21CSC201J – Data Structures & Algorithms

Unit-1

* What is DSA?

The term DSA stands for Data structures and Algorithms. As you can see, it's a combination of two separate yet interrelated topics – Data structure and Algorithms.

* What is Data Structure?

A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexity. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

* What is Algorithm?

Algorithm is defined as a process or set of well defined instructions that are typically used to solve a particular problem. In simple terms, it is

set of operations performed in a step by step manner to execute a task.

* Complexities: The primary motive of using DSA is to solve a problem effectively and efficiently. The efficiency of your program is measured in terms of complexities. Complexity is of two types:

  * Time Complexity: Time complexity is used to measure the amount of time required to execute a code.

  * Space Complexity: Space complexity means the amount of space required to execute the code successfully.

* Types of Data Structures:

  There are two types of data structures:

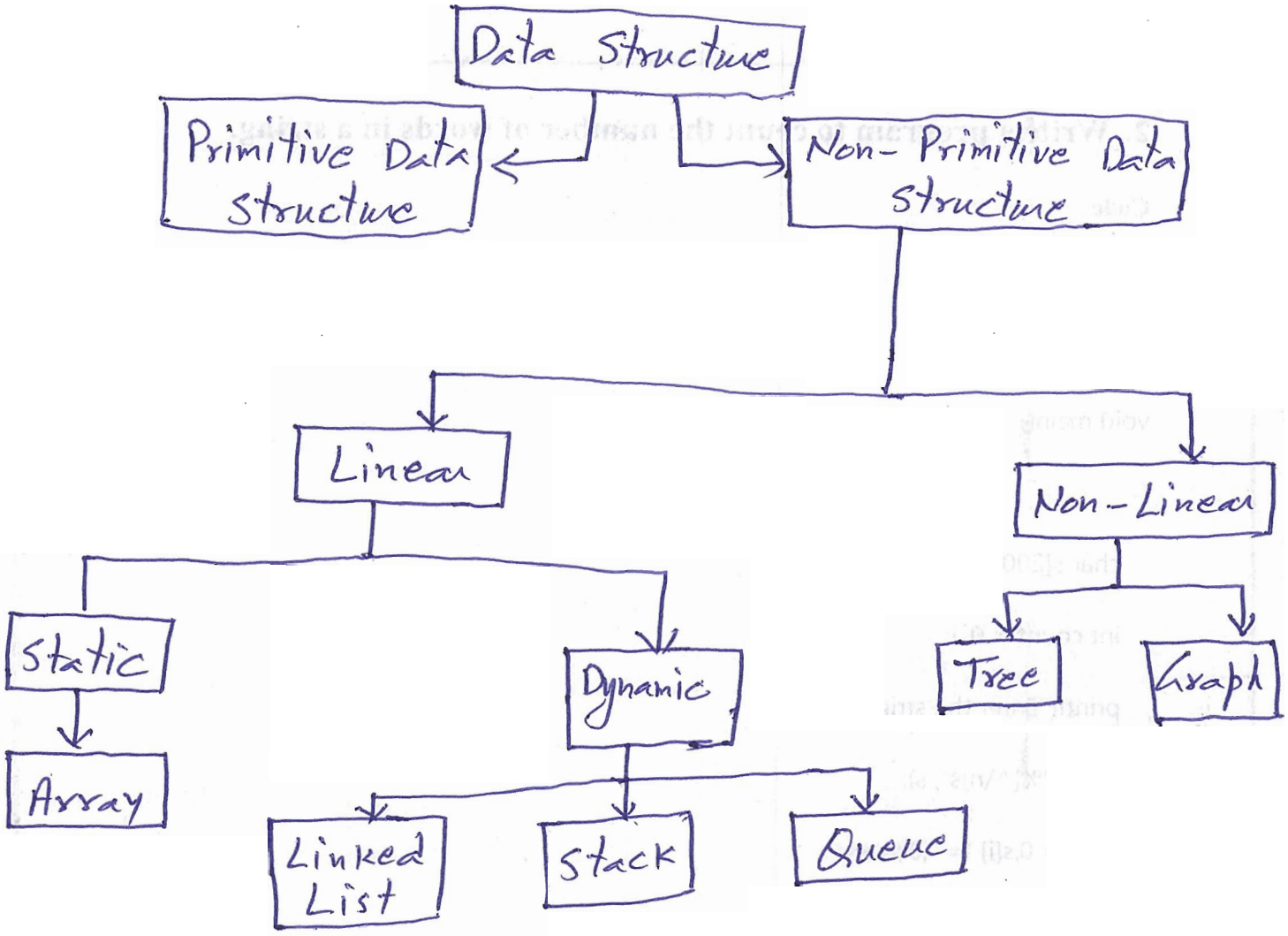
  * Primitive data structures
  * Non-Primitive data structures

* Primitive Data structures: The primitive data structures are the sets of basic data types from which all other data types are constructed. e.g: int, char, double, string etc.

* Non- Primitive Data structures:

The non-primitive data structure is further divided in two types:

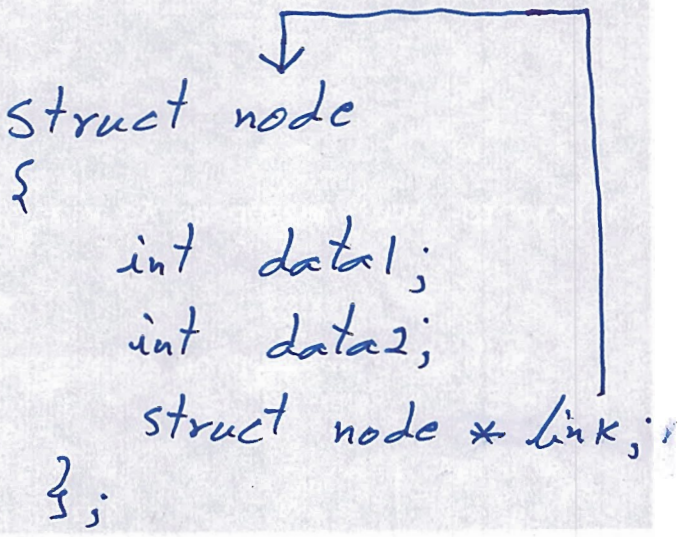
* Linear Data structure
* Non- Linear Data structure.


