DATA STRUCTURES

-
  -
  -
  -
  -

Overview of Data Structures

- Data Structures are a method of representing of logical relationships between individual data elements related to the solution of a given problem.
- A data structure is a structured set of variables associated with one another in different ways, co-operatively defining the components in the system and capable of being operated upon in the program.
- Data structures are the basis of the programming tools. The choice of data structures should provide the following:

1. The data structure should satisfactorily represent the relationship between the data elements.
2. The data structure should be easy so that the programmer can easily process the data.

Classification of Data Structures

- **Linear:** The values are arranged in a linear fashion. E.g. Arrays, Linked Lists, Stacks, Queues etc.
- **Non-Linear:** The values are not arranged in an order. E.g. Tree, Graph, Table etc.
- **Homogeneous:** Here all the values stored are of same type e.g. arrays
- **Non- Homogeneous:** Here all the values stored are of different type e.g. structures and classes.
- **Dynamic:**A *dynamic* data structure is one that can grow or shrink as needed to contain the data you want stored. That is, you can allocate new storage when it's needed and discard that storage when you're done with it. E.g. *pointers,* or *references*
- **Static:**They're essentially fixed-size and often use much space E.g. Array

Basic Notations

Algorithmic Notations:

Finite step by step list of well-defined instructions for solving a particular problem.

Formal presentation of the Algorithm consists of two parts:

1. A paragraph that tells the purpose of an algorithm. Identifies the variables which occur in the algorithm and lists the input data.
2. The list of the steps that is to be executed.

Some conventions used in Algorithms:

- Identifying Number (Algorithm 2:)
- Steps, Control and Exit (All statements are numbered, Go to and Exit)
- Comments (The comments are to be given to understand the meaning)
- Variable Names (capital letters are used for variable names)
- Assignment Statement (Max:=DATA[1])
- Input and output (Read: variable names, Write: Messages and/or variable names)
- Procedure

Basic Notations

Control Structures:

1. **Sequential Logic or Sequential Flow**

2. **Selection Logic or Conditional Flow**

   - If (condition) , then:

     [Endif]

   - If (condition) , then:

     Else :

     [Endif]

- Multiple If (condition), then : -----

  Elseif : -----

  Else :

  [Endif]

3. Iteration Logic (Repetitive Flow)

- **for loop =>**

Repeat for K=I to N:

[Module] [End of Loop]

- While loop=>

Repeat while K<= N:

