1. Introduction:

Algorithm: The word algorithm came from the name of a Persian mathematician Abu Jafar Mohammed Ibn Musa Al Khowarizmi (ninth century). An algorithm is simply s set of rules used to perform some calculations either by hand or more usually on a machine (computer).

Definition: An algorithm is a finite set of instructions that accomplishes a particular task. Another definition is a sequence of unambiguous instructions for solving a problem i.e, for obtaining a required output for any legitimate (genuine) input in a finite amount of time.

In addition all algorithms must satisfy the following criteria (characteristics).

1. Input: zero or more quantities are externally supplied as input.

Consider Fibonacci numbers program, here aim of the problem is to display ten Fibonacci numbers. No input is required; in the problem itself this is clearly mentioned as ten Fibonacci values. So zero items required for input.

Another problem is displaying given numbers of evens, so user should accept how many evens required. Based on the user input the number of evens is to be displayed. So, one data item is required as input.

2. Output: At least one quantity is produced by given algorithm as output.

In the case of Fibonacci numbers program after executing the program, first ten Fibonacci values displayed as output.

In second case, based on user input it should display given number of evens. An input of negative number is wrong, should display proper error message as output. So this program displays at least one output as error message, or number if outputs that show given number of steps.

3. Definiteness: Each instruction is clear and unambiguous i.e. each step must be easy to understand and convey only a single meaning.

4. Effectiveness: each instruction must be very basic, so that it can be carried out by a person using only pencil and paper.

This step is common in both Fibonacci and primes. For example, if user enters a negative numbers as input in evens, if you have a step like

Step: If N < 0 then

Go to ERROR

A wrong instruction given as go to ERROR, those kinds of instructions should not be there in an algorithm.

5. Finiteness: If we can trace out the instructions of an algorithm then for all cases, the algorithm terminate after a finite number of steps.

Either in the case of Fibonacci or even numbers problem should be solved in some number of steps. For example, continuous display or Fibonacci series without termination leads to abnormal termination.

Criteria for Algorithms
Input: Zero or more inputs
Output: At least one output.
Finiteness: N number of steps.
Definiteness: Clear algorithm step.
Effectiveness: A carried out step.

2. *Process for Design and analysis of algorithms:*

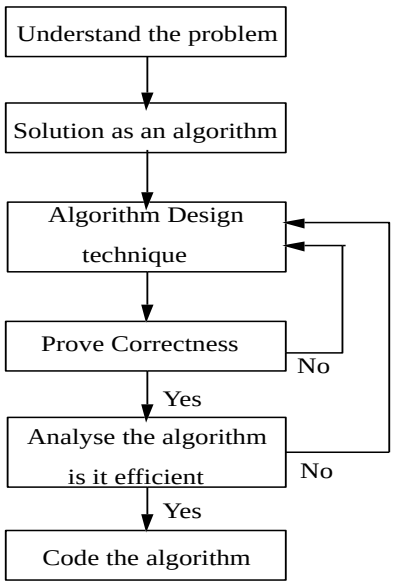


Fig: *Process for design and analysis of algorithms*

1. ***Understand the problem:*** This is very crucial phase. If we did any mistake in this step the entire algorithm becomes wrong. So before designing an algorithm is to understand the problem first.
2. ***Solution as an algorithm*** *(Exact vs approximation solving)*: Solve the problem exactly if possible. Even though some problems are solvable by exact method, but they are not faster when compared to approximation method. So in that situation we will use approximation method.
3. ***Algorithm techniques*****:** In this we will use different design techniques like,

i) Divide-and-conquer
ii) Greedy method
iii) Dynamic programming
iv) Backtracking
v) Branch and bound…. etc.,

4. ***Prove correctness:*** once algorithm has been specified, next we have to prove its correctness. Usually testing is used for proving correctness.
5. ***Analyze an algorithm:*** Analyzing an algorithm means studying the algorithm behavior ie., calculating the time complexity and space complexity. If the time complexity of algorithm is more then we will use one more designing technique such that time complexity should be minimum.
6. ***Coding an algorithm****:* after completion of all phases successfully then we will code an algorithm. Coding should not depend on any program language. We will use general notation (pseudo-code) and English language statement. Ultimately algorithms are implemented as computer programs.

