# Unit 13 Searching and Sorting

**Structure:**



## 13.1 Introduction

In computer science, a **search algorithm**, broadly speaking, is an algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions. Most of the algorithms studied by computer scientists that solve problems are kinds of search algorithms. The set of all possible solutions to a problem is called the search space. Brute-force search or "naïve"/uninformed search algorithms use the

simplest method of searching through the search space, whereas informed search algorithms use heuristic functions to apply knowledge about the structure of the search space to try to reduce the amount of time spent in searching.

Searching methods are designed to take advantage of the file organization and optimize the search for a particular record or to establish its absence. The file organization and searching method chosen can make a substantial difference to an application's performance.

**Objectives:**

At the end of this unit, you will be able to explain:

- Basics Searching Techniques.
- Sequential Search [Linear search]
- Binary Search
- Various Sorting Methods

## 13.2 Basic Searching Techniques

Consider a list of **n** elements or can represent a file of **n records**, where each element is a key / number. The task is to find a particular key in the list in the shortest possible time.

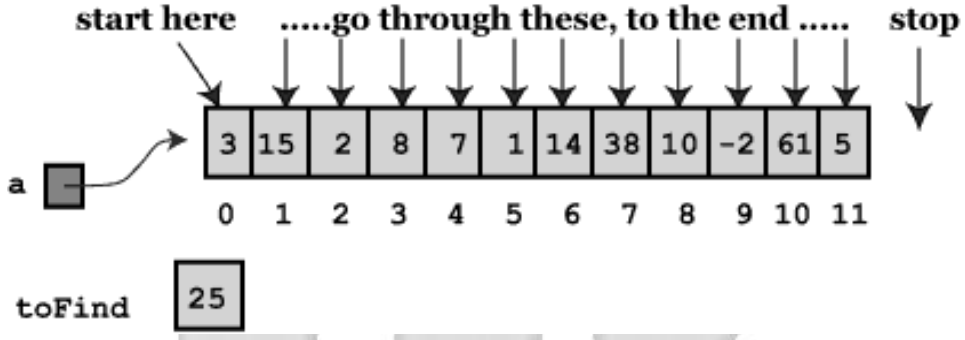
If you know you are going to search for an item in a set, you will need to think carefully about what type of data structure you will use for that set. At low level, the only searches that get mentioned are for sorted and unsorted arrays. However, these are not the only data types that are useful for searching.

- **Linear search:** Start at the beginning of the list and check every element of the list. Very slow [order O(n)] but works on an unsorted list.
- **Binary Search:** This is used for searching in a sorted array. Test the middle element of the array. If it is too big. Repeat the process in the left

half of the array, and the right half if it's too small. In this way, the amount of space that needs to be searched is halved every time, so the time is **O(log n).**

### 13.2.1 Sequential Search [Linear search]

This is the most natural searching method. Simply put it means to go through a list or a file till the required record is found or end of the list which comes earlier. It makes no demands on the ordering of records. The algorithm for a sequential search procedure is now presented.




**Linear Search (Algorithm)**

LINEAR(DATA, N, ITEM, LOC)

Step –I : Set DATA[N+1] = ITEM

Step – II: LOC:=1

Step – III: While DATA[LOC] ≠ ITEM

Set LOC :=LOC +1

[End of Loop]

Step –IV: IF LOC:=N+1 THEN Set LOC:=0

Step –V : Exit

[ DATA is a linear array, with N elements and ITEM is a given search key. LOC is the location of ITEM in DATA. If unsuccessful Search LOC =0]

**Linear Search (Analysis)**

Whether the sequential search is carried out on lists implemented as arrays or linked lists or on files, the criteria part in performance is the comparison loop step 2. Obviously the fewer the number of comparisons, the sooner the algorithm will terminate.

The fewest possible comparisons = 1. When the required item is the first item in the list. The maximum comparisons = N when the required item is the last item in the list. Thus if the required item is in position I in the list, I comparisons are required.

- Hence the average number of comparisons done by sequential search is

  $= \frac{1+2+3.....+N}{N}$

  $= \frac{N(N+1)}{2*N}$

  $= (N + 1)/2$

- Very slow [order O(n) ] but works on an unsorted list

**Example: Write a program to demonstrate Linear Search**

**[File Name: u13q1.c]**

```
/* LINEAR SEARCH */
/* U13Q1.c */
# include<stdio.h>
int search;
int flag;
int         input( int *, int , int);
int list[200];
void linear_search(int *, int , int );
void display(int *, int);
/* Definition of function */
void  linear_search(int l[], int n, int element)
```

```
{
        int k;
        flag = 1;
        for(k = 0; k< n; k++)
        {
                if(l[k] == element)
                {
                printf("\n Search is successful \n");
                printf("\n Element: %i Found at Location: %i", element,
                        k+1);
                flag = 0 ;
                }
        }
        if (flag)
           printf("\n Search is unsuccessful");
 }
void display(int list[], int n)
{
        int i;
        for(i = 0 ; i < n ; i++)
        {
                printf(" %d", list[i]);
        }
}
input(int list[], int number, int key)
{
        int i;
        key = 30;
        printf("Input the number of elements in the list:");
```

```
        number = 20;
        for(i = 0 ; i < 20; i++)
        {
                list[i] = rand() %100;
        }
        printf("\n Element to be searched: %d", key);
        search = key;
        return number;
}
void main()
{
        int number, key, list[200];
        number = input( list, number, key);
        key = search ;
        printf("\n Entered list as follows:\n");
        display(list,number);
        linear_search(list, number, key);
        printf("\n In the following list\n");
        display(list,number);
}
```

### 13.2.2 Binary Search

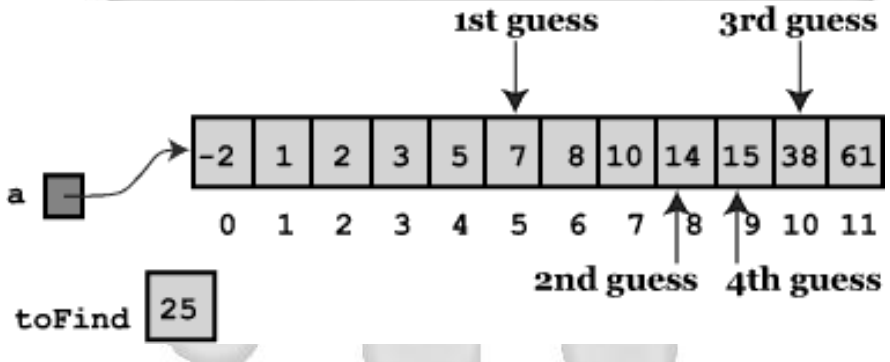
The drawbacks of sequential search can be eliminated if it becomes possible to eliminate large portions of the list from consideration in subsequent iterations. In binary search method, it halves the size of the list in subsequent iterations during the search process.