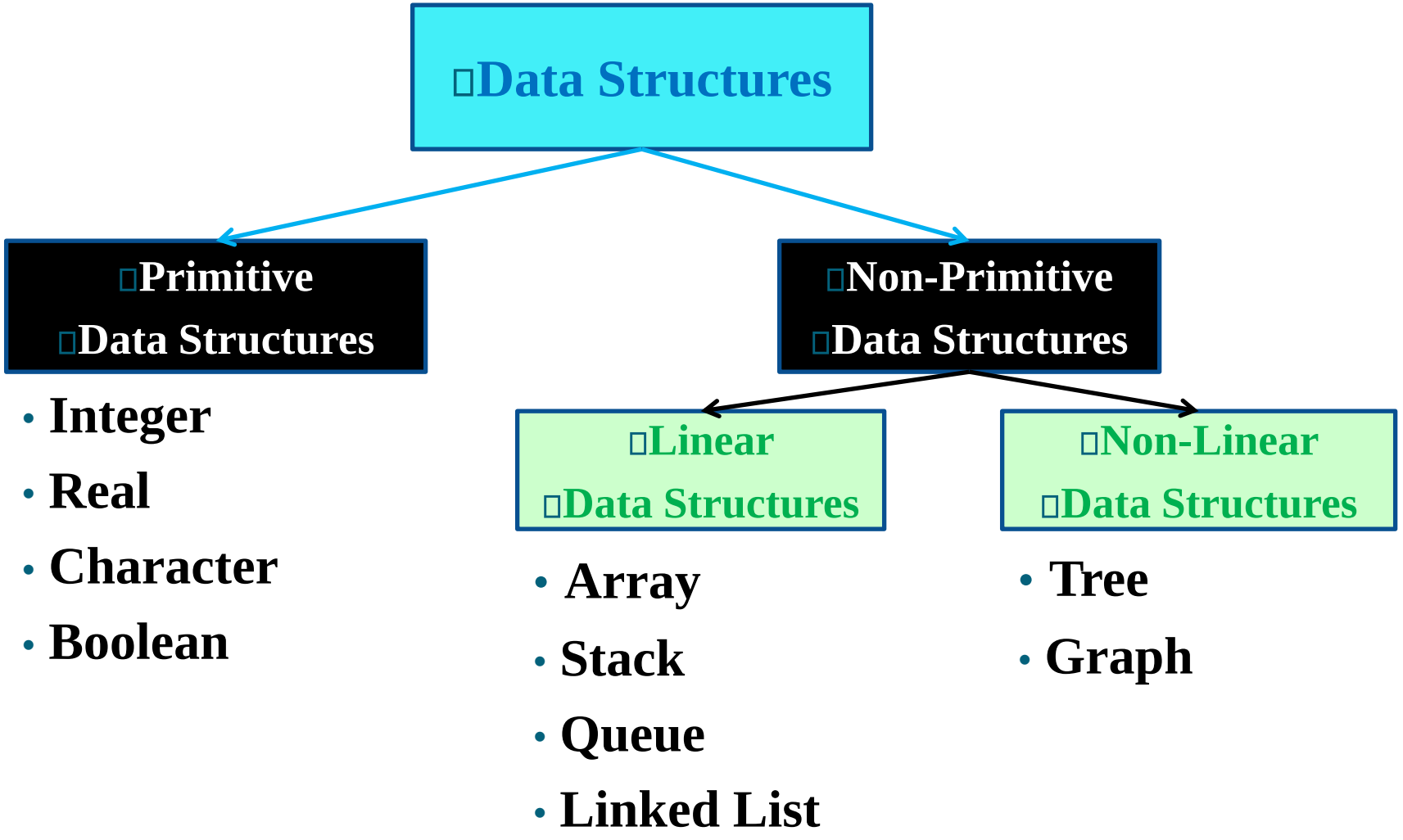
(logic)

[end of while loop]

Basic Concepts and Notations

- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data Structure: **Organization** of data needed to solve the problem

Classification of Data Structures



Data Structure Operations

Data Structures are processed by using certain operations.

1. Traversing: Accessing each record exactly once so that certain items in the record may be processed.

2. Searching: Finding the location of the record with a given key value, or finding the location of all the records that satisfy one or more conditions.

3. Inserting: Adding a new record to the structure.

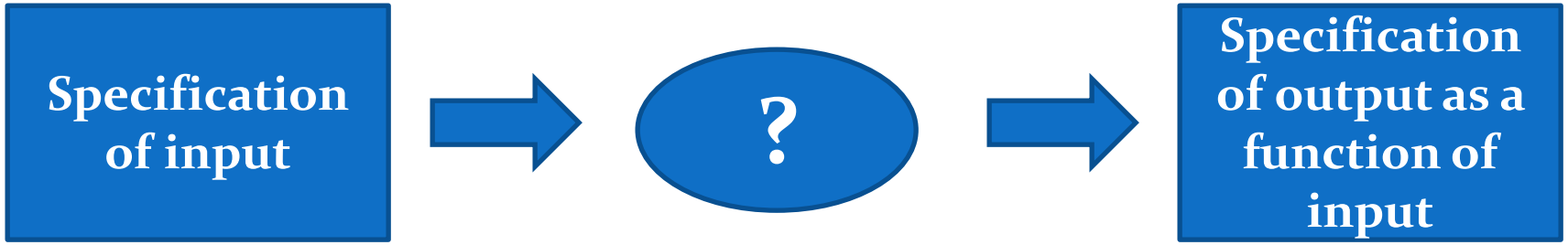
4. Deleting: Removing a record from the structure.

Special Data Structure-Operations

- Sorting: Arranging the records in some logical order (Alphabetical or numerical order).

