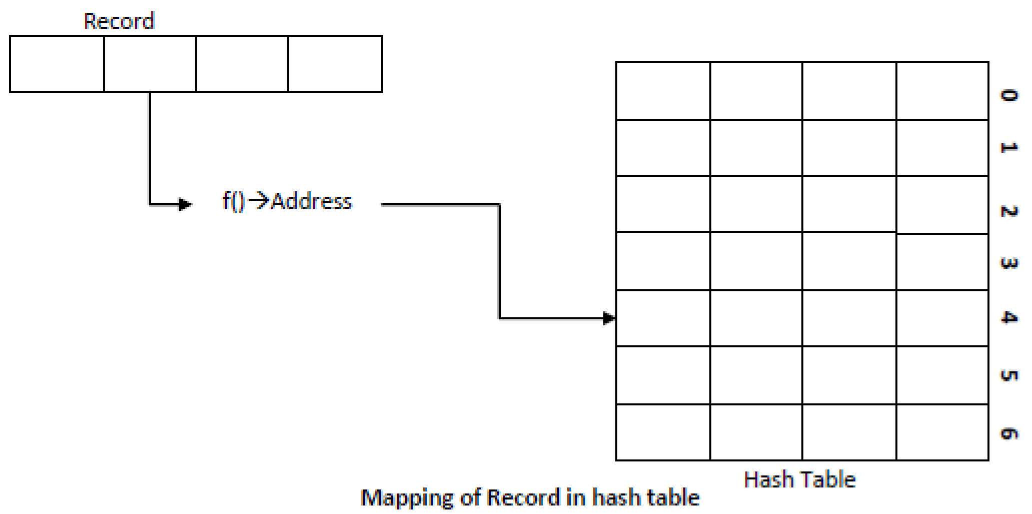
What is Hashing?

- Sequential search requires, on the average O(n) comparisons to locate an element. So many comparisons are not desirable for a large database of elements.
- Binary search requires much fewer comparisons on the average O (log n) but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require 0(n log n) comparisons.
- There is another widely used technique for storing of data called hashing. It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order (0(1)). In its worst case, hashing algorithm starts behaving like linear search.
- Best case timing behavior of searching using hashing = O(1)
- Worst case timing Behavior of searching using hashing = O(n)
- In hashing, the record for a key value "key", is directly referred by calculating the address from the key value. Address or location of an element or record, x, is obtained by computing some arithmetic function f. f(key) gives the address of x in the table.



Mapping of Record in hash table

Hash Table Data Structure:

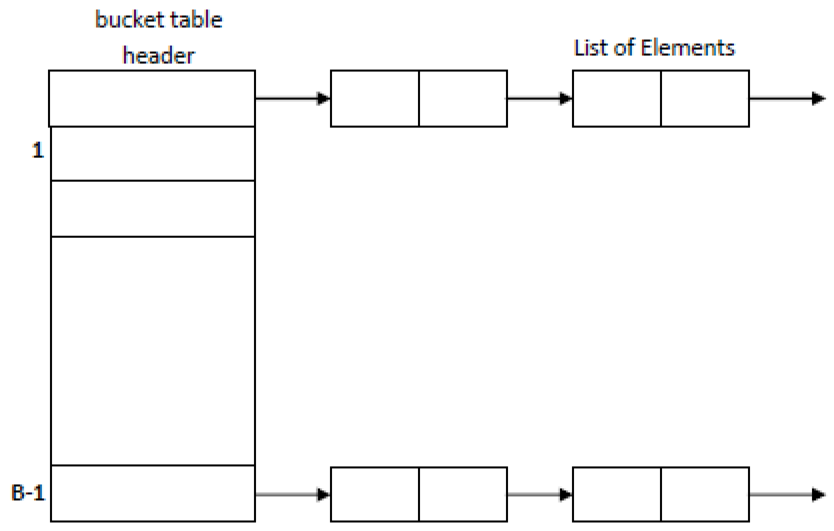
There are two different forms of hashing.

1. **Open hashing or external hashing**

   Open or external hashing, allows records to be stored in unlimited space (could be a hard disk). It places no limitation on the size of the tables.

2. Close hashing or internal hashing
   Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

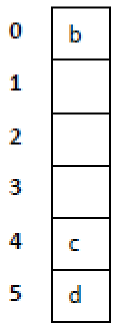
1. Open Hashing Data Structure



The open hashing data organization

- The basic idea is that the records [elements] are partitioned into B classes, numbered 0,1,2 … B-l
- A Hashing function f(x) maps a record with key n to an integer value between 0 and B-l.
- Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

2. Close Hashing Data Structure



- A closed hash table keeps the elements in the bucket itself.
- Only one element can be put in the bucket
- If we try to place an element in the bucket f(n) and find it already holds an element, then we say that a collision has occurred.
- In case of collision, the element should be rehashed to alternate empty location f1(x), f2(x), … within the bucket table
- In closed hashing, collision handling is a very important issue.