Binary search can be explained simply by the analogy of searching for a page in a book. Suppose you are searching for page 90 in book of 150 pages. You would first open it at random towards the latter half of the book. If the page is less than 90, you would open at a page to the right, if it is

greater than 90 you would open at a page to the left, repeating the process till page 90 is found. As you can see, by the first instinctive search, you dramatically reduced the number of pages to search.

Binary search requires sorted data to operate on since the data may not be contiguous like the pages of a book. We cannot guess which quarter of the data the required item may be in. So we divide the list in the centre each time.

The most efficient method of searching a sequential table without the use of auxiliary indices or tables is the binary search. Basically, the argument is compared with the key of the middle element of the table. If they are equal, the search ends successfully; otherwise, either the upper or lower half of the table must be searched in a similar manner.




**Binary Search (Algorithm)**

BINARY_SEARCH(DATA,LB,UB,ITEM,LOC)

DATA → It is the sorted array, LB → Lower bound of array, UB→ Upper bound of array

ITEM → Search Key, LOC → Location of the search key if successful search otherwise it is assigned to NULL

Step-I : Set BEG:=LB, END:=UB, MID=((BEG+END)/2)

Step-II: Repeat Step III and Step IV while BEG ≤ END and DATA[MID] ≠ ITEM

Step-III: If ITEM<DATA[MID] then

Set END:=END-1

Else

Set BEG:=MID+1

[End of if structure]

Step –IV: Set MID:=INT((BEG+END)/2)

[End of Step –II Loop]

Step-V: If DATA[MID] =ITEM then

Set LOC:=MID

Else

Set LOC:=NULL

[End of If structure]

Step-VI: Exit

**Binary Search (Analysis)**

In general, the binary search method needs no more than $[\log_2 n] + 1$ comparison. This implies that for an array of a million entries, only about twenty comparisons will be needed. Contrast this with the case of sequential search which on the average will need $\frac{(n+1)}{2}$ comparisons.

The conditions (MAX = MIN) is necessary to ensure that step 2 terminates even in the case that the required element is not present. Consider the example of Zodiac signs. Suppose the l0th item was Solar (an imaginary Zodiac sign). Then at that point we would have

MID = 10

MAX =11

MIN = 9

And from 2.2 get

MAX = MID-l = 9

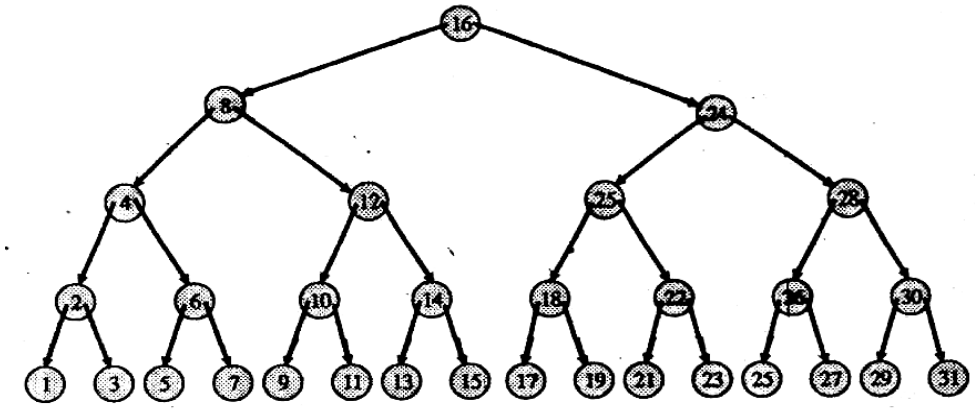
In the next iteration we get

(2.1) MID = (9 + 9) DIV 2 = 9

(2.2) MAX= 9-1 = 8.

Since MAX < MIN, the loop terminates. Since FOUND is false, we consider the target was not found.

In general, the binary search method needs no; more than **$[\log_2 n]+1$** comparisons. This implies that for an array of a million entries, only about twenty comparisons will be needed. Contrast this with the case of sequential search which on the average will need (n+1)/2 comparisons. The Binary Tree with 31 nodes is given below.




**Searching Process in Binary Search**

**Example: Write a program to illustrate the binary search process. [File Name: u13q2.c]**

```
#include <stdio.h>
int binary_search(int array[], int value, int size)
{
        int found = 0;
        int high = size, low = 0, mid;
        mid = (high + low) / 2;
```

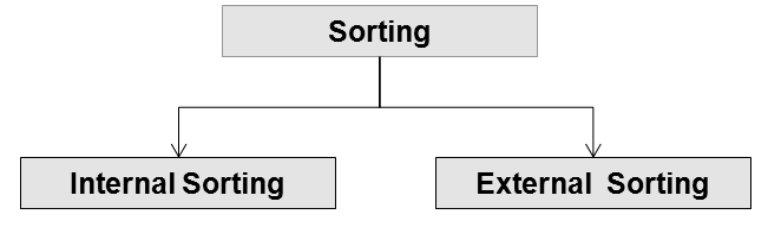
```
        printf("\n\nLooking for %d\n", value);
        while ((! found) && (high >= low))
        {
                printf("Low %d Mid %d High %d\n", low, mid, high);
                if (value == array[mid])
                        found = 1;
                else if (value < array[mid])
                        high = mid - 1;
                else
                        low = mid + 1;
                mid = (high + low) / 2;
        }
        return((found) ? mid: -1);
}
void main(void)
{
        int array[100], i;
        for (i = 0; i < 100; i++)
                array[i] = i+20;
    printf("Result of search %d\n", binary_search(array, 33, 100));
    printf("Result of search %d\n", binary_search(array, -75, 100));
    printf("Result of search %d\n", binary_search(array, 1, 100));
    printf("Result of search %d\n", binary_search(array, 1001, 100));
}
```

## 13.3 Overview of Sorting Methods

In computer science and mathematics, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to

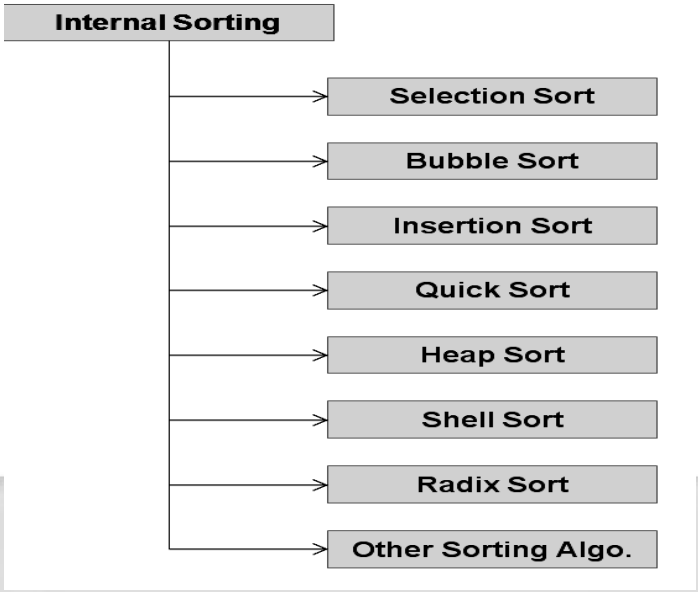
optimize the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement.




