Parul® University

Unit 1
Introduction and Searching

Computer science and Engineering

Parul®
University

DIGITAL LEARNING CONTENT

Topic-1

Introduction

Introduction

- "Data Structures and Algorithms" is one of the classic, core topics of Computer Science.
- Data structures and algorithms are central to the development of good quality computer programs.
- **Program= algorithm + Data Structure**

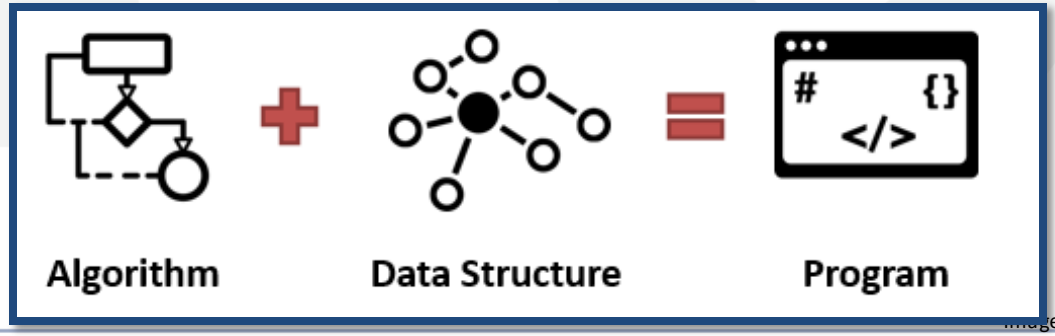


Image source : Google

Introduction

- **What is Algorithms?**

In mathematics and computer science, an **algorithm** is a set of instructions, typically to solve a class of problems or perform a computation.

- algorithm is a step by step procedure to solve a particular function.

Introduction

- It is important for every Computer Science student to understand the concept of **Information** and *how it is organized or how it can be utilized*.

- **What is Information?**

If we arrange some data in an appropriate sequence, then it forms a Structure and gives us a meaning. This meaning is called *Information*

- Two things in Information: One is **Data** and the other is **Structure** .

Introduction

What is Data?

Facts and statistics collected together for reference or analysis.

What is Data Structure?

A data structure is a systematic way of organizing and accessing data.

- A *data structure* tries to structure data!
  - Usually more than one piece of data
  - Should define legal operations on the data
  - The data might be grouped together

Introduction

- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.

- It plays an important role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Introduction

- That means, **algorithm** is a set of instruction written to carry out certain tasks & the **data structure** is the way of organizing the data with their logical relationship retained.

- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.