Hashing Functions

Characteristics of a Good Hash Function

- A good hash function avoids collisions.
- A good hash function tends to spread keys evenly in the array.
- A good hash function is easy to compute.

Different hashing functions

1. Division-Method
2. Midsquare Methods
3. Folding Method
4. Digit Analysis
5. Length Dependent Method
6. Algebraic Coding
7. Multiplicative Hashing

1. Division-Method

- In this method we use modular arithmetic system to divide the key value by some integer divisor m (may be table size).
- It gives us the location value, where the element can be placed.
- We can write,

  L = (K mod m) + 1

  where L => location in table/file

  K => key value

  m => table size/number of slots in file
- Suppose, k = 23, m = 10 then

  L = (23 mod 10) + 1= 3 + 1=4, The key whose value is 23 is placed in 4th location.

2. Midsquare Methods

- In this case, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.
- Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then you select the address as 56 (i.e. two digits starting from middle of 256).

3. Folding Method

- Most machines have a small number of primitive data types for which there are arithmetic instructions.
- Frequently key to be used will not fit easily in to one of these data types
- It is not possible to discard the portion of the key that does not fit into such an arithmetic data type

- The solution is to combine the various parts of the key in such a way that all parts of the key affect for final result such an operation is termed folding of the key.
- That is the key is actually partitioned into number of parts, each part having the same length as that of the required address.
- Add the value of each parts, ignoring the final carry to get the required address.
- This is done in two ways :
    - **Fold-shifting:** Here actual values of each parts of key are added.
        - Suppose, the key is : 12345678, and the required address is of two digits,
        - Then break the key into: 12, 34, 56, 78.
        - Add these, we get 12 + 34 + 56 + 78 : 180, ignore first 1 we get 80 as location
    - **Fold-boundary:** Here the reversed values of outer parts of key are added.
        - Suppose, the key is : 12345678, and the required address is of two digits,
        - Then break the key into: 21, 34, 56, 87.
        - Add these, we get 21 + 34 + 56 + 87 : 198, ignore first 1 we get 98 as location

4. Digit Analysis

- This hashing function is a distribution-dependent.
- Here we make a statistical analysis of digits of the key, and select those digits (of fixed position) which occur quite frequently.
- Then reverse or shifts the digits to get the address.
- For example, if the key is : 9861234. If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key. So we get, 62. Reversing it we get 26 as the address.

5. Length Dependent Method

- In this type of hashing function we use the length of the key along with some portion of the key j to produce the address, directly.
- In the indirect method, the length of the key along with some portion of the key is used to obtain intermediate value.

6. Algebraic Coding

- Here a n bit key value is represented as a polynomial.
- The divisor polynomial is then constructed based on the address range required.
- The modular division of key-polynomial by divisor polynomial, to get the address-polynomial.
- Let f(x) = polynomial of n bit key = $a_1 + a_2x + ... + a_nx^{n-1}$
- d(x) = divisor polynomial = $x^1 + d_1 + d_2x + ... + d_1x^{1-1}$
- then the required address polynomial will be f(x) mod d(x)

7. Multiplicative Hashing

- This method is based on obtaining an address of a key, based on the multiplication value.

