UNIT-4 DS

SEARCHING

Searching refers to the operation of finding a given item in a collection of items stored in any order or random manner. If item is found search is successful otherwise Search is unsuccessful.

There are three types of Searching Methods:-

1. Linear Search
2. Binary Search
3. Hashing

Linear Search

LINEAR SEARCH ALGORITHM

Linear Search(A,N,ITEM,LOC)

Here A is an array with **N** elements, ITEM is the data to be search in array A. This algorithm finds the location LOC of ITEM in A, or sets **LOC = -1** , If the search is unsuccessful.

```
Step 1.  Set K: =0 and LOC : = -1
Step 2. repeat step 3 and 4 while K < N
Step 3.  If A[K] = =ITEM     then:
                        Set   LOC : = K ,  Goto step 5
            [End of if structure]
Step 4.   Set  K: = K+1   [increment counter]
             [End of while]
Step 5. if LOC = -1 then   Write "search is unsuccessful ITEM not found "
            else write "Item is present in location " LOC
            [End of if structure]
Step 6. Exit
```

Write a program on Linear Search in C using array.

```
#include<stdio.h>

#include<conio.h>

int Lsearch(int [],int,int);
```

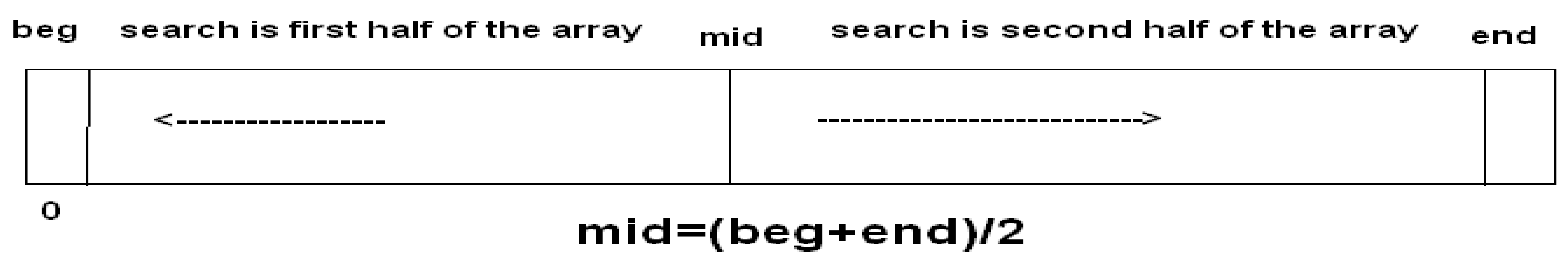
```
void main()

{

int arr[20],item,loc,i,n;

clrscr();

printf("Enter no of items \n");

scanf("%d",&n);

printf("Enter elements of the array");

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("Enter the item to search");

scanf("%d",&item);

loc=Lsearch(arr,item,n);

if(loc==-1)

printf("The item not found");

else

printf("the location ofthe item is %d",loc);

getch();

}

int Lsearch(int p[], int x,int n)

{

int i;

for(i=0;i<n;i++)

{

if(p[i]==x)

return i;
}
return (-1);
}
```

BINARY SEARCH

We can apply Binary Search Only to **SORTED ARRAY(i.e. Array in Ascending or Descending Order).**

Here we compare the item with middle element if it is equal to middle element then search is successful otherwise if it is greater than middle then search will be in the right portion of the array excluding the middle but if it is less than middle then search will be in the left portion of the array excluding middle. the process will continue till we find the element or middle element which has no left or right portion to search.

It is efficient algorithm as it reduces the no. of comparisons and runtime.



[SKETCH DIAGRAM FOR BINARY SEARCH TECHNIQUE]

Algorithm for Binary Search

```
1.  Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first a
    rray element, 'upper_bound' is the index of the last array element, 'val' is the value to search
2.  Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
3.  Step 2: repeat steps 3 and 4 while beg <=end
4.  Step 3: set mid = (beg + end)/2
5.  Step 4: if a[mid] = val
6.  set pos = mid
7.  print pos
8.  go to step 6
9.  else if a[mid] > val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit
```