Data Structure
Primitive Data structure
Non-Primitive Data structure
Linear
Non-Linear
Static
Dynamic
Tree
Graph
Array
Linked List
Stack
Queue

* Self Referential structures:

Self referential structures are those structures that have one or more pointers which points to the same type of structure, as their member.

e.g:



```
struct node
{
    int data1;
    int data2;
    struct node * link;
};
```

In other words, structures pointing to the same type of structures are self-referential in nature.

* Asymptotic Notation:

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

① Big-O Notation (O-notation) → Worst case
② Omega Notation (Ω-notation) → Best case
③ Theta Notation (Θ-notation) → Average case

In other words we can say that Asymptotic notations of an algorithm is a mathematical representation of its complexity.

① Big-O Notation: Big O notation is used to define the upper bound of an algorithm in terms of Time Complexity.

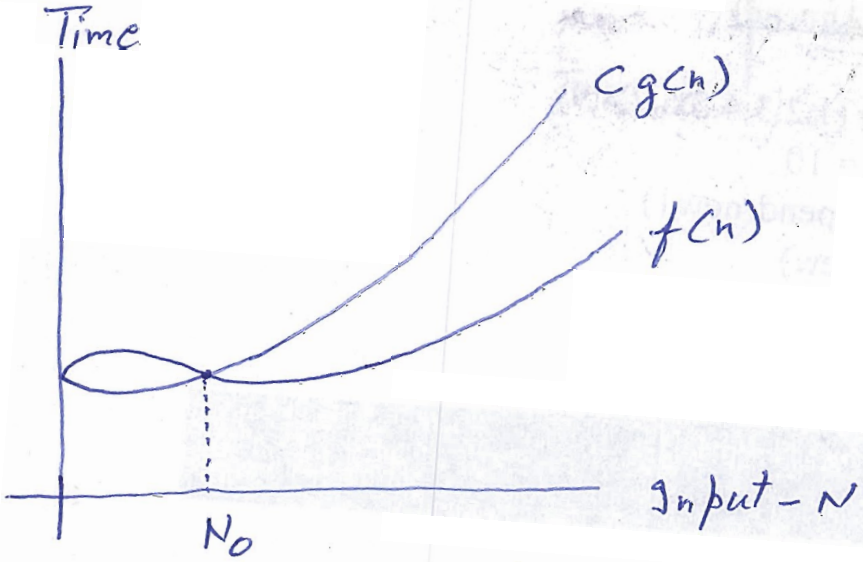
That means Big-O notation always indicates the maximum time required by an algorithm for all input values. That means Big-O notation describes the Worst case of an algorithm time complexity.

Big-O Notation can be defined as follows:

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) <= C g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $O(g(n))$.

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

Consider the following graph drawn for the values of $f(n)$ and $Cg(n)$ for input (n) values on X-Axis and time required is on Y-Axis



In the above graph after a particular input value n_0, alway $Cg(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

Example:

$f(n) = 3n + 2$

$g(n) = n$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) <= Cg(n)$ for all values of $C > 0$ and $n_0 >= 1$

$$f(n) <= Cg(n)$$
$$\Rightarrow 3n+2 <= Cn$$

Above condition is always TRUE for all values of $C = 4$ and $n >= 2$.

By us using Big O notation we can represent the time complexity as follows:

$$3n+2 = O(n)$$

(2) Omega Notation (Ω):

Omega notation is used to define the lower bound of an algorithm in terms of complexity.

That means Omega notation always indicates the minimum time required by an algorithm for all input values. That means Omega notation describes the best case of an algorithm time complexity.

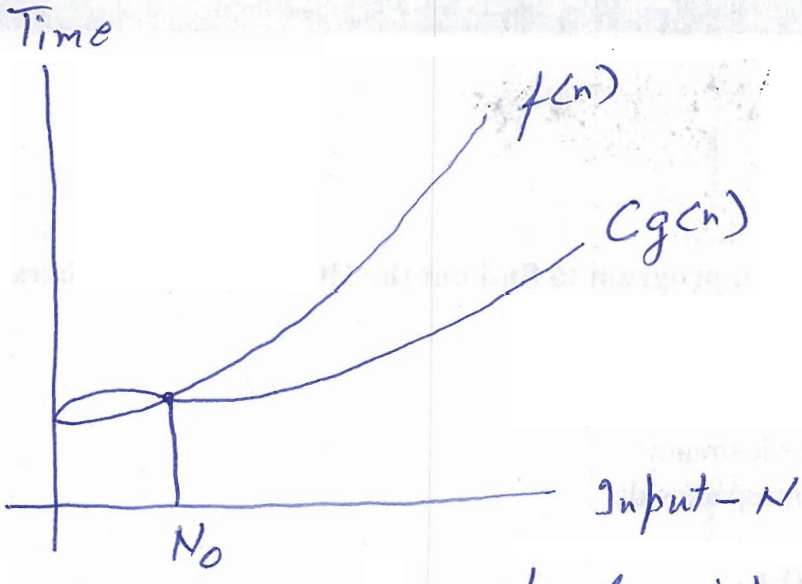
Omega Notation can be defined as follows:

Consida function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term.

If $f(n) >= Cg(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $\Omega(g(n))$.

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

Consider the following graph drawn for the values of $f(n)$ and $Cg(n)$ for input (n) value on x-Axis and time required is on y-axis.



In the above graph after a particular input value n_0, alway $Cg(n)$ is less than $f(n)$ which indicates the algorithm's lower bound.

Example:

consider the following $f(n)$ and $g(n)$.

$$f(n) = 3n + 2$$
$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) >= Cg(n)$ for all values of $C > 0$ and $n_0 >= 1$.

$f(n) >= C g(n)$

$\Rightarrow 3n+2 >= Cn$

Above condition is always true for all values of $C = 1$ and $n >= 1$.

By using Ω (Omega) notation we can represent the time complexity as follows:

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

(3) Big Theta Notation (Θ):

Big Theta notation is used to define the average bound of an algorithm in terms of Time complexity.