- *Therefore algorithm and its associated data structures form a program.*

Parul®
University

DIGITAL LEARNING CONTENT

Topic-2

Classification of Data Structure

Classification of Data Structure

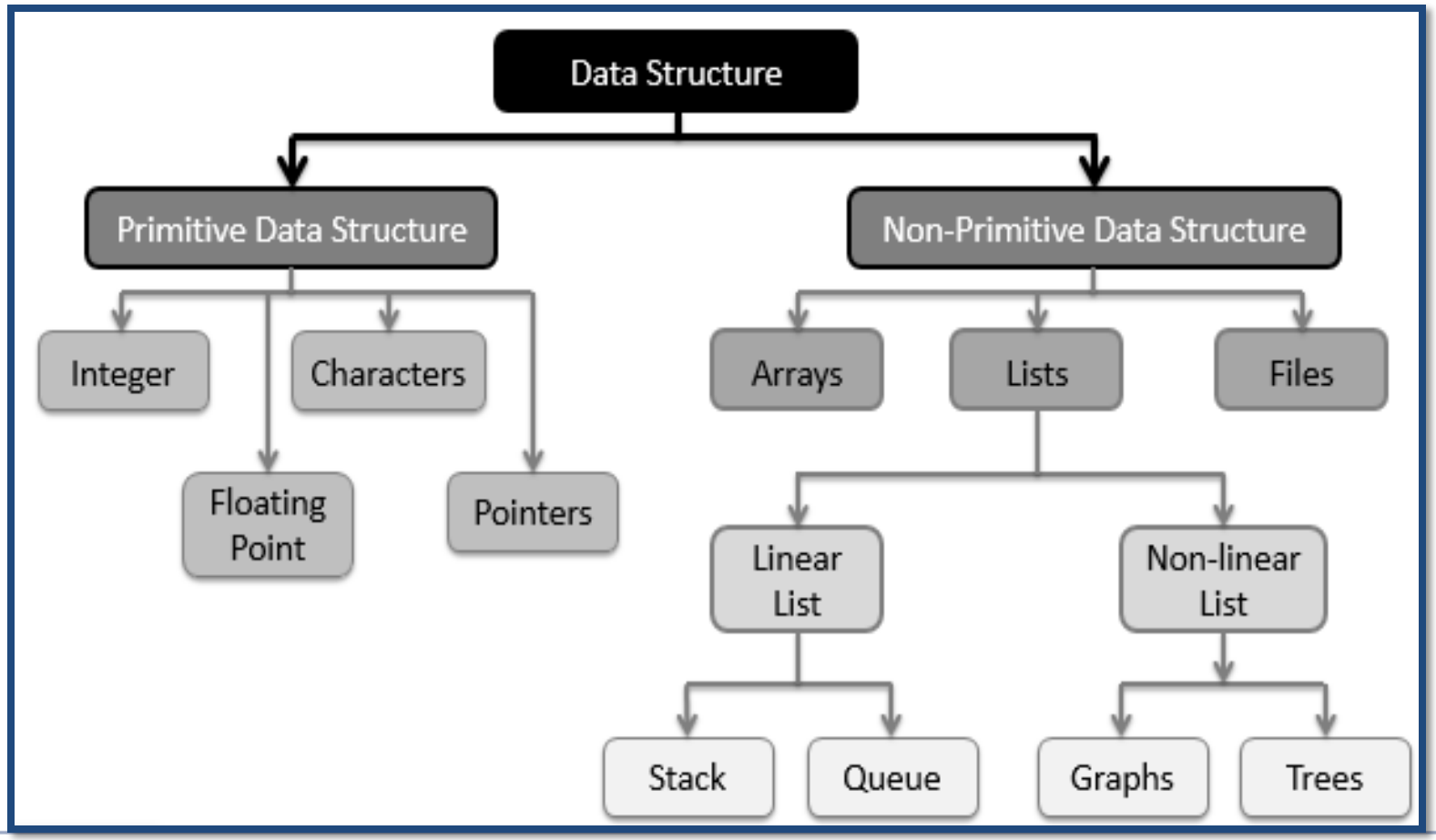


Image source : Google

Classification of Data Structure

Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- It have different representations on different computers.

- **Integer**: allows all values without fraction part.
- **Float**: used for storing fractional numbers.
- **Character**: used for character values.
- **Pointer**: holds memory address of another variable

Non Primitive Data Structure

- These are derived from primitive data structures.

- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.

- A Non-primitive data type is further divided into Linear and Non-Linear data structure

Non Primitive Data Structure

- **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
- **List:** An ordered set containing variable number of elements is called as Lists.
- **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

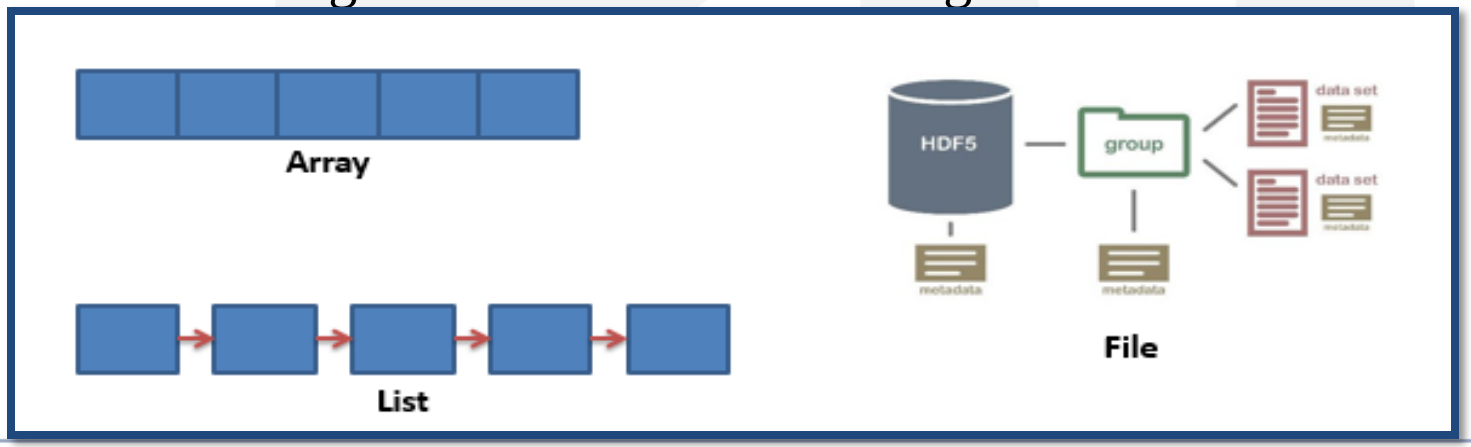


Image source : Google

Non Primitive Data Structure : Linear data structures

- Data is arranged in such a way that after one element we have just one more element that is, a single element is connected to just one more element after it.
- Examples of Linear Data Structure are Stack and Queue, linked list.