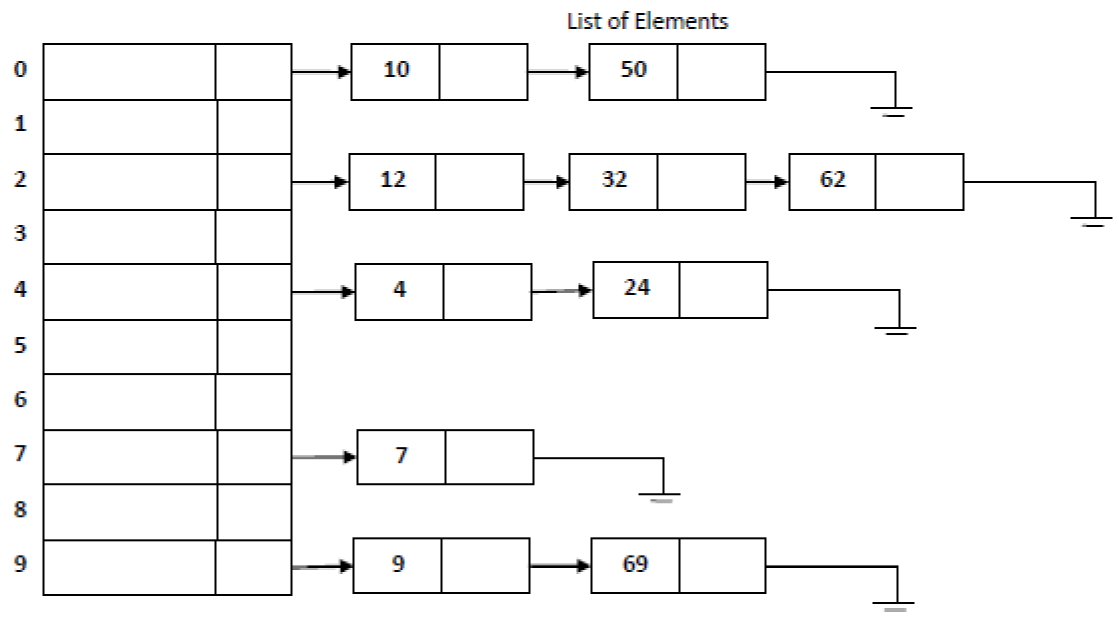
- If k is the non-negative key, and a constant c, $(0 < c < 1)$, compute kc mod 1, which is a fractional part of kc.
- Multiply this fractional part by m and take a floor value to get the address
- $\lfloor m\ (kc \bmod 1) \rfloor$
- $0 < h(k) < m$

Collision Resolution Strategies (Synonym Resolution)

- Collision resolution is the main problem in hashing.
- If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.
- There are several strategies for collision resolution. The most commonly used are :
    1. **Separate chaining** - used with open hashing
    2. **Open addressing** - used with closed hashing

1. Separate chaining

- In this strategy, a separate list of all elements mapped to the same value is maintained.
- Separate chaining is based on collision avoidance.
- If memory space is tight, separate chaining should be avoided.
- Additional memory space for links is wasted in storing address of linked elements.
- Hashing function should ensure even distribution of elements among buckets; otherwise the timing behavior of most operations on hash table will deteriorate.

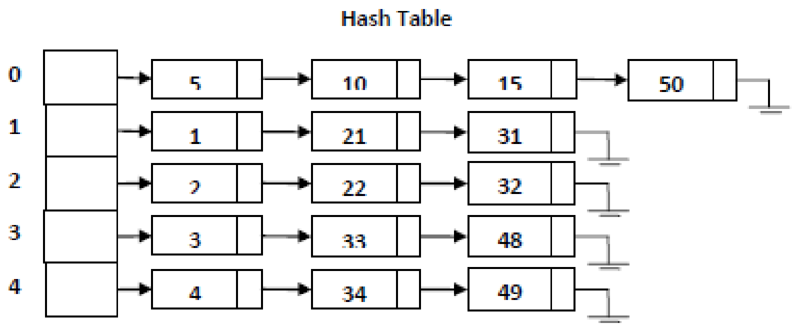


A Separate Chaining Hash Table

Example : The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function. 1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50

An element can be mapped to a location in the hash table using the mapping function **key % 10**.

Hash Table Location	Mapped element
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49



2. Open Addressing

- Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision.
- In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found.
  - a. Linear probing
  - b. Quadratic probing
  - c. Double hashing.

a) Linear Probing

- In linear probing, whenever there is a collision, cells are searched sequentially (with wraparound) for an empty cell.
- Fig. shows the result of inserting keys {5,18,55,78,35,15} using the hash function **(f(key)= key%10)** and linear probing strategy.

	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							15
1							
2							
3							
4							
5		5	5	5	5	5	5
6				55	55	55	55
7						35	35
8			18	18	18	18	18
9					78	78	78

- Linear probing is easy to implement but it suffers from "**primary clustering**"

- When many keys are mapped to the same location (clustering), linear probing will not distribute these keys evenly in the hash table. These keys will be stored in neighborhood of the location where they are mapped. This will lead to clustering of keys around the point of collision

b) ***Quadratic probing***

- One way of reducing "primary clustering" is to use quadratic probing to resolve collision.
- Suppose the "key" is mapped to the location j and the cell j is already occupied. In quadratic probing, the location j, (j+1), (j+4), (j+9), ... are examined to find the first empty cell where the key is to be inserted.
- This table reduces primary clustering.
- It does not ensure that all cells in the table will be examined to find an empty cell. Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

c) ***Double Hashing***

- This method requires two hashing functions f1 (key) and f2 (key).
- Problem of clustering can easily be handled through double hashing.
- Function f1 (key) is known as primary hash function.
- In case the address obtained by f1 (key) is already occupied by a key, the function f2 (key) is evaluated.
- The second function f2 (key) is used to compute the increment to be added to the address obtained by the first hash function f1 (key) in case of collision.
- The search for an empty location is made successively at the addresses f1 (key) + f2(key), f1 (key) + 2f2 (key), f1 (key) + 3f2(key),...

