G H RAISONI INSTITUTE OF ENGINEERING & TECHNOLOGY
(Approved by AICTE, New Delhi and Recognized by DTE, Maharashtra)
An Autonomous Institute Affiliated to Rashtrasant Tukadoji Maharaj Nagpur University, Nagpur
Accredited by NAAC with A+ Grade
Session 2023-2024

Subject : DSA **UNIT 1**

NOTES : 1.2

DSA

Insertion sort.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Code explained in Class .

Write Algorithm from code .

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where **n** is the number of items.

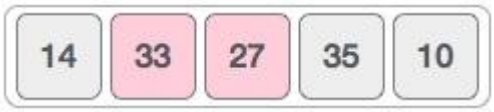
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

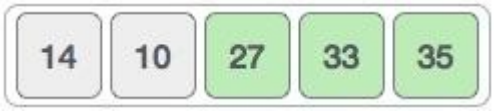
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

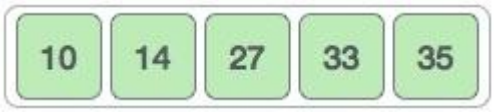
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

```
// below we have a simple C program for bubble sort
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
            // swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }

   // print the sorted array
   printf("Sorted Array: ");
   for(i = 0; i < n; i++)
   {
      printf("%d  ", arr[i]);
   }
}
int main()
{
   int arr[100], i, n, step, temp;
   // ask user for number of elements to be sorted
   printf("Enter the number of elements to be sorted: ");
   scanf("%d", &n);
   // input elements if the array
   for(i = 0; i < n; i++)
   {
      printf("Enter element no. %d: ", i+1);
      scanf("%d", &arr[i]);
   }
```

```
    // call the function bubbleSort
    bubbleSort(arr, n);
        return 0;
}
```

Convert code into Algorithm

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process

Now, let us learn some programming aspects of selection sort.

```
1.    SELECTION SORT(arr, n)
2.    Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
3.    Step 2: CALL SMALLEST(arr, i, n, pos)
4.    Step 3: SWAP arr[i] with arr[pos]
5.    [END OF LOOP]
6.    Step 4: EXIT
7.    SMALLEST (arr, i, n, pos)
8.    Step 1: [INITIALIZE] SET SMALL = arr[i]
9.    Step 2: [INITIALIZE] SET pos = i
10.   Step 3: Repeat for j = i+1 to n
11.   if (SMALL > arr[j])
```

```
12.         SET SMALL = arr[j]
13.      SET pos = j
14.      [END OF if]
15.      [END OF LOOP]
16.      Step 4: RETURN pos
```

Worst Case Time Complexity [Big-O]: **O(n_2)**

Best Case Time Complexity [Big-omega]: **O(n_2)**

Average Time Complexity [Big-theta]: **O(n_2)**

Space Complexity: **O(1)**

How Merge Sort Works?

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

To understand merge sort, we take an unsorted array as the following −

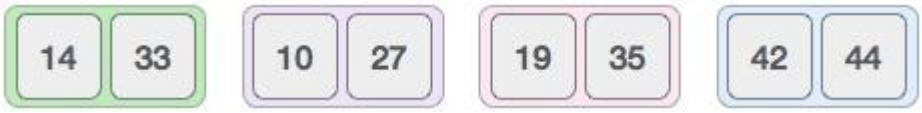
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than the insertion sort. Since we are dealing with sub-problems, we state each sub- problem as sorting a sub-array A[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through sub-problems.

To sort A[p .. r]:

1.Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two sub-arrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

2.Conquer Step

Conquer by recursively sorting the two sub-arrays A[p .. q] and A[q + 1 .. r].

