Introduction: What is an Algorithm, Algorithm Specification, Pseudo code Conventions, Recursive Algorithm, Performance Analysis, Space complexity, Time complexity, Amortized Complexity, Asymptotic Notation. Practical Complexities, Performance Measurement.

ALGORITHM

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Definition: An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

In addition, all algorithms should satisfy the following criteria.

1. **Input** → Zero or more quantities are externally supplied.
2. **Output** → At least one quantity is produced.
3. **Definiteness** → Each instruction is clear and unambiguous.
4. **Finiteness** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness** → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Areas of study of Algorithm:

1. How to devise algorithms.
   - Creating an algorithm is an art which may never be fully automated.
   - This area of study concentrates on various design techniques that have proven to be useful in that they have often yielded good algorithms.
   - It becomes easier to devise new and useful algorithm, by mastering these design strategies.
2. How to validate algorithms.
   - After devising an algorithm, it is necessary to show that it computes the correct answer for all possible legal inputs. This process is referred as *algorithm validation.*
   - The purpose of this validation is to assure us that this algorithm works correctly in the programming language it will be written in.
3. How to analyze algorithms.
   - This field of study is called *analysis of algorithms.*
   - Execution of an algorithm requires computer's central processing unit (CPU) to perform operations and its memory to hold the program and data.
   - Analysis of algorithms or performance analysis refers to task of determining how much computing time and storage an algorithm requires.
   - This is a challenging area and requires great mathematical skill.
4. How to test algorithm.
   - Testing an algorithm requires two phases
   - Debugging : It is the process of executing programs on sample data sets to determine whether faulty results occur and, of so, to correct them.
   - Profiling : It is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

ALGORITHM SPECIFICATION

Algorithm can be described in many ways.

- **Natural language like English**: When this way is chosen, we should ensure that resulting instructions are definite.
- **Graphic representations called flowchart**: This method will work well when the algorithm is small& simple.
- **Pseudo-code Method**: In this method, we should typically describe algorithms as program, which resembles language like C. The advantage of pseudo code over flowchart is that it is very much similar to the final program code. It requires less time and space to develop, and we can write it in our own way as there are no fixed rules.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces **{** and **}**. A compound statement can be represented as a block. The body of the procedure also forms a block. The statements are delimited by **;** .
3. An identifier begins with a letter. The data type of variables, or a variable is local or global, depends on the context. we assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **records**. Here is an example

```
node = record
       {    datatype_1    data1;
            datatype_2    data2;
            :
            :
            datatype_n    datan;
            node          *link;
       }
```

   link is a pointer to the record type *node*. The data items of a record can be accessed with -> and a period. For example if *p* points to a record of type node, *p->data1* stands for the value of the first field in the record. If *q* is a record of type node, *q.data1* denote its first field.

4. Assignment of values to variables is done using the assignment statement

   <variable> := < expression >;

5. There are two Boolean values **true** and **false**. In order to produce these values, the logical operators **and, or** and **not** and the relational operators **<, ≤, = , ≠, ≥,** and **>** are used.
6. Elements of the multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the $(i, j)^{th}$ element of the array is denoted as A[i, j]. Array indices start at zero.
7. The general forms of looping statements are for, while and repeat-until**.**

   The general form of **while** loop is:

```
while <condition> do
{
        <statement 1>
                :
        <statement n>
}
```

As long as <*condition*> is **true**, the statements get executed. The loop is exited whenever the condition becomes **false**.

The general form of for loop is

for *variable := value1* **to** *value2* **step** *st* **do**
{
<*statement 1*>
:
<*statement n*>
}

Here *value1*, *value2*, and *step* are arithmetic expressions. Step value can be either positive or negative. Default value of step is +1.

The general form of repeat-until statement is :

repeat
<*statement 1*>
:
<*statement n*>
until <*condition*>

The statements are executed as long as <*condition*> is false.

The **break** instruction can be used to exit out of loop.
The **return** statement is used to exit from function.