SORTING

Introduction

- Sorting is the process of arranging items in a certain sequence or in different sets .
- The main purpose of sorting information is to optimize it's usefulness for a specific tasks.
- Sorting is one of the most extensively researched subject because of the need to speed up the operations on thousands or millions of records during a search operation.

Types of Sorting :

- **Internal Sorting**

  An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer.

  This is possible whenever the data to be sorted is small enough to all be held in the main memory.

  For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it won't all fit.

  The rest of the data is normally held on some larger, but slower medium, like a hard-disk.

  Any reading or writing of data to and from this slower media can slow the sorting process considerably

- **External Sorting**

  Many important sorting applications involve processing very large files, much too large to fit into the primary memory of any computer.

  Methods appropriate for such applications are called external methods, since they involve a large amount of processing external to the central processing unit.

  There are two major factors which make external algorithms quite different: ƒ

  First, the cost of accessing an item is orders of magnitude greater than any bookkeeping or calculating costs. ƒ

  Second, over and above with this higher cost, there are severe restrictions on access, depending on the external storage medium used: for example, items on a magnetic tape can be accessed only in a sequential manner

Bubble sort

- **Bubble sort**, sometimes referred as **sinking sort.**
- It is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm gets its name from the way smaller elements "bubble" to the top of the list.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use, even compared to insertion sort.

Algorithm

```
for i ← 1 to n do
        for j ← 1 to n-i do
                If Array[j] > Array[j+1] then      /* For decreasing order use < */
                        temp ← Array[j]
                        Array[j] ← A [j+1]
                        Array[j+1] ← temp
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], n, i, j, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
        for (i = 0; I < n; i++)
        {
                scanf("%d", &array[i]);
        }
        for (i = 0 ;i< ( n - 1 );i++)
        {
                for (j = 0 ; j< n - c - 1; j++)
                {
                        if (array[j] > array[j+1])            /* For decreasing order use < */
                        {
```

```
                temp        = array[j];
                array[j]   = array[j+1];
                array[j+1] = temp;
            }
        }
    }
    printf("Sorted list in ascending order:\n");
    for (i = 0 ;i< n ;i++ )
    {
        printf("%d\n", array[i]);
    }
    getch();
}
```

Example

Consider an array A of 5 element

A[0]	45
A[1]	34
A[2]	56
A[3]	23
A[4]	12

Pass-1: The comparisons for pass-1 are as follows.

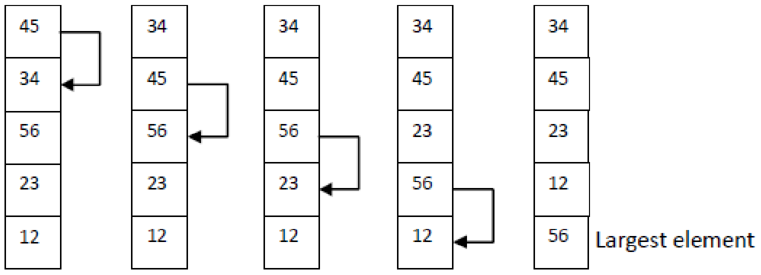
Compare A[0] and A[1]. Since 45>34, interchange them.

Compare A[1] and A[2]. Since 45<56, no interchange.

Compare A[2] and A[3]. Since 56>23, interchange them.

Compare A[3] and A[4]. Since 56>12 interchange them.

At the end of first pass the largest element of the array, 56, is bubbled up to the last position in the array as shown.

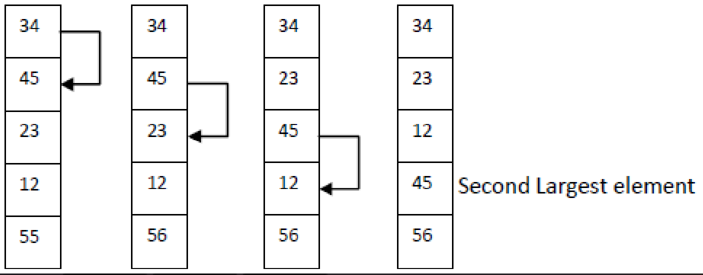


Pass-2: The comparisons for pass-2 are as follows.

Compare A[0] and A[1]. Since 34<45, no interchange.

Compare A[1] and A[2]. Since 45>23, interchange them.

