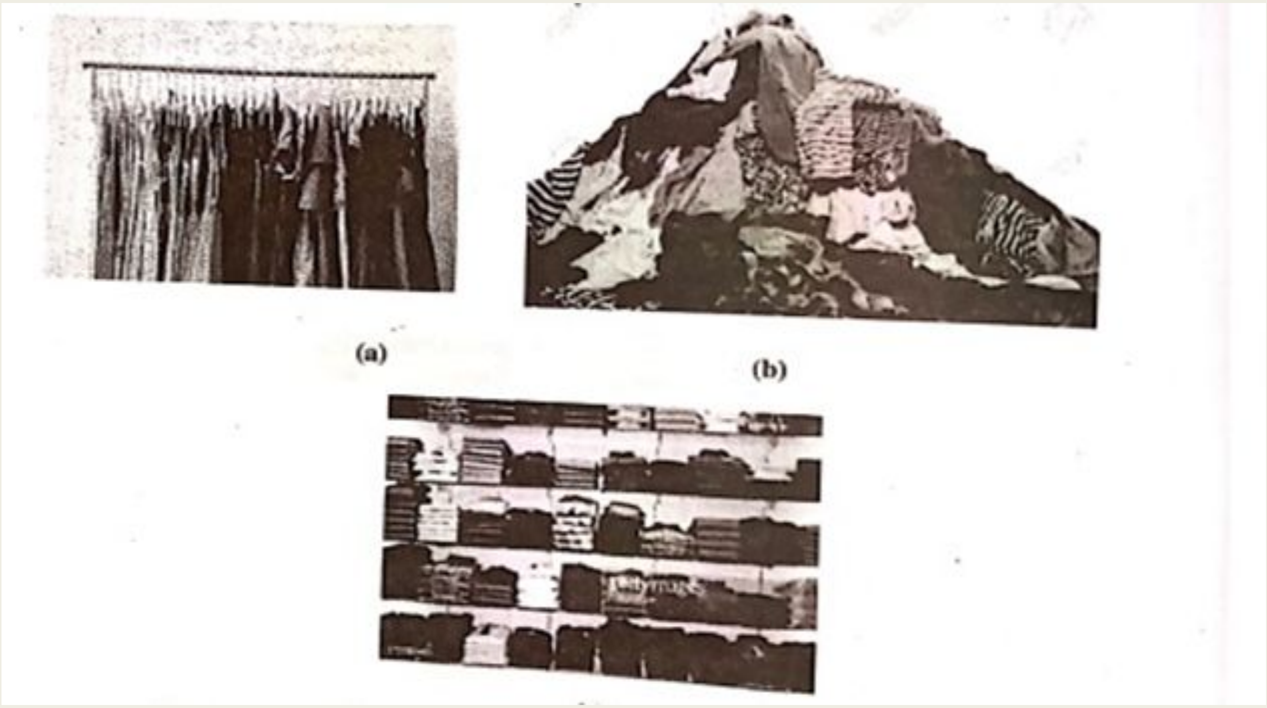
DATA STRUCTURE

UNIT 1

Sweta Suman

- A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.



Types of data structure :

1. There are two types of data structure based on **how it is arranged.**
   (i) **Linear data structure :** Here the data is stored in linear pattern or sequence. Array, stack, queue, link list are examples of linear data structure.
   (ii) **Non Linear data structure :** Here the data is stored in non linear pattern. Trees, Graph are examples of non linear data structure.
2. There are two types of data structure based on **memory type used.**
   (i) **Static data structure :** The data structure whose memory occupation is fixed i.e memory cannot be increased or decreased during run time is known as **Static Data Structure**. Array is an example of static data structure.
   (ii) **Dynámic data structure :** The data structure whose memory occupation is not fixed i.e memory can be increased or decreased during run time is known as **dynamic data structure.** Link list is an exámple of dynamic data structure.
   Other data structures like stack, queues, tree, graph can be static or dynamic depends on how they are implemented (using array or link list).
3. There are two types of data structure based on **which type of data it is holding.**
   (i) **Homogenous data structure :** The data structure which stored data of similar type is known as homogenous data structure. Array is an example of homogenous data structure.
   (ii) **Non homogenous data Structure :**
   - The data structure which stores data of different types is known as non homogenous data structure.
   - Link list is an example of non homogenous data structure.
   - Other data structures like stack, queues, tree, graph can be homogenous or non homogenous depends on how they are implemented (using array or link list).