3. Types of Algorithms:

There are four types of algorithms

1. Approximate algorithm.
2. Probabilistic algorithm.
3. Infinite algorithm.
4. Heuristic algorithm.

1. **Approximate Algorithm:** An algorithm is said to approximate if it is infinite and repeating.

   Ex: $\sqrt{2}$ = 1.414
   $\sqrt{3}$ = 1.713
   π = 3.14 etc...

1.
2. **Probabilistic algorithm:** If the solution of a problem is uncertain then it is called as probabilistic algorithm. **Ex:** Tossing of a coin.
3. **Infinite Algorithm:** An algorithm which is not finite is called as infinite algorithm.
   Ex: A complete solution of a chessboard, division by zero.
4. **Heuristic algorithm:** Giving fewer inputs getting more outputs is called the Heuristic algorithms.
   Ex: All Business Applications.

4. Criteria's (or) Issues for algorithm:

There are various issues in the study of algorithms;

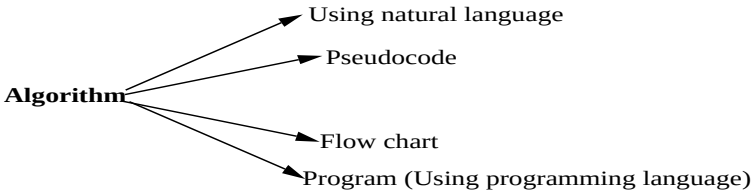
1. How to devise algorithms: The creation of an algorithm is a logical activity which may never be fully automated.
2. How to express algorithms: We shall express all of our algorithms using the best principles of structuring.
3. How to validate algorithms: After creation of algorithms is to validate algorithms. The process of checking an algorithm computes the correct answer for all possible legal inputs is called algorithm validation. The purpose of validation of algorithm is to find whether algorithm works properly without being dependent upon programming languages.
4. How to analyze algorithms: Analysis of algorithm is a task of determining how much computing time and storage is required by an algorithm. Analysis of algorithms is also called performance analysis. The behavior of algorithm in best case, worst case and average case needs to be obtained.
5. How to test a program: Testing a program really consists of two phases:

i). *Debugging:* While debugging a program, it is checked whether program produces faulty results for valid set of input and if it is found then the program has to be corrected.

ii). *Profiling or performance measuring:* Profiling is a process of measuring time and space required by a corrected program for valid set of inputs.

5 Specification of algorithm:

There are various ways by which we can specify an algorithm.



It is very easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear, and may require brief description.

Example: Write an algorithm to perform addition of two numbers.

Step 1: Read the first number, say 'a'.
Step 2: Read the second number, say 'b'.
Step 3: Add the two numbers and store the result in a variable 'c'.
Step 4: Display the result.

Such a specification creates difficulty, while actually implementing it (difficulty in converting into source code). Hence many programmers prefer to have specification of algorithm by means of pseudo-code.

Another way of representing the algorithm is by flow chart. Flow chart is a graphical representation of an algorithm, but flowchart method work well only if the algorithm is small and simple.

Pseudo-Code for expressing Algorithms

Based on algorithm there are two more representations used by programmer and those are flow chart and pseudo-code. Flowchart is a graphical representation of an algorithm. Similarly pseudo-code is a representation of algorithm in which instruction sequence can be given with the help of programming constructs. It is not a programming language since no pseudo language compiler exists.

The general procedure for writing the pseudo-code is presented below-

1. Comments begin with // and continue until the end of line
2. A block of statements (compound statement) are represented using { and } for example if statement, while loop, functions etc.,.

```
Example
{
   Statement 1;
   Statement 2;
      .....
      .....
}
```

3. The delimiters [**;**] are used at the end of the each statement.
4. An identifier begins with a letter. *Example:* sum, sum5, a; but not in 5sum, 4a etc.,.
5. Assignment of values to the variables is done using the assignment operators as **:=** or **←.**
6. There are two Boolean values TRUE and FALSE.
   Logical operators: AND, OR, NOT.
   Relational operators: <, >, ≥,≤,=,≠.
   Arithmetic operators: +, -, *, /, %;
7. The conditional statement if-then or if-then-else is written in the following form.
   If (condition) then (statement)
   If (condition) then (statement-1) else (statement-2)

'If' is a powerful statement used to make decisions based as a condition. If a condition is true the particular block of statements are execute.