## 13.4 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 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. There are a number of different types of Internal Sorting and some of the important methods are:



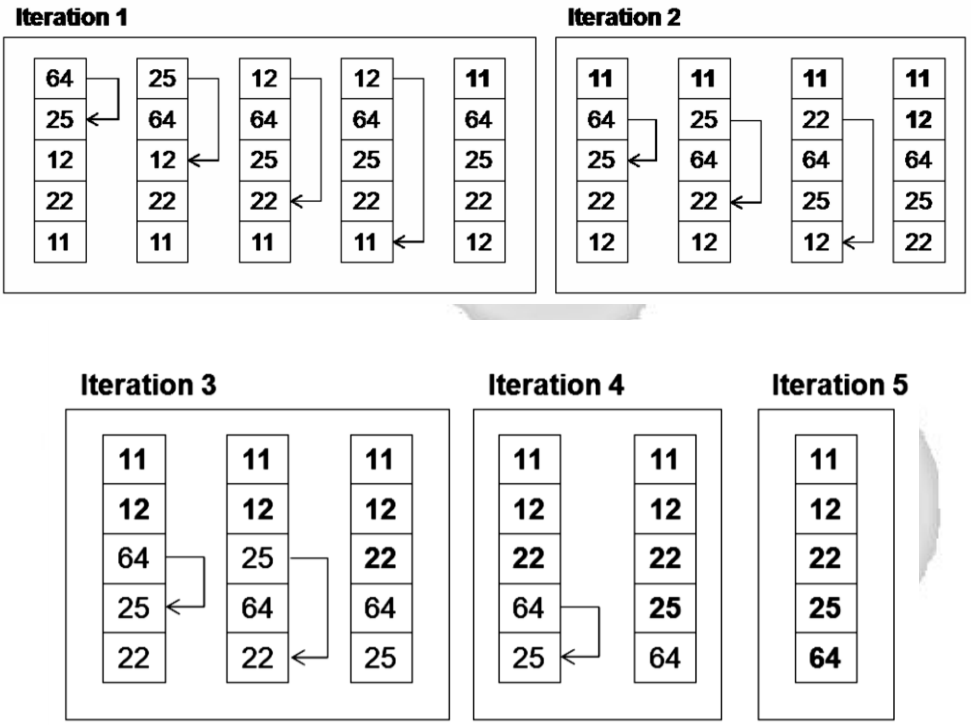

### 13.4.1 Selection Sort

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O($n^2$) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

**The algorithm works as follows:**

- Find the minimum value in the list.
- Swap it with the value in the first position.
- Repeat the steps above for remainder of the list (starting at the second position).

### Iterations of Selection Sort




### Algorithm

It is the sorting algorithm for order of values in ascending or descending order of values.

There are **three** basic steps

1. Find minimum value
2. Swap with the first elements of array
3. Increase first elements pointer by one

| **selection_sort(list,N)** | |
|---|---|
| Step I | Repeat step for ptr=1,2,…, upto N |
| Step II | **min_find** (list, st-ptr, N, LOC) |
| Step III | Temp=list[ptr]<br>List[ptr]=list[LOC]<br>List[LOC]=temp |
| Step IV | [End of Loop] |

| **min_find(list, st-ptr, N, LOC)** | |
|---|---|
| Step I | Min=list[st-ptr]<br>LOC=st-ptr |
| Step II | If min>list[st-ptr]<br>Min=list[st-ptr]<br>LOC=str-ptr<br>[End of If] |
| Step III | str-ptr=st-ptr+1 |
| Step IV | If st-ptr>N then<br>Return<br>else<br>goto Step-II |

**Example:** Write a program to sort elements using selection sort algorithm.

[File Name: u13q3.c]

```
/*selection sort*/
#include <stdlib.h>
#define ARRAY_SIZE 10
void main()
```

```
{
int list[ARRAY_SIZE],i,temp,j,loc=0;
/**** accepting random values */
for (i=0;i<ARRAY_SIZE;i++)
 list[i]=random(100);          /* accept randome no 1-100 */
/*********************************/
clrscr();
printf("\n\n\n Sorting - > Selection  Sort Algorithm\n\n");
printf("\tThe List of Numbers Before Sorting ...\n\n");
for(i=0;i<ARRAY_SIZE;i++)
printf("%5d",list[i]);
getch();
/******************SORTING STARTED **************/
for(i=0;i<ARRAY_SIZE-1;i++)
{
                loc=min_loc_search(list,i);
                temp=list[loc];
                list[loc]=list[i];
                list[i]=temp;
}
/******************SORTED COMPLETED ***************/
printf("\n\n    The List of Numbers After Sorting (SELECTION Sort
Algo.)\n\n");
for(i=0;i<ARRAY_SIZE;i++)
 printf("%5d",list[i]);      /* sorted list */
/*****************************************/
getch();
}
int min_loc_search(int list[],int i)
```

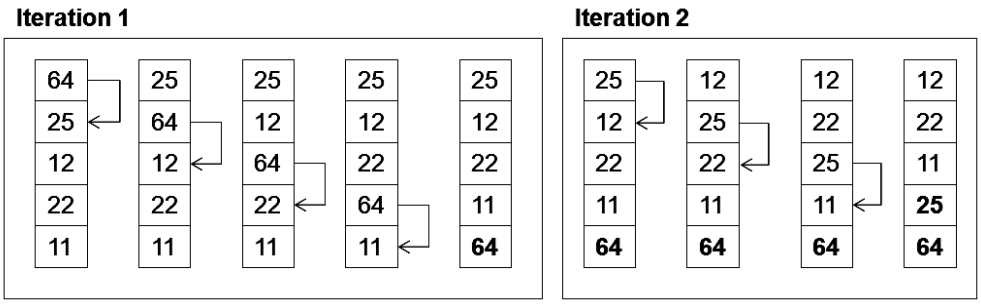
```
{
int j,min,loc;min=list[i];
loc=i;
for(j=i;j<ARRAY_SIZE;j++)
 if(list[j]<min)
        {
                min=list[j]; loc=j;
        }
return(loc);
}
```