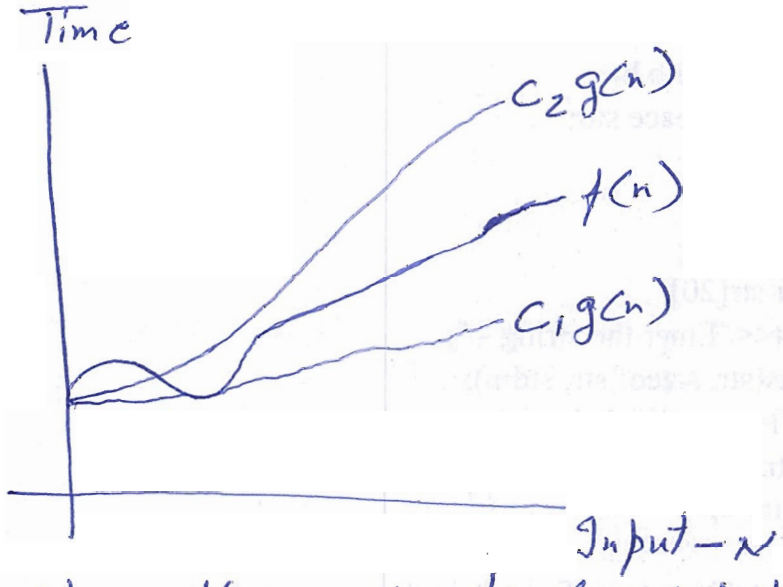
That means Big Theta notation always indicates the average time required by an algorithm for all input values. That means Big-Theta notation describes the average case of an algorithm time complexity.

Big-Theta Notation can be defined as follows:

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $C_1 g(n) <= f(n) <= C_2 g(n)$ for all $n >= n_0, C_1 > 0, C_2 > 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $\Theta(g(n))$.

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

Consider the following graph drawn for the values of $f(n)$ and $Cg(n)$ for input (n) value on X-Axis and time required is on Y-Axis.



In the above graph after a particular input value n_0, always $c_1 g(n)$ is less than $f(n)$ and $c_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

Example: Consider the following $f(n)$ and $g(n)$

$$f(n) = 3n+2$$
$$g(n) = n$$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $c_1 g(n) <= f(n) <= c_2 g(n)$ for all values of $c_1 > 0$, $c_2 > 0$ and $n_0 >= 1$.

$$c_1 g(n) <= f(n) <= c_2 g(n)$$
$$\Rightarrow c_1 n <= 3n+2 <= c_2 n$$

Above condition is always TRUE for all values of $c_1 = 1$, $c_2 = 4$ and $n >= 2$.

By using Big Theta notation we can represent the time complexity as follows:

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

Array Pgm:

```
#include<stdio.h>
int main()
{
 // You must mention the size of the array, if you want
 // more than one element initialized to 0.
 // Here, all 5 elements are set to 0.
 int arr[5] = {0};
 for(int i=0; i<5; i++)
 {
    printf("%d", arr[i]);
 } printf("\n");
   return 0;
}
```

output:

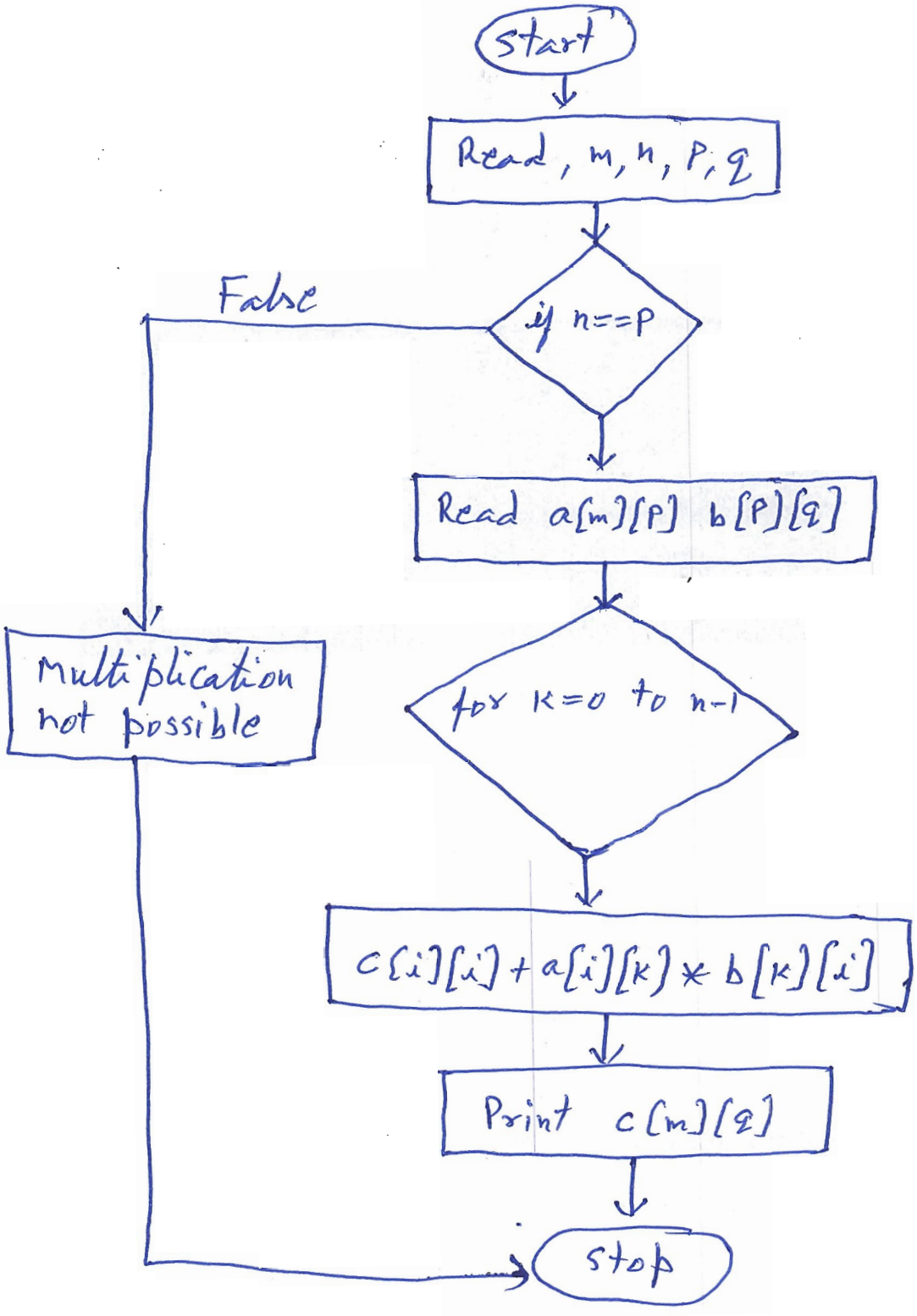
```
0
0
0
0
0
```

* Matrix Multiplication:

Algorithm:

1. start
2. Enter the value of m & n (or) order of first matrix
3. Enter the value of p & q (or) order of second matrix
4. Create a matrix of size a[m][n] and b[p][q].
5. Enter the element of matrices row wise using loops.
6. If the number of columns of the first matrix is not equal to the number of rows of the second matrix, print matrix multiplication is not possible and exit. If not, proceed to the next step.
7. Create a third matrix, c of size m x q to store the product
8. set a loop from i=0 to i=m.
9. set an inner loop for the above loop from j=0 to j=q.
10. Initialise the value of the elements (i, j) of the new matrix to 0.
11. set an inner loop inside the above loop from k=0 to k=p.
12. Using the add & assign operator (+=) store the value of a[i][k] * b[k][j] in the third matrix, c[i][j].
13. Print the third matrix.
14. stop.

* Flow chart of Matrix Multiplication:



* Program for matrix multiplication

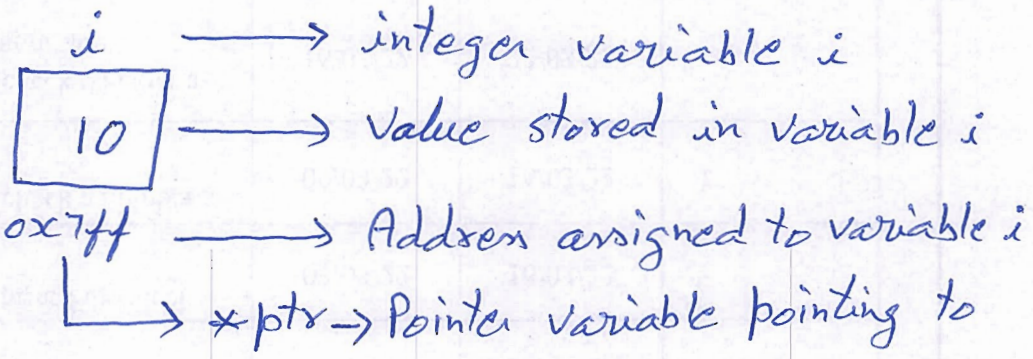
```
#include <stdio.h>
int main()
{
    int a[10][10], b[10][10], c[10][10], n, i, j, k;
    printf("Enter the value of n (n<=10):");
    scanf("%d", &n);
    printf("Enter the value of Matrix-A: \n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    printf("Enter the elements of Matrix-B: \n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            scanf("%d", &b[i][j]);
        }
    }
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            c[i][j]=0;
            for(k=0; k<n; k++
            {
                c[i][j] += a[i][k] * b[k][j];
}   }   }
```

```
printf("The product of the two matrices is:\n");
for(i=0; i<n; i++)
{
   for(j=0; j<n; j++)
   {
      printf("%d\t", c[i][j]);
   }
   printf("\n");
}
```

output:

```
Enter the value of n (n<=10): 2
Enter the elements of Matrix-A:
1 2
1 2
Enter the elements of Matrix-B:
1 2
1 1
Product of the two matrices is:
8 8
8 8
```

* Pointers : A pointer is a variable that holds the memory address of another variable. When we declare a variable of a specific data type we assign some of the memory for the variable where it can store its data. By default the memory allocation is in stack. Using different function like malloc in C or key-words like new in C++, Java etc. we can allocate memory for a variable in the heap section (also called as dynamic memory allocation. One of the major advantages of allocating memory in heap is that it allows you to access variables globally. Simply accessing the pointer that stores the address of the variable declared in heap we can directly manipulate the data of the variable in heap.



Syntax:

The syntax of pointers is similar to the variable declaration in C, but we use the (*) ~~def~~ dereferencing operator in the pointer declaration.

datatype * ptr;

Where,

ptr is the name of the pointer

datatype is the type of data it is pointing to.

We can also define pointers to functions, structures etc.

* How to use pointers:

The use of pointers can be divided into three steps.

1. Pointer Declaration
2. Pointer Initialization
3. ~~Def~~ Dereferencing

* Pointer Declaration: In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the (*) ~~def~~ dereference operator before its name.

e.g:

```
int * ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

* Pointer Initialization: Pointer initilization is the process where we assign some initial value to the pointer variable. We generally use the (&) addressof operator to get the memory address of a variable and then store it in the pointer variable.

Example:

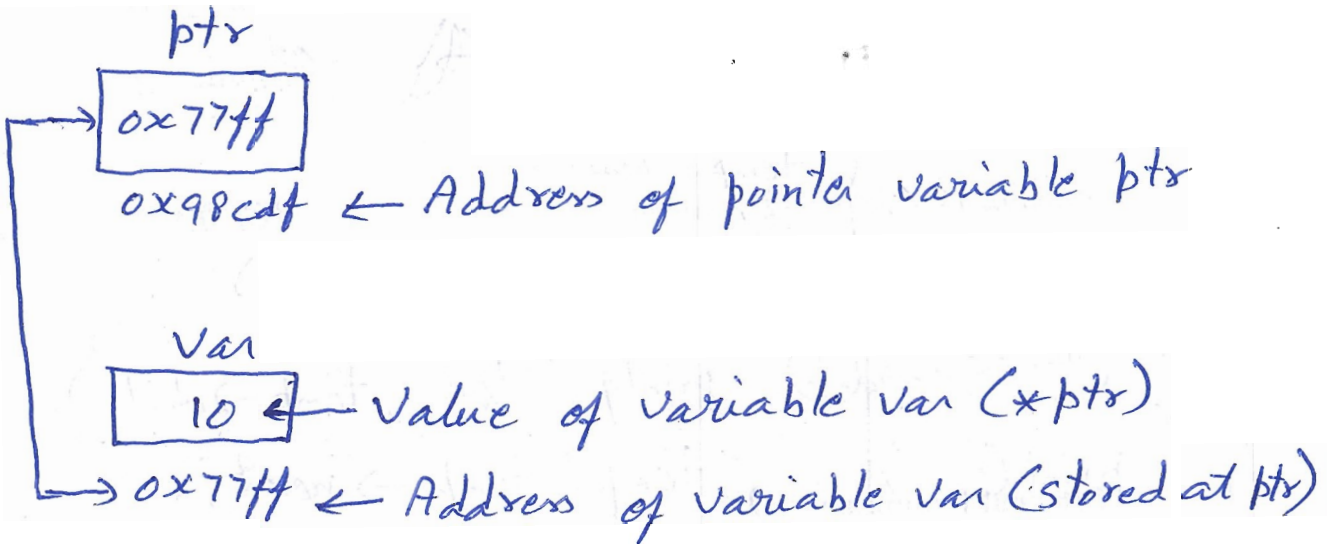
```
int var = 10;
int *ptr;
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called pointer definition as the pointer is declared and initialized at the same time.

