ALGORITHM SPECIFICATION

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

All algorithms must satisfy the following criteria:

Input. An algorithm has zero or more inputs, taken from a specified set of objects.

Output. An algorithm has one or more outputs, which have a specified relation to the inputs.

Definiteness. Each step must be precisely defined; Each instruction is clear and unambiguous.

Finiteness. The algorithm must always terminate after a finite number of steps.

Effectiveness. All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

We can describe an algorithm in many ways.

1. Natural language: use a natural language like English
2.Flow charts: Graphic representations called flowcharts , only if the algorithm is small and simple.
3.Pseudo-code: also avoids most issues of ambiguity; no particularity on syntax programming language

Example 1: Algorithm for calculate factorial value of a number:

input a number n
set variable final as 1
final <= final * n
decrease n
check if n is equal to 0
if n is equal to zero, goto step 8 (break out of loop)
else goto step 3
print the result final

Example 2 Binary search : Assume that we have n ~ 1 distinct integers that are already sorted and stored in the array list. That is, list [0] ~ list [1] ~ ... ~ list [n-l]. We must figure out if an integer searchnum is in this list. If it is we should return an index, i, such that list[i] = searchnum. If searchnum is not present, we should return -1. Since the list is sorted we may use the following method to search for the value.
Let left and right, respectively, denote the left and right ends of the list to be searched. Initially, left = 0 and right = n-1. Let middle = (left+right)/2 be the middle position in the list. If we compare list [middle] with searchnum, we will get one of three results:

searchnum < list[middle]. In this case, if searchnum is present, it must be in the positions between 0 and middle - 1. Therefore, we set right to middle - 1.
searchnum =list[middle]. In this case, we return middle.

searchnum > list[middle]. In this case, if searchnum is present, it must be in the positions between middle + 1 and n - 1. So, we set left to middle + 1.

If searchnum has not been found and there are still integers to check, we recalculate middle and continue the search.Below Program implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing searchnum to list [middle].

```
while (there are more integers to check )
middle = (left + right) / 2;
if (searchnum < list [middle] )
        right = middle - 1;
else if (searchnum == list[middle])
        return middle;
        else left = middle + 1;
```

Recursive Algorithms

A recursive algorithm calls itself which usually passes the return value as a parameter to the algorithm again. This parameter is the input while the return value is the output.

Recursive algorithm is a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is the input for the next recursion. The repletion is in the self-similar fashion. The algorithm calls itself with smaller input values and obtains the results by simply performing the operations on these smaller values. Generation of factorial, Fibonacci number series are the examples of recursive algorithms.

Example: Writing factorial function using recursion

```
int factorial(int n)
{
        return n * factorial(n-1);
}
```

DATA ABSTRACTION

Definition: An abstract data type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the opera-tions.D

The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementa-tion details. Abstract data type is implementation-independent.

Example [Abstract data type NaturaINumber]: As this is the first example of an ADT, we will spend some time explaining the notation. ADT 1.1 contains the ADT definition of NaturalNumber.

ADT NaturalNumber is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (lNT-MAX) on the computer
functions:

for all x, Y E NaturalNumber; TRUE, FALSE E Boolean and where +, -, <, and == are the usual integer operations

NaturalNumber Zero() o
Boolean IsZero(x) if (x) return FALSE

else return TRUE
Boolean Equal(x, y) if (x == y) return TRUE
else return FALSE
NaturalNumber Successor(x) if (x == INT-MAX) return x
else return x + 1
NaturalNumber Add(x, y) if «x + y) <= INT-MAX) return x + y

else return INT-MAX

NaturalNumber Subtract(x, y) if (x < y) return 0

else return x - y

end NaturalNumber

ADT : Abstract data type NaturalNumber

The ADT definition begins with the name of the ADT. There are two main sections in the definition: the objects and the functions.
The objects are defined in terms of the integers, but we make no explicit reference to their representation.
First, the definitions use the symbols x and y to denote two elements of the data type NaturalNumber, while TRUE and FALSE are elements of the data type Boolean. In addition, the definition makes use of functions that are defined on the set of integers, namely, plus, minus, equals, and less than. This is an indication that in order to define one data type, we may need to use operations from another data type. For each function, we place the result type to the left of the function name and a definition of the function to the right. The symbols "::=" should be read as "is defined as."

The first function, Zero, has no arguments and returns the natural number zero. This is a constructor function. The function Successor(x) returns the next natural number in sequence. This is an example of a transformer function. Notice that if there is no next number in sequence, that is, if the value of x is already INT-MAX, then we define the action of Successor to return INT-MAX. Some programmers might prefer that in such a case Successor return an error flag. This is also perfectly permissible. Other transformer functions are Add and Subtract. They might also return an error condition, although here we decided to return an element of the set NaturalNumber. 0

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as

A priori analysis − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.