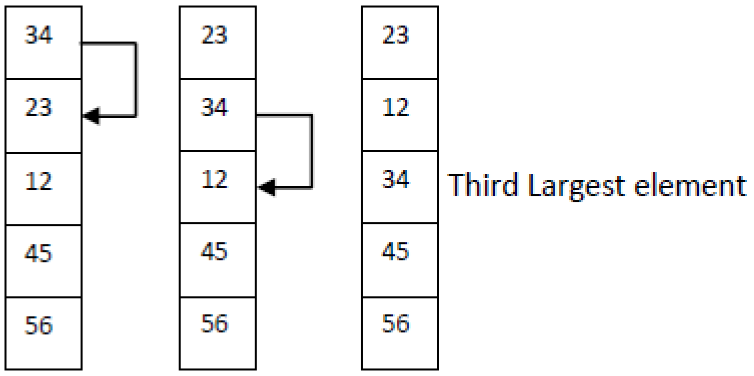
Compare A[2] and A[3]. Since 45>12, interchange them.



Pass-3: The comparisons for pass-3 are as follows.

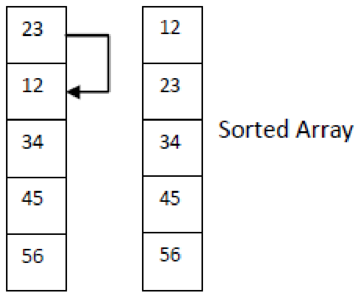
Compare A[0] and A[1]. Since 34>23, interchange them.

Compare A[1] and A[2]. Since 34>12, interchange them.



Pass-4: The comparisons for pass-4 are as follows.

Compare A[0] and A[1]. Since 23>12, interchange them.



Selection Sort

- The idea of algorithm is quite simple.
- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one.
- When unsorted part becomes empty, algorithm stops.

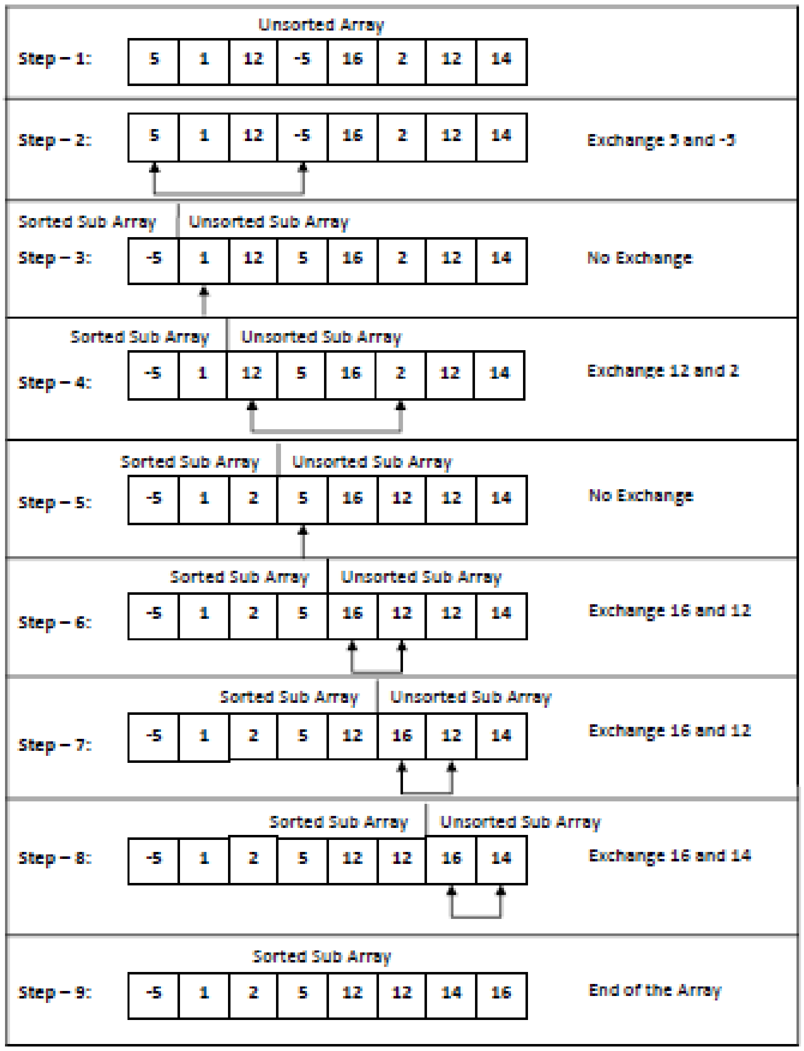
Algorithm

```
SELECTION_SORT (A)
for i ← 1 to n-1 do
          min ← i;
          for j ← i + 1 to n do
                    If A[j] < A[i] then
                              min ← j
          If min!=i then
                    temp ← A[i]
                    A[i] ← A [min]
                    A[min] ← temp
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], n, i, j, min, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
        for ( i = 0 ; i < n ; i++ )
        {
                scanf("%d", &array[i]);
        }
        for ( i = 0 ; i < ( n - 1 ) ; i++ )
        {
                min = i;
                for ( j = i + 1 ; j < n ; j++ )
                {
                        if ( array[min] > array[j] )
                        min = j;
                }
                if ( min != i )
                {
                        temp = array[i];
                        array[i] = array[min];
                        array[min] = temp;
                }
        }
        printf("Sorted list in ascending order:\n");
        for ( i = 0 ; i < n ; i++ )
        {
                printf("%d\n", array[i]);
        }
        getch();
 }
```