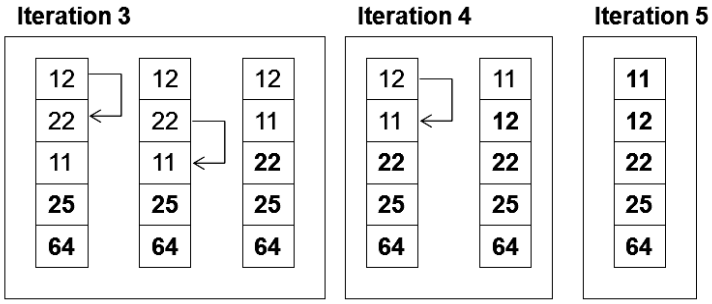
### 13.4.2 Bubble Sort

**Bubble sort** is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time 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.

- Bubble sort has worst-case complexity ***O(n²)***, where *n* is the number of items being sorted.
- There exist many other sorting algorithms with substantially better worst-case complexity *O(n log n)*, meaning that bubble sort should not be used when *n* is large.

**Iterations**






**Algorithm**

In a bubble sort, we try to improve on the performance by making more exchanges in each pass. Again, we will make several passes over the array (n -1 to be exact). For the first pass, we begin at element O and proceed forward to element n-1. Along the way we compare each element to the element that follows it. If the current element is greater than that in the next location, then they are in the wrong order, and we swap them. In this way, an element early in the array that has a very large value is able to percolate (bubble) upward. In the next pass we do exactly the same thing. But now the last element in the array is guaranteed to be where it belongs. So our pass needs only proceed as far as element n -2 of the array. This will move the second largest value in the array into the next to last position. This proceeds with each pass encompassing one less element of the array. The result, aside from a sorted array, is that we make n passes over n elements. This sort method, too, is $O(n^2)$.

**Algorithm:**

INPUT: LIST [] of N items in random order

O UTPUT: LIST [] of N items sorted in ascending order.

1. SWAP = TRUE
   PASS = 0/
2. WHILE SWAP = TRUE DO
   BEGIN.

```
2.1     FOR I = 0 TO (N-PASS) DO
          BEGIN
2.1.1   IFA[I] >A [I+1]
        BEGIN
        TMP = A[I]
        A[I] = A[I + 1]
        A[I + 1] = TMP
        SWAP = TRUE
        END
        ELSE
        SWAP = FALSE
2.1.2   PASS = PASS + 1
        END
        END
```

Total number of comparisons in Bubble sort are

$$= (N-1) + (N-2) \ldots + 2 + 1$$

$$= \frac{(N-1)*N}{2} = O(N^2)$$

This inefficiency is due to the fact that an item moves only to the next position in each pass.

**Examples: Write program to sort the elements using bubble sort algorithm. [File Name:u13q4.c]**

```
/********** Program for bubble sort of random 20 numbers ********/
#include <stdio.h>
#include <stdlib.h>
void bubble_sort(int array[], int size)  /* FUNCTION DECLARATION */
{
 int i,j,temp;
```

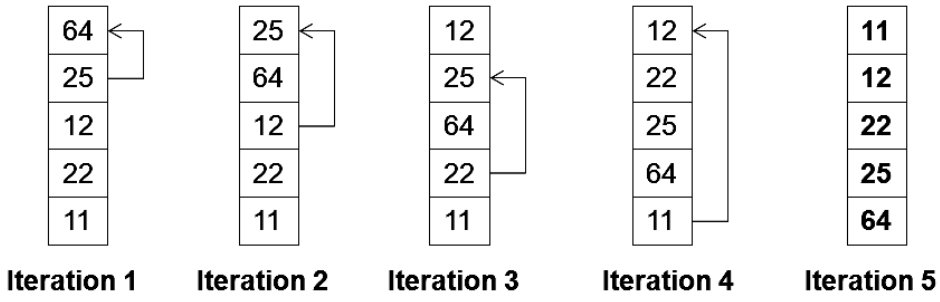
```
 for (i=0;i<size-1;i++)
 {
   for(j=0;j<size-i-1;j++)
    {
      if(array[j]<array[j+1])
      {
           temp=array[j];
           array[j]=array[j+1];
           array[j+1]=temp;
      }
    }
 }
}
/*************************************************************/
  void main()
  {
           int values[20],i;
           clrscr();
           printf(" ******** BUBBLE SORT ********\n");
           printf(" Unsorted list is as follows \n");
           for(i=0;i<20;i++)          /* Generating Random Numbers */
           {
              values[i] = rand() % 100;
              printf("%4d", values[i]);
           }
           bubble_sort(values, 20);   /* Calling the function      */
           printf("\n Sorted list is as follows \n");
           for(i=0;i<20;i++)
             printf("%4d",values[i]);
```

```
   getch();
 }
/*************************************************************/
```

### 13.4.3 Insertion Sort

**Insertion sort** is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built by one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort or merge sort, but it has various advantages:

- Efficient on (quite) small data sets.
- Efficient on data sets which are already substantially sorted: it runs in **O(*n* + *d*)** time, where d is the number of inversions.
- More efficient in practice than most other simple O($n^2$) algorithms such as selection sort or bubble sort: the average time is $n^2/4$ and it is linear in the best case.
- **Stable** (does not change the relative order of elements with equal keys)
- **In-place** (only requires a constant amount O(1) of extra memory space)
- **It is an online algorithm**, in that it can sort a list as it receives it.




**Algorithm**

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one, down the list. Insert the target in the vacated slot.