8. A conditional statement has the form

if <*condition*> **then** <*statement*>
if <*condition*> **then** <*statement1*> **else** <*statement2*>

The **case** statement takes the form

case
{
: <*condition 1*> ***:*** < *statement 1* >
:
: <*condition n*> ***:*** < *statement n* >
:else > ***:*** < *statement n+1* >
}

It is interpreted as if <*condition1*> is **true**, <*statement1*> gets executed and exit the case statement. if <*condition1*> is **false**, <*condition2*> is evaluated if it is **true**, <*statement2*> gets executed and exit the case statement. And so on. If none of the conditions are true , <*statement n+1*> is executed. **else** clause is optional.

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and body. The heading takes the form

Algorithm Name (<*parameter list*>)

Where *Name* is the name of the procedure and (<*parameter list*>) is the procedure parameters. The body of the procedure consist of one or more statements enclosed within braces **{** and **}**. Simple variables to the procedure are passed by value. Arrays and records are passed by reference.

Examples:

1. Algorithm for *finding the maximum of n given numbers.* In this algorithm (named Max), A & n are procedure parameters. Result & i are Local variables.

```
1    Algorithm Max ( A, n)
2    // A is an array of size n.
3    {
4            Result := A[1];
5            for i:= 2 to n do
6                    If A[i] > Result then Result := A[i];
7            return Result;
8     }
```

2. Algorithm for *Selection Sort* : - we can define this as
From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

```
1    Algorithm SelectionSort( a, n )
2    // sort the array a[ 1 : n ] into nondecreasing order.
3    {
4            for i:= 1 to n do
5            {
6                    j := i;
7                    for k := i+1 to n do
8                            if ( a[k] < a[j] ) then j := k;
9                    t := a[i] ; a[i] := a[j]; a[j] := t;
10           }
11   }
```

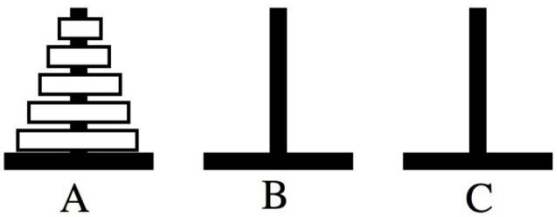
Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is *Direct Recursive*.
- Algorithm 'A' is said to be *Indirect Recursive* if it calls another algorithm which in turns calls 'A'.
- The Recursive mechanisms are extremely powerful. They can express complex process very clearly.
- The following 2 examples show how to develop a recursive algorithm.

Example 1 : Towers of Hanoi:

It consists of three towers, and a number of disks arranged in ascending order of size. The objective of the problem is to move the entire stack to another tower, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



A very elegant solution results from the use of recursion.

- Assume that the number of disks is *'n'*.
- To get the largest disk to the bottom of tower B, we move the remaining *'n-1'* disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

```
1   Algorithm TowersofHanoi(n, x, y, z)
2   //Move the top 'n' disks from tower x to tower y.
3   {
4       if(n>=1) then
5       {
6           TowersofHanoi(n-1, x, z, y);
7           Write("move top disk from tower ", x ,"to top of tower " ,y);
8           Towersofhanoi(n-1, z, y, x);
9       }
10  }
```

This algorithm is invoked by ***TowersOfHanoi(n, A, B, C)*.**

Example 2: Permutation Generator:

- Given a set of $n>=1$ elements, the problem is to print all possible permutations of this set.
- For example, if the set is *{a,b,c}*, then the set of permutation is,

  { (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}
- It is easy to see that given '*n*' elements there are *n!* different permutations.
- A simple algorithm can be obtained by looking at the case of 4 elements *(a,b,c,d).* The Answer can be constructed by writing
  - *a* followed by all the permutations of *(b,c,d)*
  - *b* followed by all the permutations of *(a,c,d)*
  - *c* followed by all the permutations of *(a,b,d)*
  - *d* followed by all the permutations of *(a,b,c)*

```
1   Algorithm perm(a,k,n)
2   {
3       if (k = n) then write (a[1:n]);        // output permutation
4       else                                   //a[k:n] has more than one permutation
5           // Generate this recursively.
6           for i := k to n do
7           {
8               t:=a[k];  a[k]:=a[i]; a[i]:=t;
9               perm(a, k+1, n);               //all permutation of a[k+1:n]
10              t:=a[k]; a[k]:=a[i]; a[i]:=t;
11          }
12  }
```