A posterior analysis − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

Time Factor − The time is measured by counting the number of key operations such as comparisons in sorting algorithm

Space Factor − The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

1)Space Complexity:

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components :

A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.

A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I) Where C is the fixed part and S(I) is the variable part of the algorithm which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop

Here we have three variables A, B and C and one constant. Hence S(P) = 1+3. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

2)Time Complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c*n, where c is the time taken for addition of two bits. Here, we observe that T(n) grows linearly as input size increases.

Asymptotic Analysis:

Asymptotic analysis of an algorithm, refers to defining the mathematical framing of its run-time performance. Using asymptotic analysis, we can conclude the best case, average case and worst case scenario of an algorithm.

Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases.

Usually, time required by an algorithm falls under three types −

Best Case − Minimum time required for program execution.

Average Case − Average time required for program execution.

Worst Case − Maximum time required for program execution.

Asymptotic Notations:

Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

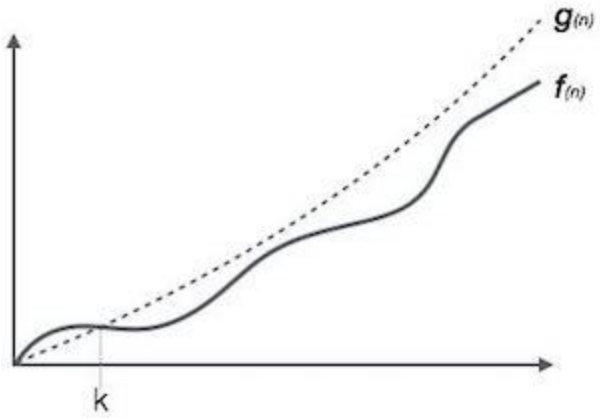
O(big O) Notation

Ω (Omega)Notation

θ (Theta)Notation

Big Oh Notation, O:

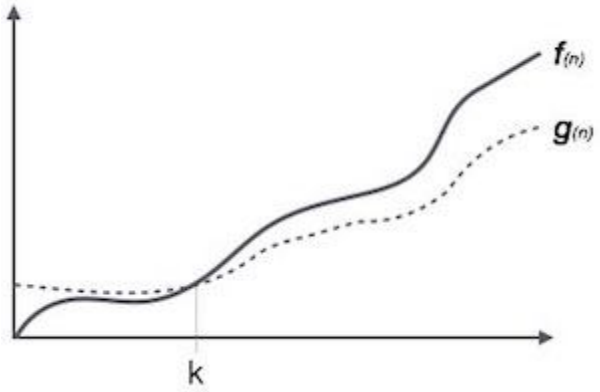
The O(n) is the way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.



for two functions f(n) and g(n)
O(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

Omega Notation, Ω:

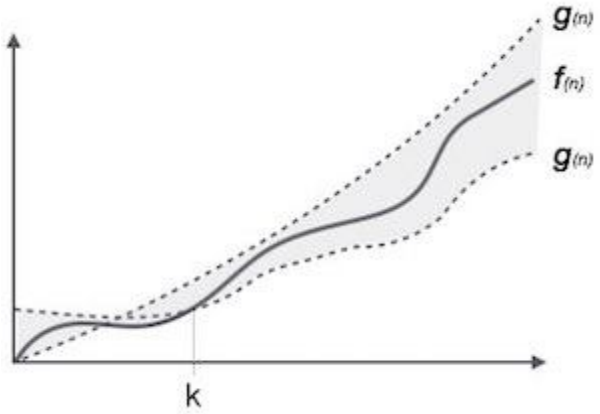
The Ω(n) is the way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.



for two functions f(n) and g(n)
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

Theta Notation, θ:

The θ(n) is the way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following −



for two functions f(n) and g(n) ,

$\theta(f(n))$ = { g(n) if and only if g(n) = $O(f(n))$ and g(n) = $\Omega(f(n))$ for all n > n0. }

Common Asymptotic Notations

Constant	−	O(1)
Logarithmic	−	O(log n)
Linear	−	O(n)
n log n	−	O(n log n)
Quadratic	−	$O(^{n2})$
Cubic	−	$O(^{n3})$
Polynomial	−	nO(1)
Exponential	−	$O(2^n)$

SEARCHING TECHNIQUES

1.Linear Search
2.Binary Search

Linear Search:

Linear search is a very simple search algorithm. Every items is checked and if a match founds then that particular item is returned otherwise search continues till the end of the data collection.

Algorithm

Linear Search (Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Binary Search:

This search algorithm works on the principle of divide and conquers. For this algorithm to work properly the data should be in sorted form.
Binary search a particular item by comparing the middle most item of the collection. If match occurs then index of item is returned. If middle item is greater than item then item is searched in sub-array to the right of the middle item otherwise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of subarray reduces to zero.