C program on Binary search

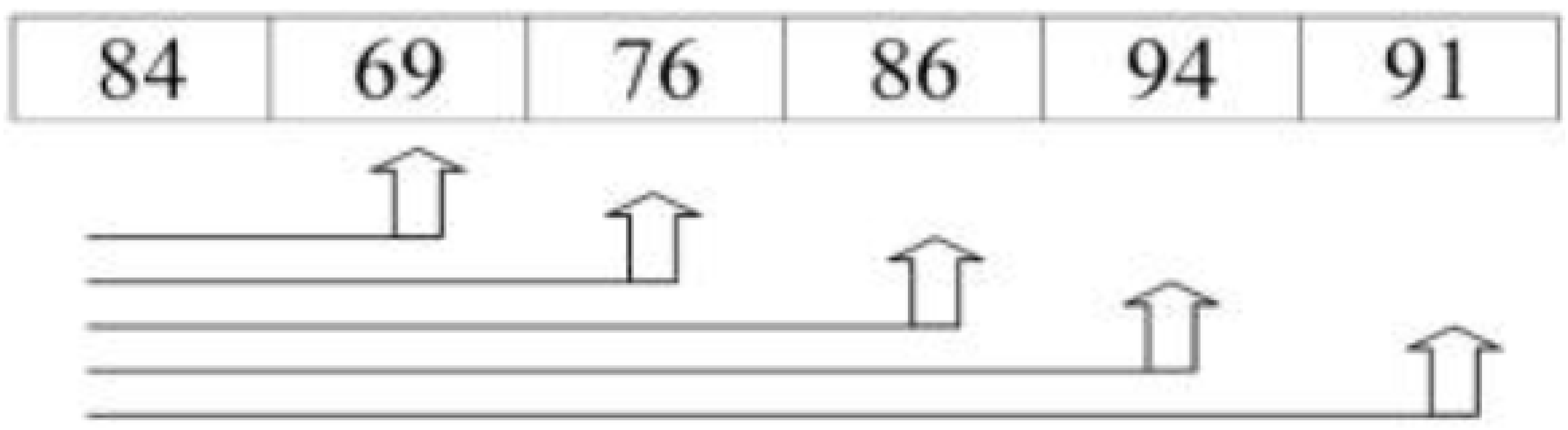
```
#include<stdio.h>
#include<conio.h>
int Bsearch(int [],int,int);
void main()
{
int arr[20],item,loc,i,n;
clrscr();
printf("Enter no of items \n");
scanf("%d",&n);
printf("Enter elements of the array in ascending order");
for(i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
printf("Enter the item to search");
scanf("%d",&item);
loc=Bsearch(arr,item,n);
if(loc==-1)
printf("The item not found");
else
printf("the location of the item is %d",loc);
getch();
}
int Bsearch(int p[], int x,int n)
{
int low,hig,mid,i;
low=0,hig=n-1;

while(low<=hig)
{
mid=(low+hig)/2;
if(x<p[mid])

{
hig=mid-1;
}
if(x>p[mid])
{
low=mid+1;
}
if(x==p[mid])
return mid;
}
return -1;
}
```

SORTING

Sorting refers to the operation of arranging data in Ascending or Descending order. Sorting is done for the purpose of searching. Sorting always on a particular key.

EXCHANGE SORT

The **exchange sort** is similar to its cousin, the bubble sort, in that it compares elements of the array and swaps those that are out of order. (Some people refer to the "exchange sort" as a "bubble sort".) The difference between these two sorts is the manner in which they compare the elements. **The exchange sort compares the first element with each following element of the array, making any necessary swaps.**



When the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array swapping elements that are out of order. This sorting process continues until the entire array is ordered.

Let's examine our same table of elements again using an exchange sort for descending order. Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements.

Array at beginning:	84	69	76	86	94	91
After Pass #1:	94	69	76	84	86	91
After Pass #2:	94	91	69	76	84	86
After Pass #3:	94	91	86	69	76	84
After Pass #4:	94	91	86	84	69	76
After Pass #5 (done):	94	91	86	84	76	69

The exchange sort, in some situations, is slightly more efficient than the bubble sort. It is not necessary for the exchange sort to make that final complete pass needed by the bubble sort to determine that it is finished.