Data Types

Primitive data type is the basic data type supported by computer. It can be operated upon by machine level instructions.

For example : Integer, character, float etc.

All primitive data type supports basic operation like addition, subtraction etc.

For **non primitive data type** we also need to define operation.

Non primitive data structure is derived from primitive data type .

The classification of the data structure is shown in Fig. 1.4.1.

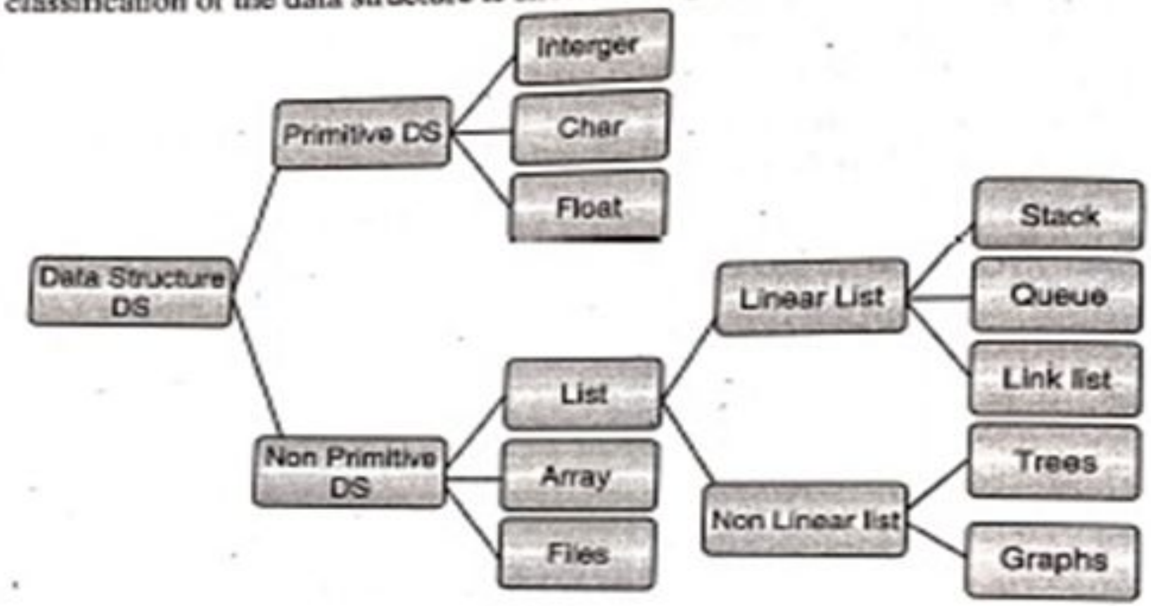


Fig. 1.4.1

Operations performed on Data Structure:

- Creation
- Insertion
- Traversing
- Searching
- Deletion
- Merging
- Copying
- Splitting
- Sorting

Algorithm

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.

- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language

- Characteristics of an Algorithm
- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

Algorithm

- **Problem** − Design an algorithm to add two numbers and display the result.

Step 1 − START

Step 2 − declare three variables **a**, **b** & **c**

Step 3 − define values of **a** & **b**

Step 4 − add values of **a** & **b**

Step 5 − store output of step 4 to **c**

Step 6 − print **c**

Step 7 − STOP

ADT

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.

- Some examples of ADT are Stack, Queue, List etc.

1D Array

Array A	10	20	30	40
Index	0	1	2	3
Memory address	2000	2004	2008	2012

Address of A[i] is calculated using below formula:

A[i]=B+w(k-lb)
B= Base Address (2000)
w=width of an element (4)
k=kth element of array (?) 3
lb=lower bound (0)   2000+4(3)=2000+12=2012

Lower bound is always zero. Upper bound = size -1

Finding address of an element

- Find the address of 12th element if the address of first element is 3572 and each element takes 2 bytes.
- A[i]=B+w(k-lb) 3572+2(12-0)=3572+24= 3596
- If the array has total 8 elements, calculate its upper bound and lower bound
- Consider that below array has 50 elements, calculate address of 32nd element. Width of each element is 4 bytes.