Example

```
if(a>b) then
 {
   write("a is big");
 }
 else
 {
    write("b is big");
 }
```

8. Case statement

```
case
{
   :(condition -1): (statement-1)
   :(condition -2): (statement-2)
   :(condition -n): (statement-n)
     .............
     .............
  else     :(statement n+1);
}
```

If condition -1 is true, statement -1 executed and the case statement is exited. If statement -1 is false, condition -2 is evaluated. If condition -2 is true, statement-2 executed and so on. If none of the conditions are true, statement –(n+1) is executed and the case statement is exited. The else clause is optional.

9. Loop statements:

For loop:

i). The general form of the for loop is

```
for variable:=value 1 to value n step do
{
   Statement -1;
   Statement -1;
     .......
     .......
   Statement -n;
}
```

Example:

```
for i:=1 to 10 do
{
  write(i); //displaying numbers from 1 to 10
   i:=i+1;
}
```

ii). **While loop:**

The general form of the while loop is

```
while <condition> do
{
  <statement 1>
  <statement 2>
   ........
   ........
 <statement n>
}
```

Example:

```
i:=1;
while(i<=10)do
{
  write (i);//displaying numbers from 1 to 10
  i:=1+1;
}
```

Note that the statements of while loop are executed as long as <condition> is true.

iii). **Repeat-until loop:**

The general form of repeat-until is-

```
repeat
{
  <statement 1>
  <statement 2>
    ......
    ......
  <statement n>
}until <condition>
```

Example

```
i:=1;
repeat
{
  write (i);
  i:=i+1;
}
until (i>10);
```

Note that the statements are executed as long as <condition> is false.

10. Break: this statement is exit from the loop.
11. Elements of array are accessed using [ ].

For example, if A is an one-dimensional array, then i^{th} element can be accessed using A[i]. If A is two-dimensional array, then (i, j)th element can be accessed using A[i,j].

12. Procedures (functions): There is only one type of procedure:

An algorithm consists of a heading and a body.

procedure name of the procedure

Syntax:

```
Algorithm Name (<parameter list>)
{
  body of the procedure
}
```

13.Compound data-types can be formed with records

Syntax:

```
Name = record
{
  data-type -1 data 1;
  data-type -2 data 2;
  data-type -n data n;
}
```

Example

```
Employee =record
{
int no;
char name[10];
float salary;
}
```

Example 1: Write an algorithm to find the sum of **n** numbers.

```
Algorithm sum(n)
 {
  total:=0;
   for i:=1 to n do
total:= total + i;
i:=i+1;
}
```

Example 2: Write an algorithm to perform Multiplication of two matrices.

```
Algorithm Multiplication (A, B, n)
        {
          for i:=1 to n do
               for j:=1 to n do
                  C[i,j]:=0;
              for k:=1 to n do
                    C[i,j]:=C[i,j]+A[i,k]*B[k,j];
        }
```

6. *Performance Analysis:*

Performance analysis or analysis of algorithms refers to the task of determining the efficiency of an algorithm i.,e how much computing time and storage an algorithm requires to run (or execute). This analysis of algorithm helps in judging the value of one algorithm over another.

To judge an algorithm, particularly two things are taken into consideration

1. Space complexity
2. Time complexity.

Space Complexity: The space complexity of an algorithm (program) is the amount of memory it needs to run to completion. The space needed by an algorithm has the following components.

1. Instruction Space.
2. Data Space.
3. Environment Stack Space.

Instruction Space: Instruction space is the space needed to store the compiled version of the program instructions. The amount of instruction space that is needed depends on factors such as-

i). The compiler used to compile the program into machine code.

ii). The compiler options in effect at the time of compilation.

iii). The target computer, i.,e computer on which the algorithm run.

Note that, one compiler may produce less code as compared to another compiler, when the same program is compiled by these two.

