Unit I

- **Algorithms-** Problem Solving, Introduction to Algorithms, Characteristics of algorithms, Algorithm design tools: Pseudo code and flowchart, Analysis of Algorithms, Complexity of algorithms- Space complexity, Time complexity, Asymptotic notation- Big-O, Theta and Omega, standard measures of efficiency.

Algorithm

- An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
- All algorithms must satisfy the following criteria:
  - **Input :** Zero or more quantities are externally supplied.
  - **Output :** At least one quantity is produced.
  - **Definiteness :** Each instruction is clear and unambiguous.
  - **Finiteness :** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
  - **Effectiveness :** Every instruction must be very basic and effective. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Uses of Algorithm

- It gives a language independent layout of the program.
- So using the basic layout we can develop the program in any desired language.
- Algorithm representation is very easy to understand.
- It facilitates or makes coding easy.

Develop an algorithm to find maximum of 10 numbers.

1. Start
2. Let count = 1
3. Read a number, say x
4. Let max = x
5. Check if count is equal to 10 , if yes go to Step 10
6. Read a number, say x
7. Add 1 to count
8. Check if x is greater than max, if yes Set max = x
9. go to step 5
10. Display max
11. Stop.

Flow chart

- A flow chart is a pictorial representation of an algorithm.
- An algorithm is first represented in the form of flowchart and flowchart is then expressed in some programming language to prepare a computer program. Since, flowcharts shows the flow of operations in diagrammatic form, any error in the logic of the procedure can be detected more easily than in case of a program. Once flowchart is ready it is very easy to write a program in terms of statements of a programming language

Pseudo code

Pseudo code is nothing but an informal way of writing a program. It is a combination of algorithm written in simple English and some programming language. In pseudo code, there is no restriction of following the syntax of the programming language. Pseudo codes can not be compiled. It is just a previous step of developing a code for given algorithm. Even sometime by examining the pseudo code one can decide which language has to select for implementation.

- e.g The following algorithm (Psuedo C code) finds and returns the maximum of n numbers;

```
Algorithm Max (A,n)
/*A is an array of size n*/
{
        Assume max as A[0] ;
        for( i = 0 ; i < n ;  i = i +1 )
        {
                if  A[i] > max  then Set max to A[i] ;
       }
       return max;
}
```

The study of algorithm involves four main areas

1. How to devise algorithms
2. How to validate algorithms
3. How to analyze algorithms
4. How to test a program

1. How to devise algorithms

This area has attracted many because the answer to the question cannot be put in algorithmic form. Creating an algorithm is an art which may never be fully automated. We can Study the various design techniques that have proven to be useful in that they have often yielded good algorithm. By mastering these design strategies, it will become easier to devise new and useful algorithms.

2. How to validate algorithms

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as algorithm validation. The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program proving or sometimes as program verification.

3. How to analyze algorithms

Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires. The efficiency is measured in best case, worst case and average case.

4. How to test a program

- Testing a program consists of two phases:
  - Debugging : It is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
  - Profiling : Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the result.

Program development cycle

- Feasibility Study
- Requirement Analysis and problem specifications
- Design
- Implementation / Coding
- Debugging
- Testing
- Maintenance
- Documentation

Performance Analysis

There are many criteria upon which we can judge an algorithm. For instance:

- Does it do what we want to do?
- Does it work correctly according to the original specifications of the task?
- Is there documentation that describes how to use it and how it works?
- Are procedures created in such a way that they perform logical subfunctions?
- Is the code readable?

There are other criteria for judging algorithms that have a more direct relationship to performance . These have to do with their computing time and storage requirements.

- **Space Complexity** : The space complexity of an algorithm is the amount of memory it needs to run to completion.
- **Time Complexity** : The time complexity of an algorithm is the amount of computer time it needs to run to completion.
- Performance evaluation can be loosely divided into two major phases :
  - A Priori estimates and
  - A posteriori testing
- We refer to these as performance analysis and performance measurement respectively.

Space Complexity

The space needed by any algorithm is seen to be the sum of following components :

- A **fixed part** that is independent of the characteristics (e.g. number,size) of the inputs and outputs. This part typically includes the instruction space (i.e. space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants, and so on…
- A **variable part** that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space.
- The space requirement S(P) of any algorithm P may therefore be written as S(P) = c + Sp(instance characteristics), where c is a constant.