Array M	20	30	35	15
Memory	1414			

1414+4(32-0)=1414+128=1542

A[i]=B+w(k-lb) B=Base address w=Width K=element lb=lower bound

Inserting an element

Array M	6	7	5	8	9
Index	0	1	2	3	4

n=4

pos=2

num = 5

M[0]=6	M[0]=6
M[1]=7	M[1]= 7
M[2]=8	M[2]= 5
M[3]=9	M[3] = 8
M[4]=	M[4]= 9

Inserting an element

Array	6	7	5	8	9
Index	0	1	2	3	4

```
n=4
index=2
num = 5

for(i=n(2) ;i>index(2) ;i--)
{
M[i]=M[i-1];  //M[3]=M[2]
}
M[index]=num;    //M[2]=5

n++;
```

Deleting in an array

Array	**2**	**6**	**8**	**8**
Index	0	1	2	3

int M[10];

n= 4

index= 1

Before Deletion(n=4)	**After Deletion(n=3)**
M[0]=2	M[0]=2
M[1]=4	M[1]=6
M[2]=6	M[2]=8
M[3]=8	M[3]=

Deleting an element

```
for(i=index; i<n-1; i++)
{
M[i]= M[i+1];  //M[2]=M[3]
}
 n--;
```

Merging 2 arrays

Array M

Array	2	4	6	
Index	0	1	2	

Array N

Array	8	10		
Index	0	1		

Array Z by merging array M and N

Array	2	4	6	8	10
Index	0	1	2	3	4

Array M	Array N	Array Z
n1= 3	n2=2	n3=(n1+n2)=5

Merging 2 arrays

- Accept array M with n1 elements n1=3
- Accept array N with n2 elements   n2=2

```
K=0;
for(i=0;  i<n1(3); i++)
{
Z[i]=M[i];
K++; k=3
}

for(i=0; i<n2(2); i++)
{
Z[K]=N[i]; //Z[4]=N[1]
K++;
}
n3=n1+n2;
```

Z[0]=M[0]=2	**K=1**
Z[1]=M[1]=4	K=2
Z[2]=M[2]=6	K=3

Z[3]=N[0]= 8	**K=4**
Z[4]=N[1]=10	K=5

Splitting an array into 2

Array Z	10	20	30	40	50
Index	0	1	2	3	4

Array M	10	20	30
Index	0	1	2

Array N	40	50
Index	0	1

Splitting an array into 2

- Index=3,n=5
- Array M (0 to index-1)

```
for(i=0;i<index;i++)
   {
      M[2]=Z[2];
      printf("%d \t",M[i]);
   }
   n1=index; j=0;
```

- Array N (Index to n-1):

```
 For(i=index ;i<n ;i++)
   {
        N[i]=Z[j]; j++;
       printf("%d \t",N[i]);
      }
           n2=n-index;
```

Array Z	10	20	30	40	50
Index	0	1	2	3	4

Array M	10	20	30
Index	0	1	2

Array N	40	50
Index	0	1

Sorting an array

Before Sorting:

Array	**7**	**5**	**4**	**3**
Index	0	1	2	3

After Sorting

Array	**3**	**4**	**5**	**7**
Index	0	1	2	3

Sorting an array

```
for(i=0;i<n;i++)
{
  for(j=i+1;j<n;j++)
{
 if(M[i]>M[j])
{
   temp=M[i];
   M[i]=M[j];
   M[j]=temp;
}
}
}
```

Array M	3	4	5	7
Index	0	1	2	3

Array M	3	4	5	7
Index	0	1	2	3

Memory Representation of 2D array

int M[i][j]; i=rows j=column

M[0][0]	**M[0][1]**	**M[0][2]**
M[1][0]	M[1][1]	M[1][2]
M[2][0]	M[2][1]	M[2][2]

1

M[0][0]	**M[0][1]**	**M[0][2]**	**M[1][0]**	**M[1][1]**	**M[1][2]**	**M[2][0]**	**M[2][1]**	**M[2][2]**

Memory Representation of 2D array

Row Major:

1	2	3
4	5	6
7	8	9

1	2	3	4	5	6	7	8	9

Memory Representation of 2D array

2) Column Major:

M[0][0]	**M[0][1]**	**M[0][2]**
M[1][0]	M[1][1]	M[1][2]
M[2][0]	M[2][1]	M[2][2]

M[0][0]	**M[1][0]**	**M[2][0]**	**M[0][1]**	**M[1][1]**	**M[2][1]**	**M[0][2]**	**M[1][2]**	**M[2][2]**

Memory Representation of 2D array

1	2	3
4	5	6
7	8	9

1	4	7	2	5	8	3	6	9

Memory Representation of 2D array

FORMULA FOR CALCULATING ADDRESS OF AN ELEMENT USING ROW MAJOR ORDER

Address of (A[i][j]) = B+w(c(i-lbr)+(j-lbc))