e.g:

```
int *ptr = &var;
```

* Dereferencing: Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same (*) dereferencing operator that we used in the pointer declaration.



* Program of Pointer in C

```
#include <stdio.h>
int main()
{
    int var = 10;
    int *ptr;        // Declare pointer variable
    ptr = &var;      // Data type of ptr & var must be same
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
    return 0;
}
```

output:

```
Value at ptr = 0x7ff10
Value at var = 10
Value at *ptr = 10
```

* Pointers in C: A pointer is a variable pointing to the address of another variable. It is declared along with an asterisk symbol (*). The syntax to declare a pointer is as follows:

Syntax:

```
datatype *var;
```

The syntax to assign the address of a variable to a pointer is:

Syntax:

```
datatype var1, *var2;

var2 = &var1;
```

* In how many ways can you access a variable?

You can access a variable in two ways:

1. Direct Access: You can directly use the variable name to access the variable.
2. Indirect Access: You use a pointer to access that variable.

Example:

```
#include<stdio.h>
int main()
{
    int a=5, *ptr;
    ptr = &a;
    printf("Direct Access, a=%d\n", a);
    printf("Indirect Access, a=%d\n", *ptr);
    return 0;
}
```

output:

Direct Access, a=5

Indirect Access, a=5.

* What is Searching:

Searching is the process of fetching a specific element in a collection of elements. The collection can be an array or a linked list. If you find the element in the list, the process is considered successful, and it returns the location of that element.

And in contrast, if you do not find the element, it deems the search unsuccessful.

Two prominent search strategies are extensively used to find a specific item on a list. However algorithm chosen is determined by the list's organization.

1. Linear search
2. Binary search

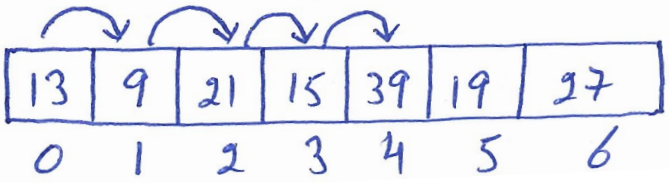
* What is Linear Search?

Linear search, often known as sequential search, is the most basic search technique. In this type of search, you go through the entire list and try to fetch a match for a single element. If you find a match, then the address of the matching target element is returned.

On the other hand, if the element is not found, then it returns a NULL value.

Following is a step-by-step approach employed to perform Linear Search Algorithm.

searched Element = 39



The procedures for implementing linear search are as follows:

step 1: First, read the search element (Target element) in the array.

step 2: In the second step compare the search element with the first element in the array.

step 3: If both are matched, display "Target element is found" and terminate the Linear search function.

step 4: If both are not matched, compare the search element with the next element in the array.

step 5: In this step, repeat steps 3 and 4 until search (Target) element is compared with the last element of the array.

step 6: If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

* Linear Search or Sequential Search Algorithm:

Linear Search (Array Arr, Value a)

step1: set i = 0
step2: if i > n then go to step 7
step3: if Arr[i] = a then go to step 6
step4: set i to i+1
step5: Goto step 2
step6: Print element a found at index i and goto step 8
step7: Print element not found
step8: Exit.

* Program for Linear search

```
#include <stdio.h>
int main()
{
    int arr[50], i, key, size;
    printf("Enter the size of The Array: ");
    scanf("%d", &size);
    printf("Enter the Element of the Array: ");
    for (i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Enter The Element to search: ");
    scanf("%d", &key);
```

```
for (i=0; i < size; i++)
{
    if (arr[i] == key)
    {
        printf("Element Found at %d Index", i);
        break;
    }
}
if (i == size)
{
    printf("Element Not Found...");
}
return 0;
}
```

* Complexity of Linear Search:

  Worst Case → O(n)

  Average Case → O(n)

  Best Case → O(1)

* Advantages of Linear Search:

  * Linear search can be used irrespective of whether the array is sorted or not.
  * Does not required any additional memory.
  * It is well-suited algorithm for small datasets.

* Drawbacks of Linear Search:
  * Linear search has a time complexity of O(n), which in turns makes it slow for large datasets.
  * Not suitable for large arrays.
* When to use Linear Search?
  * When we are dealing with a small dataset.
  * When we are searching for a dataset stored in contiguous memory.

* Binary search: Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach the element is always searched in the middle of a portion of an array.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

* Process of Binary Search:
  * Divide the search array into two halves by finding the middle index mid.

    mid = (low + high) / 2 // Take the floor value
  * Compare the middle element of the search array with the key.
  * If the key is found at middle element, the process is terminated.
  * If the process is not found at middle element, choose which half will be used as the next search array.
    * If the key is smaller than the middle element, then the left side is used for next search.
    * If the key is larger than the middle element, then the right side is used for next search.