Example


Unsorted Array
Step – 1: 5 1 12 -5 16 2 12 14
Step – 2: 5 1 12 -5 16 2 12 14
Exchange 5 and -5
Sorted Sub Array
Unsorted Sub Array
Step – 3: -5 1 12 5 16 2 12 14
No Exchange
Sorted Sub Array
Unsorted Sub Array
Step – 4: -5 1 12 5 16 2 12 14
Exchange 12 and 2
Sorted Sub Array
Unsorted Sub Array
Step – 5: -5 1 2 5 16 12 12 14
No Exchange
Sorted Sub Array
Unsorted Sub Array
Step – 6: -5 1 2 5 16 12 12 14
Exchange 16 and 12
Sorted Sub Array
Unsorted Sub Array
Step – 7: -5 1 2 5 12 16 12 14
Exchange 16 and 12
Sorted Sub Array
Unsorted Sub Array
Step – 8: -5 1 2 5 12 12 16 14
Exchange 16 and 14
Sorted Sub Array
Step – 9: -5 1 2 5 12 12 14 16
End of the Array

Quick Sort

- Quicksort is the currently fastest known sorting algorithm and is often the best practical choice for sorting, as its average expected running time is O(n log(n)).
- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Quicksort, like merge sort, is a divide-and-conquer recursive algorithm.
- The basic divide-and-conquer process for sorting a sub array A[$i..j$] is summarized in the following three easy steps:
    - **Divide:** Partition T[$i..j$] Into two sub arrays T[$i..l$-1] and T[l+1... j] such that each element of T[$i..l$-1] is less than or equal to T[l], which is, in turn, less than or equal to each element of T[l+1... j]. Compute the index l as part of this partitioning procedure
    - **Conquer:** Sort the two sub arrays T[$i..l$-1] and T[l+1... j] by recursive calls to quicksort.
    - **Combine:** Since the sub arrays are sorted in place, no work is needed to combing them: the entire array T[$i..j$] is now sorted.

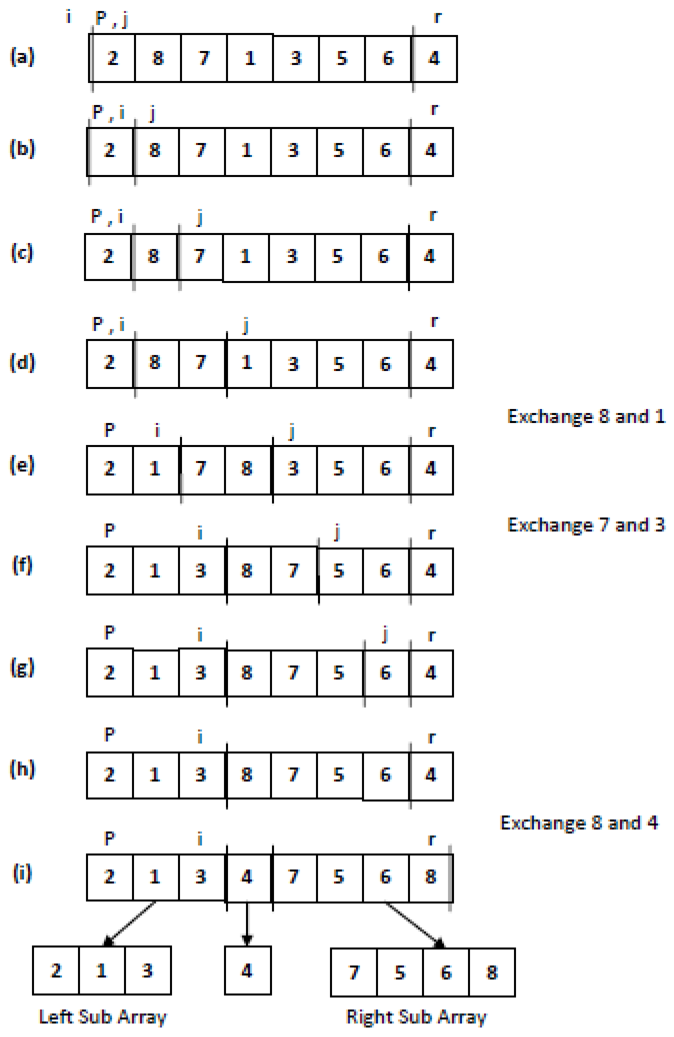
Algorithm

```
Procedure pivot (T [i… j]; var l)
{Permutes the elements in array T [i… j] and returns a value l such that, at the end, $i<=l<=j$,
T[k] <=P for all $i \le k < l$, T[l] =P, and T[k] > P for all $l < k \le j$, where P is the initial value T[i]}
P ← T[i]
K ← i; l ← j+1
Repeat k ← k+1 until T[k] > P
Repeat l ← l-1 until T[l] ≤ P
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until T[k] > P
    Repeat l ← l-1 until T[l] ≤ P
Swap T[i] and T[l]

Procedure quicksort (T [i… j])
{Sorts sub array T [i… j] into non decreasing order}
if j – i is sufficiently small then insert (T[i, …,j])
else
 pivot (T[i,…,j],l)
 quicksort (T[i,…, l - 1])
 quicksort (T[l+1,…,j]
```