Data Space: Data space is the space needed to store all constant and variable values. Data space has two components.

i). Space needed by constants, for example 0, 1, 2.134**.**

ii). Space needed by dynamically allocated objects such as arrays, structures, classes.

Environmental Stack Space: Environmental stack space is used during execution of functions. Each time function is involved the following data are saved as the environmental stack.

i). The return address.

ii). Value of local variables.

iii). Value of formal parameters in the function being invoked.

Environmental stack space is mainly used in recursive functions. Thus, the space requirement of any program **p** may therefore be written as

Space complexity S(P) = C + Sp (Instance characteristics).

This equation shows that the total space needed by a program is divided into two parts.

- Fixed space requirements(C) is independent of instance characteristics of the inputs and outputs.
  - Instruction space
  - Space for simple variables, fixed-size structure variables, constants.
- A variable space requirements (SP(1)) dependent on instance characteristics 1.
  - This part includes dynamically allocated space and the recursion stack space.

Example of instance character is:

Examples: 1

```
Algorithm NEC (float x, float y, float z)
 {
        Return (X + Y +Y * Z + (X + Y +Z)) /(X+ Y) + 4.0;
 }
```

In the above algorithm, there are no instance characteristics and the space needed by X, Y, Z is independent of instance characteristics, therefore we can write,

S(XYZ) =3+0=3

One space each for X, Y and Z

∴Space complexity is O(1).

Examples: 2

```
Algorithm ADD ( float [], int n)
 {
  sum = 0.0;
  for i=1 to n do
  sum=sum+X[i];
  return sum; }
```

Here, atleast n words since X must be large enough to hold the n elements to be summed. Here the problem instances is characterized by n, the number of elements to be summed. So, we can write,

S(ADD) =3+n

3-one each for n, I and sum

Where n- is for array X[],

∴Space complexity is O(n).

Time Complexity

The time complexity of an algorithm is the amount of compile time it needs to run to completion. We can measure time complexity of an algorithm in two approaches

1. Priori analysis or *compile time*
2. Posteriori analysis or *run* (execution) *time*.

In priori analysis before the algorithm is executed we will analyze the behavior of the algorithm. A priori analysis concentrates on determining the order if execution of statements.

In Posteriori analysis while the algorithm is executed we measure the execution time. Posteriori analysis gives accurate values but it is very costly.

As we know that the compile time does not depend on the size of the input. Hence, we will confine ourselves to consider only the run-time which depends on the size of the input and this run-time is denoted by TP(n). Hence

Time complexity T(P) = C + TP(n).

The time (T(P)) taken by a program P is the sum of the compile time and execution time. The compile time does not depend on the instance characteristics, so we concentrate on the runtime of a program. This runtime is denoted by **tp** (instance characteristics).

The following equation determines the number of addition, subtraction, multiplication, division compares, loads stores and so on, that would be made by the code for **p.**

tp(n) = CaADD(n)+ CsSUB(n)+ CmMUL(n)+ CdDIV(n)+……..

where n denotes instance characteristics, and Ca, Cs, Cm, Cd and so on…..

As denote the time needed for an addition, subtraction, multiplication, division and so on, and ADD, SUB, MUL, DIV and so on, are functions whose values are the number of additions, subtractions, multiplications, divisions and so on. But this method is an impossible task to find out time complexity.

Another method is step count. By using step count, we can determine the number if steps needed by a program to solve a particular problem in 2 ways.

Method 1: introduce a global variable “count”, which is initialized to zero. So each time a statement in the signal program is executed, count is incremented by the step count of that statement.

Example:

```
Algorithm Sum(a, n)
 {
  s:=0;

  for i:=1 to n do
  {

   s:=s+a[i];

  }


return s;
}
```

Algorithm sum with count statement added

```
count:=0;
Algorithm Sum(a,n)
 {
   s:=0;
   count:=count+1;
      for i:=1 to n do
        {
          count:=count +1;
          s:=s+a[i];
          count:=count+1;
        }
   count:=count+1; //for last time of for loop
   count:=count+1; //for return statement
   return s;
}
```

Thus the total number of steps are 2n+3

Method 2: The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

Ex:

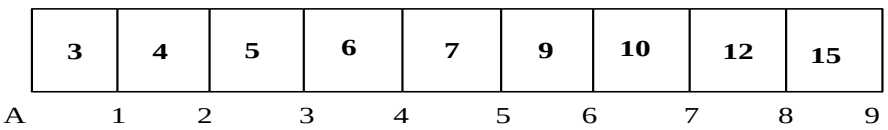
Statement	S/e	Frequency	Total steps
1. Algorithm Sum(a, n)	0	-	0
2. {	0	-	0
3. s:=0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. s:=s+a[i];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3 steps

The S/e (steps per execution) of a statement is the amount by which the count changes as a result of the execution of that statement. The frequency determines the total number of times each statement is executed.

Complexity of Algorithms:

1. Best Case: Inputs are provided in such a way that the minimum time is required to process them.
2. Average Case: The amount of time the algorithm takes on an average set of inputs.
3. Worst Case: The amount of time the algorithm takes on the worst possible set of inputs.

Example: Linear Search



Best Case: If we want to search an element 3, whether it is present in the array or not. First, A(1) is compared with 3, match occurs. So the number of comparisons is only one. It is observed that search takes minimum number of comparisons, so it comes under best case.

Time complexity is O(1).

Average Case: If we want to search an element 7, whether it is present in the array or not.

First, A(1) is compared with 7 i,.e, (3=7), no match occurs. Next, compare A(2) and 7, no match occurs. Compare A(3) and A(4) with 7, no match occurs. Up to now 4 comparisons takes place. Now compare A(5) and 7 (i.,e, 7=7), so match occurs. The number of comparisons is 5. It is observed that search takes average number of comparisons. So it comes under average case.

Note: If there are **n** elements, then we require n/2 comparisons.

$$\therefore \text{Time complexity is } O\left(\frac{n}{2}\right) = O(n) \text{ (we neglect constant)}$$

Worst Case: If we want to search an element 15, whether it is present in the array or not.

First, A(1) is compared with 15 (i.,e, 3=15), no match occurs. Continue this process until either element is found or the list is exhausted. The element is found at 9^{th} comparison. So number of comparisons are 9.

$\therefore$ Time complexity is O(n).

Note: If the element is not found in array, then we have to search entire array, so it comes under worst case.

7. Asymptotic Notation:

Accurate measurement of time complexity is possible with asymptotic notation. Asymptotic complexity gives an idea of how rapidly the space requirement or time requirement grow as problem size increase. When there is a computing device that can execute 1000 complex operations per second. The size of the problem is that can be solved in a second or minute or an hour by algorithms of different asymptotic complexity. In general asymptotic complexity is a measure of algorithm not problem. Usually the complexity of an algorithm is as a function relating the input length to the number of steps (time complexity) or storage location (space complexity). For example, the running time is expressed as a function of the input size 'n' as follows.

$$f(n)=n^4+100n^2+10^n+50 \text{ (running time)}$$

There are four important asymptotic notations.

1. Big oh notation (O)
2. Omega notation (Ω).
3. Theta notation (θ)

Let f(n) and g(n) are two non-negative functions.

Big oh notation

Big oh notation is denoted by 'O'. it is used to describe the efficiency of an algorithm. It is used to represent the upper bound of an algorithms running time. Using Big O notation, we can give largest amount of time taken by the algorithm to complete.

Definition: Let f(n) and g(n) be the two non-negative functions. We say that f(n) is said to be O(g(n)) if and only if there exists a positive constant 'c' and 'n_0' such that,

$f(n) \leq c*g(n)$ for all non-negative values of n, where $n \geq n_0$.

Here, g(n) is the upper bound for f(n).