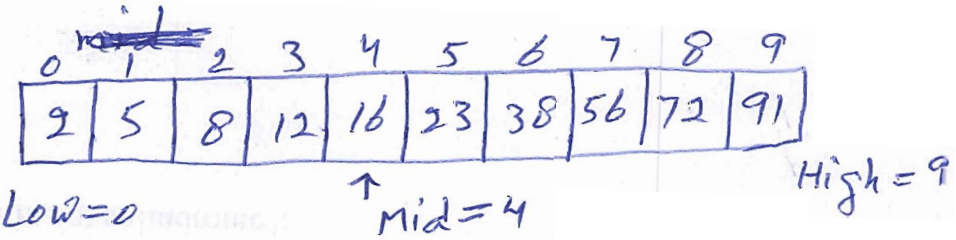
* This process is continued until the key is found or the total search array is exhausted.

Example: Let's the array be:

arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}

target = 23 = key.

step 1: Calculate the mid and compare the mid element with key. If the key is less than mid element, move to left and if it is greata than the mid then move search array to the right.



mid = (low + high) / 2

= $\frac{0+9}{2}$ = 4.5

= 4 // Take the floor value.

key > mid i.e 23 > 16

so the search array is move to right.

step 2:

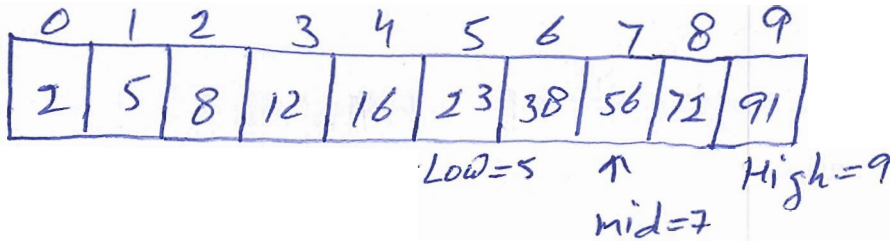
low = mid + 1 i.e 4+1 = 5

high = 9

mid = (low + high) / 2

= $\frac{5+9}{2} = \frac{14}{2} = 7$

element at index 7 is 56



Key < mid i.e 23 < 56

So the search array moves to the left.

Step 3:

~~low = mid~~

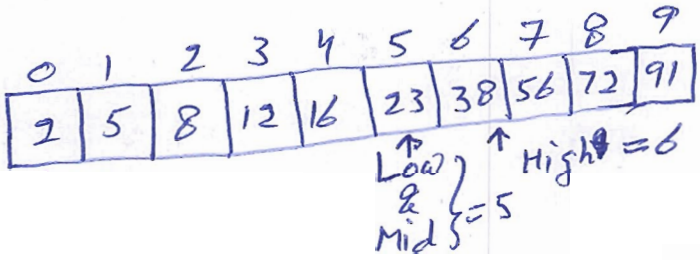
high = mid - 1 i.e 7 - 1 = 6

mid = (low + high) / 2

$= \frac{5+6}{2} = \frac{11}{2} = 5.5 = 5$

Element at index 5 is 23

~~If the key~~



If the key matches the value of the mid element, the element is found and stop search.

* So we have three case for partition of array & 4 for exit cond.

Case 1 : key == arr[mid] → When element found

Case 2 : key < arr[mid] → High = Mid - 1

Case 3 : key > arr[mid] → Low = Mid + 1

Case 4 : Low > High → When element not found.

* Advantages of Binary search Algorithm:

* Since it follows the technique to eliminate half of the array elements, it is more efficient as compared to linear search. for large data.
* Better time complexity and thus takes less compilation time.
* An improvement over linear search as it breaks the array down in half rather than sequentially traversing through the array elements.

* Limitations of Binary search Algorithm:

* Binary search Algorithm could only be implemented over a sorted array.
* Small unsorted arrays would take considerate time in sorting and then searching the desired element. So binary search is not prefered in such cases.

* Algorithm of Binary Search:

Binary_search(arr, lower_bound, upper_bound, key)

step 1: set beg = lower_bound, end = upper_bound, pos = -1
step 2: repeat step 3 and 4 while beg <= end
step 3: set mid = (beg + end) / 2
step 4: if arr[mid] == key
        set pos = mid
        print pos
        goto step 6
    else if arr[mid] > key
        set end = mid - 1
    else
        set beg = mid + 1
    [end of if]
[end of loop]
step 5: if pos = -1
        print "Value not found"
    [end of if]
step 6: exit.

* Time Complexity of Binary Search:

* Best Case → O(1)
* Average Case → O(log n)
* Worst case → O(log n)

* Program for Binary search:

```
#include<stdio.h>
int main()
{
    int arr[50], i, size, l=0, r, mid, key;
    printf("Enter the size of the array:");
    scanf("%d", &size);
    r= size-1;
    printf("Enter the element of the array:");
    for(i=0; i < size; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Enter the element to search:");
    scanf("%d", &key);
    while(l<=r)
    {
        mid= (l+r)/2;
        if (key == arr[mid]
        {
            printf("Element found at index %d", mid);
            break;
        }
        else if (key < arr[mid]
        {
            r= mid-1;
        }
        else
        {
            l= mid+1;
        }
```

```
    }
    if (l > r)
    {
      printf("Element Not found...");
    }
    return 0;
}
```

* Binary Search Example:

	0	1	2	3	4	5	6	7	8	9
Arr =	5	9	17	23	25	45	59	63	71	89

n = 10

Key = 59

l → 0, mid → 4, r → 9

l	r	mid = $\text{floor}\left(\frac{l+r}{2}\right)$
0	9	4
5	9	7
5	6	5
6	6	6 ← stop

For every step three cases are there:

Case I: key == Arr[mid] → Stoping condition

Case II: key < Arr[mid]

Case III: key > Arr[mid]

Here Case III is true as key i.e 59 is greater than 25 i.e the mid value. We can say this as the array is sorted.