The pseudocode of binary search algorithm should look like this −

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value ot be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found

      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint

   end while

end procedure
```

SORTING TECHNIQUES

1) **Bubble sort:**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n are no. of items.

Algorithm

assume list is an array of n elements. Assume that swapfunction, swaps the values of given array elements.

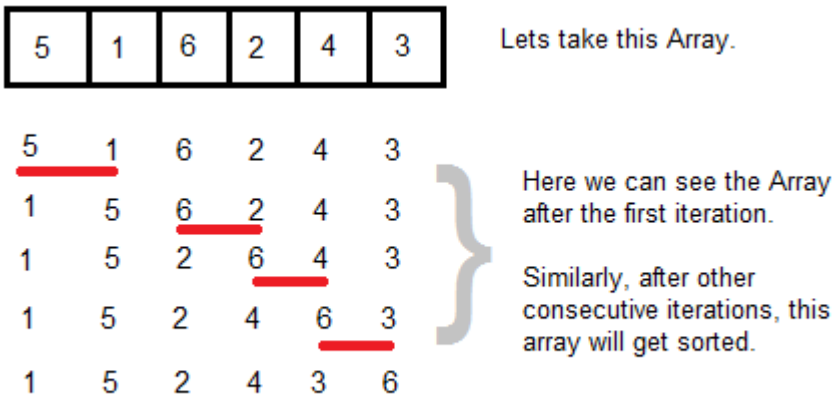
```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Example: {5, 1, 6, 2, 4, 3}



2)Insertion Sort:

The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array). This algorithm is not suitable for large data sets.
Here, a sub-list is maintained which is always sorted. A element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and insert it there. Hence the name insertion sort.

Algorithm

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Example: sort the sequence {3, 7, 4, 9, 5, 2, 6, 1}.

3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 5 7 9 2 6 1
2 3 4 5 7 9 6 1
2 3 4 5 6 7 9 1
1 2 3 4 5 6 7 9

3)Selection Sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

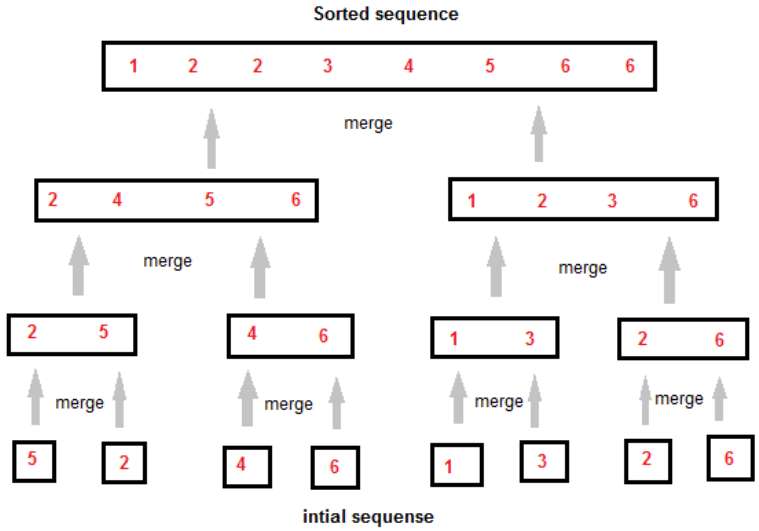
Example: 64 25 12 22 11
11 64 25 12 22
11 12 64 25 22
11 12 22 64 25
11 12 22 25 64

4)Merge Sort:

Merge sort is a sorting technique based on divide and conquer technique. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then merge sort combines smaller sorted lists keeping the new list sorted too.
Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.



5) Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than specified value say pivot based on which the partition is made and another array holds values greater than pivot value.
The quick sort partitions an array and then calls itself recursively twice to sort the resulting two sub array. This algorithm is efficient for large sized data sets as its average and worst case complexity are of O(nlogn) where n are no. of items.

The pivot value divides the list in to two parts. And recursively we find pivot for each sub-lists until all lists contains only one element.

QuickSort Pivot Algorithm

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

6)Radix Sort:

Radix Sort is an algorithm that sorts a list of numbers and comes under the category of distribution sort.
This sorting algorithm doesn't compare the numbers but distributes them, it works as follows:

1. Sorting takes place by distributing the list of number into a bucket by passing through the individual digits of a given number one-by-one beginning with the least significant part. Here, the number of buckets are a total of ten, which bare key values starting from 0 to 9.
2. After each pass, the numbers are collected from the buckets, keeping the numbers in order
3. Now, recursively redistribute the numbers as in the above step '1' but with a following reconsideration: take into account next most significant part of the number, which is then followed by above step '2'.