Data Structures and Algorithms Analysis

- A program is written in order to solve a problem. A solution to a problem actually consists of two things:
  - A way to organize the data
  - Sequence of steps to solve the problem
- Given a problem, the first step to solve the problem is obtaining one's own abstract view, or *model, of the problem*. This process of modeling is called ***abstraction***.
- **Abstraction** is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.
- **Algorithms**
- An algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output.

Properties of an algorithm

- **Finiteness**: Algorithm must complete after a finite number of steps.
- **Definiteness**: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.
- **Sequence**: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Feasibility**: It must be possible to perform each instruction.

- **Correctness**: It must compute correct answer for all possible legal inputs.
- **Language Independence**: It must not depend on any one programming language.
- **Completeness**: It must solve the problem completely.
- **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.
- **Efficiency**: It must solve with the least amount of computational resources such as time and space.
- **Generality**: Algorithm should be valid on all possible inputs.
- **Input / Output**: There must be a specified number of input values, and one or more result values.

Algorithm Analysis Concepts

- Algorithm analysis refers to the process of determining the amount of computing time and storage space required by different algorithms
- It's a process of predicting the resource requirement of algorithms in a given environment.
- In order to solve a problem, there are many possible algorithms. One has to be able to choose the best algorithm for the problem at hand using some scientific method.
- To classify some data structures and algorithms as good, we need precise ways of analyzing them in terms of resource requirement
- **The main resources are**:
  - Running Time

- Memory Usage
- Communication Bandwidth

- There are **two approaches** to measure the efficiency of algorithms:
  - **Empirical**: Programming competing algorithms and trying them on different instances (total running time of the program).
  - **Theoretical**: Determining the quantity of resources required mathematically (Execution time , memory space, etc.) needed by each algorithm.
- However, it is difficult to use actual clock-time as a consistent measure of an algorithm's efficiency, because clock-time can vary based on many things. For example,
  - Specific processor speed
  - Current processor load
  - Specific data for a particular run of the program
  - Input Size
  - Input Properties
  - Operating Environment
- Accordingly, we can analyze an algorithm according to the number of operations required, rather than according to an absolute amount of time involved. This can show how an algorithm's efficiency changes according to the size of the input.

Complexity Analysis

- **Complexity Analysis** is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.
- There are two things to consider:
  - **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
  - **Space Complexity**: Determine the approximate memory required to solve a problem of size n.
- Complexity analysis involves two distinct phases:
  - **Algorithm Analysis**: Analysis of the algorithm or data structure to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.
  - **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs.
- There is no generally accepted set of rules for algorithm analysis.

Analysis Rules:

1 .We assume an arbitrary time unit.

2. Execution of one of the following operations takes time 1:

- Assignment Operation
- Single Input / Output Operation
- Single Boolean Operations
- Single Arithmetic Operations

- Function Return

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loops: Running time for a loop is equal to the running time for the statements inside the loop * number of iterations.
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body. ***Examples 1:*** int x=10; x=x+10; cout<< x;

Time Units to Compute

- ✓ 1 for the assignment statement: int x=10;
- ✓ 1 for the single arithmetic statement: x +10
- ✓ 1 for the assignment statement: x=x+10;
- ✓ 1 for the output statement: cout<< x; **T (n)= *1+1+1+1 = 4***

Examples 2:

for(i=1; i<=4; i++) cout<<i;

Time Units to Compute

- ✓ 1 for the initialization expression: i=1 ;
- ✓ 5 for the test expression: i<=4 ;
- ✓ 4 for the increment expression: i++ ;

✓ 4 for the output statement: cout<<i;

T (n)= *1+5+4+4 = 14*

Examples 3:

for(i=1; i<=n; i++) cout<<i;

Time Units to Compute

- 1 for the initialization expression: i=1 ;
- n+1 for the test expression: i<= n;
- n for the increment expression: i++ ;
- n for the output statement: cout<<i;

T (n)= *1+ n+1+n+n = 3n +2*

- Some times to find Order of Magnitude of an Algorithm is a bit complex. However, it can be simplified by using some formal approach
- **for Loops:**
  - In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

Example

for (int i = 1 ; i < = n ; i + +) cout<<i ; *n*

$\sum_{i=}^{n} 1 = n$ =**O(n)**

- **Nested Loops**
  - Nested for loops translate into multiple summations, one for each for loop.

Example

```
for ( int i = 1 ; i < = n ; i+ + ) for ( int j = 1 ; j < = n ; j+ + )
        cout<<1;
```

$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = nn$ = **O(n^2)**

- **for Loops:**
  - If the increment expression changes.