Algorithm on Exchange sort

```
1. Set i to 0
        2. Set j to i + 1
        3. If a[i] > a[j], exchange their values
        4. Set j to j + 1. If j < n goto step 3
        5. Set i to i + 1. If i < n - 1 goto step 2
        6. a is now sorted in ascending order.
```

Note: n is the number of elements in the array

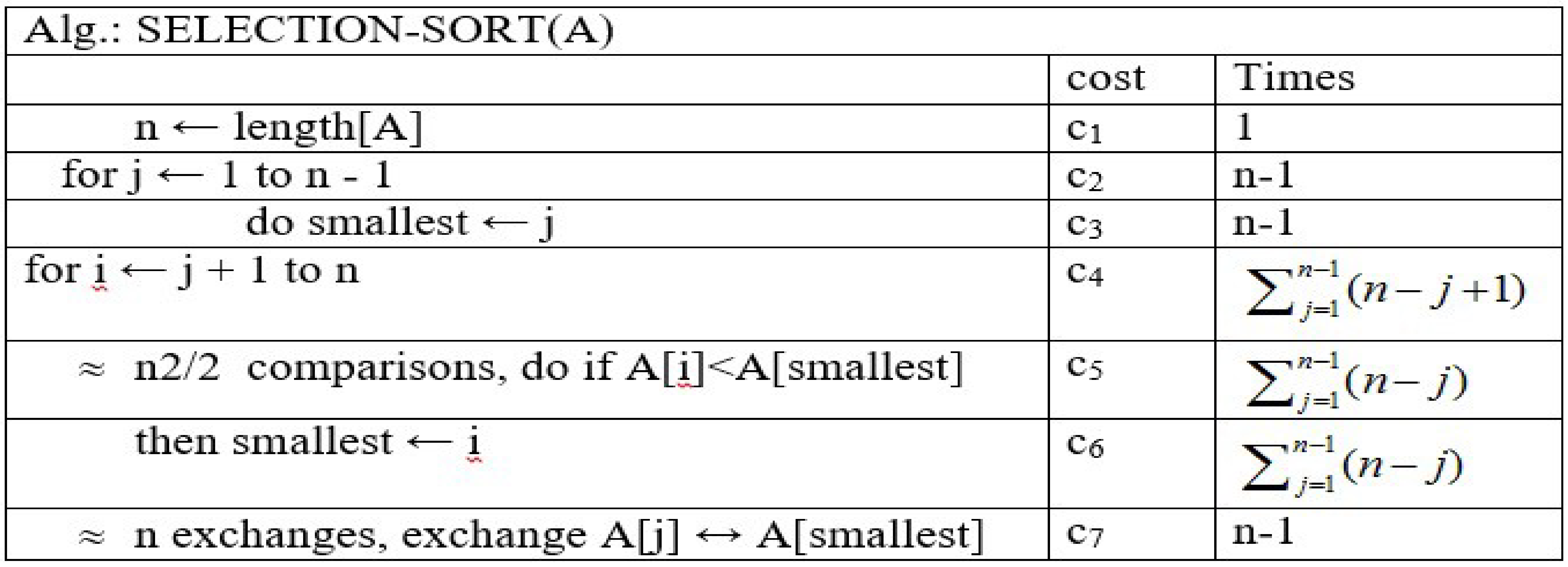
Program on Exchange sort

```
#include<stdio.h>
#include<conio.h>

Void main()
{
        int array[5];          // An array of integers.
        int length = 5;        // Length of the array.
        int i, j;
        int temp;

         //Some input
        for (i = 0; i < 5; i++)
        {
                printf("Enter a number: ");
                scanf("%d", &array[i]);
        }

        //Algorithm
        for(i = 0; i < length -1; i++)
        {
                for (j=(i + 1); j < length; j++)
                {
                        if (array[i] < array[j])
                        {
                                temp = array[i];
                                array[i] = array[j];
                                array[j] = temp;
                        }
                }
        }
        //Some output
        for (i = 0; i < 5; i++)
        {
                printf("%d\n",array[i]);
        }
getch();
}
```

SELECTION SORT

The selection sort is also known as push down list. This sort consist entirely of a selection phase in which largest of remaining elements, large is repeatedly placed in its proper position. Let a be an array of n elements.