Time Complexity

The time T(P) taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This runtime is denoted by tp(instance characteristics).

Analysis Basics

- Many algorithms that can solve a given problem.
- Will have different characteristics that will determine how efficiently each will operate.
- When we analyze an algorithm, we first have to show that the algorithm does properly solve the problem because if it doesn't, its efficiency is not important.
- Next, we look at how efficiently it solves the problem.

- Analyzing an algorithm determines the amount of “time” that algorithm takes to execute.
- This is not really a number of seconds or any other clock measurement but rather an approximation of the number of operations that an algorithm performs.

Algorithms classification

- Repetitive algorithms
  - Have loops and conditional statements as their basis, and so their analysis will entail determining the work done in the loop and how many times the loop executes
- Recursive algorithms
  - Solve a large problem by breaking it into pieces and then applying the algorithm to each of the pieces.
  - Analyzing will entail determining the amount of work done to produce the smaller pieces, and then putting their individual solutions together to get the solution to the whole problem.
  - Combining this information with the number of the smaller pieces and their sizes, we can produce a recurrence relation for the algorithm.
  - This recurrence relation can then be converted into a closed form that can be compared with other equations.

What is analysis

- Analysis of an algorithm provides background information that gives us a general idea of how long an algorithm will take for a given problem set.
- Studying the analysis of algorithms gives us the tools to choose between algorithms.

Algorithm to find the largest of 4 nos

```
largest = a
if b > largest then
    largest = b
end if
if c > largest then
    largest = c
end if
if d > largest then
    largest = d
end if
return largest
```

Algorithm to find the largest of 4 nos

```
if a > b then
      if a > c  then
            if a > d then
                  return a
            else
                  return d
            end if
      else
            if c > d then
                  return c
            else
                  return d
            end if
      end if
else
   Continued
```

```
else
      if b > c  then
            if b > d then
                  return b
            else
                  return d
            end if
      else
            if c > d then
                  return c
            else
                  return d
            end if
      end if
end if
```

- The purpose of analysis of algorithms is not to give a formula that will tell us exactly how many seconds or computer cycles a particular algorithm will take.
- This is not useful information because we would then need to talk about the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, whether it has a complex or reduced instruction set processor chip, and how well the compiler optimizes the executable code.
- All of those will have an impact on how fast a program for an algorithm will run.
- To talk about analysis in those terms would mean that by moving a program to a faster computer, the algorithm would become better because it now completes its job faster.
- That’s not true, so , we do our analysis without regard to any specific computer.

What to count and consider

- Deciding what to count involves two steps
  - Choosing the significant operation or operations
  - Deciding which of those operations are integral to the algorithm and which are overhead or bookkeeping.
- Two classes of operations
  - Comparison
  - Arithmetic (additive , multiplicative operators)

Input classes

- Input plays an important role in analyzing algorithms because it is the input that determines what the path of execution through an algorithm will be.
- When looking at the input, we will try to break up all the different input sets into classes based on how the algorithm behaves on each set.

Cases to consider

- Choosing what input to consider when analyzing an algorithm can have a significant impact on how an algorithm will perform.
- We will actually look for those input set that allow an algorithm to perform the most quickly and most slowly.
- We will also consider an overall average performance of the algorithm as well.

Best Case

- Is the input that requires the algorithm to take the shortest time.
- This input is the combination of values that causes the algorithm to do the least amount of work.
- Because the best case for an algorithm will usually be a very small and frequently constant value, we will not do a best-case analysis very frequently.

Worst Case

- Important analysis because it gives us an idea of the most time an algorithm will ever take.
- Worst case analysis requires that we identify the input values that cause an algorithm to do the most work.
- The worst case gives us an upper bound on how slowly parts of our programs may work based on our algorithm choices.

Average Case

- Toughest to do because there are a lot of details involved.
- The basic process begins by determining the number of different groups into which all possible input sets can be divided.
- The second step is to determine the probability that the input will come from each of these groups.
- The third step is to determine how long the algorithm will run for each of these groups.

Average Case

- All of the input in each group should take the same amount of time ,and if they do not, the group must be split into two separate groups.
- When all of this has been done, the average case time is given by the following formula:

$$A(n) = \sum_{i=1}^{m} p_i * t_i$$

- Where
  - n : size of the input
  - m : no. of groups
  - p_i : probability that the input will be from group i
  - t_i : time that the algorithm takes for input from group i.

