MODULE 1

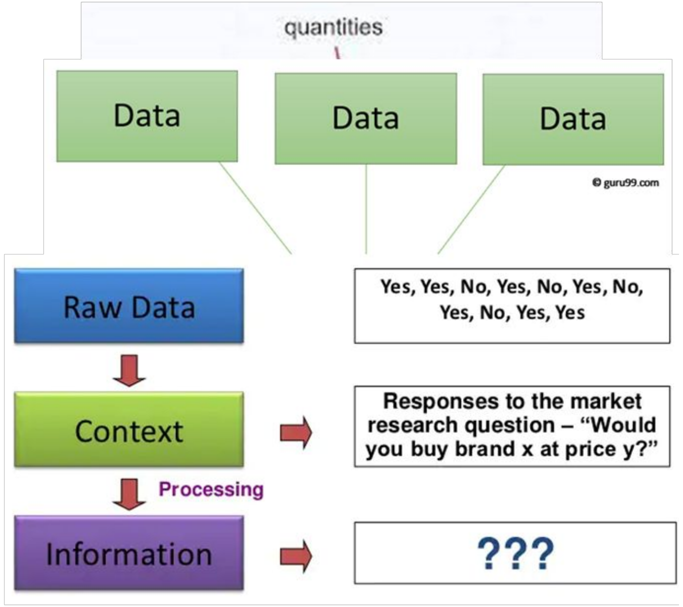
DATA STRUCTURES

OUTLINE

- Introduction – Definition of Data structures
- Data structure Types
- Algorithm Design
- Array Definition
- Memory representation of 1D Array
- Array operations - Insertion, Deletion, Search and Traversal
- Two Dimensional Arrays
- Function Associated with Arrays - Arrays as Parameters
- Recursive Functions.

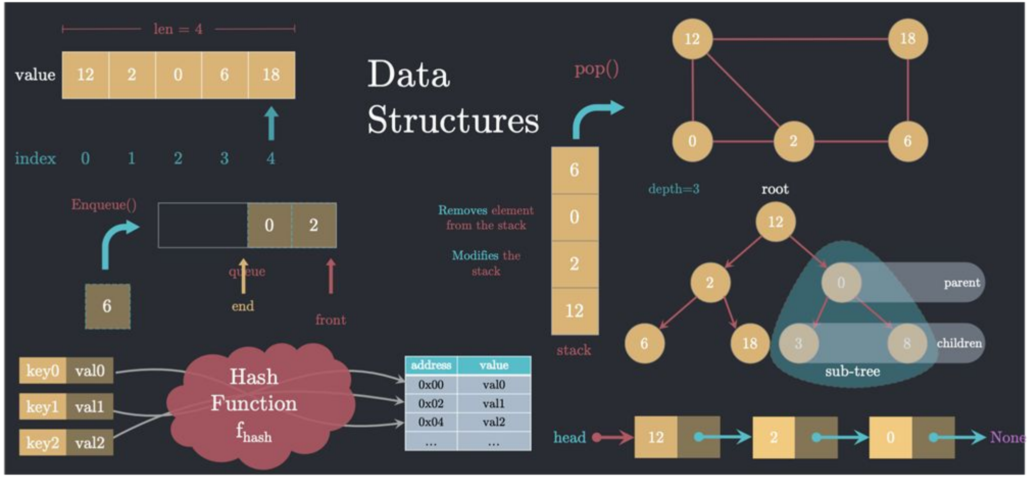
Introduction: Basic Terminology

- Data
  - Data are raw facts and set of values. 0-9, a to z,@#
- Information
  - The processed data is known as information.
- Data Item:
  - It refers to single unit of values
  - *Group Item*: Item that can be sub divided into sub items.
  - *Elementary Item*: can not be sub divided into sub items.



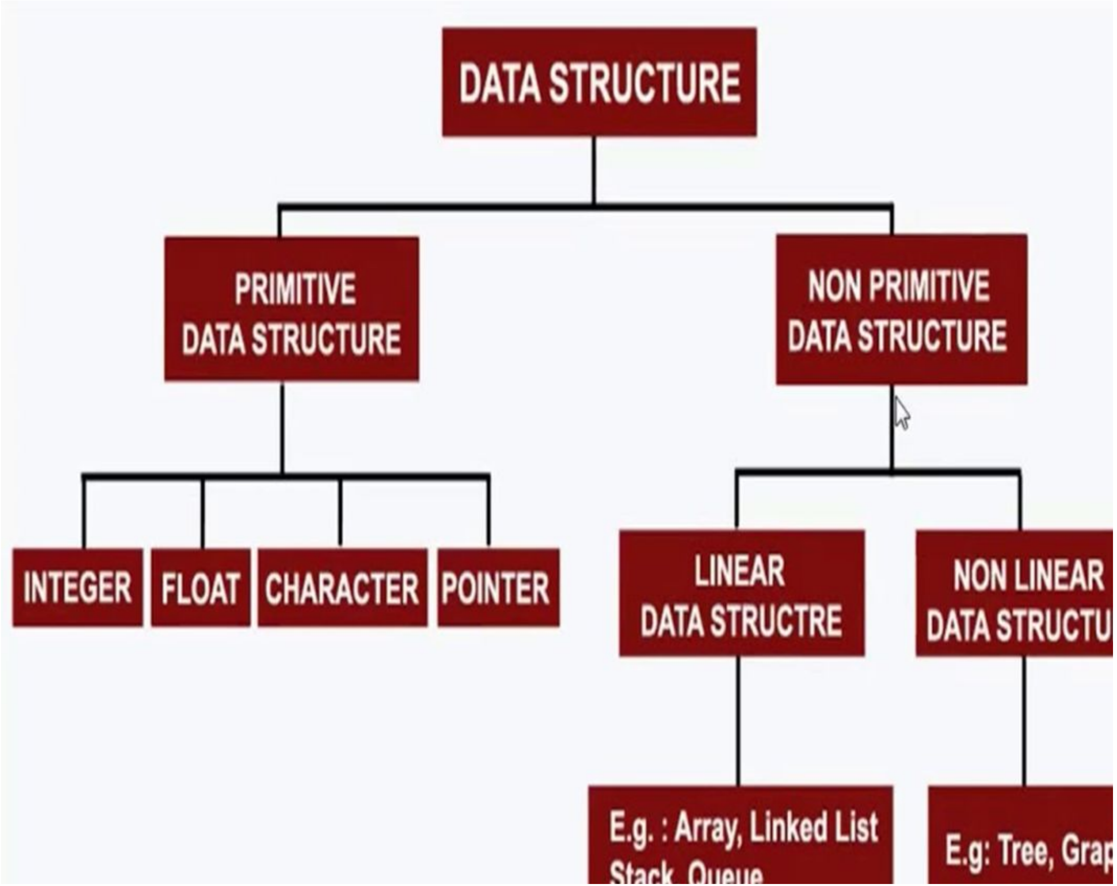
What is Data Structure?