Where,

B= Base element, contains address of first element of the array

W=Size of element i.e. number of memory cells

c= number of columns

lbr= lower bound of row

lbc= lower bound of column

Number of columns= ubc-lbc+1

Memory Representation of 2D array

FORMULA FOR CALCULATING ADDRESS OF AN ELEMENT USING ROW MAJOR ORDER

- There is an array A[4,7:-1,3] requires 2 byte to store each element. Calculate the address of A[6][2]. Given base address is 100

  Address of A[6][2]= B+w(c(i-lbr)+(j-lbc))

- An array X[-15,10:15,40] requires 1 byte of storage for each element. Calculate address of X[15][20]. Given base address is 1500.

Memory Representation of 2D array

- **FORMULA FOR CALCULATING ADDRESS OF AN ELEMENT USING COLUMN MAJOR ORDER**

Address of A[i][j]= B+w((i-lbr)+r(j-lbc))

Where,

B= Base element, contains address of first element of the array

W=Size of element i.e. number of memory cells

r= number of rows

lbr= lower bound of row

lbc= lower bound of column

R= Number of rows= UBR-LBR+1

Accepting a 2D array

```
int main()
{
int A[10][10], i, j, r, c;
printf("Enter number of rows for array A : ");
scanf("%d",&r);
printf("\n Enter number of columns for array A : ");
scanf("%d",&c);
for(i=0;i<r(3);i++)
{
for(j=0;j<c(2);j++)
{
printf("Enter value for A %d %d \n",i,j);
scanf("%d",&A[i][j]);
}
}
```

2D Array

- Addition of 2D arrays

- Addition is possible only when order of both the arrays are same i.e. No.of rows of array 1 is equal to No.of rows of array 2 and No.of columns of array1 is equal to No.of columns of array 2

- We can not add 2*3 array to 3*2 array.

- Accept first array M (rows r1, cols c1)
- Accept second array N (rows r2, cols c2)
- Check of rows of array M = rows of array N and columns of array M = columns of array N
- If(r1==r2 && c1==c2)
- Add elements of array M and array N and store in third array Z
- Z[i][j]=M[i][j]+N[i][j];
- Print Array Z which contains addition of array M and array N

Array M	Col 0	Col 1
Row 0	1	2
Row 1	3	4

+

Array N	Col 0	Col 1
Row 0	5	6
Row 1	7	8

=

Array Z	Col 0	Col 1
Row 0	6	8
Row 1	10	12

Sparse Array:

- Majority elements are zero element

6	0	1	0
0	3	0	1
0	0	2	0
0	0	0	7

- A sparse matrix store only non zero elements
- Diagonal matrix – All diagonal elements are non zero
- Upper triangular matrix – all the entries above diagonal are zero
- Lower triangular matrix - all the entries below diagonal are zero

Multiplication of 2 Array:

- Number of columns of first array should be equal to number of rows of second array
- Consider array M has r1 rows and c1 columns and array N has r2 rows and c2 columns
- i.e. c1==r2 then we can multiply array M & N

7	8	9
1	2	3

1	2
3	4
5	6

2X3

3x2

Equal

7*1 + 8*3 + 9*5 = 76	7*2 + 8*4 + 9*6 = 100
1*1 + 2*3 + 3*5 = 22	1*2 + 2*4 + 3*6= 28

2X2

Multiplication

Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 0 & 7 \end{bmatrix} = \begin{bmatrix} 1*5 + 2*0 & 1*6 + 2*7 \\ 3*5 + 4*0 & 3*6 + 4*7 \end{bmatrix} = \begin{bmatrix} 5 & 20 \\ 15 & 46 \end{bmatrix}$$

1	2	3	4
5	6	7	8

1	2
3	4
5	6
7	8

Advantages Of Array

- Simple & Convenient
- Easy to implement
- Contiguous memory location hence address of any element can be easily calculated
- No memory overflow
- Different dimensions available

Disadvantages Of Array

- Only Homogeneous elements can be stored
- It is static
- Operations like insertion, deletion are tedious to perform

- Accept first array M (rows r1, cols c1)
- Accept second array N (rows r2, cols c2)
- Check no.of columns M = No.of rows of N
- If(c1==r2)
- Multiply elements of array M and array N
- for(i=0; i<r1; i++)

  for(j=0 ;j<c2; j++)

  for(k=0; k<r2; k++)