Algorithm on Selection Sort

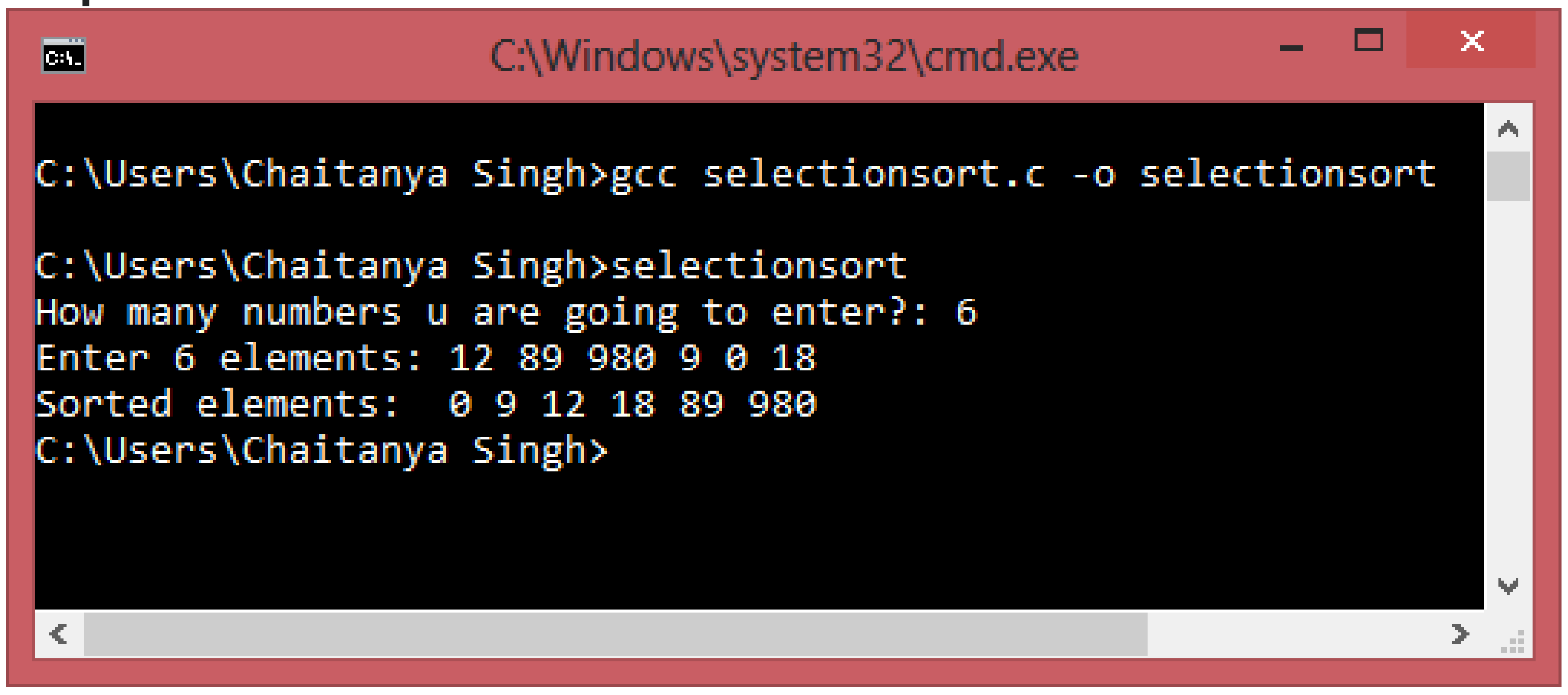


Alg.: SELECTION-SORT(A)		
	cost	Times
n ← length[A]	c_1	1
for j ← 1 to n - 1	c_2	n-1
do smallest ← j	c_3	n-1
for i ← j + 1 to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
≈ n2/2 comparisons, do if A[i]<A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
≈ n exchanges, exchange A[j] ↔ A[smallest]	c_7	n-1

```
#include<stdio.h>
void main()
{
  /* Here i & j for loop counters, temp for swapping,
   * count for total number of elements, number[] to
   * store the input numbers in array. You can increase
   * or decrease the size of number array as per requirement
   */
  int i, j, count, temp, number[25];

  printf("How many numbers u are going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  // Loop to get the elements stored in array
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  // Logic of selection sort algorithm
  for(i=0;i<count;i++){
    for(j=i+1;j<count;j++){
      if(number[i]>number[j]){
        temp=number[i];
        number[i]=number[j];
```

```
            number[j]=temp;
          }
        }
      }

      printf("Sorted elements: ");
      for(i=0;i<count;i++)
        printf(" %d",number[i]);
    }
```