- The logical or mathematical model of storing and organizing data in memory is called data structure.
  - List of names of months in a year
  - Location of historical places in a country.



Why data structure?

- Data structure, way in which data are stored for efficient search and retrieval.
- Meant to be an *organization for the collection of data items*.
- To solve real life problems efficiently
  - Insertion, deletion, search and sort
- Different data structures are suited for different problems.
- Some data structures are useful for simple general problems, such as retrieving data that has been stored with a specific identifier.

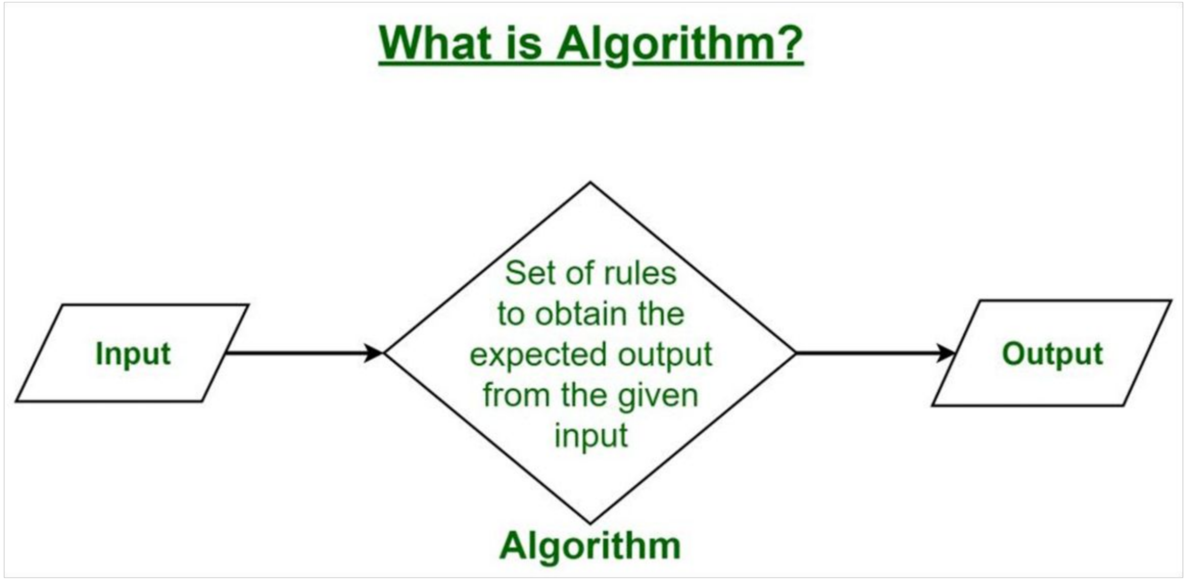

DATA STRUCTURE
PRIMITIVE
DATA STRUCTURE
NON PRIMITIVE
DATA STRUCTURE
INTEGER
FLOAT
CHARACTER
POINTER
LINEAR
DATA STRUCTRE
NON LINEAR
DATA STRUCTU
E.g. : Array, Linked List
E.g: Tree, Grap

Data Structure Types

- Primitive DS
  - Its a data structure that can hold a single value in a specific location
  - Eg float, character, integer and pointer.
- Non-primitive DS
  - Created from primitive DS
  - Can hold multiple values either in a contiguous location or random locations
  - defined by the programmer
  - further classified into two categories, i.e., linear and non-linear data structure.

Defining algorithm ??

- An Algorithm is step by step instruction to solve the given problem



Expressing Algorithm

- Reasoning about an algorithm is impossible without a careful description of the sequence of steps to be performed.
- The three most common forms of algorithmic notation are:
- I. English,
- II. Pseudocode,
- III. A real programming language.

Expressing Algorithm- English

- Step1:start
- Step2:take var n1 &n2
- Step3:calculate the sum of n1&n2
- Step4:calculate the sub of n1&n2
- Step5:calculate the div of n1&n2
- Step 6:calculate the product of n1&n2
- Step7:print the output sum,sub,div,product
- Step 8:stop

Expressing Algorithm- Pseudocode

- Read n1 & n2
- Sum=n1+n2
- Sub=n1-n2
- Div=n1/n2
- Product=n1*n2
- Print sum,sub,div,product

Algorithm Characteristics

- A clearly specified set of instructions to solve a problem.

Well defined zero or more quantities are externally supplied

Input

Each instruction is clear and unambiguous

Definiteness

The algorithm terminates in a finite number of steps.

Finiteness

Each instruction must be primitive and feasible

Effectiveness

At least one quantity is produced

Output

Characteristics of an Algorithm

Preliminaries of algorithm

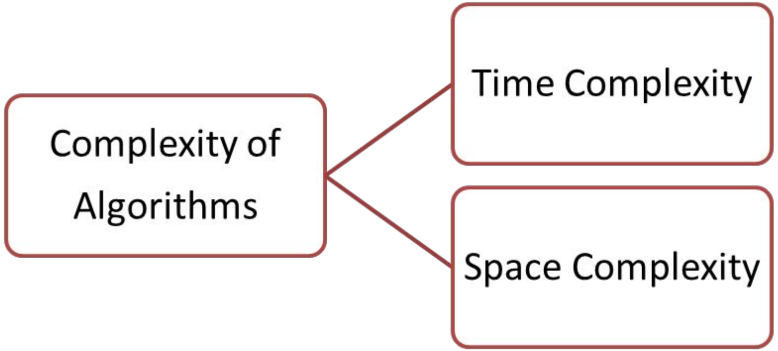
An Algorithm is simply a set of instructions. Genreally, the instructions must be specific enough that one can estimate the amount of work required to complete each step. Multiple algorithms can exist to solve the same problem or complete the same task. How do we choose which algorithm to use?

