```
    strcopy (msg, "Able Was I") ;
    printf("The message now is %s \n", msg) ;
    msg = (char *) realloc (msg, 50) ;
    strcopy(msg, "Ere I Saw Elba....") ;
    printf("\n The message is now %s", msg) ;
    free(msg) ;
    getch( ) ;
}
```

The output of the above program is :

```
The message is now Able Was I
The message is now Ere I Saw Elba...
```


1.6 ALGORITHMS

An algorithm named after ninth century mathematician Al-Khowarizmi, is defined as follows :

- An algorithm is a set of rules for carrying out calculations either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an obstruction of a program to be executed on a physical machine.

The most famous algorithm in history dates from well before the time of the ancient Greeks : this is Euclids algorithm for calculating the greatest common divisors of *two* integers.

The Classical Multiplication Algorithm

1. *Multiplication, the American way.* Multiply the multiplicand one after another by each digit of the multiplier taken from right to left.

```
          9 8 1
        1 2 3 4
-----------------
        3 9 2 4
      2 9 4 3
    1 9 6 2
    9 8 1
-----------------
1 2 1 0 5 5 4
```

2. *Multiplication, the English way.* Multiply the multiplicand one after another by each digit of the multiplier taken from left to right

```
          9 8 1
        1 2 3 4
-----------------
    9 8 1
    1 9 6 2
      2 9 4 3
        3 9 2 4
-----------------
1 2 1 0 5 5 4
```

Algorithm is a branch of Computer Science that consists of designing and analyzing computer algorithms :

1. The "design" pertains to :
   (*i*) the description of algorithm at an abstract level by means of a pseudo language and
   (*ii*) Proof of correctness that is, the algorithm solves the given problem in all cases.
2. The "analysis" deals with performance evaluation (complexity analysis).

Algorithm is of *two* different types :

- Incremental Algorithm
- Divide and Conquer Algorithm

1.6.1 Incremental Algorithm

Insertion sort uses an incremented 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]**.

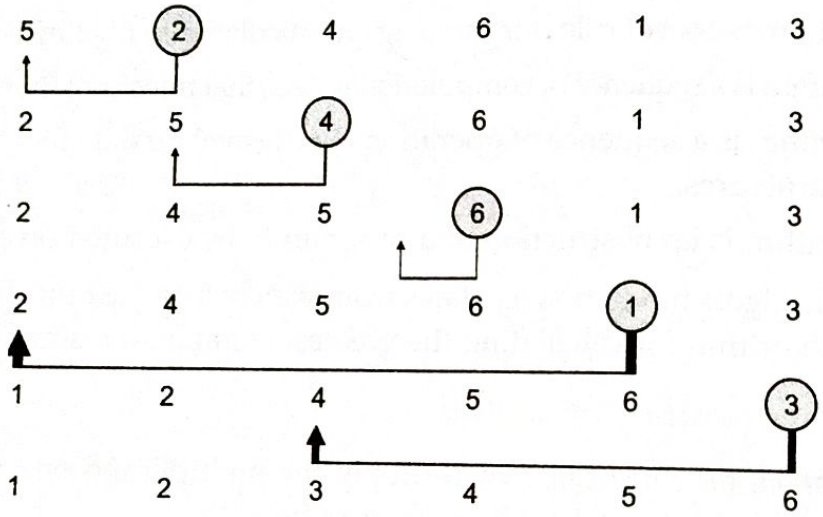

Figure 1.7 Insertion Sort

1.6.2 Divide and Conquer Algorithm

Many useful algorithms are recursive in structure, to solve a given problem, they call them solve, recursive one or more time to deal with closely related subproblems. These algorithms follow a divided & conquer approach.

They break the problem into several sub problems that are similar to original problem but smaller in size, solve the sub problem recursively, and then combine these solutions to create a solution to the original problem.

Divide : the problem into a number of sub problems.