We now present the algorithm for insertion sort.

| **Insert(A,N)** | |
|---|---|
| Step – I | Set A[]:= -α [ The number which is minimum in list] |
| Step – II | Repeat Step III to Step - V For K=1,2,3,…,N |
| Step – III | Set TEMP:=A[K] and PTR := K-1 |
| Step – IV | Repeat while TEMP < A[PTR]<br>Set A[PTR+1]:=A[PTE]<br>PTR:=PTR -1<br>[End of Step –IV Loop] |
| Step – V | Set A[PTR+1]:= TEMP<br>[End of Step – II Loop] |
| Step – VI | Return |

**Example:** Write a program to sort elements using insertion sort algorithm [File Name:u13q5.c]

```
/* INSERTION SORT */
# include<stdio.h>
# include<stdlib.h>
int key;
int  insertion_sort(int *);
// prototype
void display(int *, int);
/* Definition of the insertion sort function */
int insertion_sort(int list[])
{
        int i = 0;
        int j, k ;
        printf("\nElement to be inserted Break condition is -0 : ");
        scanf("%d", &key);
        printf("\n Selected Element is: %d", key);
```

```
        while(key != -0 ) /* -0 is break condition */
        {
                k = i - 1;
                while((key < list[k]) && (k >= 0))
                {
                        list[k+1] = list[k];
                        --k;
                }
                list[k+1] = key ;
                printf("\n List after inserting an element ");
                for(j = 0 ; j<=i; j++)
                        printf(" %d", list[j]);
                printf("\n Element to be inserted Break condition is -0: ");
                scanf("%d", &key);
                printf("\n Selected Element is: %d", key);
                i ++;
        }
        return i;
}
/* End of the insertion sort function */
/* Definition of the function */
void display(int list[], int n)
{
        int j;
        printf("\n Final sorted list is as follows:\n");
        for(j = 0 ; j < n ; j++)
                printf(" %d", list[j]);
}
/* End of the display function */
```

```
void main()
{
        int list[200];
        int n ;
        n = insertion_sort(list);
        display(list,n);
}
```

### 13.4.4 Quick Sort

Quick sort doesn't look at all like any of the sorts we have examined up until now. It is a partition sort. It is one of the speediest of these 'in place' array sorts. We say that it is faster than its siblings, but in fact, it is also an $O(n^2)$ algorithm. It gets its reputation for speed from the fact that it is $O(n^2)$. Only in the worst case, this almost never occurs. In practice quick sort is an O(n Log n) algorithm.

Quick sort begins by picking an element to be the **pivot** value. There is much debate, about how to best pick this initial element. In terms of understanding how quick sort works, it really doesn't matter. They can just pick the first element in the array. With the pivot to work with, the array is divided into three parts: all values less than the pivot, the pivot and values greater than the pivot. When this is finished, the pivot value is in its proper position in the sorted array. Quick sort, then, recursively applies this same process to the first partition, containing low values, and to the second position, which contains high values. Each time at least one element (the pivot) finds its final resting place and the partitions get smaller. Eventually the partitions are of size one and the recursion ends.

The iteration process is given below:

| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|---|---|---|---|---|---|---|---|---|

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array

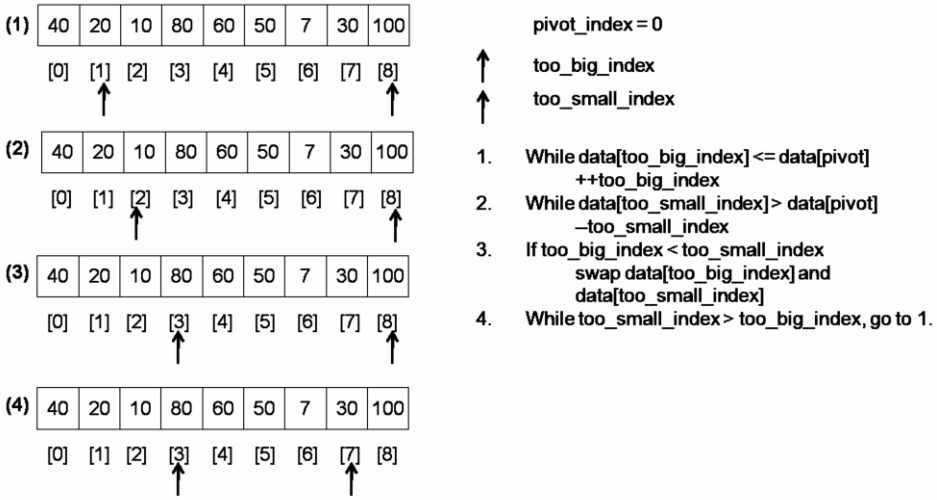
Given a pivot, partition the elements of the array such that the

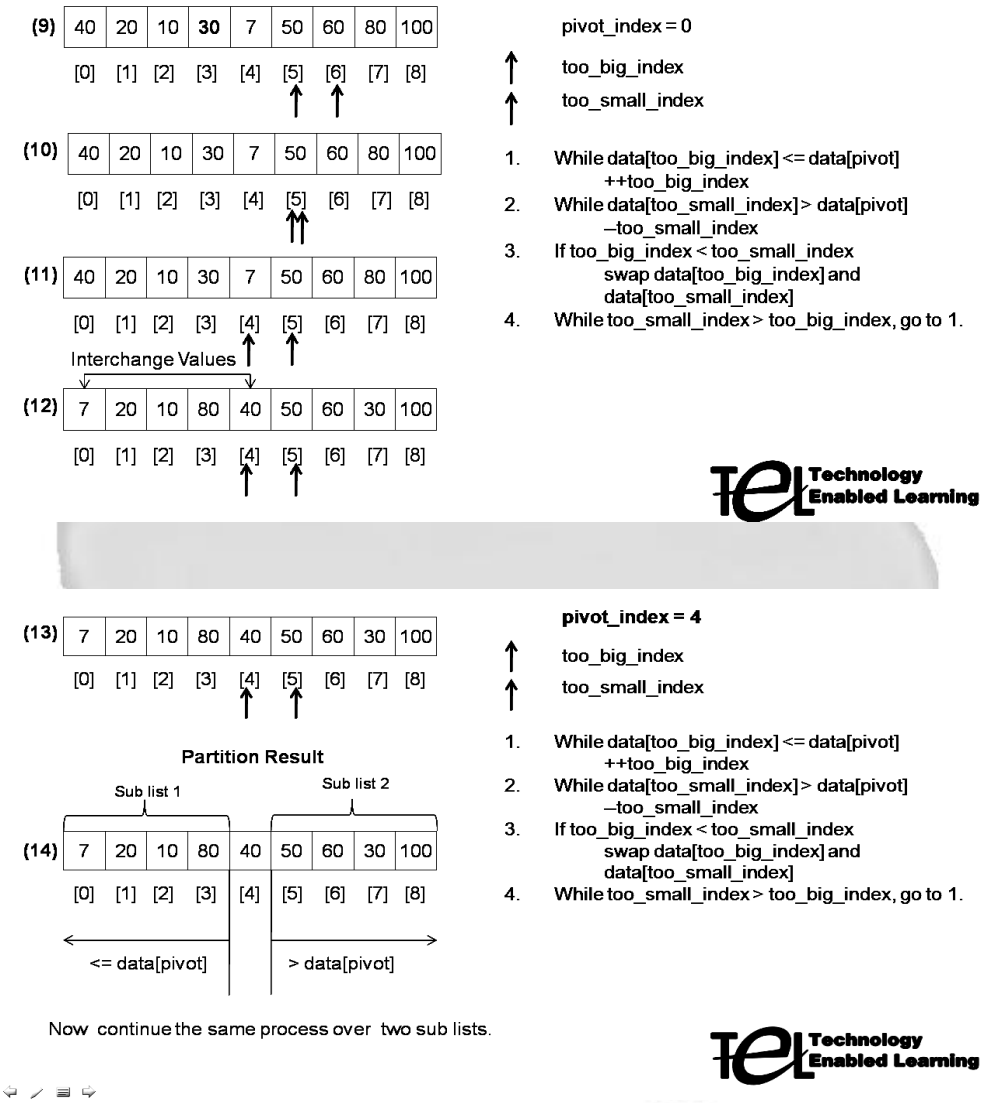
resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.