The appropriate algorithm can be determined based on an number of factors:

- How long the algorithm takes to run
- What resources are required to execute the algorithm
- How much space or memory is required
- How exact is the solution provided by the algorithm (is estimation involved, what about floating point rounding?)

Algorithm Analysis and its complexity

- Analysis: predict the cost of an algorithm in terms of resources and performance
- Efficiency or complexity of an algorithm is analyzed in terms of
  - cpu time and
  - memory.



- Amount of computational time required by an algorithm to perform complete task.
- Amount of memory required by an algorithm to complete its execution.

3 cases to analyse an algorithm

1) *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.

2) *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.

3) *Average Case Analysis*
   - we take all possible inputs and calculate computing time for all the inputs.
   - Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

Algorithm complexity

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Space Complexity: It's the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Time Complexity: Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes its execution in the minimum time possible.

Finding f(n)

We can compare two data structures for a particular operation by comparing their f(n) values.

We are interested in growth rate of f(n) with respect to n because it might be possible that for smaller input size, one data structure may seem better than the other but for larger input size it may not.

This concept is applicable in comparing the two algorithms as well.

Example

$$f(n) = 5n^2 + 6n + 12$$

For $n = 1$

$$\% \text{ of running time due to } 5n^2 = \frac{5}{5 + 6 + 12} \times 100 = 21.74\ \%$$

$$\% \text{ of running time due to } 6n = \frac{6}{5 + 6 + 12} \times 100 = 26.09\ \%$$

$$\% \text{ of running time due to } 12 = \frac{12}{5 + 6 + 12} \times 100 = 52.17\ \%$$

Squared term consumes 99% of running time (below table)

EXAMPLE

$f(n) = 5n^2 + 6n + 12$

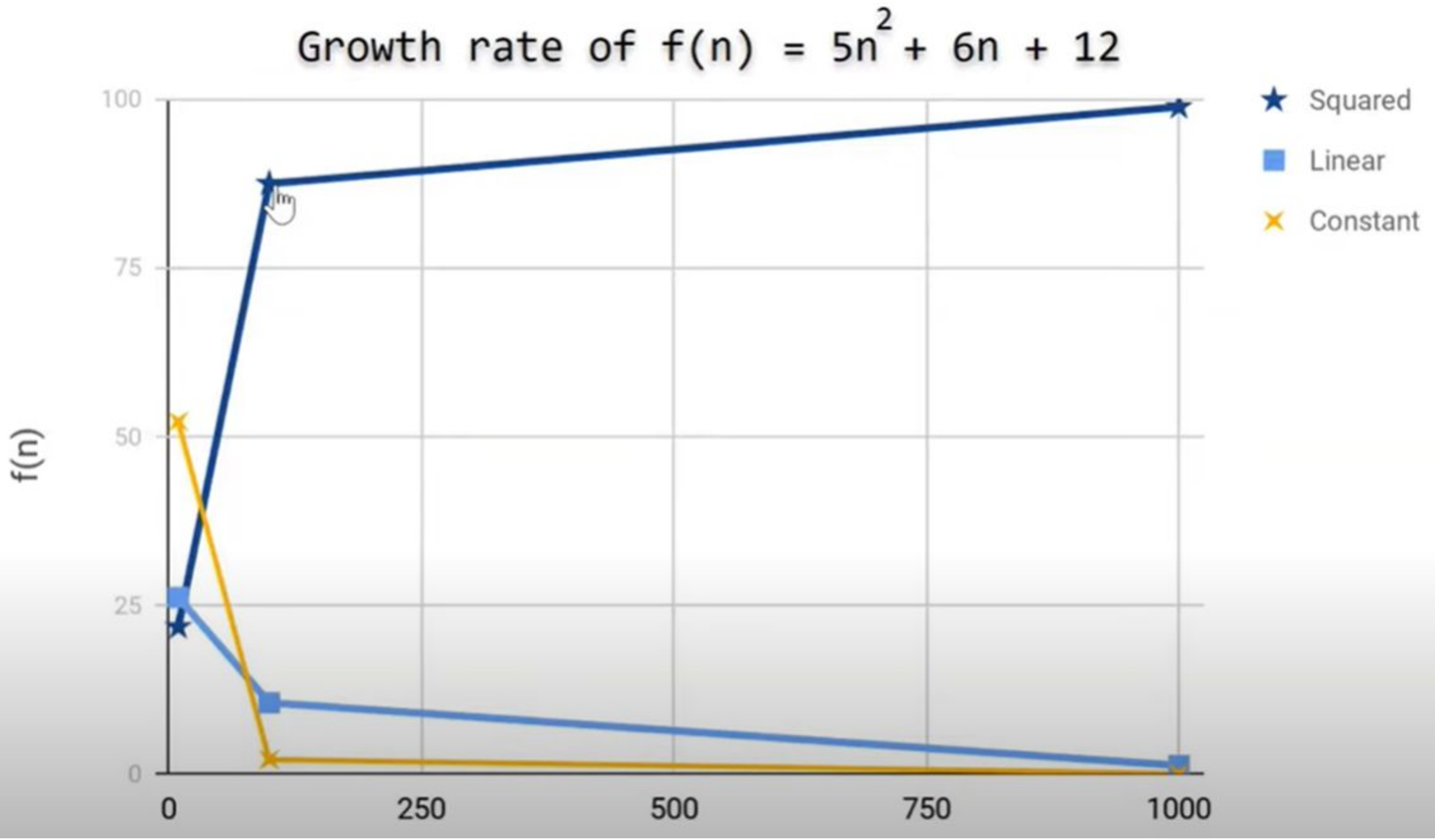
n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

Example

$$f(n) = 5n^2$$

We are getting the approximate time complexity and we are satisfied with this result because this approximate result is very near to the actual result.