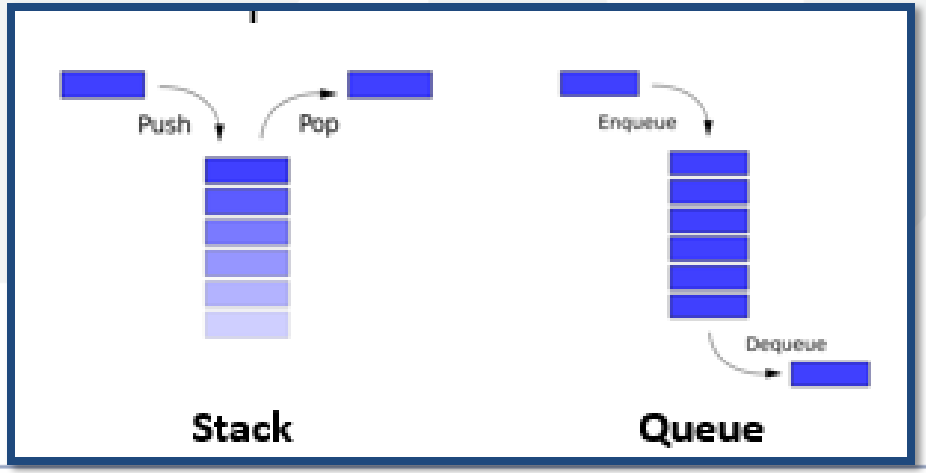


Image source : Google

Non Primitive Data Structure : Linear data structures

• **Stack**: Stack is ordered linear data structure which is modified form of an array. a data structure in which insertion and deletion operations are performed at one end only. Stack is also called as Last in First out (LIFO) data structure.

• **Queue**: The data structure which permits the insertion at one end and Deletion at another end, known as Queue. Queue is also called as First in First out (FIFO) data structure.

Non Primitive Data Structure : Non Linear data structures

- If after one element, we have connection to multiple elements then such data structures are called as non linear data structures.
- Examples of Non-linear Data Structure are Tree and Graph.

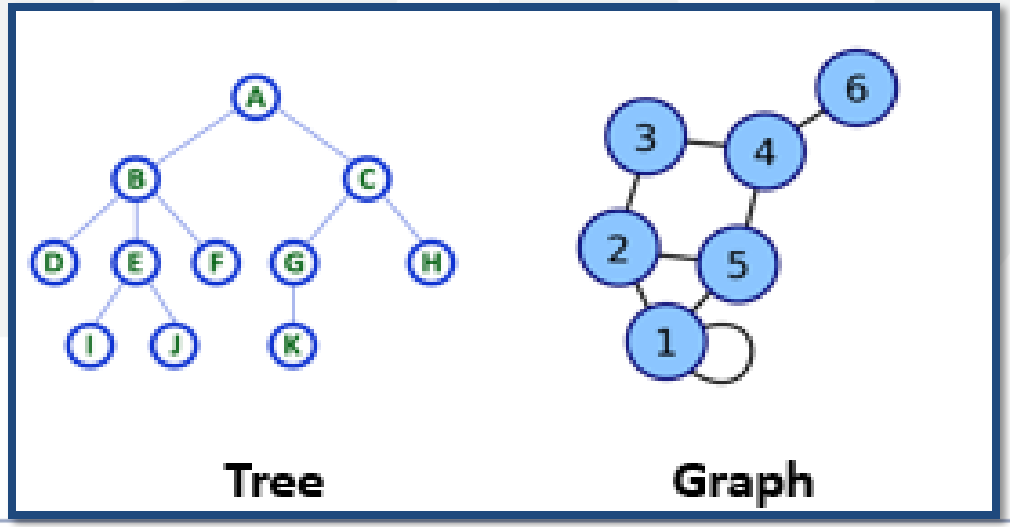


Image source : Google

Non Primitive Data Structure : Non Linear data structures

- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.

- Trees represent the hierarchical relationship between various elements.

- Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

Non Primitive Data Structure : Non Linear data structures

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
- A tree can be viewed as restricted graph.

- Graphs have many types:
  - Un-directed Graph
  - Directed Graph
  - Mixed Graph
  - Multi Graph
  - Simple Graph
  - Null Graph
  - Weighted Graph

Difference between Linear and Non Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

Parul®
University

DIGITAL LEARNING CONTENT

Topic-3

Operation on Data Structures

Operation on Data Structures

- Operation means processing the data in the data structure. The following are some important operations
- **Create**
  - The create operation results in reserving memory for program elements.
- **Destroy**
  - Destroy operation destroys memory space allocated for specified data structure
- **Selection**
  - Selection operation deals with accessing a particular data within a data structure

Operation on Data Structures

- **Traversing**
  - Traversal is a process of visiting each and every node of a list in systematic manner
- **Updation**
  - It updates or modifies the data in the data structure.
- **Searching**
  - To search for a particular value in the data structure for the given key value.
- **Sorting**
  - To arrange the values in the data structure in a particular order.

Operation on Data Structures

- **Inserting**
  - To add a new value to the data structure
- **Deleting**
  - To remove a value from the data structure
- **Splitting**
  - Splitting is a process of partitioning single list to multiple list.
- **Merging**
  - To join two same type of data structure values

Parul®
University

DIGITAL LEARNING CONTENT

Topic-4

Analysis of an Algorithm

Algorithm

- Algorithms are the ideas behind computer programs.

- **What is algorithm? :** It is a finite set of instruction for performing a particular task.

- Data structures are implemented using algorithms.

The efficient algorithm

- There is always more than one algorithm to solve particular problem.

- Now there question out of many choices, which algorithm is to be used? Answer is, for that we need to analysis algorithms.

- To compare the performance of different algorithm and choose the best one to solve a particular problem, analysis of algorithm is needed.

The efficient algorithm

What is to be analyzed?

Algorithm Analysis Measures

1. Input size
2. Measuring Time complexity
3. Measuring Space complexity
4. Computing Best case, Average case, Worst case
5. Computing order of growth of algorithm.

1. Input size

- Input size depends on the problem being studied

- For many problems, such as **sorting** input size is the number of items in the input—for example, the array size n for sorting.