## Algorithm

Here A is an array with N elements. Parameters BEG and END contains the boundary values of the sub list of A to which this procedure applies. LOC keeps the track of the position of the first elements A[BEG] of the sub list during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

The task has been done in two Steps which is given below:

| | |
|---|---|
| Step A | QUICK(A,N,BEG,END,LOC) |
| Step – I | [Initialized]<br>Set LEFT:=BEG, RIGHT:=END and LOC:=BEG |
| Step –II | [Scan from right to left]<br>Repeat while A[LOC]≤A[RIGHT] and LOC≠RIGHT<br>RIGHT = RIGHT -1<br>[End of Loop]<br>If LOC = RIGHT then return<br>If A[LOC] > A[RIGHT] then<br>[Interchange A[LOC] and A[RIGHT]]<br>TEMP:=A[LOC],A[LOC]:=A[RIGHT]<br>A[RIGHT]:=TEMP<br>Set LOC:=RIGHT<br>Go to Step – III<br>[End of If Structure] |
| Step – III | [Scan from Left to Right]<br>Repeat while A[LEFT] ≤ A[LOC] and LEFT≠LOC<br>LEFT:=LEFT+1<br>[End of Loop]<br>If LOC = LEFT then Return<br>IF A[LEFT]>A[LOC] then<br>[Interchange A[LEFT] and A[LOC]]<br>TEMP:=A[LOC],A[LOC]:=TEMP<br>A[LEFT]:=TEMP<br>Set LOC=LEFT<br>Go to Step – II<br>[End of If Structure] |
| Step B | This algorithm sorts are array A with N elements |
| Step – I | [Initialize]<br>TOP:=NULL |
| Step –II | If N>1 then<br>TOP:=TOP+1,LOWER[1]:=1,UPPER[1]:=N |

| | |
|---|---|
| | [End of If Structure] |
| Step – III | Repeat Steps –IV to VIII which TOP ≠ NULL |
| Step – IV | [POP sub list from Stack]<br>Set BEG:=LOWER[TOP]<br>END := UPPER[TOP]<br>TOP:=TOP -1 |
| Step – V | Call QUICK (A,N,BEG,END,LOC) |
| Step –VI | If BEG<(LOC-1) then<br>TOP:=TOP+1, LOWER[TOP]:=BEG,<br>UPPER[TOP]=LOC-1<br>[End of If Structure]<br>[End of Step III Loop] |
| Step – VIII | Exit |

**Example:** Write a program to sort elements using quick sort algorithm. [File Name :u13q6.c]

```
/* quick.c */
#include <stdio.h>
#include <stdlib.h>
void quick_sort(int array[], int first, int last)
{
    int temp, low, high, list_separator;
    low = first;
    high = last;
    list_separator = array[(first + last) / 2];
    do {
        while (array[low] < list_separator)
          low++;
        while (array[high] > list_separator)
          high--;
        if (low <= high)
        {
```

```
                temp = array[low];
                array[low++] = array[high];
                array[high--] = temp;
            }
    } while (low <= high);
    if (first < high)
            quick_sort(array, first, high);
    if (low < last)
            quick_sort(array, low, last);
}
void main(void)
{
    int values[100], i;
    printf("\n Unsorted list is as follows \n");
    for (i = 0; i < 20; i++)
    {
            values[i] = rand() % 100;
            printf(" %d", rand() %100);
    }
    quick_sort(values, 0, 19);
    printf("\n Sorted list as follows\n");
    for (i = 0; i < 20; i++)
            printf("%d ", values[i]);
}
```

### 13.4.5 Heap Sort

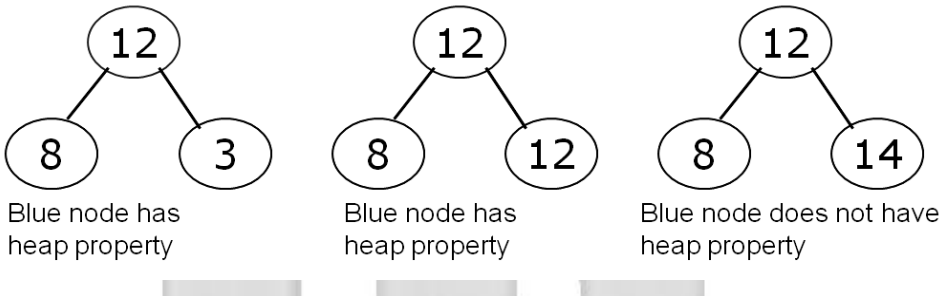
Heap sort (method) is a comparison-based sorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of quick sort, it has the advantage of

a worst-case O(*n* log *n*) runtime. Heap sort is an in-place algorithm, but is not a stable sort.

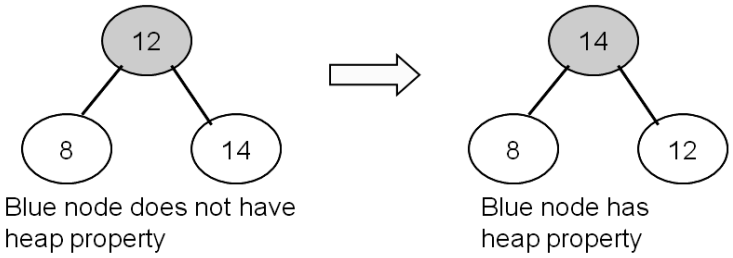
**Plan of attack**

- First, we will learn how to turn a binary tree into a heap.
- Next, we will learn how to turn a binary tree *back* into a heap after it has been changed in a certain way.
- Finally we will see how to use these ideas to sort an *array*.