Example

Sort given array using Quick Sort:

2	8	7	1	3	5	6	4



Apply same method for left and right sub array finally we will get sorted

Merge Sort

- The merge sort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:
    - **DIVIDE:** Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.
    - **CONQUER:** Sort the two subsequences recursively using the merge sort.
    - **COMBINE:** Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.
- Note that recursion "**bottoms out**" when the sequence to be sorted is of unit length.
- Since every sequence of length 1 is in sorted order, no further recursive call is necessary.
- The key operation of the merge sort algorithm is the merging of the two sorted sub sequences in the "combine step".
- To perform the merging, we use an auxiliary procedure Merge (A,p,q,r), where A is an array and p,q and r are indices numbering elements of the array such that procedure assumes that the sub arrays A[p..q] and A[q+1...r] are in sorted order.
- It merges them to form a single sorted sub array that replaces the current sub array A[p..r]. Thus finally, we obtain the sorted array A[1..n], which is the solution.

Algorithm

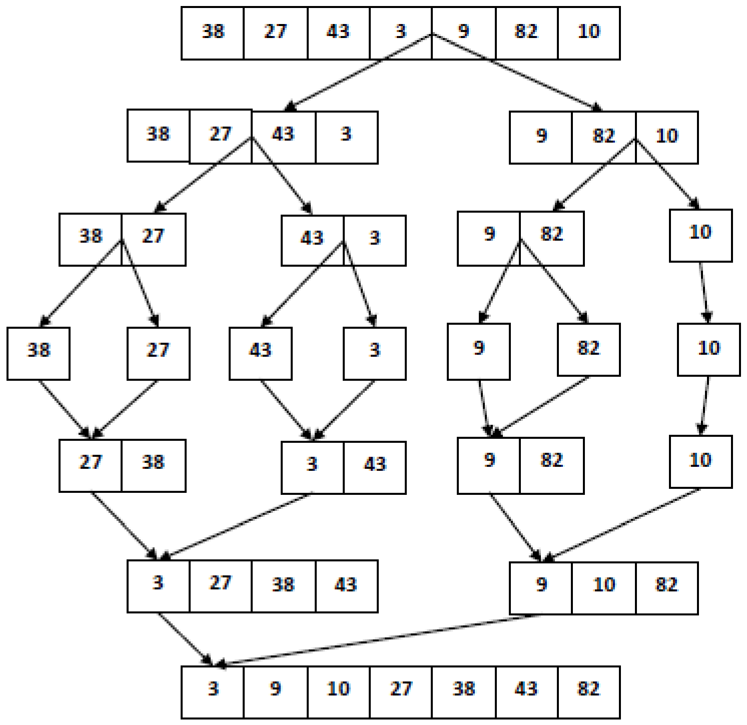
```
MERGE (A,p,q,r)
n1 = q  -p + 1
n2 = r – q
let L[1…n1+1] and R[1…n2+1] be new arrays
for i = 1 to n1
        L[i] = A[p+i-1]
for j = 1 to n2
        R[j] = A[q+j]
L[n1+1] = infinite
R[n2+1]= infinite
i=1
j=1
for k = p to r
        if L[i] ≤ R[j]
                A[k]=L[i]
                i = i +1
        else A[k] = R[j]
                j = j + 1
```

```
MERGE SORT (A,p,r)
if p < r
        then q<-- [ (p + r) / 2 ]
        MERGE SORT(A,p,q)
        MERGER SORT(A,q + 1,r)
        MERGE(A,p,q,r)
```

Example

Sort given array using merge sort

38	27	43	3	9	82	10



Linear/Sequential Search

- In computer science, linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.
- Linear search is the simplest search algorithm.
- It is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list.