- If the input to an algorithm is a **graph**, the input size can be described by the numbers of vertices and edges in the graph

2. Time complexity

- The **amount of time** required by an algorithm to be executed is called its time complexity

- Time complexity is commonly estimated by counting the **number of elementary operations** performed on a particular input by the algorithm.

Time complexity- Example

- *Time complexity using frequency count*

```
for(i=0;i<n;i++)
{
printf("hello");
}
```

statement	count
i=0	1
i<n	n+1
i++	n
printf("hello")	n

Total =1+(n+1)+n+n=3n+2=O(n)

Image source : Google

Time complexity- Example

- *Time complexity using frequency count*

```
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
Count++
}
}
```

statement	count
i=0	1
i<n	n+1
i++	n
J=0	n
j<n	n(n+1)
j++	n*n
Count++	n*n

Total =1+(n+1)+n+n+n(n+1)+n*n+n*n

$=3n^2+4n+2 \qquad =O(n^2)$

Image source : Google

3. Space complexity

• Space Complexity of an algorithm is total **space taken** by the algorithm with respect to the input size

• We often speak of **"extra" memory** needed, not counting the memory needed to store the input itself.

•Space complexity is sometimes **ignored** because the space used is minimal and/or obvious, but sometimes it becomes an important issue as time.

Space complexity: Which one is better?

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

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

Image source : Google

4. Computing Best case, Average case, Worst case

- The best, worst and average cases of a given algorithm express what the resource usage is **at least, at most and on average,** respectively

- The resource being considered is running time, but it could also be memory or the other resource

Computing Best case

- In the best case analysis, we calculate lower bound on running time of an algorithm.

- We must know the case that causes minimum number of operations to be executed.

- In the linear search problem, the best case occurs when x is present at the first location.

Computing Average case

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.

- Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

- For the linear search problem we sum all the cases and divide the sum by (n)

Computing Worst case

- In the worst case analysis, we calculate upper bound on running time of an algorithm.

- We must know the case that causes maximum number of operations to be executed.

- e.g. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array.
- When x is not present, the search () functions compares it with all the elements of array one by one

5. Order of growth of algorithm

- Order of growth in algorithm means how the time for computation increases when you increase the input size.

- It really matters when your input size is very large.

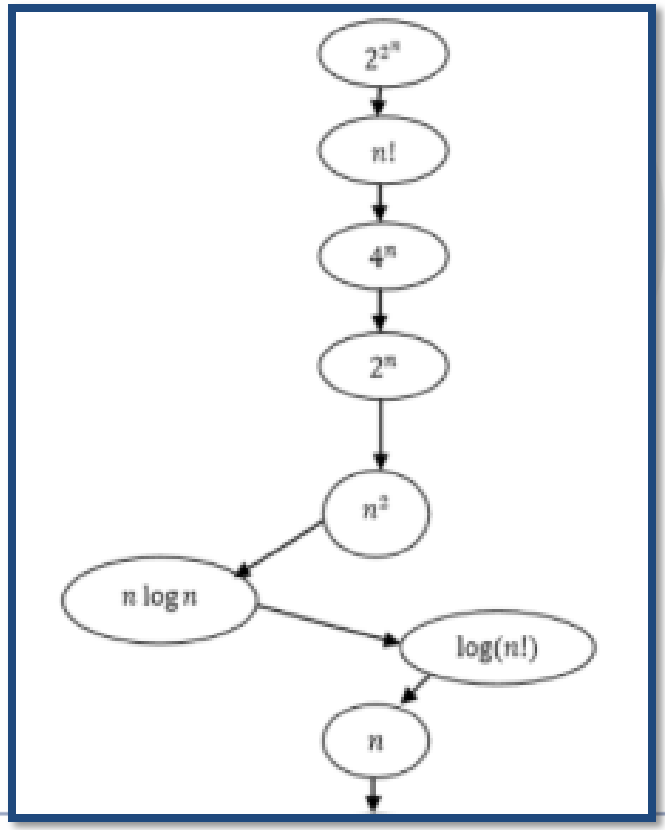
Order of growth of algorithm: example

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Image source : Google

Order of growth of algorithm:

Relationship between different rates of growth



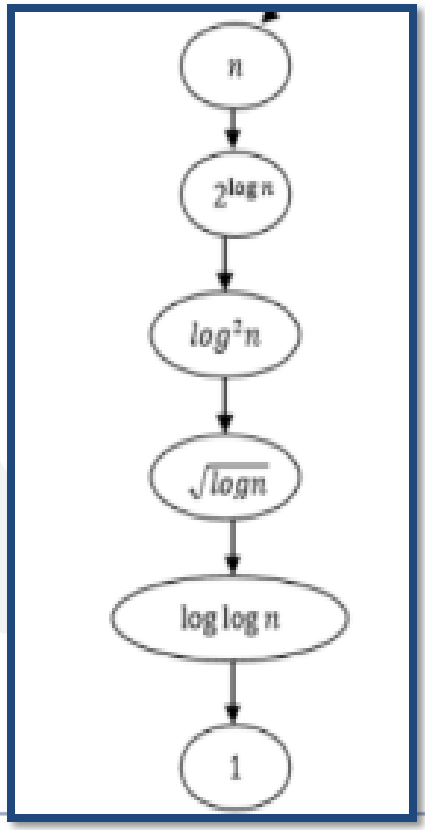


Image source : Google

Asymptotic notation

- It is used to express the rate of growth of an algorithm's running time in terms of the input size n.