  *sum = sum + M[i][k]*N[k][j];*

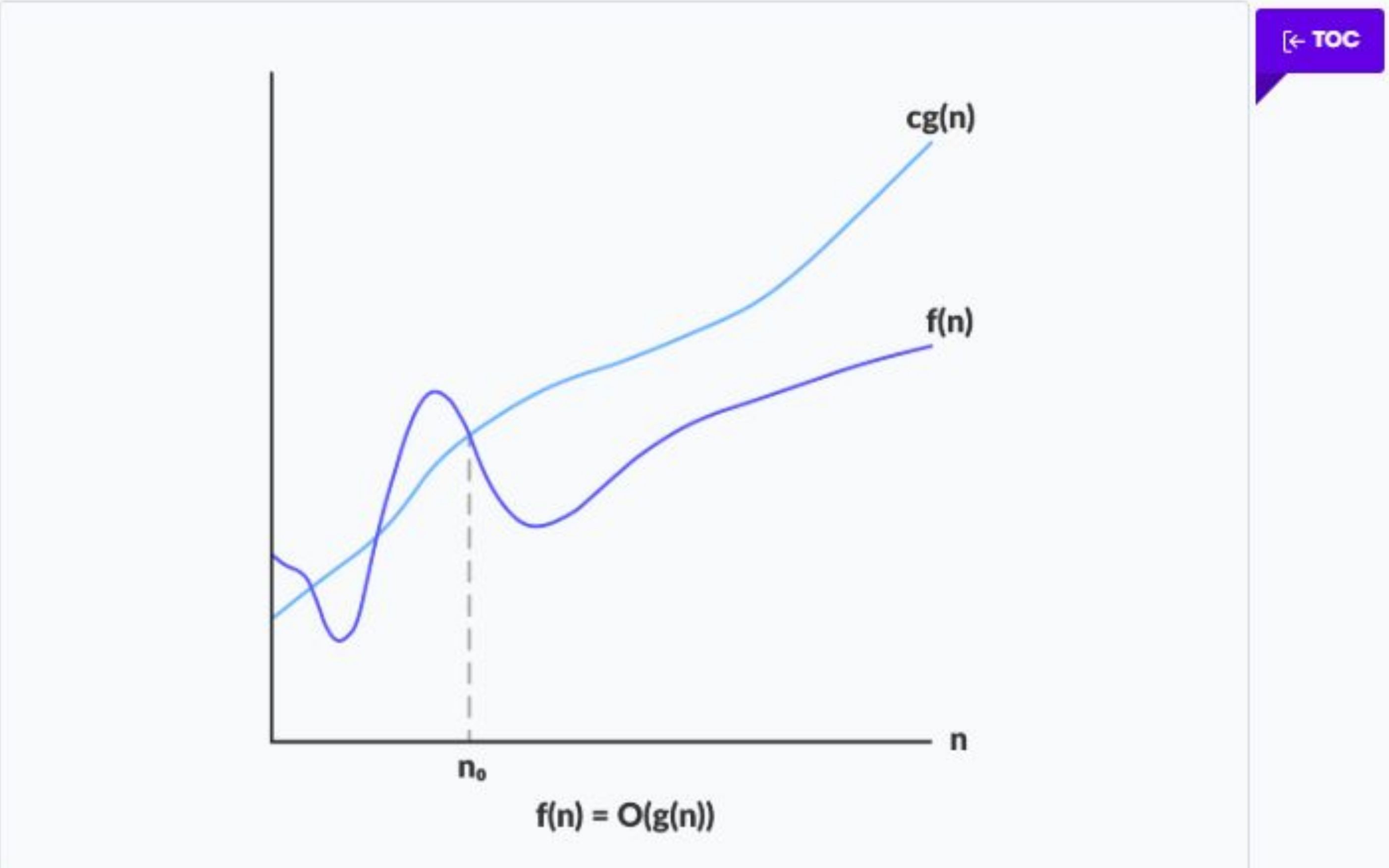
Asymptotic Notation

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value

- An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change

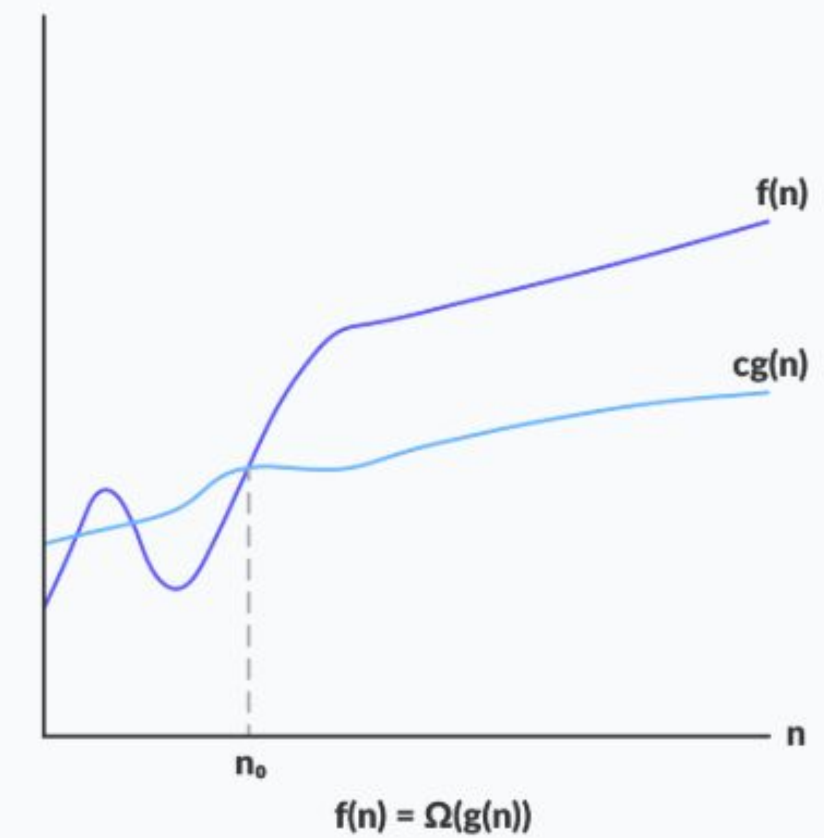