This approximate measure of time complexity is called
Asymptotic Complexity


Growth rate of f(n) = 5n^2 + 6n + 12
100
75
50
25
0
f(n)
0
250
500
750
1000
Squared
Linear
Constant

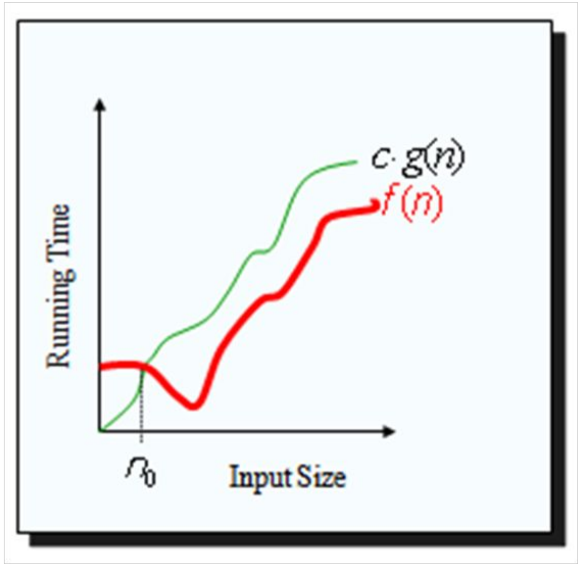
Asymptotic Notation: Big-oh Notation (O)

- Asymptotic upper bound used for worst-case analysis
- Let f(n) and g(n) are functions over non-negative integers
- if there exists constants c and n_0, such that

  $f(n) \le c\, g(n)$ for $n \ge n_0$
- Then we can write $f(n) = O(g(n))$
- Example $f(n) = 2n + 1$, $g(n) = 3n$

  i.e $f(n) \le 3n$

so we can say $f(n) = O(n)$ when $c = 3$



Big O Notation

Big O notation is used to measure the performance of any algorithm by providing the order of growth of the function.

It gives the upper bound on a function by which we can make sure that the function will never grow faster than this upper bound.

We want the approximate runtime of the operations performed on data structures.
We are not interested in the exact runtime.

Big O notation will help us to achieve the same.

Big O Notation

If $f(n)$ and $g(n)$ are the two functions, then

$$f(n) = O(g(n))$$

If there exists constants c and n_o such that
$f(n) \leq c.g(n)$, for all $n \geq n_o$

Examples

$f(n) = n$ $\qquad$ $g(n) = 2n$

Is $f(n) = O(g(n))$?

$f(n) \leq c.g(n)$

$n \leq c.2n$ $\qquad$ For $c = 1$ and $n_o = 1$

Examples

$f(n) = 4n + 3$ $\qquad$ $g(n) = n$

Is $f(n) = O(g(n))$?

$f(n) \le c.g(n)$

$4n+3 \le c.n$

Take $c = 5$

$4n+3 \le 5n$

$3 \le 5n - 4n$

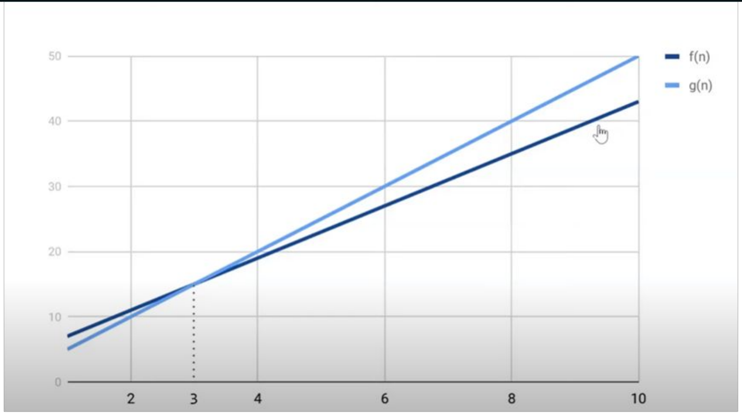
$n \ge 3$

Therefore,

$f(n) \le c.g(n)$

For all $n \ge 3$

where $c = 5$ and $n_o = 3$



$f(n) = 4n + 3$ $\quad$ $g(n) = n$

Is $f(n) = O(g(n))$?

$f(n) \leq c.g(n)$

$4n+3 \leq c.n$

Take $c = 5$

$4n+3 \leq 5n$

$3 \leq 5n - 4n$

$n \geq 3$

Therefore,

$f(n) \leq c.g(n)$

For all $n \geq 3$

where $c = 5$ and $n_o = 3$

$$\boxed{f(n) = O(n)}$$

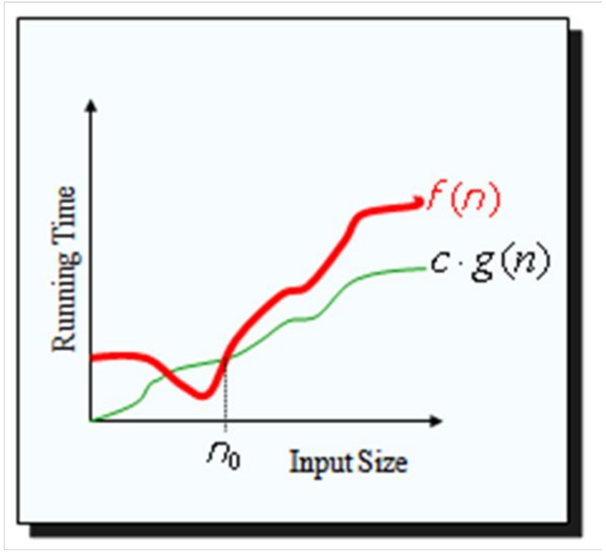
Asymptotic Notation: Omega Notation (Ω)

- Asymptotic lower bound used to describe best-case running times
- Let f(n) and g(n) are functions over non-negative integers
- if there exists constants c and n_0, such that

  $c\, g(n) \le f(n)$ for $n \ge n_0$
- Then we can write $f(n) = \Omega(g(n))$,
- Example $f(n) = 18n+9$, $g(n) = 18n$

  i.e $f(n) \ge 18n$

so we can say $f(n) = \Omega(n)$ when $c = 18$



Asymptotic Notation: Omega Notation (Ω)

- Consider $f(n) = 2n^2 + 5$, $g(n) = 7n$. Find some constant c such that $f(n) \geq g(n)$
- For n=0, f(n) = 0+5=5, g(n) = 0 -> f(n)>g(n)
- For n=1, f(n) = 2*1*1+5 = 7, g(n) = 7 -> f(n) = g(n)
- For n=3, f(n) = 2*3*3+5 = 23, g(n) = 7*3 =21 -> $f(n) \geq g(n)$

- Thus for n > 3, $f(n) \geq g(n)$ ->always lower bound.