Asymptotic notation

- To compare two algorithms with running times f(n) and g(n), we need a rough measure that characterizes **how fast each function grows.**

- O notation: asymptotic “less than”: (big oh)

  f(n)=O(g(n)) implies: f(n) “≤” g(n)

- Ω notation: asymptotic “greater than”: (omega)

  f(n)= Ω (g(n)) implies: f(n) “≥” g(n)

Asymptotic notation

- Θ notation: asymptotic “equality”: (theta)

  f(n)= Θ (g(n)) implies: f(n) “=” g(n)

- o notation: (small oh)

  f(n)=0(g(n)) implies: f(n) “<” g(n)

- ω notation: (little omega)

  f(n)= ω (g(n)) implies: f(n) “>” g(n)

O-Notation (Upper Bound)

- For a given function g(n), we denote by **O(*g*(*n*))**

  O(*g*(*n*)) = { f(n) : there exist positive constants c and n_0 such that 0 ≤ f (n) ≤ cg(n) for all n ≥ n_0 }

- We use O notation to give an upper bound on a function, to within a constant factor. For all values of n to the right of n_0, the value of the function f(n) is on or below g(n).

O-Notation (Upper Bound)

- We write f(n) = O(g(n))
- We say that g(n) is an asymptotically upper bound for f(n).

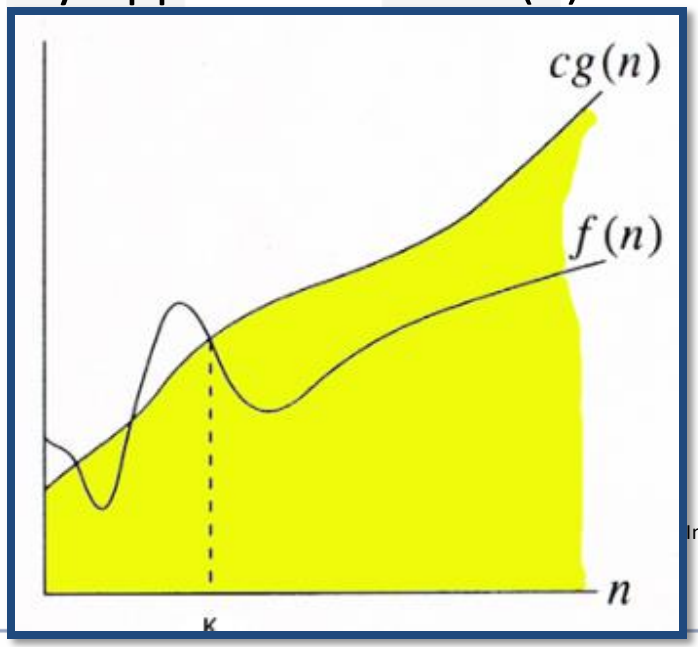


Image source : Google

O-Notation (Upper Bound): example

Let $f(n)=n^2$ and $g(n)=2^n$

n	$f(n)=n^2$	$g(n)=2^n$	
1	1	2	$f(n) < g(n)$
2	4	4	$f(n) = g(n)$
3	9	8	$f(n) > g(n)$
4	16	16	$f(n) = g(n)$
5	25	32	$f(n) < g(n)$
6	36	64	$f(n) < g(n)$
7	49	128	$f(n) < g(n)$

Here for $n \geq 4$ we have behavior $f(n) \leq g(n)$
Where $n_0=4$

Ω-Notation (Lower Bound)

- For a given function g(n), we denote by $\boldsymbol{\Omega(g(n))}$ the set of functions

- **$\Omega\,(g(n))$ = { f(n) : there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }**

- Ω Notation provides an asymptotic lower bound. For all values of n to the right of n_0, the value of the function f(n) is on or above cg(n).

Ω-Notation (Lower Bound)

- $f(n) = \Omega(g(n))$
- We say that g(n) is an asymptotically lower bound for f(n).