3.Combine Step

Combine the elements back in A[p .. r] by merging the two sorted sub-arrays A[p .. q]
and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

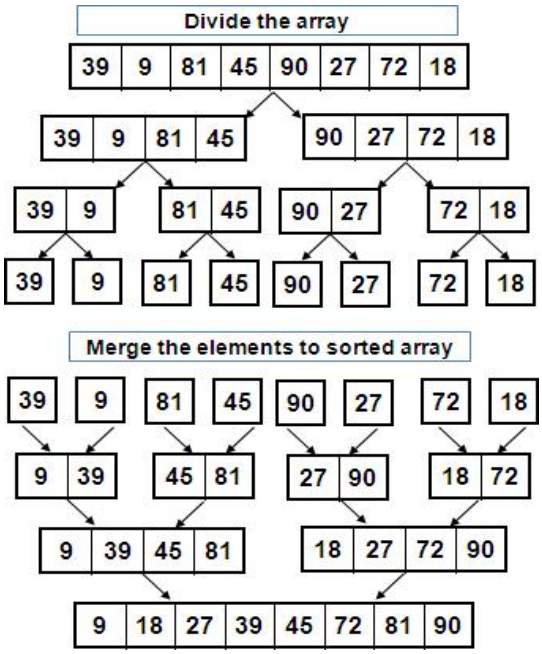
Note that the recursion bottoms out when the sub-array has just one element, so that it is trivially sorted.

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

```
IF p < r                          // Check for base case
  THEN q = FLOOR[(p + r)/2]       // Divide step
       MERGE (A, p, q)            // Conquer step.
       MERGE (A, q + 1, r)        // Conquer step.
       MERGE (A, p, q, r)         // Conquer step.
```

Ex:- A list of unsorted elements are: 39 9 81 45 90 27 72 18



Sorted elements are: 9 18 27 39 45 7

Write a program to implement Merge sort.

```
/* program to sort elements of an array using Merge Sort */
#include<stdio.h>void disp( );
void mergesort(int,int,int);void msortdiv(int,int);
int a[50],n; void main( )
{
 int i; clrscr( );
 printf("\nEnter the n value:");scanf("%d",&n);
 printf("\nEnter elements for an array:");for(i=0;i<n;i++)
  scanf("%d",&a[i]);
  printf("\nBefore Sorting the elements are:");disp( );
  msortdiv(0,n-1);
  printf("\nAfter Sorting the elements are:");disp( );
 getch( );
 }
 void disp( )
 {
  int i; for(i=0;i<n;i++) printf("%d ",a[i]);
 }
```

```
void mergesort(int low,int mid,int high)
{
 int t[50],i,j,k;i=low; j=mid+1; k=low;
 while((i<=mid) && (j<=high))
 {
  if(a[i]>=a[j])
   t[k++]=a[j++];
  else t[k++]=a[i++];
 }
 while(i<=mid) t[k++]=a[i++];
 while(j<=high) t[k++]=a[j++];
 for(i=low;i<=high;i++)a[i]=t[i];
}
 void msortdiv(int low,int high)
  {
   int mid; if(low!=high)
   {
    mid=((low+high)/2);
    msortdiv(low,mid);
     msortdiv(mid+1,high);
```

```
    mergesort(low,mid,high);
   }
  }
```

OUTPUT:

How many elements you want to sort ? : 7

Enter elements for an array : 88 45 54 8 32 6 12

After Sorting the elements are : 6 8 12 32 45 54 88

Convert code into Algorithm

Formulate recursive algorithm for binary search with its timing analysis.

Binary search is quicker than the linear search. However, it cannot be applied on unsorted data structure. The binary search is based on the approach **divide-and-conquer**. The binary search starts by testing the data in the middle element of the array. This determines target is whether in the first half or second half. If target is in first half, we do not need to check the second half and if it is in second half no need to check in first half. Similarly we repeat this process until we find target in the list or not found from the list. Here we need 3 variables to identify first, last and middle elements.

To implement binary search method, the elements must be in sorted order. Search is performed asfollows:

- The key is compared with item in the middle position of an array
- If the key matches with item, return it and stop
- If the key is less than mid positioned item, then the item to be found must be in first halfof array, otherwise it must be in second half of array.
- Repeat the procedure for lower (or upper half) of array until the element is found.