Frequency Count

- It is the number of times the instruction (Step in algorithm) is executed in program (Algorithm).
- Total Computational time for an instruction = Instruction execution time * Frequency count.
- Now instruction execution time depends upon
  - speed of processor (instruction per millisecond)
  - instruction set of the processor
  - time required to execute instruction
  - compiler for translating the instruction into machine language.
- All these factors independent of the algorithm and external to it. Hence performance of an algorithm (time complexity) is measured in terms of Frequency count. Hence only Frequency count is important in deciding the performance (time complexity) of an algorithm.

Stmt. No	Statement	s / e	Frequency	Total Steps
1	**Algorithm** Sum(a,n)	0	--	0
2	**{**	0	--	0
3	s = 0;	1	1	1
4	for i = 1 to n do	1	n+1	n + 1
5	s = s + a[i];	1	n	n
6	return s;	1	1	1
7	**}**	0	--	0
Total				**2n + 3**

Stmt. No	Statement	s / e	Frequency	Total Steps
1	**Algorithm** Add(a,b,c,m,n)	0	--	0
2	**{**	0	--	0
3	for i = 1 to m do	1	m+1	m+1
4	for j = 1 to n do	1	m(n+1)	mn+m
5	c[i,j] = a[i,j] + b[i,j];	1	mn	mn
6	**}**	0	--	0
Total				**2mn + 2m + 1**

Determine the frequency count for each statement in the following piece of code. Obtain its time complexity in terms of ‘n’

```
for(  i=1; i ≤ n ; i++)
{
    for(  j=1;  j ≤ i ; j++)
      {
          for(  k=1 ;  k  ≤ j  ;  k++)
            {
                  s = x + 1;
            }
      }
}
```

Determine the frequency count for each statement in the following piece of code. Obtain its time complexity in terms of ‘n’

```
i=0
while(i ≤ n)
{
        j = i;
        while( j  ≤ i +1 )
          {
                  a = a * a;
                  j ++;
          }
          i++;
}
```

Stmt. No	Statement	s / e	Frequency		Total Steps	
			n = 0	n > 0	n = 0	n > 0
1	**Algorithm** RSum(a,n)	0	--	--	0	0
2	**{**	0	--	--	0	0
3	if (n <= 0) then	1	1	1	1	1
4	return 0.0;	1	1	0	1	0
5	else	0	--	--	--	--
6	return RSum(a,n-1) + a[n];	1 + x	0	1	0	1 + x
7	**}**	0	--	--	0	0
Total					**2**	**2 + x**

Where x = tRsum(n-1)

Mathematical Background

$$\sum_{i=1}^{N} 1 = N$$

$$\sum_{i=1}^{N} C = C \times N$$

$$\sum_{i=1}^{N} i = N(N+1) / 2$$

$$\sum_{i=1}^{N} i^2 = N(N+1)(2N+1) / 6$$

Mathematical Background

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=1}^{N} A^i = (A^{N+1} - 1) \ / \ (A - 1)$$

$$\sum_{i=1}^{N} i\, 2^i = (N-1)\, 2^{N+1} + 2$$

$$\sum_{i=1}^{N} 1 / i = \ln N$$

Mathematical Background

$$\sum_{i=1}^{N} \lg i \approx N \lg N - 1.5$$

Asymptotic Notation (**O Ω Θ**)

[Big “oh”] : This notation is used to express an upper bound on computing time of an algorithm. When we say that time complexity of Selection sort algorithm is **$O(n^2)$** , means that for sufficiently large values of ‘n’ , computation time will not exceed some constant time * **n^2** i.e proportional to **n^2** (Worst Case).

[Big “oh”] **O**

- Definition : The function f(n) = O(g(n)) (read as “f of n is big oh of g of n”) iff (if and only if) there exist positive constants c and n0 such that f(n) <= c * g(n) for all n, n >= n0.
- **Example:**
- The function 3n+2 = O(n) as 3n + 2 <= 4n for all n >= 2
- The function 3n+3 = O(n) as 3n + 3 <= 4n for all n >= 3
- The function 100n+6 = O(n) as 100n + 6 <= 101n for all n >= 6
- The function $10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 <= 11n^2$ for all n >= 5
- The function $1000n^2+100n-6 = O(n^2)$ as $1000n^2+100n-6 <= 1001n^2$ for all n >= 100
- The function $6*2^n + n^2 = O(2^n)$ as $6*2^n + n^2 <= 7*2^n$ for all n >= 4

[Omega] **Ω**

This notation is used to express an lower bound on computing time of an algorithm. When we say that best case time complexity of insertion sort is **Ω (n)**, means that for sufficiently large values of ‘n’, minimum computation time will be some constant time * n i.e proportional to n.