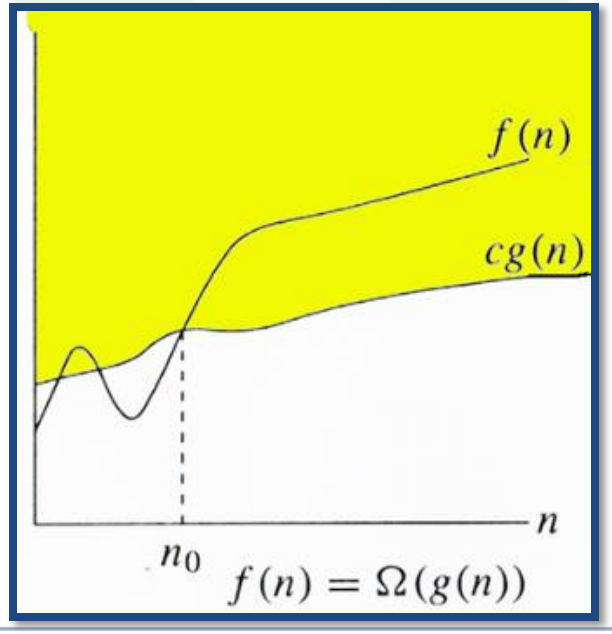


Image source : Google

Ω-Notation (Lower Bound): example

Let $f(n)= 2^n$ and $g(n)= n^2$

n	$f(n)= 2^n$	$g(n)= n^2$	
1	2	1	$f(n) > g(n)$
2	4	4	$f(n) = g(n)$
3	8	9	$f(n) < g(n)$
4	16	16	$f(n) = g(n)$
5	32	25	$f(n) > g(n)$
6	64	36	$f(n) > g(n)$
7	128	49	$f(n) > g(n)$

Here for $n \geq 4$ we have behavior $f(n) \geq g(n)$

Where $n_0=4$

Image source : Google

Θ-Notation

- For a given function g(n), we denote by **Θ(*g*(*n*))**

Θ(*g*(*n*)) = { f(n) : there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }

- This notation bounds a function to within constant factors. We say f(n) = Θ(g(n)) if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of f(n) always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.

Θ-Notation

- We write f(n) = **Θ** (g(n))