Asymptotic Notation: Theta Notation (Θ)

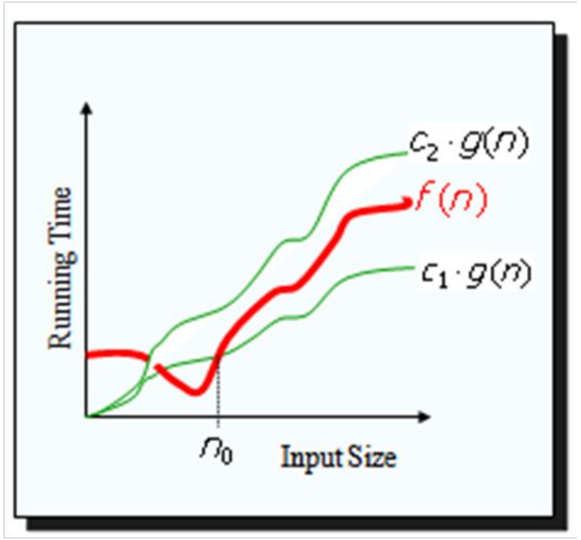
- Asymptotically tight bound used for average-case analysis
- Let f(n) and g(n) are functions over non-negative integers
- if there exists constants c_1, c_2, and n_0, such that

  $c_1\ g(n) \le f(n) \le c_2\ g(n)$ for $n \ge n_0$
- $f(n)=\Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- Example $f(n) = 18n+9$, $c_1\ g(n) = 18n$, $c_2\ g(n) = 27n$

  $f(n) > 18n$ and $f(n) \le 27n$

so we can say $f(n) = O(n)$ and $f(n) = \Omega(n)$

i.e $f(n) = \Theta(n)$



Recursion

Recursion is **the process of repeating items in a self-similar way**. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
 void recursion() {
recursion(); /* function calls itself */
}

int main()
{
recursion();
 }
```

TYPES OF RECURSION

1. Direct recursion
2. Indirect recursion
3. Tail recursion
4. Non-tail recursion

1

Direct recursion

A function is called direct recursive if it calls the same function again.

Structure of Direct recursion:

```
fun() {
    //some code

    fun();

    //some code
}
```

2 Indirect recursion

A function (let say fun) is called indirect recursive if it calls another function (let say fun2) and then fun2 calls fun directly or indirectly.

Structure of Indirect recursion:

```
fun() {
    //some code

    fun2();

    //some code
}
```

```
fun2() {
    //some code

    fun();

    //some code
}
```

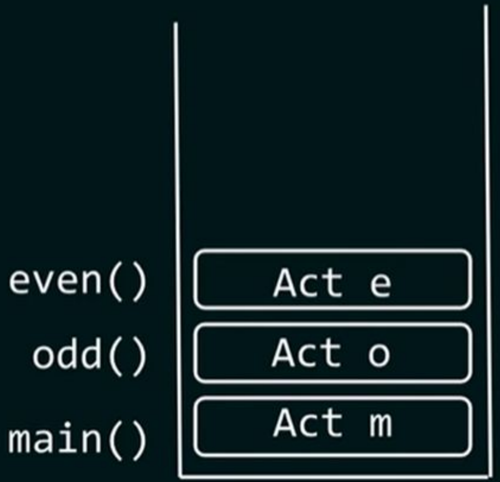
Program to understand indirect recursion

```
void odd();
void even();
int n=1;

void odd() {
    if(n <= 10) {
        printf("%d ", n+1);
        n++;
        even();
    }
    return;
}

void even() {
    if(n <= 10) {
        printf("%d ", n-1);
        n++;
        odd();
    }
    return;
}

int main() {
    odd();
}
```



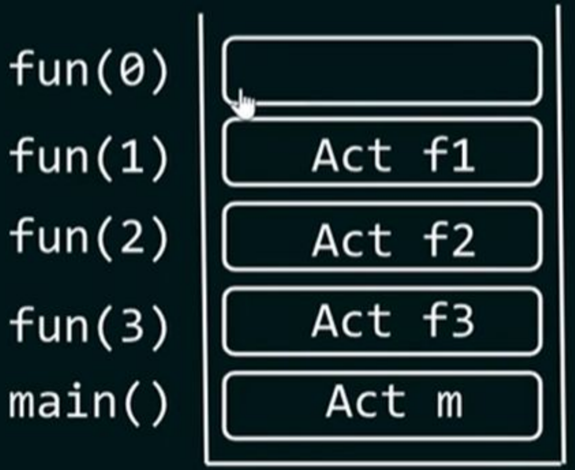
Act - Activation record
o - odd(), e - even()

Output: 2 1 4 3 6 5 8 7 10 9

DEFINITION

A recursive function is said to be tail recursive if the recursive call is the last thing done by the function. There is no need to keep record of the previous state.

```
void fun(int n) {
    if(n == 0)
        return;
    else
        printf("%d ", n);
    return fun(n-1);
}
int main() {
    fun(3);
    return 0;
}
```

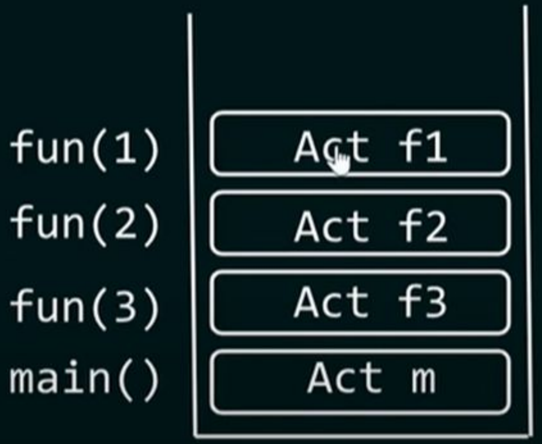


Output: 3 2 1

DEFINITION

A recursive function is said to be **non-tail recursive** if the recursive call is not the last thing done by the function. After returning back, there is some something left to evaluate.