Recursive Algorithm:

Binary_Search(a,key,lb,ub)

begin

Step 1: [initialization]

lb=0

ub=n-1;

Step 2: [search for the item]

Repeat through step 4 while lower bound(lb) is less than upper bound.

Step 3: [obtain the index of middle value]mid = (lb+ub)/2

Step 4: [compare to search for item] if(key < a[mid]) then

ub=mid-1

otherwise if(key > a[mid]) then lb=mid+1;

otherwise if(key==a[mid]) Write “match found” return (mid)

return Binary_Search(a,key,lb,ub)

Step 5: [unsuccessful search]

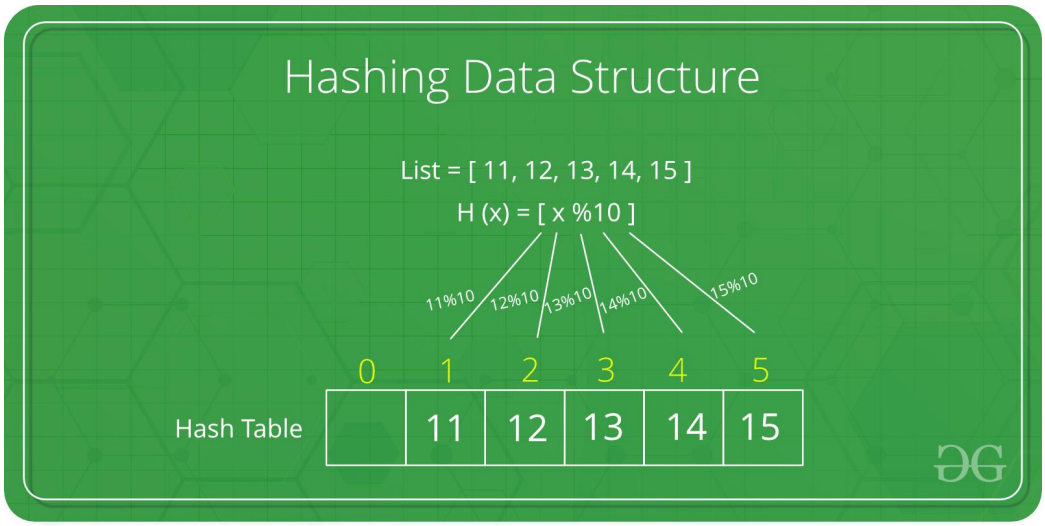
Write “match not found”

Step 6: [end of algorithm]

What is Hashing?

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.
Hash Function is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a significant number or string to a small integer that can be used as the index in the hash table.

Let a hash function H(x) maps the value x at the index x%10 in an Array. For *example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.*



different hash functions:

1. ***Division Method.***
2. ***Mid Square Method.***
3. ***Folding Method.***
4. ***Multiplication Method.***

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

$h(K) = k \bmod M$

Here,

k is the key value, and

M is the size of the hash table.

Example:

k = 12345
M = 95
h(12345) = 12345 mod 95
= 90

k = 1276
M = 11
h(1276) = 1276 mod 11
= 0

Mid Square Method

The steps involved in computing this hash method include the following

- Squaring the value of k (like k*k)

1. Extract the hash value from the middle r digits.

Formula - h(K) = h(k x k)

(where k = key value)

k = 60
k x k = 60 x 60
= 3600
h(60) = 60

The hash value obtained is 60

2. **Digit Folding Method:**

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.

2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

$k = k1, k2, k3, k4, ..., kn$

$s = k1 + k2 + k3 + k4 + ...+ kn$

$h(K) = s$

Here,

s is obtained by adding the parts of the key **k**

$k = 12345$

$k1 = 12, k2 = 34, k3 = 5$

$s = k1 + k2 + k3$

$= 12 + 34 + 5$

$= 51$

$h(K) = 51$

Commonly used hash functions:

1. SHA (Secure Hash Algorithm): SHA is a family of cryptographic hash functions designed by the National Security Agency (NSA) in the United States. The most widely used SHA algorithms are SHA-1, SHA-2, and SHA-3.