**The Heap Property**: A node has the heap property if the value in the node is as large as or larger than the values in its children

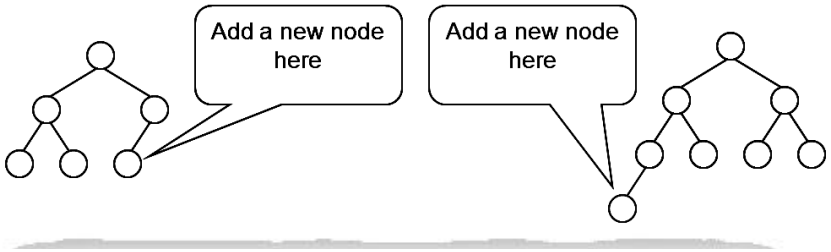



**ShiftUp**: Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child. This is sometimes called **sifting up**.



**Heap Construction Rules (1/2)**:

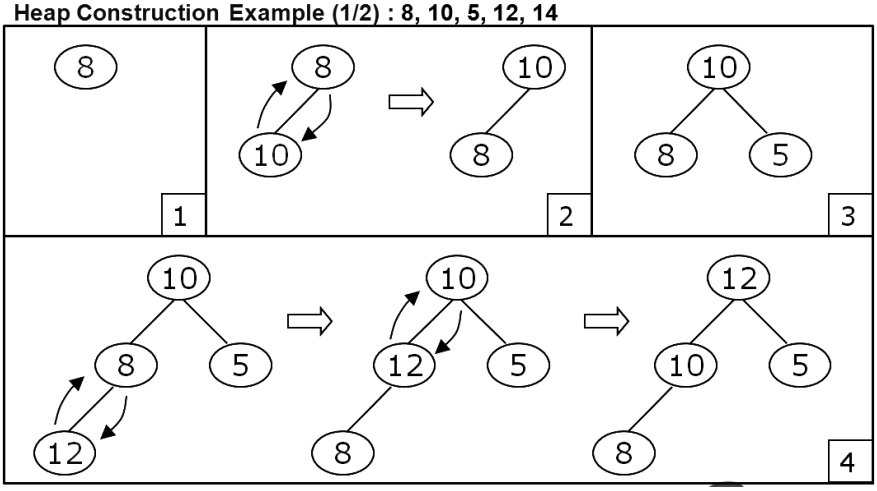
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- **Examples:**



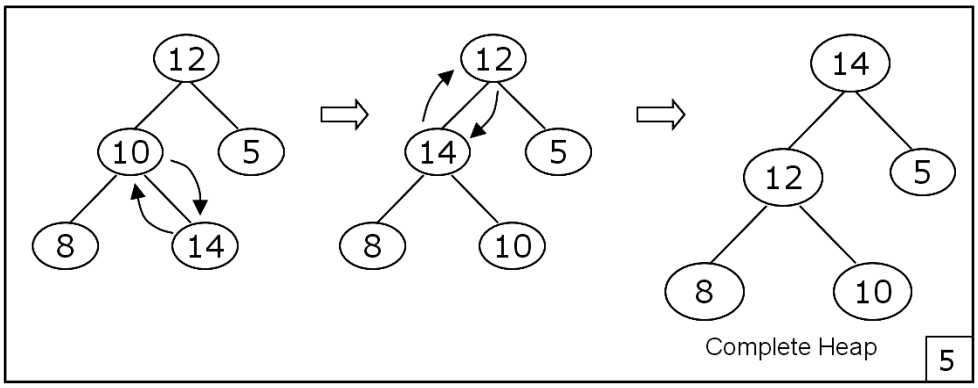

**Heap Construction Rules (2/2)**:

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
- We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
- We reach the root

**Heap Construction Example (1/2) : 8, 10, 5, 12, 14**

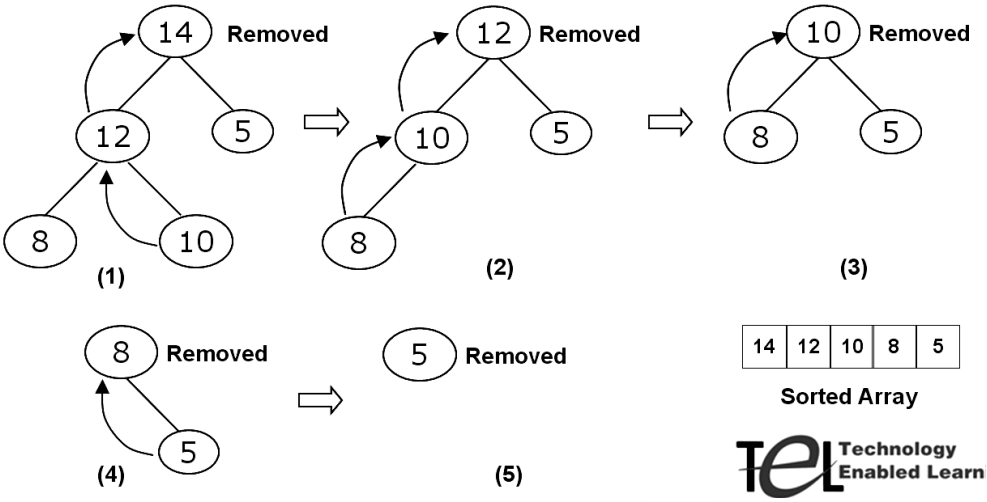


**Heap Construction Example (2/2) : 8, 10, 5, 12, 14**




**Notice that heapified does *not* mean sorted**

**Sorting and Reheapifying (1/1):**




Technology Enabled Learn

**Example:** Write a program to sort the elements using Heap Sort algorithm.

[File Name: u13q7.c]

```
/* HEAP SORT */
# include<stdio.h>
void  heap_sort(int *, int );
void create_heap(int *, int);
```

```
void display(int *, int);
/*      Definition of the function */
void create_heap(int list[], int n )
{
        int k, j, i, temp;
        for(k = 2 ; k <= n;  ++k)
        {
                i = k ;
                temp = list[k];
                j = i / 2 ;
                while((i > 1) && (temp > list[j]))
                {
                  list[i] = list[j];
                  i = j ;
                  j = i / 2 ;
                  if ( j < 1)
                          j = 1 ;
                }
                list[i] = temp ;
        }
}
/* End of heap creation function */
/* Definition of the function */
void heap_sort(int list[], int n)
{
        int k, temp, value, j, i, p;
        int step = 1;
        for(k = n ; k >= 2; –k)
        {
```

```
            temp = list[1] ;
            list[1] = list[k];
            list[k] = temp ;
            i = 1;
            value = list[1];
            j = 2 ;
            if((j+1) < k)
              if(list[j+1] > list[j])
                    j ++;
            while((j <= ( k-1)) && (list[j] > value))
            {
              list[i] = list[j];
              i = j;
              j = 2*i ;
              if((j+1) < k)
                    if(list[j+1] > list[j])
                          j++;
                    else
                          if( j > n)
                                j = n;
              list[i] = value;
            } /* end of while statement */
            printf("\n Step = %d ", step);
            step++;
            for(p = 1; p <= n; p++)
              printf(" %d", list[p]);
      } /* end for loop */
}
/* Display function */
```

```
void display(int list[], int n)
{
        int i;
        for(i = 1 ; i <= n; ++ i)
        {
                printf("  %d", list[i]);
        }
}
/* Function main */
void main()
{
        int list[100];
        int i, size = 13 ;
        printf("\n Size of the list: %d", size);
        for(i = 1 ; i <= size ; ++i)
        {
                list[i] = rand() % 100;
        }
        printf("\n Entered list is as follows:\n");
        display(list, size);
        create_heap(list, size);
        printf("\n Heap\n");
        display(list, size);
        printf("\n\n");
        heap_sort(list,size);
        printf("\n\n Sorted list is as follows :\n\n");
        display(list,size);
}
```