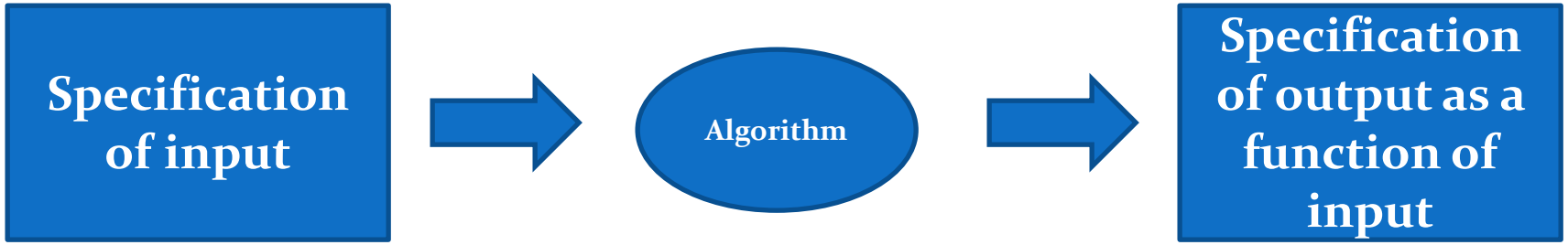
- Merging: Combining the records in two different sorted files into a single sorted file.

Algorithmic Problem



- Infinite number of input instances satisfying the specification. For example: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
  - 1, 20, 908, 909, 100000, 1000000000.
  - 3.

Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- Efficient:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - The number of bits in an input number
  - Number of data elements(numbers, points)

MCQs

. Which of these best describes an array?
a) A data structure that shows a hierarchical behaviour
b) Arrays are immutable once initialised
c) Container of objects of similar types
d) Array is not a data structure

- How do you initialize an array in C?
  a) int arr[3] = (1,2,3);
  b) int arr(3) = {1,2,3};
  c) int arr[3] = {1,2,3};
  d) int arr(3) = (1,2,3);

- How do you instantiate an array in Java?
  a) int arr[] = new int(3);
  b) int arr[];
  c) int arr[] = new int[3];
  d) int arr() = new int(3);

- Which of the following is a correct way to declare a multidimensional array in Java?
  a) int[] arr;
  b) int arr[[]];
  c) int[][]arr;
  d) int[[]] arr;

- What are the advantages of arrays?
  a) Objects of mixed data types can be stored
  b) Elements in an array cannot be sorted
  c) Easier to store elements of same data type
- d) Index of first element of an array is 1

- What are the disadvantages of arrays?
  a) Data structure like queue or stack cannot be implemented
  b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size
  c) Index value of an array can be negative
  d) Elements are sequentially accessed

- Assuming int is of 4bytes, what is the size of int arr[15];?
  a) 15
  b) 19
  c) 11
  d) 60

Complexity analysis

- Why we should analyze algorithms?
  - Predict the resources that the algorithm requires
    - Computational time (CPU consumption)
    - Memory space (RAM consumption)
    - Communication bandwidth consumption
  - The running time of an algorithm is:
    - The total number of primitive operations executed (machine independent steps)
    - It is a determination of order of magnitude of statement.
    - Also known as algorithm complexity

Time Complexity

- Worst-case
  - An upper bound on the running time for any input of given size
- Average-case
  - Assume all inputs of a given size are equally likely
- Best-case
  - The lower bound on the running time

Time Complexity – Example

- Sequential search in a list of size n
  - Worst-case:
    - `n` comparisons
  - Best-case:
    - `1` comparison
  - Average-case:
    - `n/2` comparisons

time space tradeoff

- A time space tradeoff is a situation where the memory use can be reduced at the cost of slower program execution (and, conversely, the computation time can be reduced at the cost of increased memory use).

- A **space-time** or **time**-memory **tradeoff** is a way of solving a problem or calculation in less **time** by using more storage **space** (or memory), or by solving a problem in very little **space** by spending a long **time**.

time space tradeoff

- As the relative costs of CPU cycles, RAM space, and hard drive space change—hard drive space has for some time been getting cheaper at a much faster rate than other components of computers—the appropriate choices for time space tradeoff have changed radically.