Ex: Let $f(n) = 2n^4 + 5n2 + 2n + 3$

$\leq 2n^4 + 5n^4 + 2n^4 + 3n^4$

$\leq (2+5+2+3)n^4$

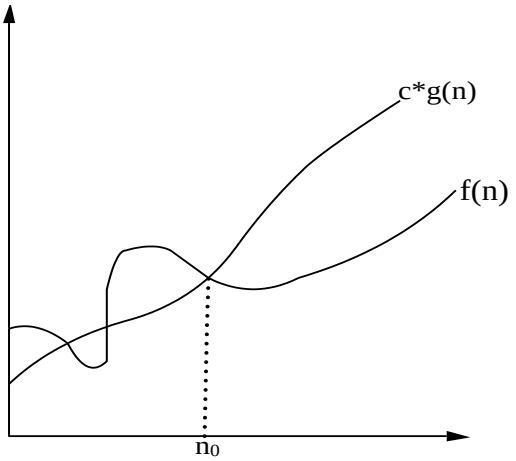
$\leq 12n^4$.

$\therefore f(n)=12n^4$

This implies $g(n)=n^4$, $n \geq 1$

$\therefore c=12$ and $n_0=1$

$\therefore f(n)=O(n^4)$



The above definition states that the function 'f' is almost 'c' times the function 'g' when 'n' is greater than or equal to n_0.

This notion provides an upper bound for the function 'f' i.,e, the function g(n) is an upper bound on the value of f(n) for all n, where $n \geq n_0$.

Big omega notation

Big omega notation is denoted by 'Ω'. It is used to represent the lower bound of an algorithms running time. Using big omega notation we can give shortest amount of time taken by the algorithm to complete.

Definition: The function $f(n)= \Omega(g(n))$ (read as for of n is omega of g of n) if and only if there exist positive constants 'c' and 'n_0' such that,

$f(n) \geq c*g(n)$ for all n, $n \geq n_0$

Example:

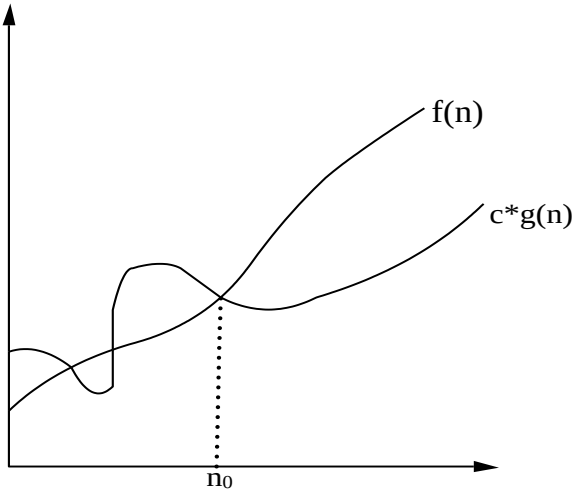
Let $f(n) = 2n^4 + 5n2 + 2n + 3$

$\geq 2n^4$ (for example as $n \rightarrow \infty$, lower order oterms are insignificant)

$\therefore f(n) \geq 2n^4$, $n \geq 1$

$\therefore g(n)=n^4$, $c=2$ and $n_0=1$

$\therefore f(n)= \Omega(n^4)$



Big Theta notation

The big theta notation is denoted by 'θ'. It is in between the upper bound and lower bound of an algorithms running time.

Definition: Let f(n) and g(n) be the two non-negetive functions. We say that f(n) is said to be θ(g(n)) if and only if there exists a positive constants 'c_1' and 'c_2', such that,

$c_1g(n) \leq f(n) \leq c_2g((n)$ for all non-negative values n, where $n \geq n_0$.

The above definition states that the function f(n) lies between 'c_1'times the function g(n) and 'c_2', times the function g(n) where 'c_1' and 'c_2' are positive constants.

This notation provides both lower and upper bounds for the function f(n) i.,e, g(n) is both lower and upper bounds on the value of f(n), for large n. in other words theta notation says that f(n) is both O(g(n)) and Ω(g(n)) for all n, where $n \geq n_0$.

This function f(n) = θ(g(n)) iff g(n) is both upper and lower bound an f(n).