This algorithm is invoked by ***perm(a, 1, n)***

EXERCISES

1. Define algorithm and List out the criteria's of an algorithm. (features / properties)
2. What are the four distinct areas of study of algorithm?
3. Define debugging and profiling.
4. Describe pseudo code conventions for specifying algorithms. **
5. The factorial function n! has value 1 when n <=1 and value n * (n – 1)! when n>1. Write both recursive and iterative algorithm to computer n!.
6. What is pseudo-code? Explain with an example.
7. Distinguish between algorithm and pseudo code.
8. Devise an algorithm that inputs three integers and outputs them in nondecreasing order.
9. Present an algorithm that searches an unsorted array a[1:n] for the element x. If x occurs, then return a position in the array; else return zero.
10. The Fibonacci numbers are defined as $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both recursive and iterative algorithms to compute f_i.
11. Devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
12. Write a recursive algorithm to solve Towers of Hanoi problem with an example.

PERFORMANCE ANALYSIS

Computing time and storage requirement are the criteria for judging algorithms that have direct relationship to performance.

Performance evaluation can be divided into two major phases

1. a priori estimates (performance analysis) and
2. a posteriori testing (performance measurement).

Space Complexity:

Space Complexity of an algorithm is the amount of memory it needs to run to completion.

The space needed by any algorithm is the sum of the following components.

1. A **fixed part**, that is independent of the characteristics of inputs and outputs. This part includes space for instructions(code), space for simple variables, & fixed size component variables, space for constants etc.
2. A **variable part**, which consists of space needed by component variables whose size, is dependent on the particular problem instance being solved, space for reference variables and recursion stack space etc.

The Space requirement *S(P)* of an algorithm *P* may be written as

$$S(P) = c + S_P(\text{instance characteristics})$$ where *'c'* is a constant.

When analyzing the space complexity of an algorithm first we estimate $S_P(\text{instance characteristics})$. For any given problem, we need to determine which instance characteristics to use to measure the space requirements.

Example 1:

```
1  Algorithm abc(a, b, c)
2  {
3     return a+b+b*c + (a+b –c) / ( a+b) + 4.0;
4  }
```

It is characterized by values of *a, b, c*. If we assure that one word is needed to store the values of each *a, b, c* , *result* and also we see S_P*(instance characteristics)=0* as space needed by *abc* is independent of instance characteristics; So 4 words of space is needed by this algorithm.

Example 2:

```
1  Algorithm sum(a, n)
2  {
3     s := 0.0;
4     for i := 1 to n do
5           s := s+a[i];
6     return s;
7  }
```

This algorithm is characterized by *'n'* (number of elements to be summed). The space required for *'n'* is 1 word. The array *a[]* of float values require atleast *'n'* words.
So, we obtain

$$S_{sum}(n) \geq n+3$$

(*n* words for *a*, 1 word for each of *n, i, s*)

Example 3:

```
1  Algorithm Rsum(a, n)
2  {
3     if ( n ≤ 0 ) then  return 0.0;
4     else  return Rsum(a, n-1) + a[n];
5  }
```

Instances of this algorithm are characterized by *n*. The recursion stack space includes space for formal variables, local variables, return address (1-word). In the above algorithm each call to *Rsum* requires 3 words (for *n*, return address, pointer to *a[]*). since the depth of the recursion is *n+1*, the recursion stack space needed is $\geq 3(n+1)$

Time Complexity :

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

- The time *T(P)* taken by a program *P* is the sum of the *compile time* and *Run time*. Compile time does not depend on instance characteristics and a compiled program will be run several times without recompilation, so we concern with just run time of the program. Run time is denoted by T_P***(instance characteristics)***.
- The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function. The number of steps is computed as a function of some subset of number of inputs and outputs and magnitudes of inputs and outputs.

- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.
- The number of steps any problem statement is assigned depends on the kind of statement.
  For example, comments → 0 steps.
  Assignment statements → 1 step
  Iterative statements such as for, while & repeat-until→ 1 step for control part of the statement.

- Number of steps needed by a program to solve a particular problem is determined in two ways.

Method 1 : (Step count calculation using a variable)

- In this method, a new global variable count with initial value '0' is introduced into the program.
- Statements to increment count are also added into the program.
- Each time a statement in the original program is executed, count is incremented by the step count of the statement.