Output:



```
C:\Users\Chaitanya Singh>gcc selectionsort.c -o selectionsort

C:\Users\Chaitanya Singh>selectionsort
How many numbers u are going to enter?: 6
Enter 6 elements: 12 89 980 9 0 18
Sorted elements:  0 9 12 18 89 980
C:\Users\Chaitanya Singh>
```

As you can see that we have entered 6 elements in random order and the program sorted them in ascending order by using selection sort algorithm which we have implemented in the program. You can also modify this same program to sort the elements in descending order as well.

Bubble sort

This algorithm sorts the array A with N elements.

ALGORITHM

```
step 1. set i=0  [initialization]
Step 2.repeat step 3 to 4 until i < N-1 [ for total no. of passes]
Step 3. set j=0 [initialization]
Step 4. repeat step 4 until j < N- i -1 [for total no of  comparisons‘ in each pass]
        if A[ j ]  > A[ j + 1]  then
        set  temp =A [j ]
        set A [j ]= A[ j +1]
        set A[j+1] = temp
```

```
            [end if]
             j=j+1
             [end of loop1]
             i=i+1
             [End of loop2]
    Step 5. Exit
```

Program on bubble sort

```
#include<stdio.h>

#include<conio.h>

void bubble(int [],int);

#define  MAX 100

void main()

{

int arr[MAX],item,loc,i,n;

clrscr();

printf("Enter no of items \n");

scanf("%d",&n);

printf("Enter elements of the array");

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

bubble(arr,n);

printf("the sorted list is\n");

for(i=0;i<n;i++)
printf("\t%d",arr[i]);
getch();
}
void bubble(int p[],int n)
{
int i,j,temp,xchanges;
for(i=0;i<n-1;i++)
{
  xchanges=0;
  for(j=0;j<n-i-1;j++)
  {
     if(p[j]>p[j+1])
     {
     temp=p[j];
     p[j]=p[j+1];
     p[j+1]=temp;
     xchanges++;
     }}
  if(xchanges==0)
  break;
}
}
```

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(n^2), where **n** is the number of items.This method can be explained in similar way the arrangement of cards by a card player. The card player pics a card and insert it into proper position.

Algorithm for Insertion sort

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert

  for i = 1 to length(A) inclusive do:

    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i

    /*locate hole position for the element to be inserted */

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while

    /* insert the number at hole position */
    A[holePosition] = valueToInsert
```

```
   end for
end procedure
```

Write a program on insertion sort using c language.

```
#include<stdio.h>
#include<conio.h>
void insertion(int [],int);
#define  MAX 100
void main()
{
int arr[MAX],i,n;
clrscr();
printf("Enter no of items \n");
scanf("%d",&n);
printf("Enter elements of the array");
for(i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
insertion(arr,n);
printf("the sorted list is\n");
for(i=0;i<n;i++)
printf("\t%d",arr[i]);
getch();
}
```

```
void insertion(int p[],int n)

{

int i,k,j;

for(i=1;i<n;i++)

{

k=p[i];

for(j=i-1;j>=0 && k<p[j];j–)

p[j+1]=p[j];

p[j+1]=k;

