly than powers.
7. The product of upper bound of functions gives an upper bound for the product of the functions. For example, if f is $O(n^2)$ and g is $O(\log n)$, then fg is $O(n^2 \log n)$.

Limitation of Big-oh Notation

Big-Oh Notation has following two basic limitations:

1. It contains no effort to improve the programming methodology. Big-Oh Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyse and calculate the efficiency (by finding time complexity) of the program.
2. It does not exhibit the potential of the constants. For example, one algorithm is taking 1000 n^2 time to execute and the other n^3 time. The first algorithm is $O(n^2)$, which implies that it will take less time than the other algorithm which is $O(n^3)$. However in actual execution the second algorithm will be faster for $n < 1000$.

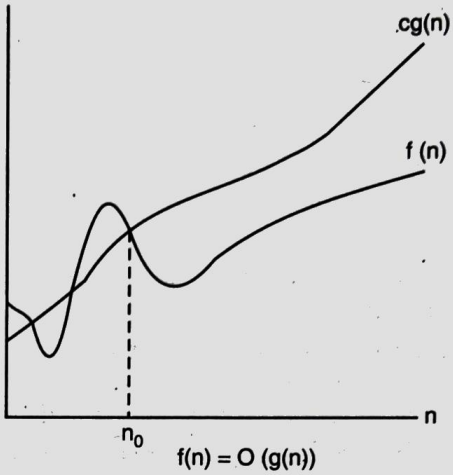


Fig. 1.8.

(b) Big-omega Notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound.***

For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n")

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that:

$0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.

This is almost the same definition as Big-oh except that "$f(n) \ge g(n)$", this makes $g(n)$ a lower bound function instead of an upper bound function. It describes the best that can happen for a given data size.

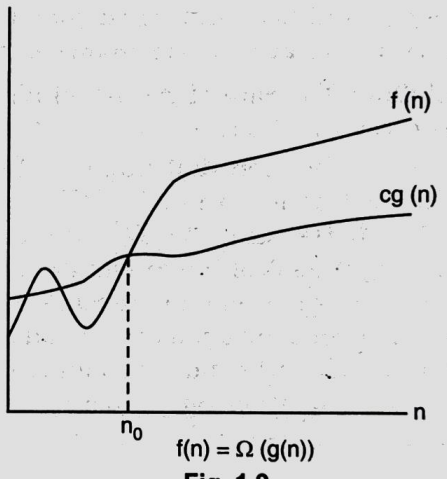


Fig. 1.9.

Because we are interested in efficiency, $\Omega(f)$ will not be of much interest to us because $\Omega(n^2)$, for example, includes all functions that grow faster than n^2 including n^3 and 2^n.

(c) Theta Notation

The lower and upper bound for the function T is provided by the theta notation. For a given function $g(n)$.

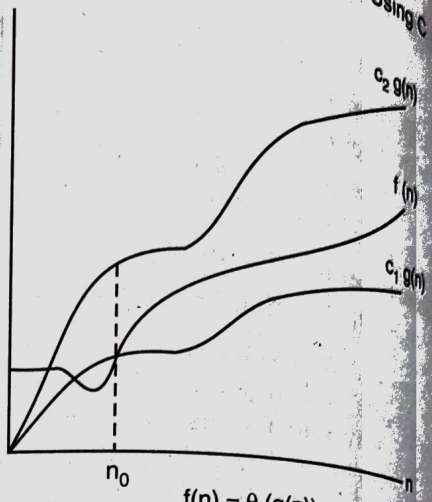
We denote by $\theta(g(n))$ the set *of functions* as:

$\theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that:
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

A function $f(n)$ belongs to the set $\theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

Formally, this class of functions is defined as the place where big-omega and big oh overlap, so $\theta(f) = \Omega(f) \cup O(f)$.

When we consider algorithms, we will be interested in finding algorithms that might do better than the one we are considering. So, finding one that is in big theta (in other words, is of the same complexity) is not very interesting.



$f(n) = \theta(g(n))$
Fig. 1.10.

Thus asymptotic notation gives us a way to express their relationship.

- If $f(n)$ is $O(g(n))$ this means f(n) grows no faster than g(n).
- If $f(n)$ is $\Omega(g(n))$ this means f(n) grows no slower than g(n).
- If $f(n)$ is $\theta(g(n))$ this means f(n) and g(n) grow at the same rate.

EXAMPLE 1.2. Suppose that it is known that running time of one algorithm is always N log N and the running time of another algorithm is always about N^3. What can you say about the relative performance of the algorithm?

Solution: The rate of growth of algorithms are shown in the given Fig. 1.11.

It can be seen that number of comparisons required in N logN is less than N^3. Thus, it can be seen that N logN is better than N^3. N^3 running time shows that the running time is of linear equation whereas N logN is the recursive equation and the running time of the recursive function depends upon divide and conquer technique. We know divide and conquer technique is always better than the linear equation. So, the performance of N logN is always better than the N^3.

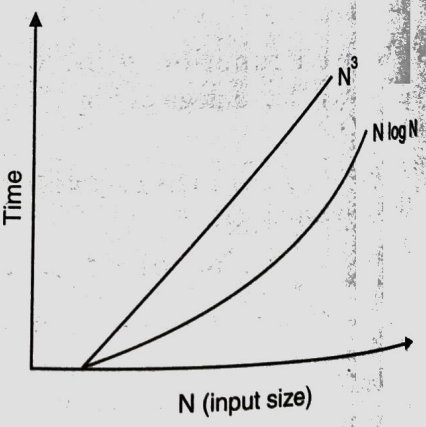


Fig. 1.11.

1.14 ABSTRACT DATA TYPES (ADT)

An Abstract Data Type (ADT) is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. We can think of an *Abstract Data Type* (ADT) as a mathematical model with a collection of operations defined on that model. Sets of integers, together with the operations of union, intersection, and set difference, form a simple example of an ADT. So, a useful tool for specifying the logical properties of a data type is abstract data type.

An Abstract Data Type (ADT) is the specification of the data type which specifies the logical and mathematical model of the data type.