Conquer : the sub problem by solving them recursively

Combine : the solutions to the sub problem into the solution for the original problem

$$A = \{5, 2, 4, 6, 1, 3, 2, 6\}$$

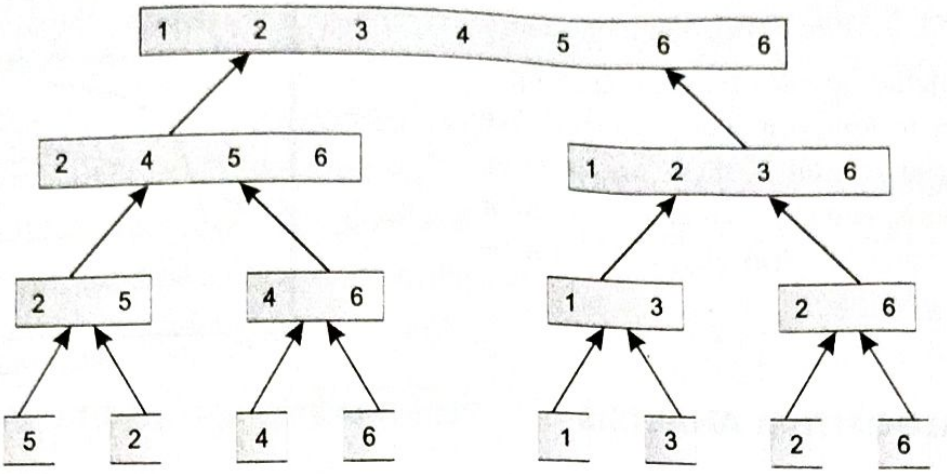

Figure 1.8 Merge Sort

1.7 ALGORITHM PERFORMANCE

The performance evaluation of an algorithm is obtained by totalling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size **n** and is to be considered modulo a multiplicative constant.

The following notations are commonly used in performance analysis and used to characterize the complexity of an algorithm :

θ-Notation (Tightly Bound)

This notation bounds a function to within constant factors. We say $f(n) = \theta(g(n))$ if there exists positive constant n_0, C_1 and C_2 such that to the right of n_0 the value of $f(n)$ always lies between $C_1\ g(n)$ and $C_2\ g(n)$ inclusive

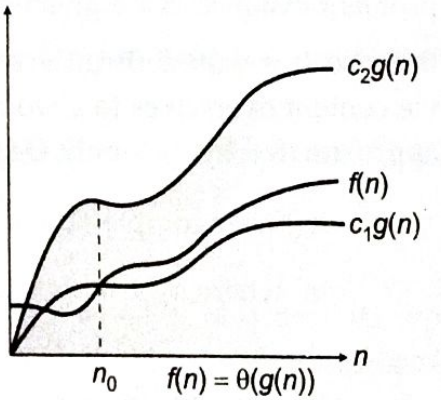

Figure 1.9

O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and C such that to the right of n_0, the value of $f(n)$ always lies on or below $Cg(n)$.

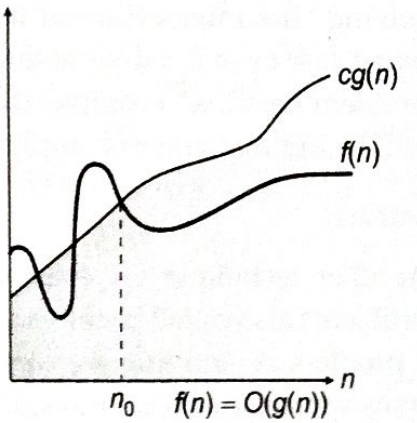

Figure 1.10

Ω –Notation (Lower Bound)

This notation gives a Lower bound for a function to within a constant factor. We write $f(n) = \Omega\, g(n)$ if there are positive constants n_0 and C such that to the right of n_0, the value of $f(n)$ always lies on or above $Cg(n)$.