}

}
```

MERGE SORT

Merge-sort is based on the **divide-and-conquer technique**. It can be described into 3 steps :-

1. Divide Step

Divide the n elements into two sub array of n/2 each.

2. Conquer step

Sort the two sub array recursively.

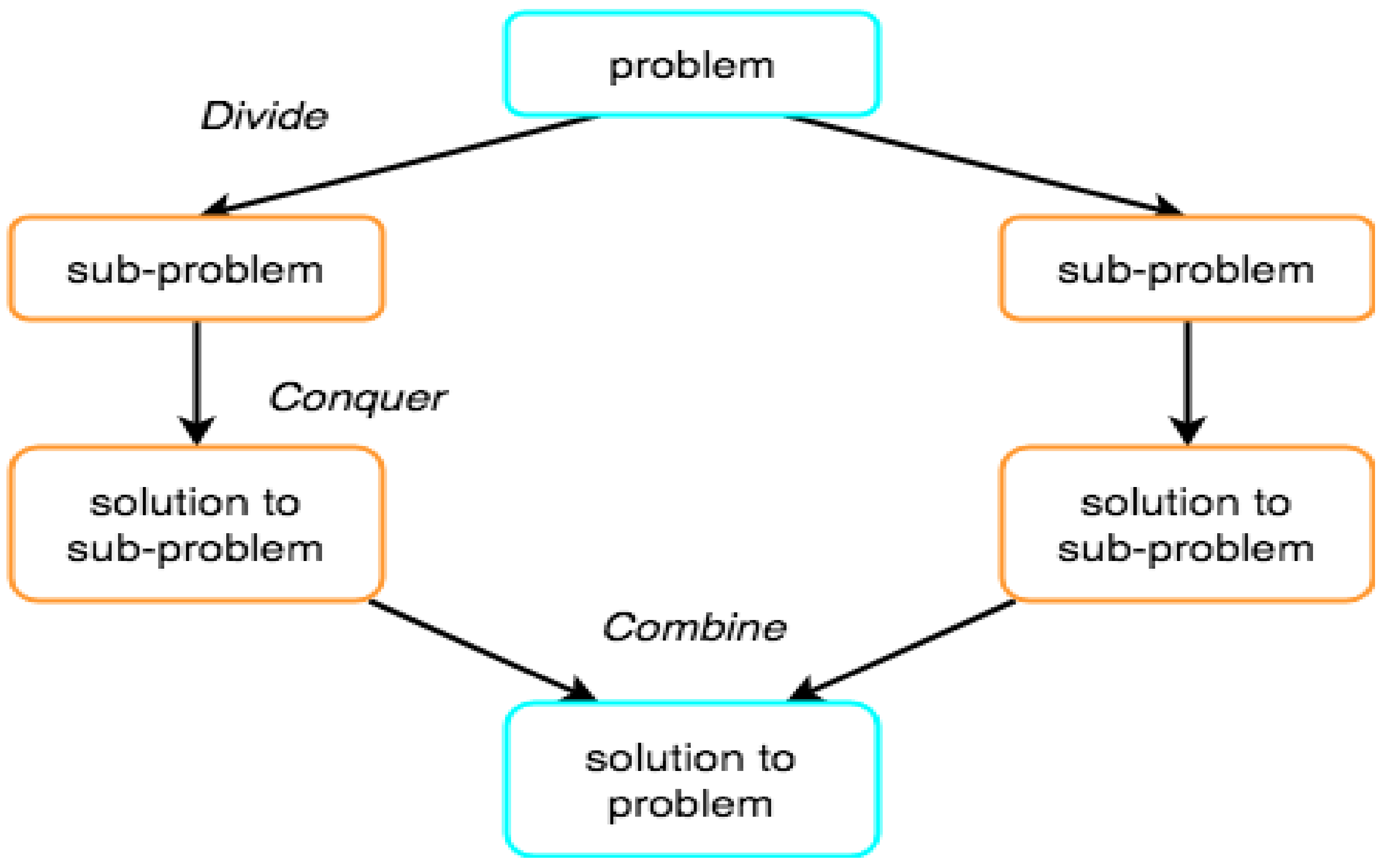
3. Conquer Step

Merge the two sorted array.

In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these sub arrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actual problem.



<u>**Algorithm for Merge Sort**</u>

```
Merge sort (int a[ ], int low, int high)
{
Int mid
If (low !=high)
{
Mid=(low+high)/2
Merge-sort (low,mid)
Merge-sort(mid+1, high)
Merge (low, mid, high)
}
```

Write a program on merge sort using C language.

```
#include<stdio.h>
#include<conio.h>
void mergesort(int[],int,int);
void merge(int[],int,int,int);
void main()
{
int arr[20],n,i;
clrscr();
printf("enter the number of elements to be sorted\n");
```

```
scanf("%d",&n);
printf("Enter the elements\n");
for(i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
mergesort(arr,0,n-1);
printf("\n The sorted array is \n");
for(i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}
getch();
}
void mergesort(int arr[],int low, int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
mergesort(arr,low,mid);
mergesort(arr,mid+1,high);
merge(arr,low,mid,high);
}
}
void merge(int arr[],int low,int mid,int high)
{
int b[20],x,y,i;
x=low;
y=mid+1;
```