Example:

```
1   Algorithm Sum (a, n)
2   {
3      s:=0.0;
4      count :=count + 1;  //count is global; initially 0
5      for i := 1 to n do
6      {
7         count:=count+1;  //for for
8         s:=s+a[i]; count:=count+1; //for assignment
9      }
10     count :=count + 1;  //for last time of for
11    count :=count + 1;  //for the return
12    return s;
13  }
```

The change in the value of *count* by the time this program terminates is the number of steps executed by the algorithm.

The value of count is increment by ***2n*** in the for loop.

At the time of termination the value of count is ***2n+3***.

So invocation of Sum executes a total of **2n+3** steps.

Example 2:

```
1   Algorithm RSum(a, n)
2   {
3      count:=count+1; //for the if conditional
4      if ( n ≤ 0) then
5      {
6            count:=count+1;  //for the return
7            return 0.0;
8      }
9      else
10     {
11           count:=count+1; //for addition, function call, return
12           return RSum(a,n-1)+a[n];
13     }
14  }
```

Let $t_{RSum}(n)$ be the increase in the value of *count* when the algorithm terminates.

When n=0, $t_{RSum}(0) = 2$
When n>0, *count* increases by 2 plus $t_{RSum}(n-1)$

When analyzing a recursive program for its step count, we often obtain a recursive formula.

For example,

$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

- These Recursive formulas are referred to as *recurrence relations*

To solve it, use repeated substitutions

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2(2) + t_{RSum}(n-2) \\ &: \\ &= n(2) + t_{RSum}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

So the step count of *RSum* is *2n+2*.
This step count is telling *the run time for a program with the change in instance characteristics*.

Method 2 : (Step count calculation by building a table)

In this method, the step count is determined by building a table. We list total number of steps contributed by each statement in the table. The table is built in this order.

1. Determine the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed.
   (*The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.)*
2. The total contribution of the statement is obtained by combining these two quantities.
3. Step count of the algorithm is obtained by adding the contribution of all statements.

Example 1:

	Statement	s/e	Frequency	Total Steps
1	**Algorithm** sum(*a, n*)	*0*	_	*0*
2	{	*0*	_	*0*
3	*s* := 0.0;	*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*

Example 2:

	Statement	s/e	Frequency		Total Steps	
			n=0	n>0	n=0	n>0
1	**Algorithm** Rsum(*a, n*)	*0*				
2	{	*0*				
3	**if** ($n \leq 0$) **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*				
	Total:				*2*	*2 + x*

$x = t_{RSum}(n-1)$

Example 3:

	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)*	*m(n + 1)*
5	*c[i,j] := a[i,j]+b[i,j];*	*1*	*mn*	*mn*
6	}	*0*	-	*0*
	Total:			*2mn+2m+1*

Exercise :

1. Define time complexity and space complexity.
2. What is space complexity? Illustrate with an example for fixed and variable part in space complexity.
3. Explain the method of determining the complexity of procedure by the step count approach. Illustrate with an example.

4. Implement iterative function for sum of array elements and find the time complexity use the increment count method.
5. Using step count method, analyze the time complexity when 2 m X n matrix added.
6. Write an algorithm for linear search and analyze the algorithm for its time complexity.
7. Give the algorithm for matrix multiplication and find the time complexity of the algorithm using step-count method.
8. Determine the frequency counts for all statements in the following algorithm segment.

```
i : = 1
while ( i <= n) do
{
      x := x + 1
      i := i + 1
}
```

9. Write a recursive algorithm to find the sum of first n integers and Derive its time complexity.
10. Implement an algorithm to generate Fibonacci number sequence and determine the time complexity of the algorithm using the frequency method.
11. What is the time complexity of the following function.

```
int fun() {
      for ( int i = 1; i<=n; i++ )
      {
            for ( int j = 1; j < n; j += i)
            {
                  sum = sum + i * j;
            }
      }
      return(sum);
}
```

12. Give the algorithm for transpose of a matrix m X n and determine the time complexity of the algorithm by frequency-count method.
13. Give the algorithm for matrix addition and find the time complexity of the algorithm using frequency-count method
14. Explain recursive function analysis with an example

Amortized Analysis

- Amortized Analysis not just considers one operation, but a sequence of operations on a given data structure. It averages cost over a sequence of operations.
- In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The amortized cost of an operation is the total cost charged to it.
- The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs.
- The only requirement of amortized complexity is that sum of the amortized complexities of all operations in any sequence of operations be greater than or equal to their sum of the actual complexities.