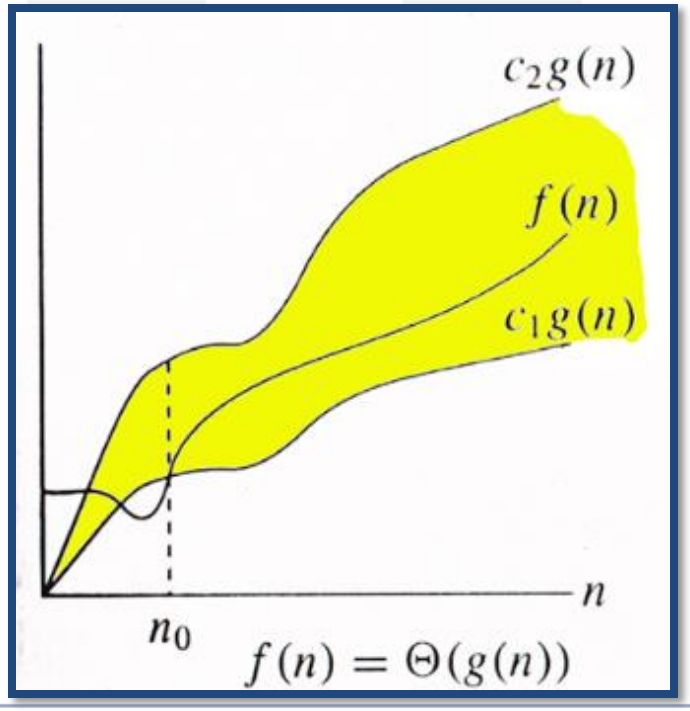


Image source : Google

Big-O complexity chart

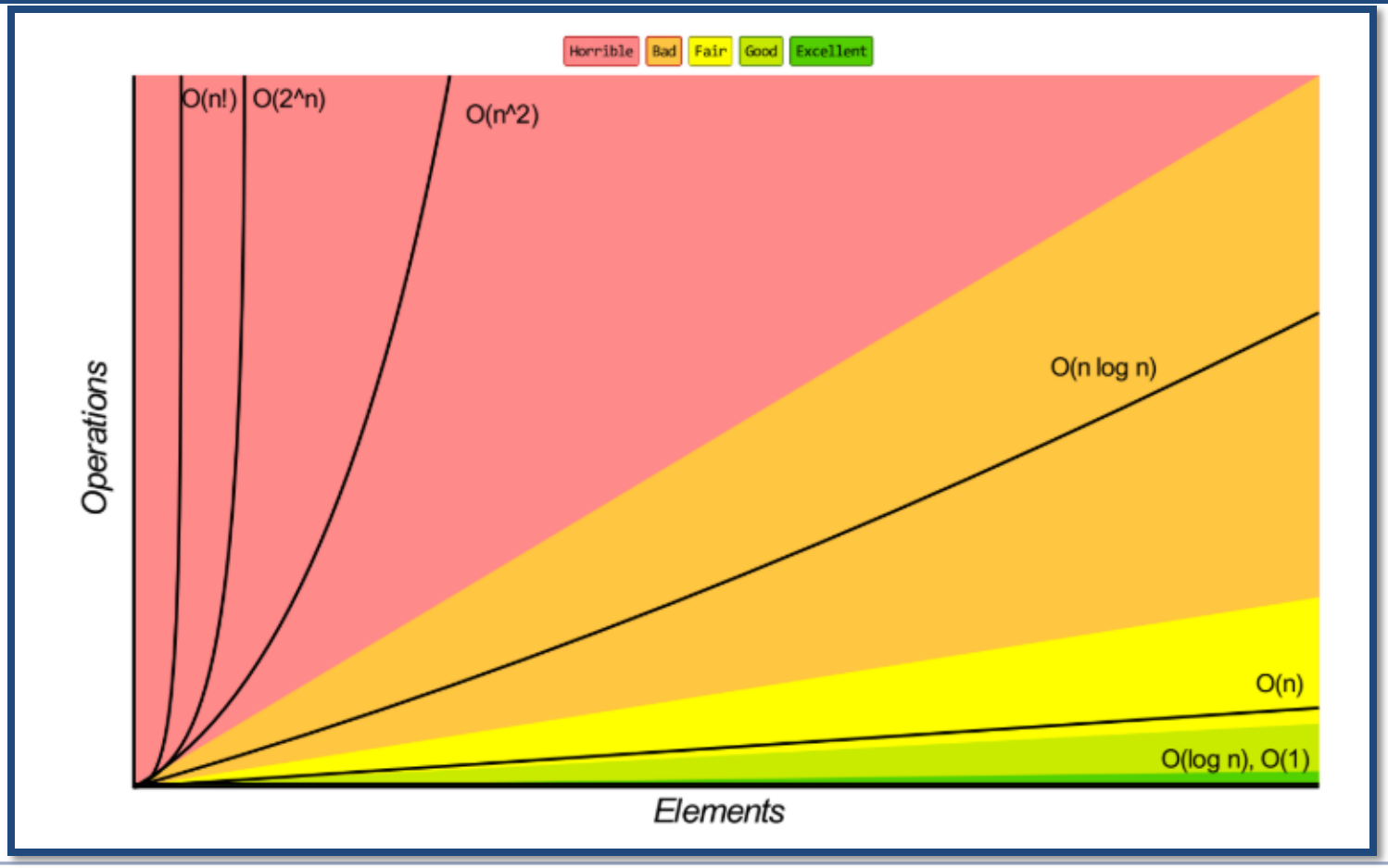


Image source : Google

Parul®
University

DIGITAL LEARNING CONTENT

Topic-5

Searching

Searching problem

- Searching algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

- Two categories
  - Linear search
  - Binary search

linear Search(sequential search)

- A search traverses the collection until
  - The desired element is found
  - Or the collection is exhausted

- If the collection is ordered, I might not have to look at all elements
  - I can stop looking when I know the element cannot be in the collection.

linear Search

- **Problem:** Given an array arr[] of n elements, write a function to search a given element x in arr[].

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
  - If x matches with an element, return the index.
  - If x doesn't match with any of elements, return -1.

linear Search: example

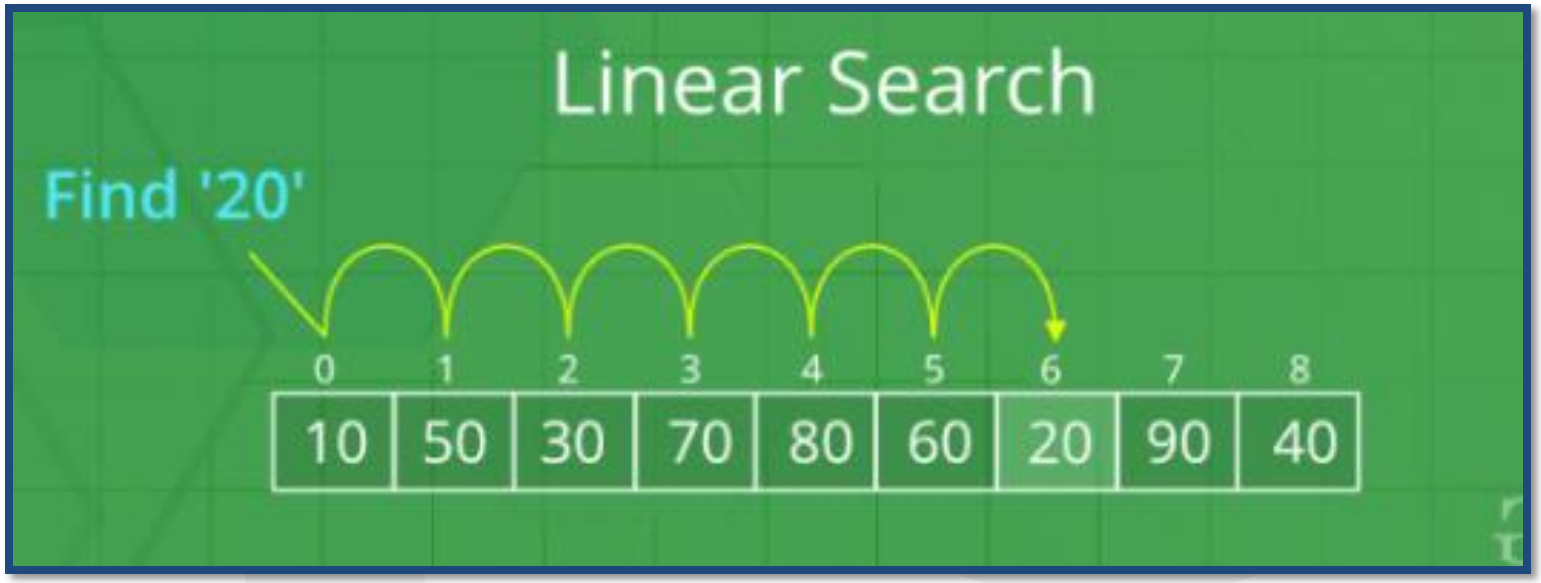


Image source :www.geeksforgeeks.org

linear Search: Complexity Analysis

- Worst case time complexity: **O(N)**
- Average case time complexity: **O(N)**
- Best case time complexity: **O(1)**
- Space complexity: **O(1)**

Binary Search (interval search)

- Search a sorted array by repeatedly dividing the search interval in half.