```
void fun(int n) {
    if(n == 0)
        return;
    fun(n-1);
    printf("%d ", n);
}
int main() {
    fun(3);
    return 0;
}
```



OUTPUT : 1 2 3

Output: 1

Linear Recursion

A function is called the linear recursive if the function makes a single call to itself at each time the function runs and grows linearly in proportion to the size of the problem.

The factorial function is a good example of linear recursion

Recursive algorithms for factorial function

```
#include<stdio.h>
int fact(int n)
{
  if(n>0)
     return(n*fact(n-1));
  else
     return(1);
}

/* 5=5*4*3*2*1=120
    fact(5)
      5*fact(4)
        4*fact(3)
          3*fact(2)
            2*fact(1)
              1* fact(0)
                1
                 */
```

```
int main()

{
 int i,fact=1,number;
 printf("Enter a number: ");
  scanf("%d",&number);
    for(i=1;i<=number;i++){
      fact=fact*i;
  }
  printf("factorial of %d is %d",number, fact);
}
```

Binary recursion

In binary recursion, the function calls itself twice in each run. As a result, the calculation depends on two results from two different recursive calls to itself

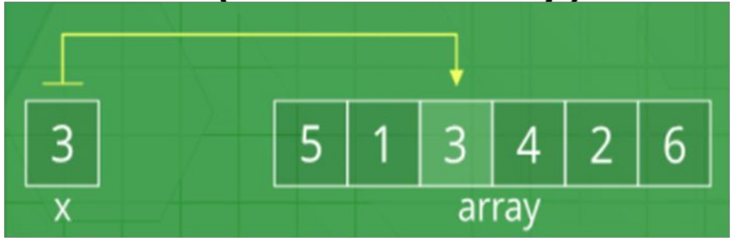
Fibonacci sequence generation is an example for binary recursive function.

Fibonacci sequence

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int);
int main(){
        int n, i;
        printf("Enter the number of element you want in series :\n");
        scanf("%d",&n);
        printf("fibonacci series is : \n");
        for(i=0;i<n;i++) {
                printf("%d ",fibonacci(i));
        }
        getch();
}
 int fibonacci(int i){
        if(i==0) return 0;
        else if(i==1) return 1;
        else return (fibonacci(i-1)+fibonacci(i-2));
}
```

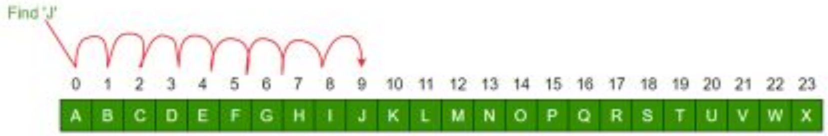
Array -Search

- Searching an element in an array at a particular location.
- 3 types
- Linear search (unsorted array)
- Binary search (sorted array)
- Fibonacci Search(sorted array)



Linear Search

- Performed on unsorted array



Linear Search

	i = 0	i = 1	i = 2	i = 3	i = 4
A	32	15	48	27	74
	0	1	2	3	4

Size = 5
Search for 27

For i = 0 to 4 *(size -1)*

i = 0 Check if A[0] is 27

i = 1 Check if A[1] is 27

i = 2 Check if A[2] is 27

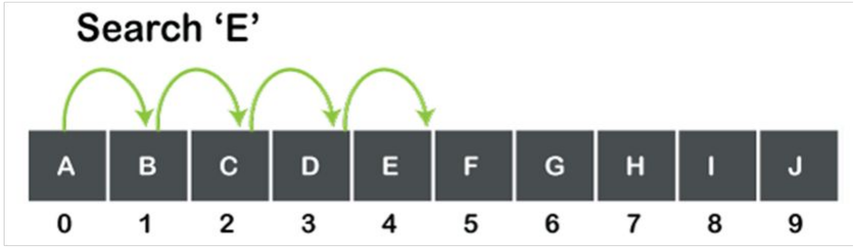
i = 3 Check if A[3] is 27 // Element found at index 3

i = 4 Check if A[4] is 27

Linear Search

```
#include<stdio.h>
void linear_search(int  *a, int n, int item){
  int i, count = 0;
  for(i=0;i<n;i++)
  {
                    if(a[i] == item)
                    {
                      printf("The  item found at index %d.\n",i);
                      count = count+1;
     }
   }
   if(count == 0)
     printf("Element  not found");
}

int main()
{
  int X[10]={2,4,6,8,10};
  int  n=5, k, item;
  printf("\n  Enter the element to be searched  ");
  scanf("%d",&item);
  linear_search(X,n,item);
  return 0;
}
```



for i=0 to 9 (size -1)

i =0, check if A[0] = 'E'

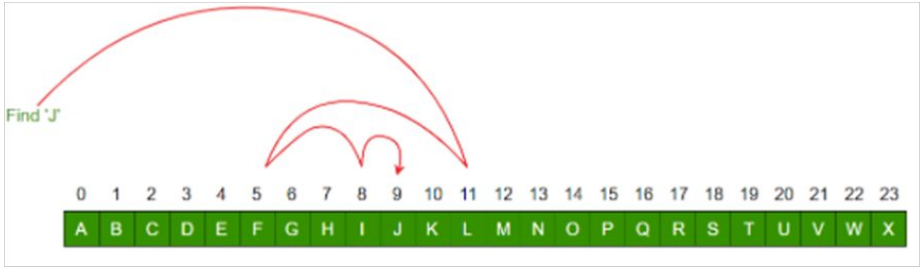
i =1, check if A[1] = 'E'

i =4, check if A[4] = 'E'