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis

Big 0



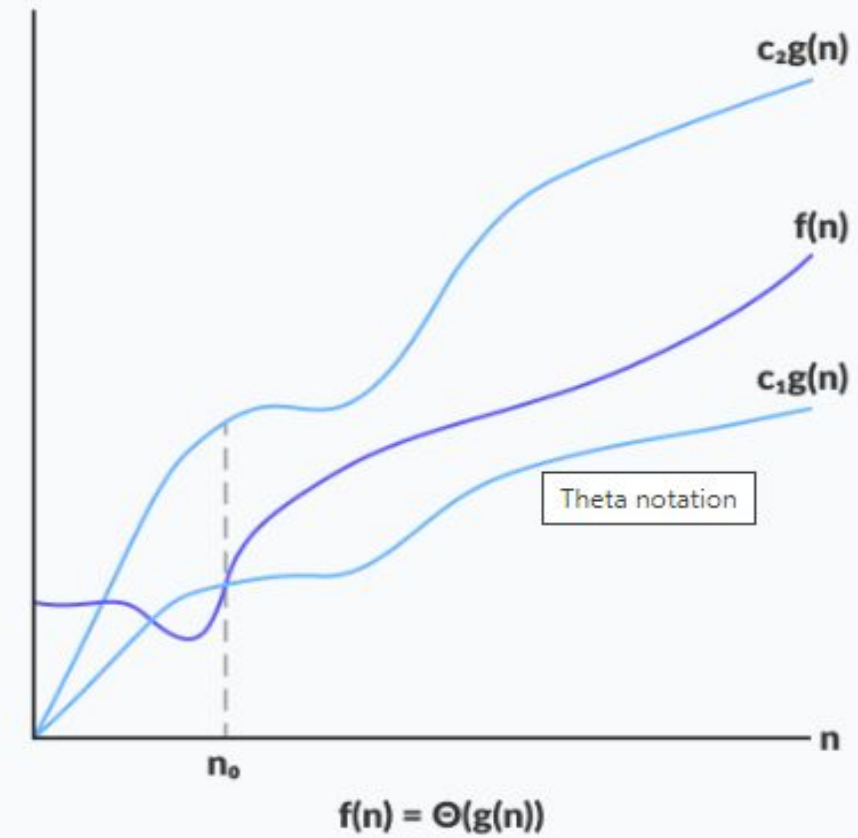
Big-O gives the upper bound of a function

Big omega



Omega gives the lower bound of a function

Big Theta



Theta bounds the function within constants factors