- Begin with an interval covering the whole array.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Binary Search

```
Binary_search ( x,n,A){
        low=1,high=n;          //x is search key
        while( low<=high ){
                mid=(low+high)/2;
                if( A[mid] > x ){
                        high= mid-1; }
                else if( A[mid] < x ){
                        low= mid+1; }
                else
                        return(mid);
        }
        return(false); }
```

Binary Search: example

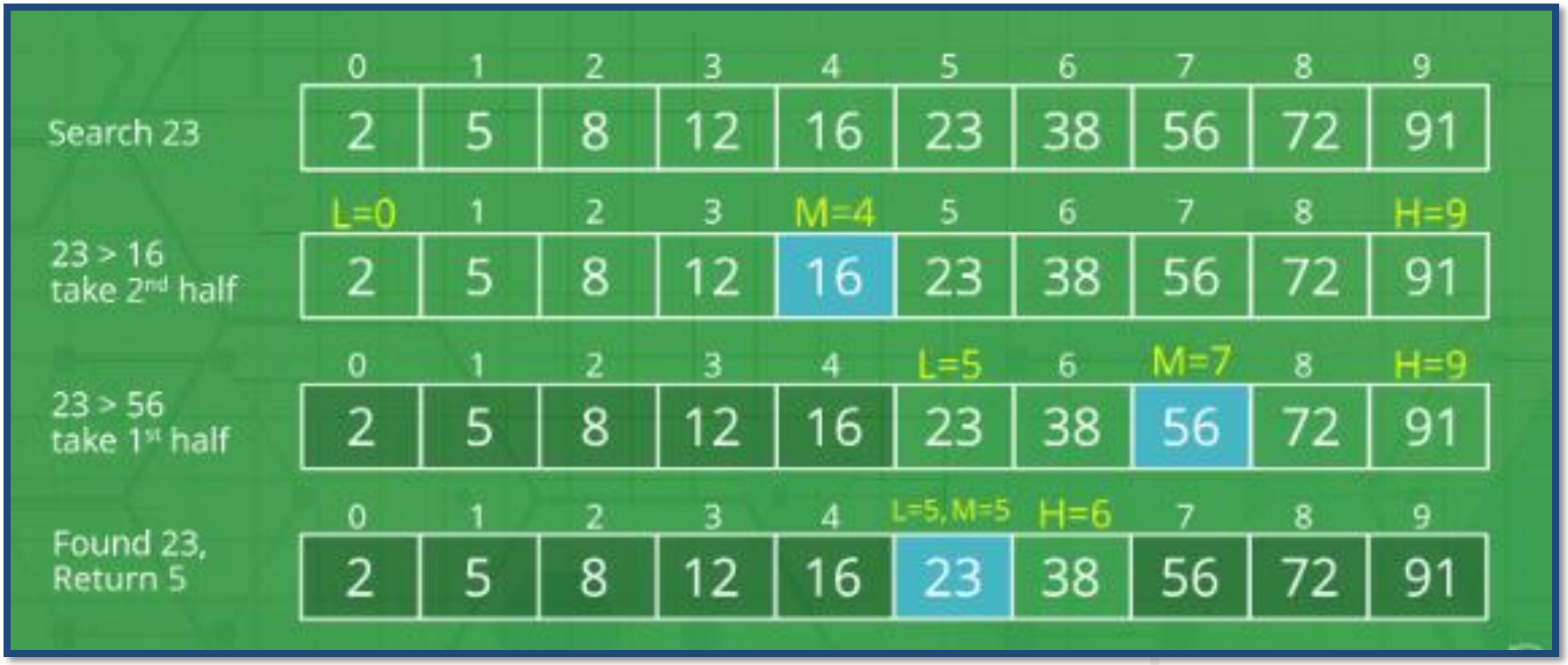


Image source :www.geeksforgeeks.org

linear Search: Complexity Analysis

- Worst case time complexity: **$O(\log_2(n))$**
- Average case time complexity: **$O(\log_2(n))$**
- Best case time complexity: **O(1)**
- Space complexity: **O(1)**

Binary Search: Calculating Time complexity

- At each iteration, the array is divided by half. So let's say the length of array at any iteration is **n**
- At Iteration 1,
  - Length of array = n
- At Iteration 2,
  - Length of array = n/2

Binary Search: Complexity Analysis

- At Iteration 3,

  •Length of array = (n/2)/2= $n/2^2$

- Therefore, after **Iteration k**

  •Length of array = $n/2^k$

- Also, we know that after

  •After k divisions, the length of array becomes 1

Binary Search: Complexity Analysis

- Therefore
  - Length of array = $n/2^k = 1$

    $\Rightarrow n = 2^k$

- Applying log function on both sides:

  $\Rightarrow \log_2(n) = \log_2(2^k)$

  $\Rightarrow \log_2(n) = k^* \log_2(2)$

- Therefore,

  $\Rightarrow k = \log_2(n)$

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in