Now the search Array is:

	0	1	2	3	4	5	6	7	8	9
	5	9	17	23	25	45	59	63	71	89

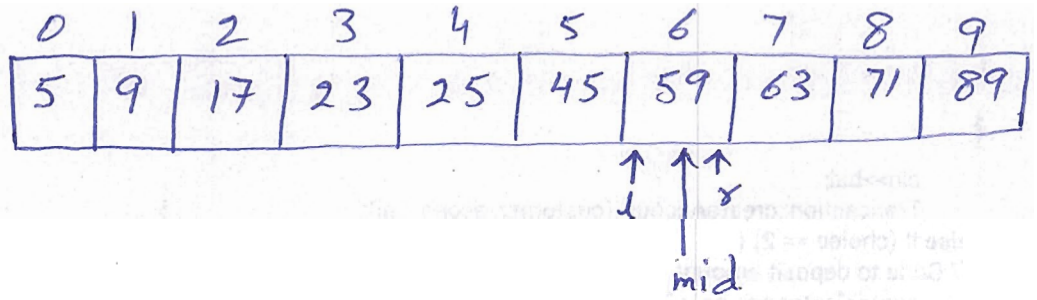
l → 5, mid → 7, r → 9

if key > mid then l = mid+1 for next step & r remain the same

if key < mid then r = mid−1 next step & l remain the same

	0	1	2	3	4	5	6	7	8	9
	5	9	17	23	25	45	59	63	71	89

l → 5, mid → 6, r → 7

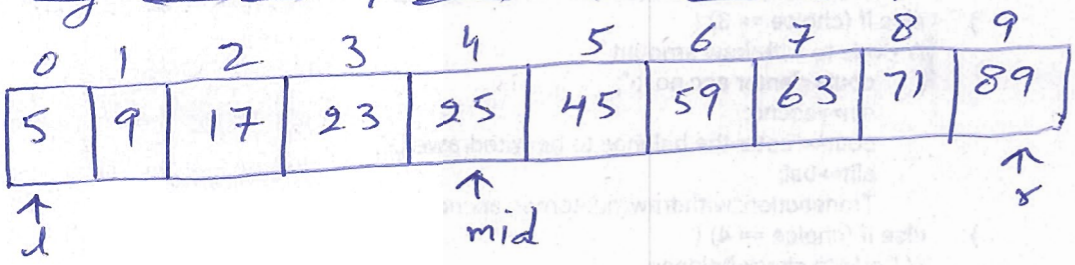


* When Key is not present in the array.

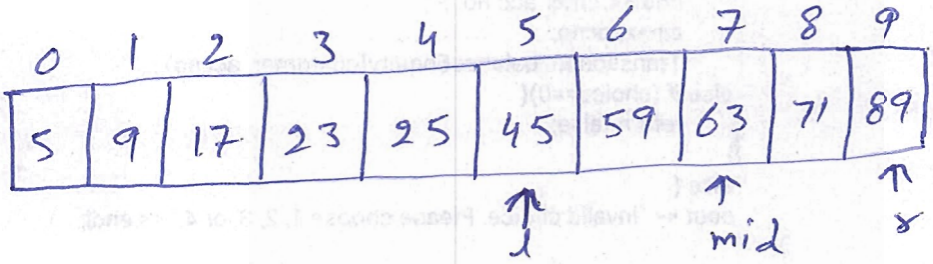
Arr =

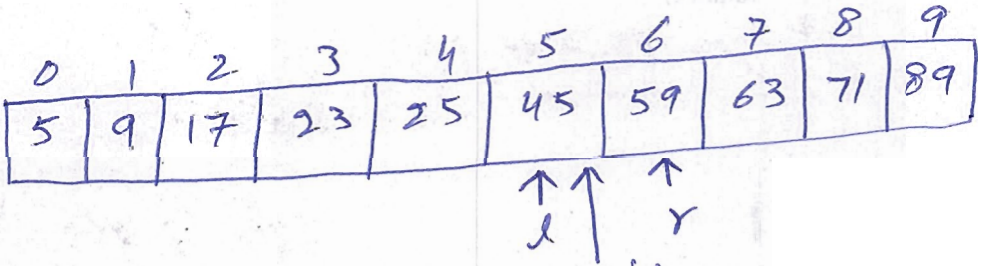
n=10

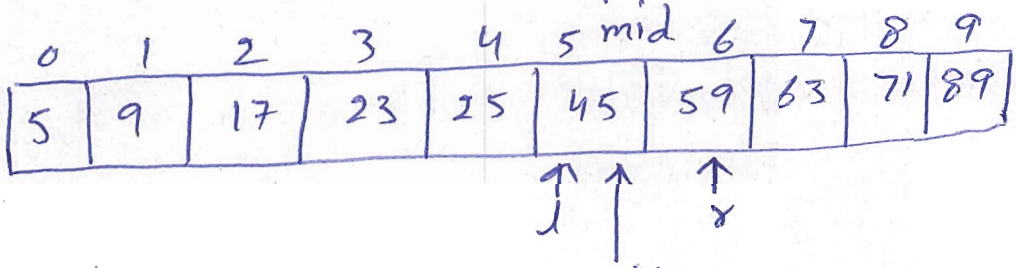
Key=60

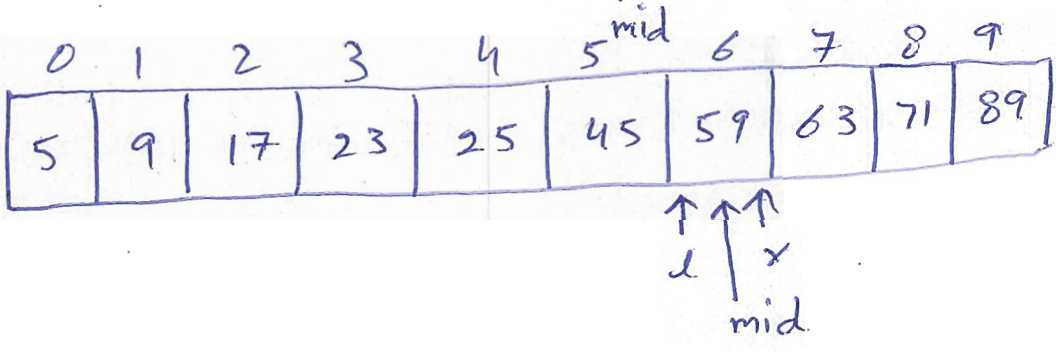


l	r	mid
0	9	4
5	9	7
5	6	5
6	6	6
→ 7	6	









Stop if $l > r$ i.e Key is not present.