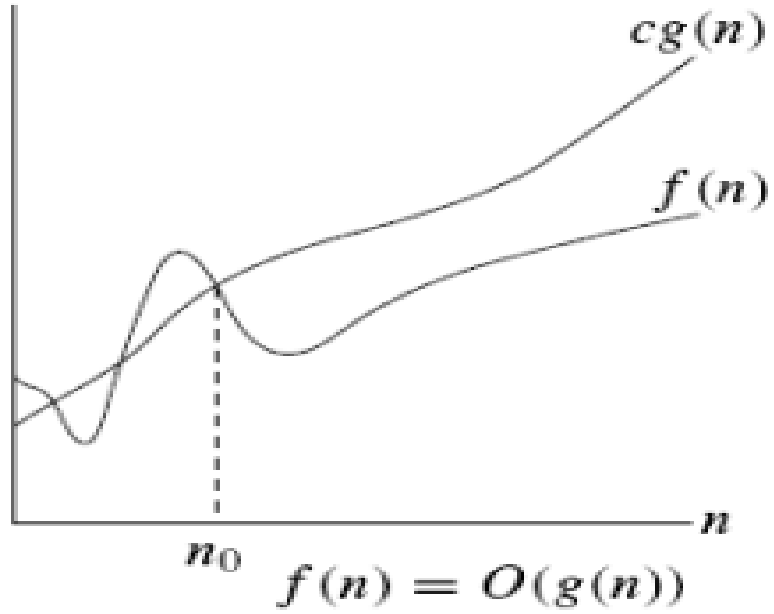
- Often, by exploiting a time space tradeoff, a program can be made to run much faster.

Asymptotic notations

- Algorithm complexity is rough estimation of the number of steps performed by given computation depending on the size of the input data
  - Measured through **asymptotic notation**
    - `O(g)` where `g` is a function of the input data size
  - Examples:
    - Linear complexity `O(n)` – all elements are processed once (or constant number of times)
    - Quadratic complexity $O(n^2)$ – each of the elements is processed `n` times

Big-O Notation (O)

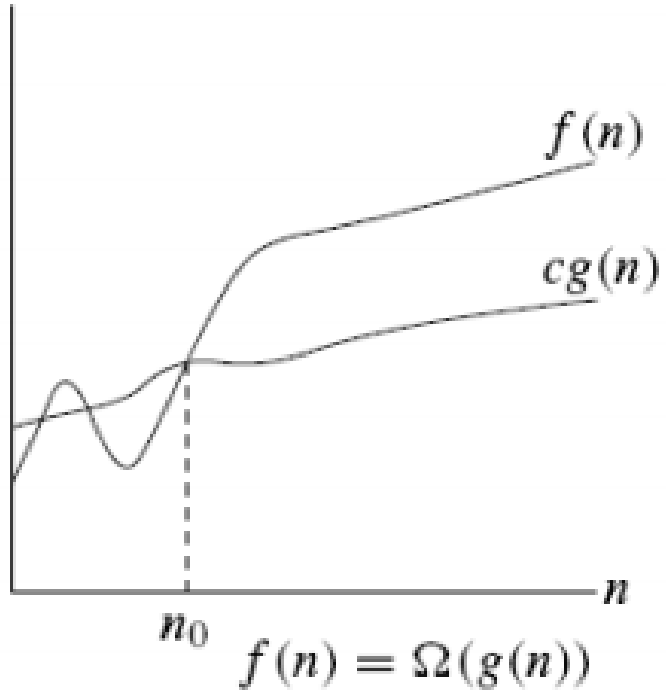
Asymptotic Upper Bound



$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\} .$$

Omega Notation (Ω)