Algorithm

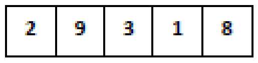
```
# Input: Array A, integer key
# Output: first index of key in A,
# or -1 if not found

Algorith: Linear_Search
for i = 0 to last index of A:
        if A[i] equals key:
                return i
return -1
```

Program

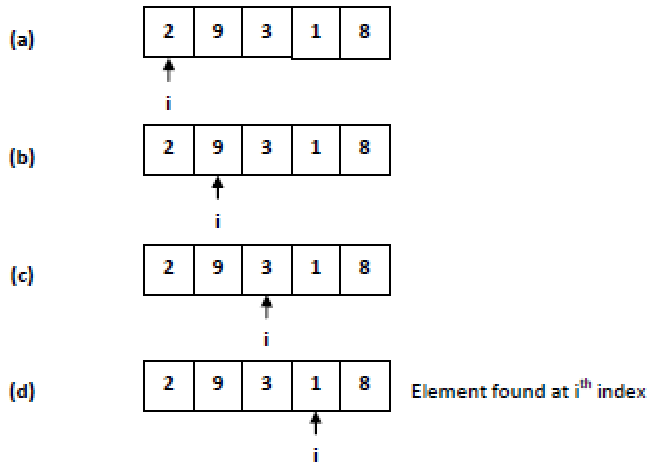
```
#include <stdio.h>
void main()
{
	int array[100], key, i, n;

	printf("Enter the number of elements in array\n");
	scanf("%d",&n);

	printf("Enter %d integer(s)\n", n);

	for (i = 0; i < n; i++)
	{
		printf("Array[%d]=", i);
		scanf("%d", &array[i]);
	}

	printf("Enter the number to search\n");
	scanf("%d", &key);


	for (i = 0; i < n; i++)
	{
		if (array[i] == key)    /* if required element found */
		{
			printf("%d is present at location %d.\n", key, i+1);
			break;
		}
	 }
	if (i == n)
	{
		printf("%d is not present in array.\n", search);
	}
	getch();
}
```

Example

Search for 1 in given array:



Comparing value of i^{th} index with element to be search one by one until we get seache element or end of the array



Binary Search

- If we have an array that is sorted, we can use a much more efficient algorithm called a Binary Search.
- In binary search each time we divide array into two equal half and compare middle element with search element.
- If middle element is equal to search element then we got that element and return that index otherwise if middle element is less than search element we look right part of array and if middle element is greater than search element we look left part of array.

Algorithm

```
# Input: Sorted Array A, integer key
# Output: first index of key in A, or -1 if not found

Algorith: Binary_Search (A, left, right)
while left <= right
        middle = index halfway between left, right
        if D[middle] matches key
                return middle
        else if key less than A[middle]
                right = middle -1
        else
                left = middle + 1
return -1
```

Program

```
#include <stdio.h>
void main()
{
           int i, first, last, middle, n, key, array[100];

          printf("Enter number of elements\n");
          scanf("%d",&n);
          printf("Enter %d integers in sorted order\n", n);
          for ( i = 0 ; i < n ; i++ )
          {
                    scanf("%d",&array[i]);
          }
          printf("Enter value to find\n");
          scanf("%d",&key);
```

```
          first = 0;
          last = n - 1;
          middle = (first+last)/2;

           while( first <= last )
          {
                    if (array[middle] == key)
                    {
                              printf("%d found at location %d.\n", key, middle+1);
                              break;
                    }
                    else if ( array[middle]>key )
                     {
                              Last=middle - 1;
                    }
                    else
                              first = middle + 1;

                    middle = (first + last)/2;
           }
          if ( first > last )
          {
                    printf("Not found! %d is not present in the list.\n", key);
          }
          getch();
 }
```

Example

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 --> (middle element is 19 > 6): Search in left part

-1 5 6 18 **19** 25 46 78 102 114

Step 2 --> (middle element is 5 < 6): Search in Right part

-1 **5** 6 18

Step 3 --> (middle element is 6 == 6): Element Found

6 18