Array in Data Structure

Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

In C programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects. Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

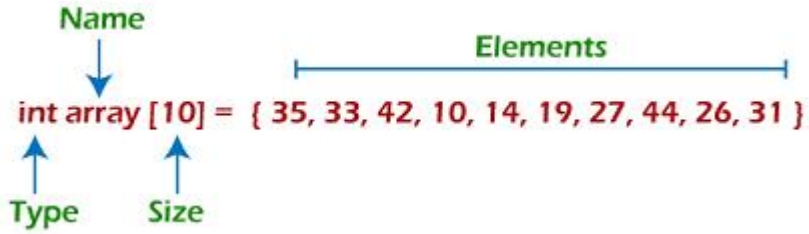
Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Why are arrays required?

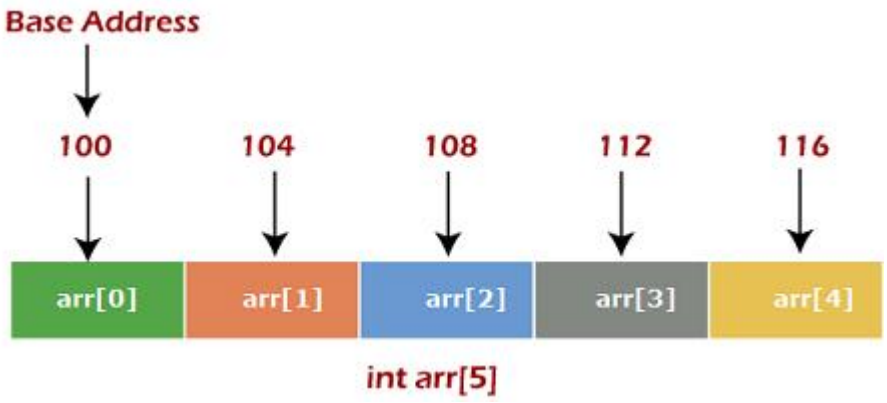
Arrays are useful because -

- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- **Arrays are good for storing multiple values in a single variable -** In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Memory allocation of an array

As stated above, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory. Each element of the array is represented by proper indexing.

We can define the indexing of an array in the below ways -

1. 0 (zero-based indexing): The first element of the array will be arr[0].
2. 1 (one-based indexing): The first element of the array will be arr[1].
3. n (n - based indexing): The first element of the array can reside at any random index number
4. 

How to access an element from the array?

We required the information given below to access any random element from the array -

- Base Address of the array.
- Size of an element in bytes.
- Type of indexing, array follows.

The formula to calculate the address to access an array element -

1. Byte address of element A[i] = base address + size * (i - first index)

Basic operations

Now, let's discuss the basic operations supported in the array -

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.
- Update - It updates an element at a particular index.

Traversal operation

This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program -

```c
#include <stdio.h>
void main() {
  int Arr[5] = {18, 30, 15, 70, 12};
int i;
  printf("Elements of the array are:\n");
  for(i = 0; i<5; i++) {
    printf("Arr[%d] = %d,  ", i, Arr[i]);
  }
```

```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12,
```

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array. Now, let's see the implementation of inserting an element into the array.

```c
#include <stdio.h>
int main()
{
   int arr[20] = { 18, 30, 15, 70, 12 };
   int i, x, pos, n = 5;
   printf("Array elements before insertion\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");

   x = 50; // element to be inserted
   pos = 4;
   n++;

   for (i = n-1; i >= pos; i--)
      arr[i] = arr[i - 1];
   arr[pos - 1] = x;
   printf("Array elements after insertion\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
```

```
21.         printf("\n");
22.         return 0;
23.     }
```

Output

```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Search operation

This operation is performed to search an element in the array based on the value or index.