Asymptotic Lower Bound

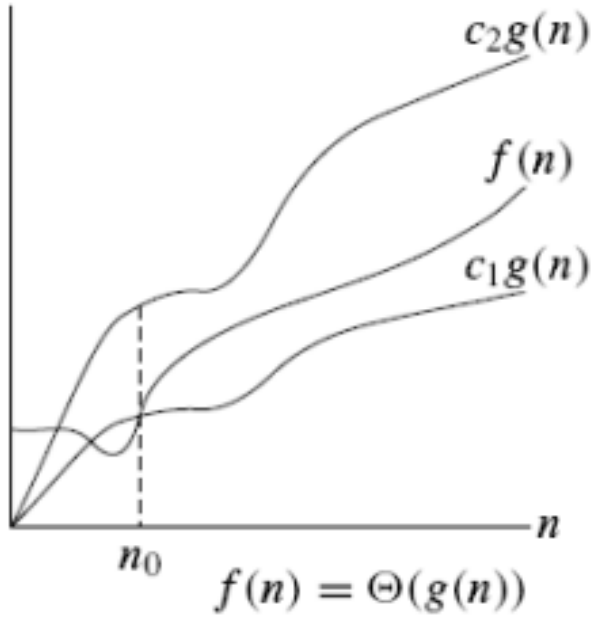


$f(n) = \Omega(g(n))$

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

Theta Notation (Θ)

g(n) is an asymptotically tight bound of f(n)



$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1g(n) \le f(n) \le c_2g(n)$ for all $n \ge n_0\}$.[1]

Big O notation

- f(n)=O(g(n)) iff there exist a positive constant c and non-negative integer n_o such that

$$f(n) \leq cg(n) \quad \text{for all } n \geq no.$$

- g(n) is said to be an upper bound of f(n).

Basic rules

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).

Example 1

```
//linear
for(int i = 0; i < n; i++) {
        cout << i << endl;
}
```

- Ans: O(n)

Example 2

```
//quadratic
for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++){
                //do swap stuff, constant time
        }
}
```

- Ans O(n^2)

Example 3

- for(int i = 0; i < 2*n; i++) {
- 	cout << i << endl;
- }

- At first you might say that the upper bound is O(2n); however, we drop constants so it becomes O(n)

Example 4

```
//linear
for(int i = 0; i < n; i++) {
        cout << i << endl;
}

//quadratic
for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++){
                //do constant time stuff
        }
}
```

- Ans : In this case we add each loop's Big O, in this case n+n^2. O(n^2+n) is not an acceptable answer since we must drop the lowest term. The upper bound is O(n^2). Why? Because it has the largest growth rate

Example 5

```
for(int i = 0; i < n; i++) {
        for(int j = 0; j < 2; j++){
                //do stuff
        }
}
```

- Ans: Outer loop is 'n', inner loop is 2, this we have 2n, dropped constant gives up O(n)

Example 6

```
x=y+z;

for(i=1; i<= n; i++)
    x=y+z;

for(i=1; i<=n/2; i++)
    for(j=1; j<=n; j++)
        x=y+z;
```

Example 7

```
x=y+z;

    for(i=1; i<= n; i++)
        x=y+z;

    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
            x=y+z;
            a=b+c;
```

Example 8

```
while(n>1)
   {
      n=n-1;
      a=b+c;
   }
```

Example 10

```
while(n>=1)
   {
      n=n-20;
      n=n-5;
      n=n-2;

   }
```

Example 11

```
while(n>=1)
   {
      n=n-20;
      n=n+5;
      n=n-30;

   }
```

Example 12

```
while(n>=1)
   {
      n=n/2;
   }
```

Example 13

```
while(n>=1)
   {
      n=n/2;
      n=n/3;
   }
```

Example 14

```
while(n>=1)
   {
         n=n-2;
         n=n/2;

   }
```

Example 15

- for(int i = 1; i < n; i *= 2) {
- 	cout << i << endl;
- }

- There are n iterations, however, instead of simply incrementing, 'i' is increased by 2*itself each run. Thus the loop is log(n).

Example 16

```
for(int i = 0; i < n; i++) { //linear
        for(int j = 1; j < n; j *= 2){ // log (n)
                //do constant time stuff
        }
}
```

- Ans: n*log(n)

Example 17

```
while(n>2)
   {
      n=√n;
   }
```

Example 18

```
while(n>2)
   {
      n=n^2;
      n=√n;
      n=n-2;
   }
```

Example 19

```
x=y+z;

   for(i=1; i<= n; i++)
     {
        j=2;
        while(j<=n)
        {
           j=j2;
        }

     }
```