That is $$\sum_{1 \le i \le n} amortized(i) \ge \sum_{1 \le i \le n} actual(i) \quad \text{--(1)}$$

- Where *amortized(i)* and *actual(i)*, respectively denote the amortized and actual complexities of the i^{th} operation in a sequence of *n* operations.
- For this reason, we may use the sum of the amortized complexities as an upper bound on the complexity of any sequence of operations.

- Amortized cost of an operation is viewed as the amount you charge the operation rather than the amount the operation costs. You can charge an operation any amount you wish so long as the amount charged to all operations in the sequence is at least equal to the actual cost of the operation sequence.
- Relative to the actual and amortized costs of each operation in a sequence of n operations, we define a potential function P(i) as below

$$P(i) = amortized(i) - actual(i) + P(i-1) \quad \text{-- (2)}$$

- That is, the i^{th} operation causes the potential function to change by the difference between the amortized and actual costs of that operation.

- If we sum the equation (2) for $1 \le i \le n$, we get

$$\sum_{1 \le i \le n} P(i) = \sum_{1 \le i \le n} (amortized(i) - actual(i) + P(i-1))$$

or

$$\sum_{1 \le i \le n} (P(i) - P(i-1)) = \sum_{1 \le i \le n} (amortized(i) - actual(i))$$

or

$$P(n) - P(0) = \sum_{1 \le i \le n} (amortized(i) - actual(i))$$

From equation (1), if follows that $P(n) - P(0) \ge 0$ -- (3)

Under the assumption P(0)=0, the potential P(i) is the amount by which the first *i* operations have been overcharged.
Generally, when we analyze the complexity of a sequence of *n* operations, *n* can be any nonnegative integer. Therefore equation (3) can hold for all nonnegative integers.

There are **three** popular methods to arrive at amortized costs for operations.
1. **Aggregate method**: In this we determine the upper bound [*UpperBoundOnSumOfActualCosts(n)*]for sum of the actual costs of the n operations.
The amortized cost of each operation is set equal to *UpperBoundOnSumOfActualCosts(n) / n*

2. **Accounting Method**: In this we assign amortized costs to the operations (by guessing), compute the *p(i)*s and show that $p(n) - p(0) \ge 0$.

3. **Potential method:** we start with a potential function that satisfies $p(n) - p(0) \ge 0$. And compute the amortized complexities using

$$P(i) = amortized(i) - actual(i) + p(i-1)$$

Asymptotic notations:

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

The following asymptotic notations are mostly used to represent time complexity of algorithms.

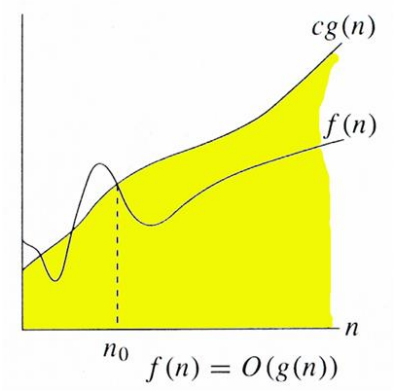
[Big-Oh] O-notation:

- The Big O notation defines an upper bound of an algorithm; it bounds a function only from above.
- Big-Oh notation is used widely to characterize running time and space bounds in terms of some parameter *n*, which varies from problem to problem.
- Constant factors and lower order terms are not included in the big-Oh notation.
- For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.

Def: The function $f(n)=O(g(n))$ iff there exists positive constants c and n_0 such that

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

Figure shows that, for all values n at and to the right of n_0, the value of function $f(n)$ is on or below $c.g(n)$.



Example : $7n-2$ is $O(n)$
Proof: By the big-Oh definition, we need to find a real constant $c>0$ and an integer constant $n_0 \geq 1$ such that $7n-2 \leq cn$ for every integer $n \geq n_0$.
Possible choice is $c=7$ and $n_0=1$.