```
1.      #include <stdio.h>
2.      void main() {
3.        int arr[5] = {18, 30, 15, 70, 12};
4.        int item = 70, i, j=0 ;
5.
6.        printf("Given array elements are :\n");
7.
8.        for(i = 0; i<5; i++) {
9.          printf("arr[%d] = %d,  ", i, arr[i]);
10.       }
11.        printf("\nElement to be searched = %d", item);
12.       while( j < 5){
13.         if( arr[j] == item ) {
14.           break;
15.         }
16.
17.         j = j + 1;
18.       }
19.
20.       printf("\nElement %d is found at %d position", item, j+1);
21.     }
```

Output

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Element to be searched = 70
Element 70 is found at 4 position
```

Advantages of Array

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

Disadvantages of Array

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store less number of elements than the declared size.

2D Array– Defined

2 Dimensional arrays are often defined as an array of arrays. A 2D array is also called a matrix. A matrix can be depicted as a table of rows and columns.

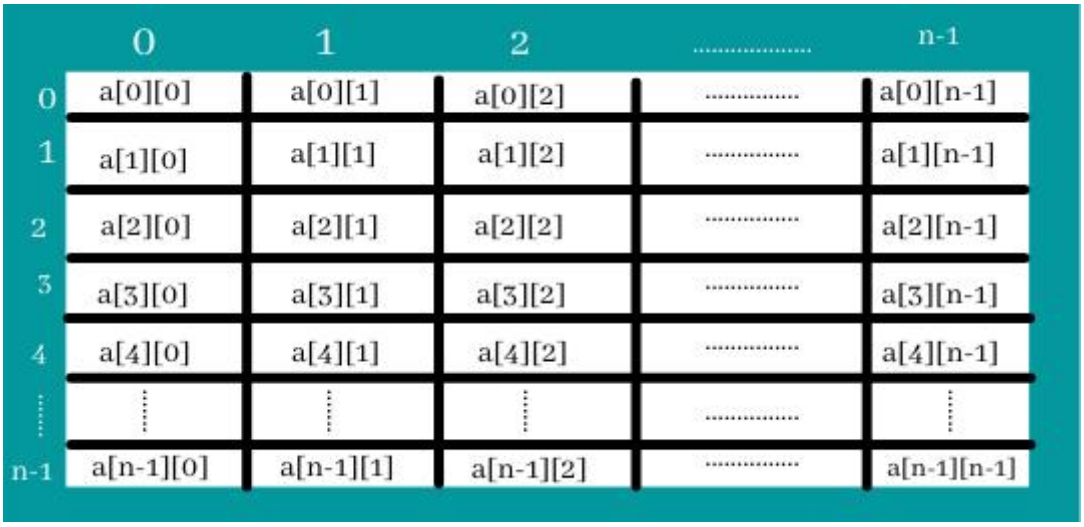
How to declare 2D Array?

The syntax of declaring a 2D array is very much similar to that of an 1D array.

Syntax –

int arr[max_rows][max_columns];

It produces data structure as shown below:




```
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

How to access data in a 2D array?

As in a one-dimensional array, data can be accessed by using only an index, and similarly, in a two-dimensional array, we can access the cells individually by using the indices of the cells. There are two indices attached to a single cell, one is its row number, and the other one is its column number.

Syntax to store the value stored in any particular cell of an array –

```
int x = a[i][j];
```

Here i and j are the row and column indices, respectively.

Initializing 2D array –

When we declare a one-dimensional array, there's no need to specify the size of the array, but it's not the same for a two-dimensional array. For a 2D array, we need to specify at least the row size, i.e., the second dimension.

Syntax to declare 2D array –

```
int arr[2][2] = {1,2,3,4}
```

The number of elements present in a 2D array will always be equal to (**number of rows * number of columns**).

Two-dimensional array example in C

```
#include<stdio.h>
int main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++){
 for(j=0;j<3;j++){
   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
 }//end of j
}//end of i
return 0;
}
```

Output

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```