```
i=low;
while((x<=mid) && (y<=high))
{
 if(arr[x]<arr[y])
 {
 b[i]=arr[x];
 x=x+1;
 i=i+1;
 }
 else
 {
 b[i]=arr[y];
 y=y+1;
 i=i+1;
 }
 }
 while(x<=mid)
 {
 b[i]=arr[x];
 x=x+1;
 }
 while(y<=high)
 {
 b[i]=arr[y];
 y=y+1;
 i=i+1;
 }
 for(i=low;i<=high;i++)
 arr[i]=b[i];
 }
```

QUICK SORT

Quick sort method was developed by C.A.R. Hoare. It implements **divide and conquer technique**. It is a 3 step process for sorting a array.

Divide:- Divide the array into two sub-array.

Conquer:- The two sub arrays are sorted recursively.

Combine:- Combine each sub array.

←------------------ n ---------------→

C program of Quick Sort

```
//quick sort  program in C

#include<stdio.h>

#include<conio.h>

void quicksort(int[],int,int);

int partition(int[],int,int);

void main()

{

int arr[20],i,n;
clrscr();

printf("enter number of elements\n");

scanf("%d",&n);

printf("Enter elements of the array\n");

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

quicksort(arr,0,n-1);

printf("The sorted array is \n");

for(i=0;i<n;i++)

{

printf("%d\t",arr[i]);

}

getch();

}

void quicksort(int arr[],int low, int up)

{

int pivloc;

if(low>=up)

return;

pivloc=partition(arr,low,up);

quicksort(arr,low,pivloc-1);

quicksort(arr,pivloc+1,up);

}

int partition(int arr[],int low, int up)
```

```
{
int temp,i,j,pivot;
i=low+1;
j=up;
pivot=arr[low];
while(i<=j)
{
while((arr[i]<pivot) && (i<up))
i++;
while(arr[j]>pivot)
j–;
if(i<j)
{
temp=arr[i];
arr[i]=arr[j];
arr[j]=temp;
i++;
j–;
}
else
i++;
}
arr[low]=arr[j];
arr[j]=pivot;
return j;
}
```

Example of quick sort:-—

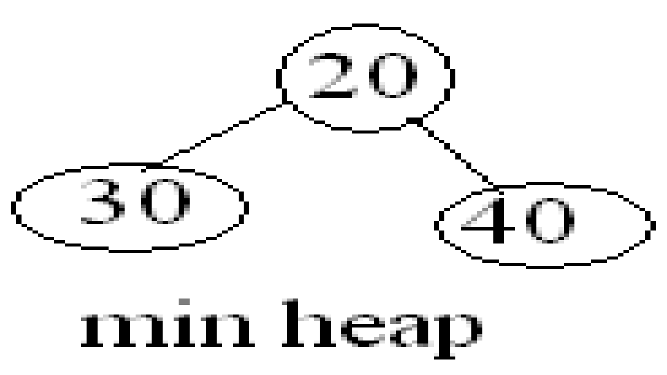
HEAP SORT

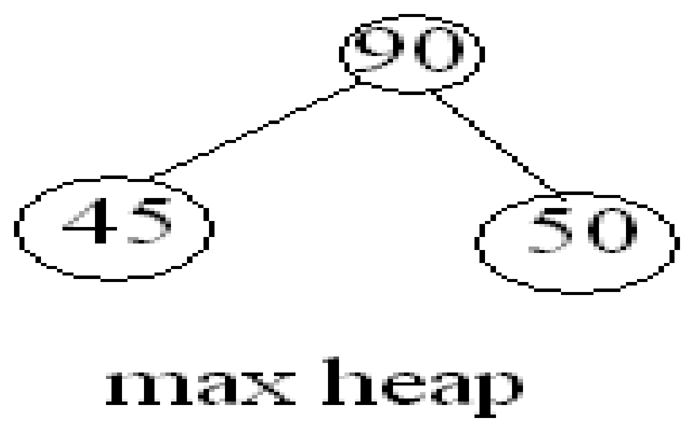
In this method an array is to be sorted as a binary tree in a sequential representation.

Before proceed for heap sort 1st of all we have to create a heap.

A heap is an complete binary tree with n nodes , it is of two types :-

1. Max heap or descending heap or Top down
2. Min heap or ascending heap or Bottom up
3. **Max heap**:- **if the parent node is greater than equal to its child node**.
4. A[parent[i]] > = A[i]
5. **Min heap**:- **if the parent node is less than equal to its child node.**
6. A[parent[i]] <= A[i]





ALGORITHM FOR HEAP SORT(RECURSIVE ALGORITHM)

Here Build Heap is a procedure to build a heap tree *and* Heapify is a procedure to adjust the tree to make it heap tree.

A- array , n- for max elements, B- another array

Heap sort(A)

1. Build Heap (A)   [constructing the heap tree]
2. n= length (A) , heap-size = length (*A*)   *[initialize variables]*
3. for i= 0 to n   by i = i + 1   [Root node to n node]
4. B[n] =A[i]   [store the last element into an array]
5. Do exchange A[n] and A[i]   [Root node with last node of the tree]
6. heap-size = heap-size -1
7. n=n-1

8. Heapify(A,n)

Write a program on Heap sort using C language .