Example: $20n^3+10n \log n+5$ is $O(n^3)$
Proof: $20n^3+10n \log n+5 \leq 35n^3$, for $n \geq 1$
In fact, any polynomial $a_k n^k + a_{k-1}n^{k-1} + \ldots + a_0$ will always be $O(n^k)$.

Here is a list of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first. k is some arbitrary constant.

Notation	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Polylogarithmic
O(n^2)	Quadratic
O(n^3)	Cubic
O(n^k) $(k \geq 1)$	Polynomial
O(k^n) $(k>1)$	exponential

Instead of always applying the big-Oh definition directly to obtain a big-Oh characterization, we can use the following rules to simplify notation.

Theorem: Let $d(n)$, $e(n)$, $f(n)$ and $g(n)$ be functions mapping nonnegative integers to nonnegative reals. Then

1. If $d(n)$ is $O(f(n))$, then $kd(n)$ is $O(f(n))$, for any constant $k > 0$.
2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.

3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n).e(n)$ is $O(f(n).g(n))$.
4. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.
5. If $f(n)$ is a polynomial of degree d (that is, $f(n)=a_0 + a_1n + ...+ a_dn^d$), then $f(n)$ is $O(n^d)$.
6. n^x is $O(a^n)$ for any fixed $x>0$ and $a>1$.
7. $\log n^x$ is $O(\log n)$ for any fixed $x>0$.
8. $\log^x n$ is $O(n^y)$ for any fixed constants $x>0$ and $y>0$.

Example : $2n^3 + 4n^2 \log n$ is $O(n^3)$
Proof:

- $\log n$ is $O(n)$
- $4n^2 \log n$ is $O(4n^3)$
- $2n^3 + 4n^2 \log n$ is $O(2n^3 + 4n^3)$
- $2n^3 + 4n^3$ is $O(n^3)$
- $2n^3 + 4n^2 \log n$ is $O(n^3)$

[Omega] Ω-notation:

- Ω-notation provides an asymptotic lower bound.

Def: The function $f(n)=\Omega(g(n))$ iff there exists positive constants c and n_0 such that

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

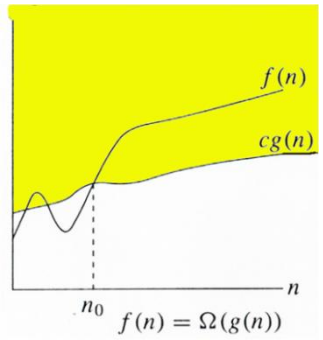


Figure shows the intuition behind Ω-notation for all values n at or to the right of n_0, the value of $f(n)$ is on or above $c.g(n)$.

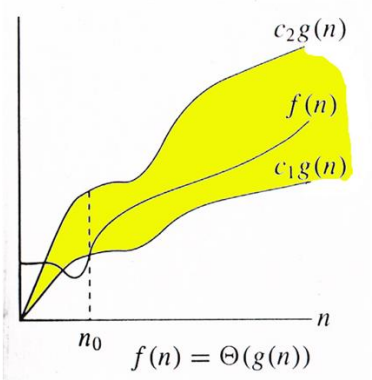
- If the running time of an algorithm is $\Omega(g(n))$, then the meaning is, "the running time on that input is atleast a constant times $g(n)$, for sufficiently large n".
- It says that, $\Omega(n)$ gives a lower-bound on the best-case running time of an algorithm.
- Eg: Best case running time of insertion sort is $\Omega(n)$.

Example: 3n+2 is $\Omega(n)$.
Proof: $3n+2 \geq 3n$ for $n \geq 1$ ($c=3$, $n0=1$)

[Theta] Θ - notation :

Def: The function $f(n)=\Theta(g(n))$ iff there exists positive constants c_1, c_2 and n_0 such that
$c1*g(n) \leq f(n) \leq c2*g(n)$ for all n, $n \geq n_0$

Following figure gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n)=\Theta(g(n))$.



- For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c1.g(n)$ and at or below $c2.g(n)$.

- For all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.
- We can say that $g(n)$ is an asymptotically tight bound for $f(n)$.

Example :

$$3n + 2 = \Theta(n)$$

find c_1, c_2, n_0 such that

$$\left.\begin{aligned} 3n + 2 &\geq c_1 . g(n) \\ 3n + 2 &\leq c_2 . g(n) \end{aligned}\right\} \quad \forall n \geq n_0$$

$c_1 = 3, c_2 = 4, n_0 = 2$ satisfies the above.