Example:

$f(n) = 2n^4 + 5n^2 + 2n + 3$

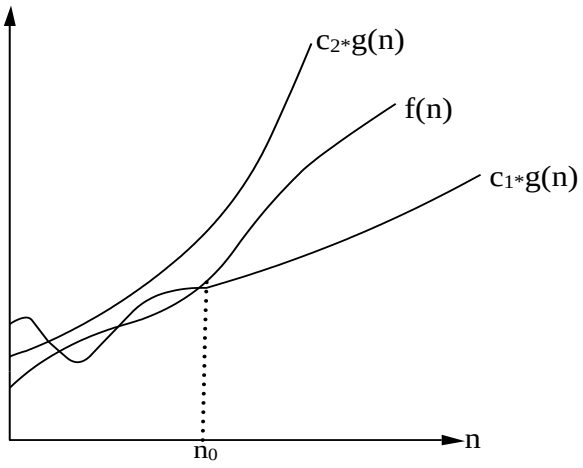
$\Rightarrow 2n^4 \leq 2n^4 + 5n^2 + 2n + 3 \leq 12n^4$

$\Rightarrow 2n^4 \leq f(n) \leq 12n^4$, $n \geq 1$

$\therefore g(n) = n^4$

$\therefore$ c1=2, c2=12 and n0=1

$\therefore f(n)=(n^{4)}$



Little 'oh' notation

Little oh notation is denoted by "**o**". the asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight.

Definition: f(n)=o(g(n)), iff f(n)<c.g(n) for any positive constants c>0, n_0>0 and n> n_0.

Or

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Find the time complexity for sum f given array elememts

The aymptotic complexity of sum is as follows

Ex:

Statement	S/e	Frequency	Total steps
1. Algorithm Sum(a, n)	0	-	θ(0)
2. {	0	-	θ(0)
3. s:=0;	1	1	θ(1)
4. for i:=1 to n do	1	n+1	θ(n)
5. s:=s+a[i];	1	n	θ(n)
6. return s;	1	1	θ(1)
7. }	0	-	θ(0)
Total			θ(n)

The time complexity can be calculated as follows-

First find out the basic operation of the above algorithm is-

S:=S+ a[i] i.e., addition in the loop is the basic operation.

The basic operation is executed every time the loop is executed.

Thus time complexity $T(n) = \sum_{i=1}^{n} 1$

$= 1 + 1 + 1 + 1 \cdots\cdots\cdots\cdots n$

$= n$

$\therefore T(n) = O(n)$

8 *Probabilistic Analysis:*

Probabilistic analysis of algorithms is an approach to estimate the complexity of an algorithm. It uses the probability in the analysis of problems. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to compute an expected running time of a known algorithm.

The following is the simple example as probabilistic average case analysis.

Example: Consider linear search algorithm which searches a target element say x, in the given list of size n. in the worst case, the algorithm will examine all n elements in the list before terminating.

For a probabilistic average-case analysis, it is generally assumed that all possible terminations are equally likely-that is, the probability that x, will be found at position 1 is 1/x at position 2 is 1/n and so on.

The average search cost is therefore the sum of all possible search costs each multiplied by their associated probability.

For example, if n=5, we would have

Average search cost=1/5(1 +2 +3 +4 +5)=3.

In general case we have

Average search cost =1/n(n(n+1)/2)=(n+1)/2

Probabilistic analysis is mainly useful in estimate running time of an algorithm, calculating search costs in a searching algorithm etc.

9. *Amortized Analysis:*

Amortized analysis refers to finding the average running time per operation, over a worst case sequence of operations. That is the main goal of amortized analysis is to analyze the time per operation for a series of operations. Sometimes single operation might be expensive; in that case amortized analysis specifies average over a sequence of operations. Amortized cost per operation for a sequence of n operations is the total cost of operations divided by n.

For example, if we have 100 operations at cost 1, followed by one operation at cost 100, then amortized cost per operation is 200/101 < 2. Amortized analysis does not allow random selection of input.

The average case analysis and amortized analysis are different. In average case analysis, we are averaging over all possible inputs whereas in amortized analysis we are averaging over a sequence of operations.

Amortized analysis does not allow random selection of input.

There are several techniques used in amortized analysis.

1. Aggregate Analysis: In this type of analysis upper bound T(n) on the total cost of a sequence of n operations is decided, then the average cost is calculated as T(n)/n.

2. Accounting Method: In this method the individual cost of each operation is determined, by combining immediate execution time and its influence on the running time of future operations.

3. Potential Method: It is like the accounting method, but overcharges operations early to compensate for undercharges later.