Example

```
for ( int  i = 1 ; i < = n ; i =i*2 )
    cout<<i ; logn
```

$\sum_{i=0}^{logn} 1 = logn$ = **O($\log_2 n$)**

Measures of Times

- In order to determine the running time of an algorithm it is possible to define three functions *Tbest(n)*, *Tavg(n)* and *Tworst(n)* as the best, the average and the worst case running time of the algorithm respectively.
  - **Average Case** (Tavg): The amount of time the algorithm takes on an "average" set of inputs.
  - **Worst Case** (Tworst): The amount of time the algorithm takes on the worst possible set of inputs.

- **Best Case** (Tbest): The amount of time the algorithm takes on the smallest possible set of inputs.

Asymptotic Analysis

- **Asymptotic analysis** is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.
- There are five notations used to describe a running time function. ***These are:***
  - Big-Oh Notation (O)
  - Big-Omega Notation (Ω)
  - Theta Notation (Θ)
  - Little-o Notation (o)
  - Little-Omega Notation (ω)

The Big-Oh Notation

- **Big-Oh notation** is a way of comparing algorithms and is used for computing the complexity of algorithms; i.e., the amount of time that it takes for computer program to run.
- It is only concerned with what happens for very a large value of n.
- For example, if the number of operations in an algorithm is $\mathbf{n^2 - n}$, **n** is insignificant compared to $\mathbf{n^2}$ for large values of n. Hence the n term is ignored.
- **Big-Oh** is mainly concerned with large values of **n**.
- **Big-O** expresses an upper bound on the growth rate of a function, for sufficiently large values of **n**.

- Formal Definition: f (n)= O (g (n)) if there exist c, k ε $\mathcal{R}$+ such that for all n≥ k, f (n) ≤ c.g (n).

1 . f(n)=10n+5 and g(n)=n. Show that f(n) is O(g (n)).

To show that f(n) is O(g(n)) we must show that constants c and k such that f(n) <=c.g(n) for all n>=k

or 10n+5<=c.n for all n>=k

Try c=15. Then we need to show that 10n+5<=15n Solving for n we get: 5<5n or 1<=n.

So f(n) =10n+5 <=15.g(n) for all n>=1.

Each of the following functions is big-O of its successors:

➢ K ➢ $\log_b n$ ➢ n ➢ $n\log_b n$	➢ n2 ➢ n to higher powers ➢ 2n ➢ 3n ➢ larger constants to the nth power ▪ n! ▪ nn

- **Example**

T(n)= $3n\log_b n + 4\log_b n+2$ is $O(n\log_b n)$

T(n)= $3n\log_b n + 4\log_b n+2n^2$ is $O(n^2)$

T(n)= $3n! + 4\log_b n+2^n$ is O(n!)

Big-Omega Notation``

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

- Formal Definition: A function f(n) is Ω(g (n)) if there exist constants c and k ε $\mathcal{R}$+ such that f(n) >=c. g(n) for all n>=k.
- f(n)= Ω(g (n)) means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.
- **Example: If f(n) =n^2, then f(n)= Ω(n)**
- In simple terms, f(n)= Ω(g (n)) means that the growth rate of f(n) is greater that or equal to g(n).

Theta Notation

- A function f (n) belongs to the set of Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.
- Formal Definition: A function f (n) is Θ (g(n)) if it is both O(g(n)) and Ω(g(n)). In other words, there exist constants c1, c2, and k >0 such that c1.g(n) <= f(n) <= c2.g(n) for all n >= k
- If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n).

Little-o Notation

- Big-Oh notation may or may not be asymptotically tight, for example:
  - $2n^2 = O(n^2)$
    $= O(n^3)$
- f(n)=o(g(n)) means for all c>0 there exists some k>0 such that f(n) < c.g(n) for all n>=k.
- as n increases g(n) grows faster than f(n)

- computes non-tight upper bound of f(n)

Example: f(n)=3n+4 is o(n^2)

- In simple terms, f(n) has less growth rate compared to g(n).
- f(n)= $2n^2$ f(n) =o(n^3), O(n^2), f(n) is not o(n^2).

Little-Omega (ω notation)

- Little-omega (ω) notation is to big-omega (Ω) notation as littleo notation is to Big-Oh notation. We use ω notation to denote a lower bound that is not asymptotically tight.
- Formal Definition: f(n)= ω (g(n)) if there exists a constant n>0 such that 0 < = c. g(n) < f(n) for all n >= k.
- Example: f(n)= $2n^2$ ω(n) but it is not ω (n^2).
- As n increases f(n) grows faster than g(n)
- computes non-tight lower bound of f(n)