$\therefore 3n + 2 = \Theta(n)$

[Little-oh] o-notation :

- O-notation provides asymptotic upper bound. It may or may not be asymptotically tight.
- The bound $2n^2 = O(n^2)$ is asymptotically tight, but bound $2n = O(n^2)$ is not.

- o-notation is used to denote an upper bound that is not asymptotically tight.

Def: The function $f(n)=o(g(n))$ iff there exists a constant $n_0>0$ such that $0 \leq f(n) \leq c.g(n)$ for all $n>n_0$ and for any positive constant $c>0$.

The function $f(n)=o(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Example: The function $3n+2 = o(n^2)$

Since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$

[little-omega] ω - notation :

- ω - notation is used to denote a lower bound that is not asymptotically tight.
- By analogy, ω - notation to Ω-*notation* as *o-notation* to *O-notation*.

Def: The function $f(n) = \omega(g(n))$ iff there exists a constant $n_0>0$ such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$ and for any positive constant $c>0$.

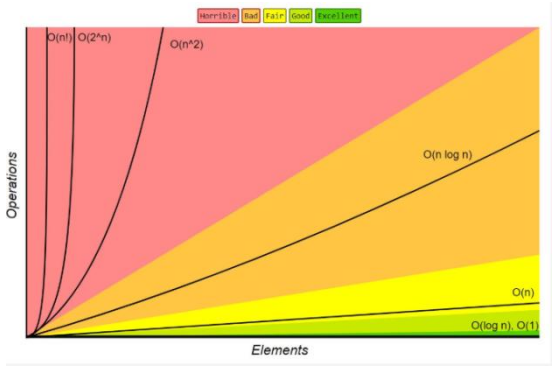
The function $f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Practical complexities

- Time complexity of an algorithm is generally some function of the instance characteristics.
- This function is useful
  - in determining how the time requirements vary as the instance characteristics change.
  - in comparing two algorithm P and Q which perform the same task,
- Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. we can assert that algorithm P is faster than algorithm Q for sufficiently large n.

Following table shows how various functions grow with n.

n	log n	n	n log n	n2	n3	2n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84467E+19
128	7	128	896	16384	2097152	3.40282E+38
256	8	256	2048	65536	16777216	1.15792E+77
512	9	512	4608	262144	134217728	1.3408E+154



It is very clear from the table that the function 2^n grows very rapidly with n.

eg:
Let a computer can execute 1 billion steps per second
And n = 40
If algorithm needs 2^n steps for execution, then the number of steps needed = $1.1 * 10^{12}$ which takes 18.3 minutes.

Important Questions :

1. Write about three popular methods to arrive at amortized costs for operations with example.
2. What is amortized analysis and Explain with and example. ****
3. What is amortized analysis of algorithms and how is it different from asymptotic analysis.
4. What are asymptotic notations? And give its properties.
5. Give the definition and graphical representation of asymptotic notations.
6. Describe and define any three asymptotic notations.
7. Give the Big – O notation definition and briefly discuss with suitable example.
8. Define Little Oh notation with example.
9. Write about big oh notation and also discuss its properties.
10. Define Omega notation
11. Explain Omega and Theta notations.
12. Compare Big-oh notation and Little-oh notation. Illustrate with an example.
13. Differentiate between Bigoh and omega notation with example.
14. Show that the following equalities are incorrect with suitable notations.

$$10\,n^2 + 9 = O(n)$$
$$n^2 \log n = \theta(n^2)$$
$$n^2 / \log n = \theta(n^2)$$
$$n^3 2^n + 5\,n^2 3^n = O(n^3 2^n)$$

15. Describe best case, average case and worst case efficiency of an algorithm.
16. Find big-oh and little-oh notation for $f(n) = 7\,n^3 + 50\,n^2 + 200$
17. What are different mathematical notations used for algorithm analysis
18. Prove the theorem if $f(n) = a_m n^m + \ldots + a_1 n + a_0$, where $f(n) = O(n^m)$.
19. Prove the theorem if $f(n) = a_m n^m + \ldots + a_1 n + a_0$, where $f(n) = \theta(n^m)$.