[Omega] Ω

- Definition : The function f(n) = Ω(g(n)) (read as “f of n is omega of g of n”) iff there exist positive constants c and n0 such that f(n) >= c * g(n) for all n, n >= n0.
- Example:
- The function 3n+2 = Ω (n) as 3n + 2 >= 3n for all n >= 1
- The function 3n+3 = Ω (n) as 3n + 3 >= 3n for all n >= 1
- The function 100n+6 = Ω (n) as 100n + 6 >= 100n for all n >= 1
- The function $10n^2+4n+2$ = Ω (n^2) as $10n^2+4n+2$ >= n^2 for all n >= 1
- The function $6*2^n + n^2$ = Ω (2^n) as 6*2^n + n^2 >= 2^n for all n >= 1

[Theta] **Θ**

- This notation is used to express time complexity of an algorithm when it is same for worst & Best cases. For example best and worst case time complexities for selection sort is **$O(n^2)$** & **$\Omega(n^2)$** i.e. it can be expressed as **$\Theta(n^2)$**.
- Definition : The function f(n) = Θ(g(n)) (read as "f of n is theta of g of n") iff there exist positive constants c1, c2 and n0 such that c1 *g(n) <= f(n) <= c2 * g(n) for all n, n >= n0.

- Example:
- The function $3n+2 = \Theta(n)$ as $3n + 2 >= 3n$ for all $n >= 2$ and $3n + 2 <= 4n$ for all $n >= 2$.
- The function $3n+3 = \Theta(n)$
- The function $10n^2+4n+2 = \Theta(n^2)$
- The function $6*2^n + n^2 = \Theta(2^n)$

Recurrence Relations

- Recurrence relations can be directly derived from a recursive algorithm, but they are in a form that does not allow us to quickly determine how efficient the algorithm is.
- To do that we need to convert the set of recursive equations into what is called closed form by removing the recursive nature of the equations.
- This is done by a series of repeated substitutions until we can see the pattern that develops.
- The easiest way to see this process is by a series of examples.
- Examples :
  - $T(n) = 2\,T(n-2) - 15$

    $T(2) = 40$

    $T(1) = 40$
  - $T(n) = 4$ if $n \leq 4$

    $= 4\,T(n/2) - 1$ otherwise

Consider the following recurrence relation

– $T(n) = 2\,T(n-2) - 15$

$T(2) = 40$

$T(1) = 40$

- Substitution

$T(n-2) = 2\,T(n-2-2) - 15 = 2\,T(n-4) - 15$

$T(n-4) = 2\,T(n-6) - 15$

$T(n-6) = 2\,T(n-8) - 15$

$T(n-8) = 2\,T(n-10) - 15$

$T(n-10) = 2\,T(n-12) - 15$

Now we begin to substitute back into the original equation.

- $T(n) = 2T(n-2) – 15 = 2(2T(n-4) – 15) – 15$
- $T(n) = 4T(n-4) – 2*15 – 15$

- $T(n) = 4(2T(n-6) – 15) – 2 * 15 - 15$
- $T(n) = 8T(n-6) – 4 * 15 - 2*15 – 15$

- $T(n) = 8(2T(n-8) – 15) – 4 * 15 - 2*15 – 15$
- $T(n) = 16T(n-8) – 8*15 - 4 * 15 - 2*15 – 15$

- $T(n) = 16(2T(n-10) – 15) – 8*15 - 4 * 15 - 2*15 – 15$
- $T(n) = 32T(n-10) – 16*15 - 8*15 - 4 * 15 - 2*15 – 15$

- $T(n) = 32(2T(n-12)–15) – 16*15 - 8*15 - 4 * 15 - 2*15 – 15$
- $T(n) = 64T(n-12)–32*15 -16*15 - 8*15 - 4 * 15 - 2*15 – 15$

- How many times we have to substitute back into this equation to get to either of two values ?
- If n is even (2 = n – (n-2))
- This seems to indicate that we would substitute back into this equation [(n-2) / 2] – 1 times giving n/2 – 1 terms based on -15 in the equation, and the power of coefficient of T will be n/2 - 1.
- If n is odd, the only thing that would change is that T would have a value of 1 instead of 2, but by our equations, n/2 – 1 is 5 (not 6) when n is 13.
- For odd n, we will use n/2 instead of n/2 – 1.
- We will have two cases in our answer.