Element found at index 4

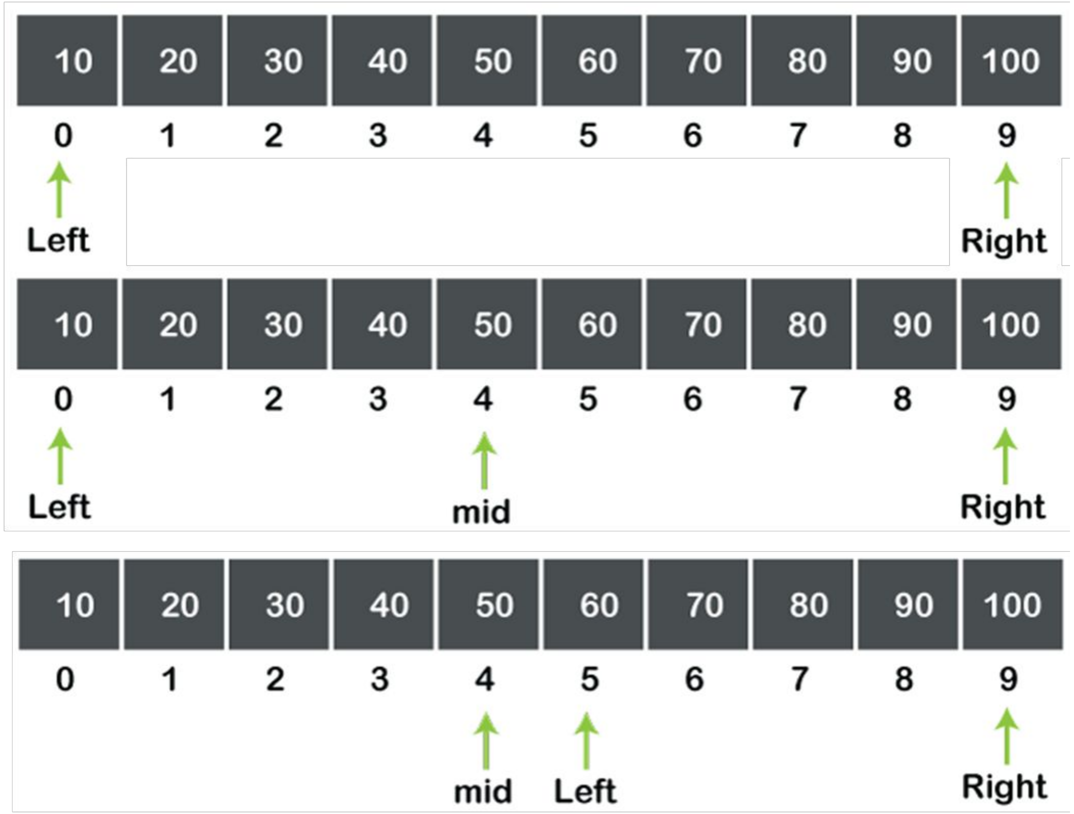
Binary Search

- Performed on sorted array
- Divide and conquer technique



Binary Search

Search item 70

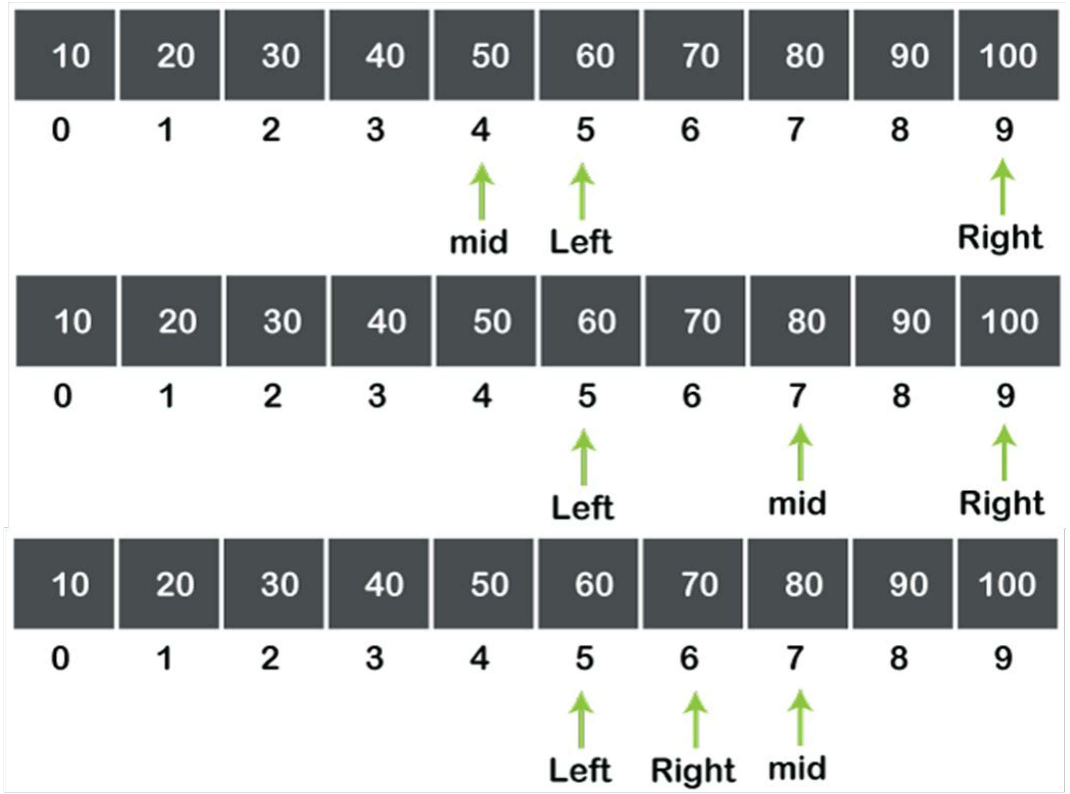


Step 1 : Calculate middle element
mid = (left + right)/2 = 4

Step 2 : Check if a[4] = 70
if not
Check if item > a[mid]
if yes,
Left = mid + 1

Binary Search

Search item 70



Step 1 : Calculate middle element
mid = (left + right)/2 = 7

Step 2 : Check if a[7] = 70
if not
Check if item > a[mid]
if yes,
Left = mid + 1
else
Right = mid - 1