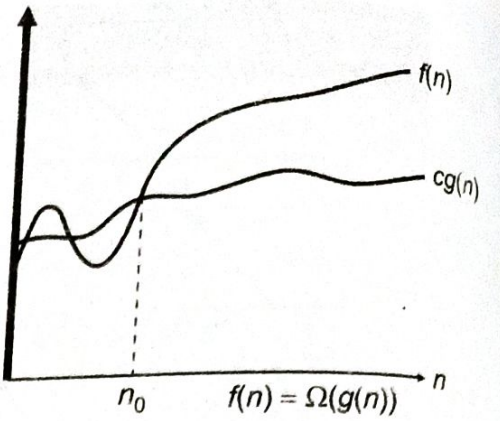

Figure 1.11

1.8 ALGORITHM ANALYSIS

The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operations (or running time) performed by an algorithm when the input size is n. There are two interpretators of upper bound.

Worst Case Complexity

The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.

Average Case Complexity

The running time for any given size input will be the average number of operations over all problem instances for a given size.

Because, it is quite difficult to estimate the statistical behaviour of the input, most of the time we content overselves to a worst case behaviour. Most of the time, the complexity of $g(n)$ is approximated by its family $O(f(n))$ where $f(n)$ is one of the following functions :

n (linear complexity), $\log n$ (logrithmic complexity),

n^a where $a \geq 2$ (polynomial complexity), a^n (exponential complexity).

Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the logarithm's solving this problems. For example, any algorithm solving "**the intersection of n segments**" problem will execute at least n^2 operations in the worst case even if it does nothing but prints the output. This is abbreviated by saying that the problem has $\Omega(n^2)$ complexity. If one finds an $O(n^2)$ algorithm that solves this problem, it will be optimal and of complexity $\theta(n^2)$

Reduction

Another technique for estimating the complexity of a problem is the transformation of problems also called problem reduction. As an example, suppose we know a lower bound for a problem A, and that we would like to estimate a lower bound for a problem B. If we can transform A into B by transformation step whose cost is less that for solving A into B by a transformation step whose cost is less than that for solving A, then B has the same bound as A.

1.9 Categories of Algorithms

Based on Big Oh notation, the algorithms can be categorized as follows :

- Constant time ($O(1)$) algorithms.
- Linear time ($O(n)$) algorithms.
- Logrithmic time ($O(\log n)$) algorithms.
- Polynomial time ($O(n^k)$, for $k > 1$ algorithms.
- Exponential time ($O(k^n)$, for $k > 1$) algorithms.

Many algorithms are $O(n \log n)$.

1.10 DATA STRUCTURE OPERATIONS

The data which is stored in our data structures are processed by certain set of operations. While selecting a particular data structure for the application we choose a given data structure largely on the frequency with which specific operations are performed. The following operations we can perform on the data structures :

(*i*) ***Traversing.*** Accessing each data exactly once in the data structure so that each data item is traversed or visited.

(*ii*) ***Searching.*** Finding the location of data within the data structure which satisfy the searching condition or the criteria.

(*iii*) ***Inserting.*** Adding a new data in the data structure is referred as Insertion.

(*iv*) ***Deleting.*** Removing a data from the data structure is referred as deletion.

(*v*) ***Sorting.*** Arranging the data in some logical order for example, in numerical increasing order or alphabetically.

(*vi*) ***Merging.*** Combining the data of two different sorted files into a single sorted file.

1.11 ABSTRACT DATA TYPES

A useful tool for specifying the logical properties of a data type is the abstract data type or ADT. Fundamentally, a data type is a collection of values and a set of operations on those values. That collection & those operations form a mathematical collection that may be implemented using a particular hardware or software data structure. The term "abstract data type" refers to the basic mathematical concept, that defines the data type.

Example

How might we actually implement fraction data type.

Implementation 1 :

```
type def strict { int numerator, denomination ; fraction ; }
main( )
{
      fraction f ;
      f.numerator = 1 ;
      f.denominator = 2 ;
      .......
}
```