- $T(n) = 2^{(n/2)-1}\, T(2) - 15 \sum_{i=0}^{(n/2-1)} 2^i$ if n is even
- $T(n) = 2^{(n/2)-1} * 40 - 15 \sum_{i=0}^{(n/2-1)} 2^i$ if n is even
- $T(n) = 2^{(n/2)}\, T(1) - 15 \sum_{i=0}^{n/2} 2^i$ if n is odd
- $T(n) = 2^{(n/2)-1} * 40 - 15 \sum_{i=0}^{n/2} 2^i$ if n is odd

- Now applying the equation for an even value of n we get

$$T(n) = 2^{(n/2)-1} * 40 - 15* (2^{n/2} - 1)$$

$$= 2^{n/2} * 20 - 2^{n/2} *15 + 15$$

$$= 2^{n/2} (20 - 15) + 15$$

$$= 2^{n/2} * 5 + 15$$

- for an odd value of n we get

$$T(n) = 2^{n/2} * 40 - 15* (2^{n/2 + 1} - 1)$$

$$= 2^{n/2} * 40 - 2^{n/2} *30 + 15$$

$$= 2^{n/2} (40 - 30) + 15$$

$$= 2^{n/2} * 10 + 15$$

Consider the following recurrence relation

– $T(n) = 5$ if $n \leq 4$

$= 4\,T(n / 2) - 1$ otherwise

- Substitution

$T(n/2) = 4T(n/4) - 1$

$T(n/4) = 4T(n/8) - 1$

$T(n/8) = 4T(n/16) - 1$

$T(n/16) = 4T(n/32) - 1$

$T(n/32) = 4T(n/64) - 1$

Now we begin to substitute back into the original equation.

- $T(n) = 4T(n/2) – 1 = 4 (4 T(n/4) – 1) – 1$
- $T(n) = 16T(n/4) – 4*1 – 1$

- $T(n) = 16(4T(n/8) – 1) – 4 * 1 – 1$
- $T(n) = 64T(n/8) – 16 * 1 – 4 * 1 – 1$

- $T(n) = 64(4T(n/16) - 1) – 16 * 1 – 4 * 1 – 1$
- $T(n) = 256T(n/16) – 64*1 – 16 * 1 – 4 * 1 – 1$

- $T(n) = 256(4T(n/32)-1) – 64*1 – 16 * 1 – 4 * 1 – 1$
- $T(n) = 1024T(n/32) – 256 * 1 – 64*1 – 16 * 1 – 4 * 1 – 1$

- $T(n) = 1024(4T(n/64)-1) – 256*1 – 64*1 – 16*1 – 4*1 – 1$
- $T(n) = 4096T(n/64) – 1024*1– 256*1–64*1–16*1–4*1–1$

- We notice that the coefficient of -1 increases by a power of 4 each time we substitute, and it is the case that the power of 2 that we divide n by is 1 greater than the largest power of 4 for this coefficient.
- Also, we notice that the coefficient of T is the same power of 4 as the power of 2 that we divide n by.
- When we have T(n/2), its coefficient will be 4^i and we will have terms from $-(4^{i-1})$ to -1

- Now , for what value of i can we stop the substitution?
- Well, because we have the direct case specified for $n \leq 4$, we can stop when we get to $T(4) = T(n / 2^{\lg n - 2})$.
- Putting together we get.
- $T(n) = 4^{\lg n - 2} T(4) - \sum_{i=0}^{\lg n - 2} 4^i$

- Using the value for the direct case, and main eqn we get

$$T(n) = 4^{\lg n - 2} * 5 - (4^{\lg n - 2} - 1) / (4 - 1)$$

$$= 4^{\lg n - 2} * 5 - (4^{\lg n - 2} - 1) / 3$$

$$= (15 * 4^{\lg n - 2} - 4^{\lg n - 2} + 1) / 3$$

$$= (4^{\lg n - 2} * (15 - 1) + 1) / 3$$

$$= (4^{\lg n - 2} * 14 + 1) / 3$$