Binary Search

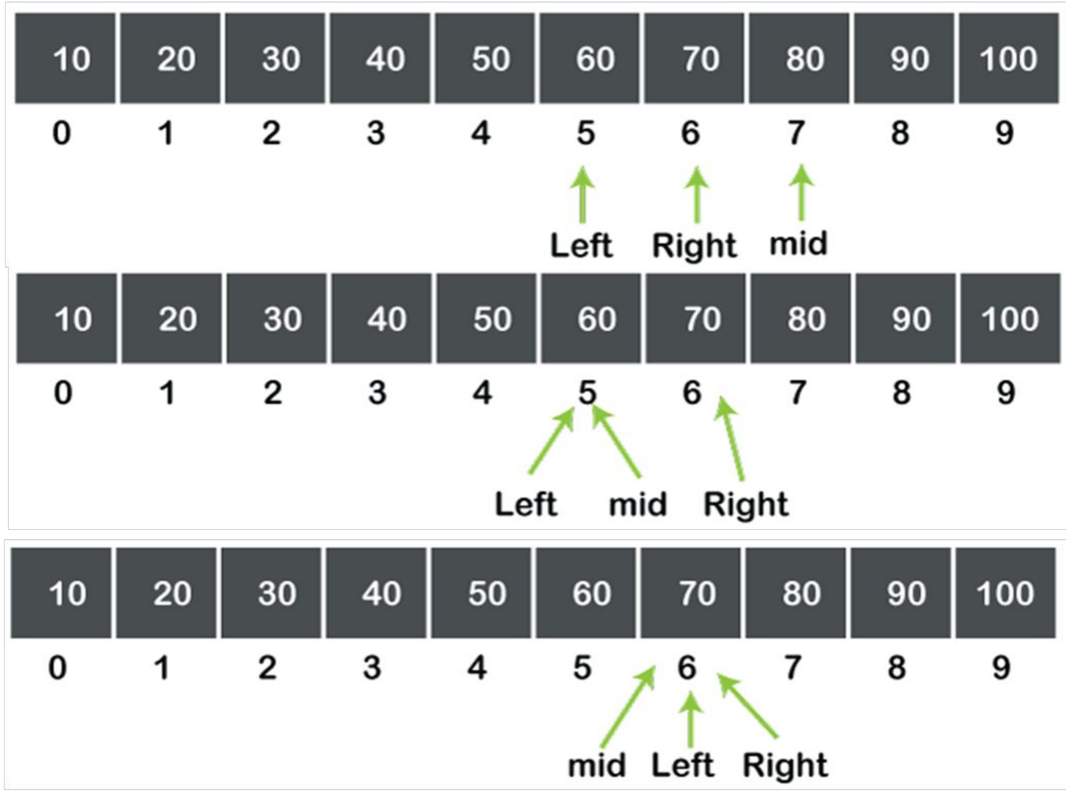
Search item 70

Step 1 : Calculate middle element
mid = (left + right)/2 = 5

Step 2 : Check if a[5] = 70
if not
Check if item > a[mid]
if yes,
Left = mid + 1
else
Right = mid - 1

Step 1 : Calculate middle element
mid = (left + right)/2 = 6

Step 2 : Check if a[6] = 70



Search item 70

Step 1 : Calculate middle element

mid = (left + right)/2 = 5

Step 2 : Check if a[5] = 70

if not

Check if item > a[mid]

if yes,

Left = mid + 1

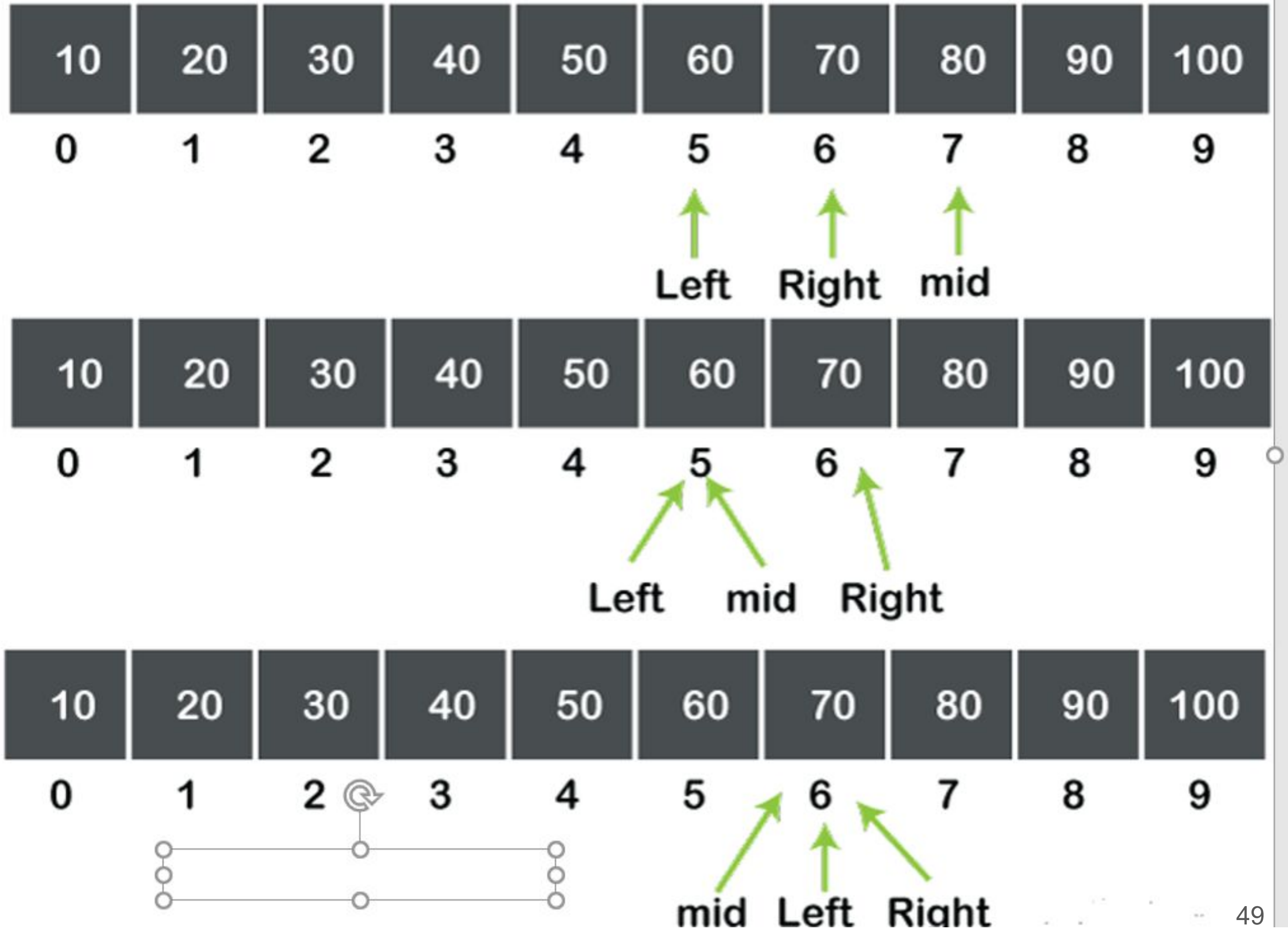
else

Right = mid - 1

Step 1 : Calculate middle element

mid = (left + right)/2 = 6

Step 2 : Check if a[6] = 70



Binary Search Algorithm

Binary_search(A, n, item)

1. Assign counters left = 0, right = n – 1, mid = (left + right)/2;
2. Repeat steps 2.1 to while (left <= right)

    2.1 if(a[mid] < item), then left = mid + 1;

    else if (a[mid] == item), then item found at mid

    else right = mid - 1;

    2.2 mid = (left + right)/2;

3. if(left > right), then Item is not in the array

Fibonacci Search

Steps:

1. Find F(k){k^{th} Fibonacci number}, Which is greater than or equal to n
2. If F(K) = 0
   stop and print message as element not found
3. Offset = -1

4. i = min(offset + F(k-2) , n-1)

5. If S == A[i]
   return i and **stop the search**
   If S > A[i]
   k = k -1, offset = i and repeat steps 4,5
   If S < A[i]
   k = k-2 repeat steps 4,5

THANK YOU