* Insertion Sort: Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

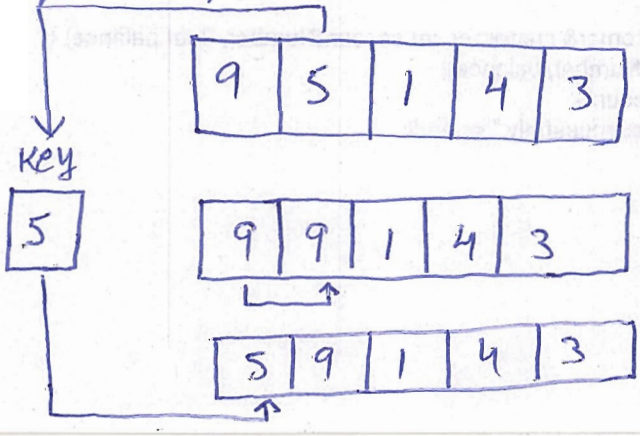
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

* Working of Insertion Sort:

Suppose we need to sort the following array.

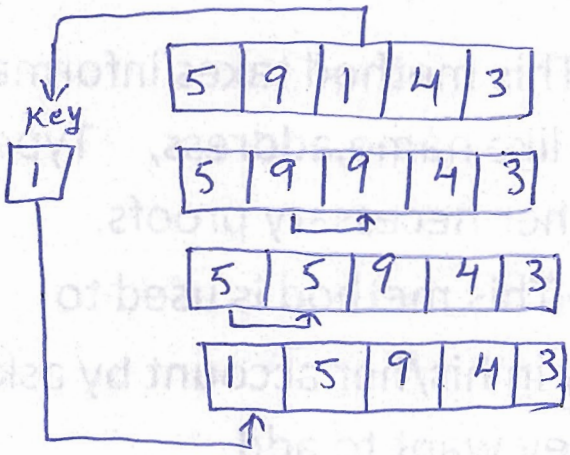
Step 1: The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



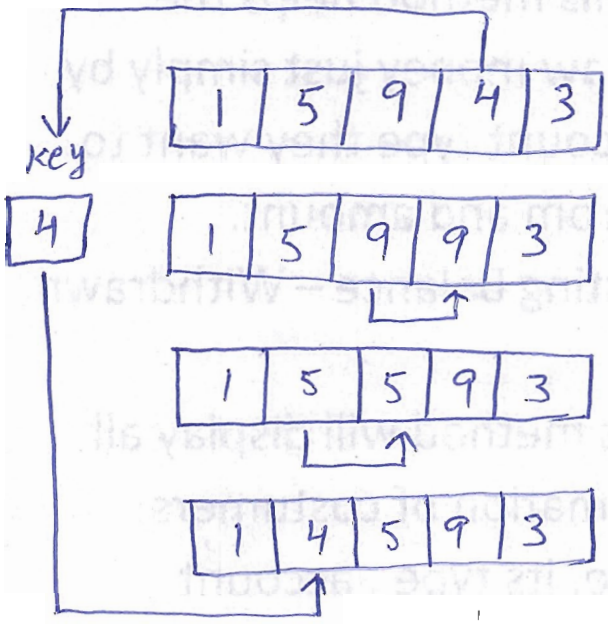
If the first element is greater than key, then key is placed in front of the first element.

Step 2: Now, the first two elements are sorted.
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



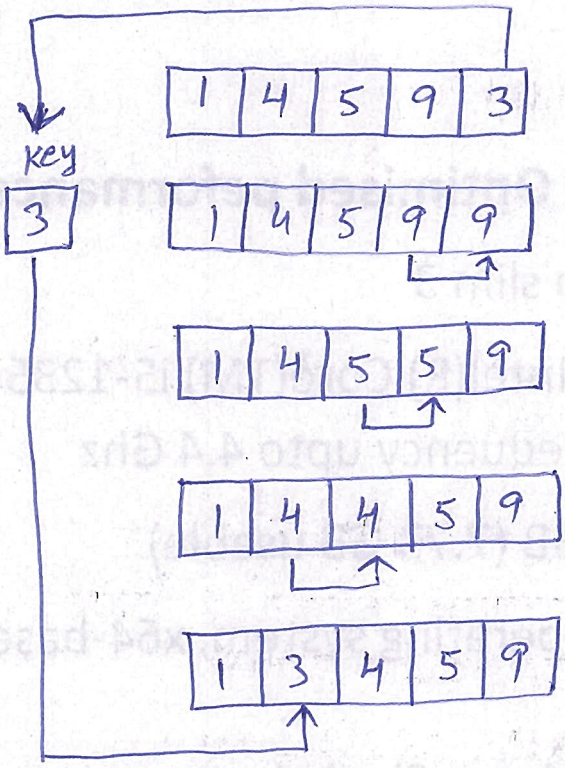
Place 1 at the beginning

Step 3: Similarly, place every unsorted element at its correct position.



Place 4 behind 1

step 4:



Place 3 behind 1 and the array is sorted.

* Insertion Sort Algorithm:

step 1: If it is the first element, it is already sorted.
step 2: Pick a next element.
step 3: Compare with all elements in the sorted sub-list.
step 4: Shift all the elements in the sorted sub-list that is greater than the value to be sorted.
step 5: Insert the value
step 6: Repeat until list is sorted.

OR

```
insertionSort(array)
    mark first element as sorted
    for each unsorted element x
        extract the element x
        for j <-
```

* Insertion Sort Algorithm (Another Approach)

InsertionSort(arr, size)
consider 0^{th} element as sorted
for each element from i=2 to n-1
temp = arr[i]
for j = i-1 to 0
if a[j] > temp then right shift it by one position
put temp at current j.

* Insertion Sort Complexity:

Best → $O(n)$
Worst → $O(n^2)$
Average → $O(n^2)$

* Characteristics of Insertion Sort:

* This algorithm is one of the simplest algorithms with a simple implementation.
* Basically, Insertion sort is efficient for small data sets.
* Insertion sort is adaptive in nature, i.e it is appropriate for data sets that are already partially sorted.

* Program for Insertion sort.

```
#include<stdio.h>
int main()
{
    int arr[50], i, j, size, temp;
    printf("Enter The size of the Array: ");
    scanf("%d", &size);
    printf("\nEnter the elements of the array: ");
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }
    for(i=1; i<size; i++)
    {
        temp= arr[i];
        j= i-1;
        while(j>=0 && arr[j] > temp)
        {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1]= temp;
    }
    printf("The sorted array is: \n");
```

```
for(i=0; i < size; i++)
{
    printf("%d", arr[i]);
}
 return 0;
}
```