Example 20

- Main()
- {
- While(n>2)
- {
- n=√n;
- n=n/2;
- n=n-2;
- }
- }

Example 21

```
main()
{
While(n>1)
{
n=n/2;
a=a+b;
}
}
```

Example 22

- main()
- {
- While(n>=1)
- {
- n=n/2;
- }
- }

Example 23

- main()
- {
- While($n^3>1$)
- {
- n=n-1;
- a=b+c;
- X=x+y;
- }
- }

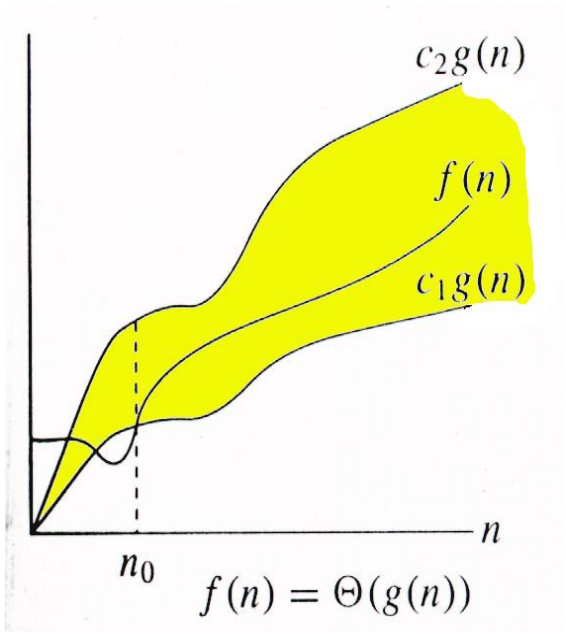
Example 24

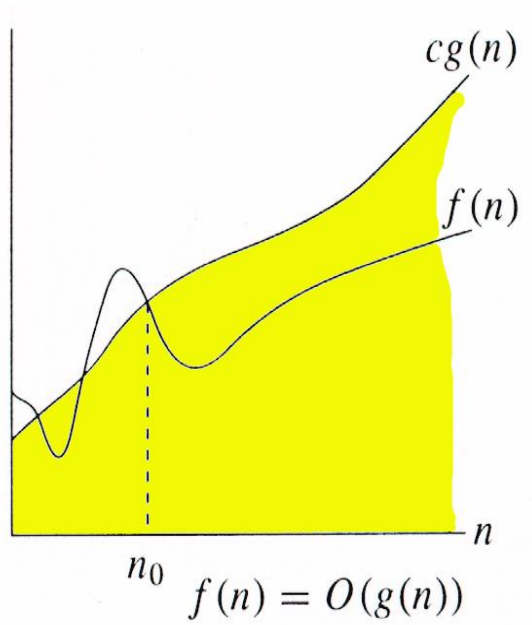
```
main()
{
  for(i=1;i<=n;i=3*i)
   {j=1;
    While(j<=n)
    {
      j=j*3;
    }
  }
}
```

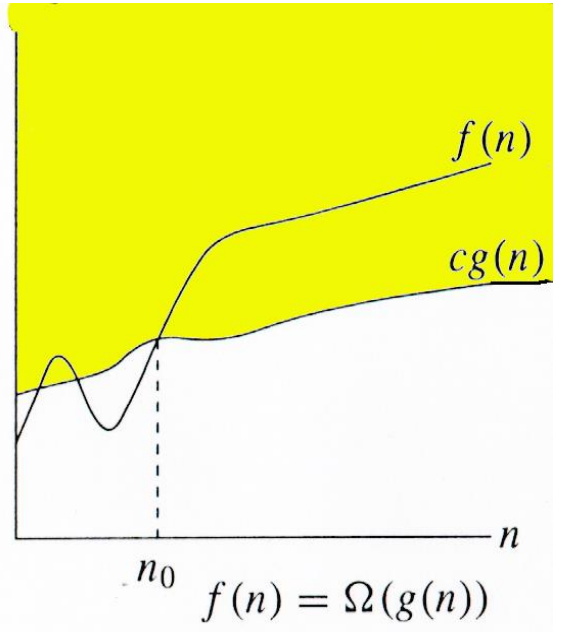
Example 25

```
main()
{
for(i=1;i<=n;i++)
{
   for(j=1;j<=i;j++)
    {
     for(k=1;i<=143;k++)
      {
        x=y+z;
      }
    }
}
}
```

Relations Between Θ, O, Ω







Thank You