```
#include<stdio.h>

#include<conio.h>

int Left(int);

int Right(int);

void Heapify(int[],int,int);

void BuildHeap(int[],int);

void Heapsort(int [],int);

void main()

{

int arr[20],n,i;

clrscr();

printf("enter the number of elements to be sorted\n");

scanf("%d",&n);

printf("Enter the elements\n");

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

Heapsort(arr,n);

printf("\n The sorted array is \n");

for(i=0;i<n;i++)

{

printf("%d\t",arr[i]);

}

getch();
```

```
}


int Left(int i)

{

return (2*i+1);

}

int Right(int i)

{

return (2*i+2);

}

void Heapify(int arr[],int i, int n)

{

int l,r,large,temp;

l=Left(i);

r=Right(i);

if((l<=n-1) && (arr[l]>arr[i]))

large=l;

else

large=i;

if((r<=n-1) && (arr[r]>arr[large]))

large=r;

if(large!=i)

{

temp=arr[i];

arr[i]=arr[large];

arr[large]=temp;

Heapify(arr,large,n);
}

}

void BuildHeap(int arr[],int n)

{

int i;

for(i=(n-1)/2;i>=0;i–)

{

Heapify(arr,i,n);

}

}

void Heapsort(int arr[],int n)

{

int i,m,temp;

BuildHeap(arr,n);

m=n;

for(i=n-1;i>=1;i–)

{

temp=arr[0];

arr[0]=arr[i];

arr[i]=temp;

m=m-1;

Heapify(arr,0,m);

}

}
```

RADIX SORT

Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.[1] Radix sorting algorithms came into common use as a way to sort punched cards as early as 1923.

Radix sort is one of the **sorting algorithms** used to **sort** a list of integer numbers in order. In **radix sort** algorithm, a list of integer numbers will be **sorted** based on the digits of individual numbers.

Algorithm for Radix sort

Let **A** be a linear array of **n** elements **A[1], A[2], A[3]............A[n]**. Digit is the total number of digit in the largest element in array **A**.

1. Input n number of elements in an array A.
2. Find the total number of digits in the largest element in the array.
3. Initialize i=1 and repeat the steps 4 and 5 until (i<=Digit).
4. Initialize the bucket j=0 and repeat the steps 5until (j<n).
5. Compare the ith position of each element of the array with bucket number and place it in the corresponding bucket.
6. Read the elements (S) of the bucket from 0th bucket to 9th bucket and from the first position to the higher one to generate new array A.
7. Display the sorted array A.
8. Exit.

Write a program on C using Radix sort.

// Radix Sort in C Programming

```
#include <stdio.h>
#include<conio.h>
// Function to get the largest element from an array
int getMax(int array[], int n) {
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}
// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place) {
  int output[size + 1];
  int max = (array[0] / place) % 10;

  for (int i = 1; i < size; i++) {
    if (((array[i] / place) % 10) > max)
    max = array[i];
  }
  int count[max + 1];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  // Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;
      // Calculate cummulative count
  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i–) {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]–;
  }
  for (int i = 0; i < size; i++)
    array[i] = output[i];
}
// Main function to implement radix sort
void radixsort(int array[], int size) {
  // Get maximum element
  int max = getMax(array, size);
  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}
// Print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
// Driver code
int main() {
  int array[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n);
  printArray(array, n);
}
```

****