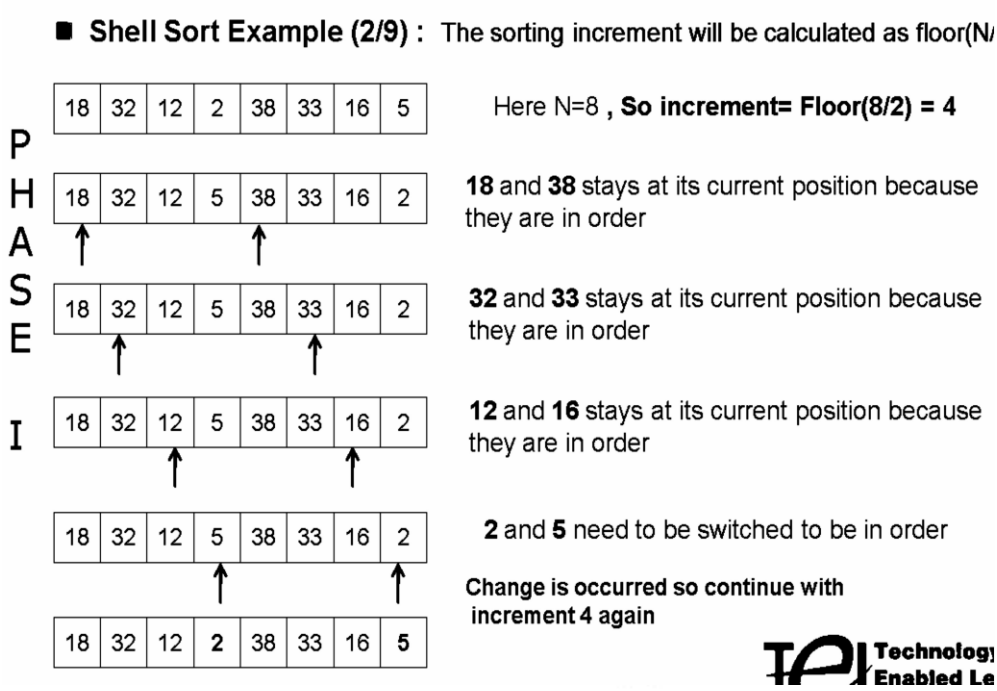
### 13.4.6 Shell Sort

Founded by Donald Shell and named the sorting algorithm after him in 1959. Shell sort is a sorting algorithm that is a generalization of insertion sort, with two observations:

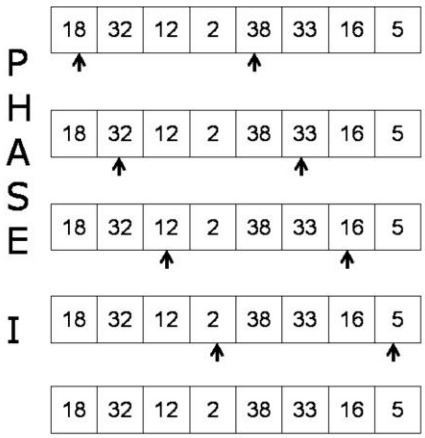
- Insertion sort is efficient if the input is "almost sorted", and
- Insertion sort is typically inefficient because it moves values just one position at a time.

**Best Case:** The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less.

**Worst Case:** The running time of Shell sort depends on the choice of increment sequence. The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect.



■ **Shell Sort Example (3/9) :** **Increment Remains = 4**




**18** and **38** stays at its current position because they are in order

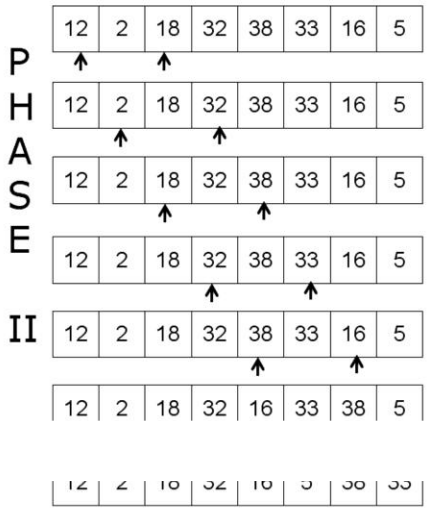
**32** and **33** stays at its current position because they are in order

**12** and **16** stays at its current position because they are in order

**2** and **5** stays at its current position because they are in order

**No Change is occurred**

■ **Shell Sort Example (4/9) :** Now **increment= Floor(4/2) = 2**




**12** and **18** stays at its current position because they are in order

**2** and **32** stays at its current position because they are in order

**18** and **38** stays at its current position because they are in order

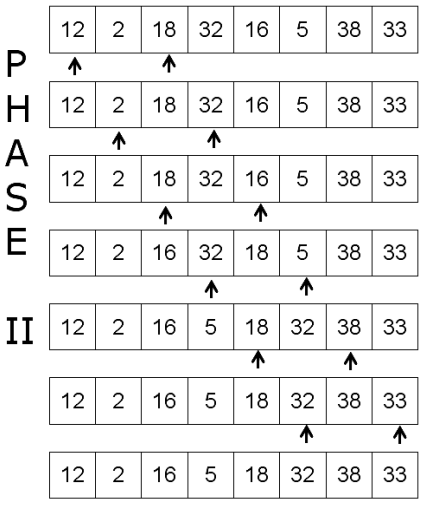
**32** and **33** stays at its current position because they are in order

**38** and **16** need to be switched to be in order

**33** and **5** need to be switched to be in order

**ıange is occurred so continue with increment 2 again**

**Increment Remains = 2**




**12** and **18** stays at its current position because they are in order

**2** and **32** stays at its current position because they are in order

**18** and **16** need to be switched to be in order

**32** and **5** need to be switched to be in order

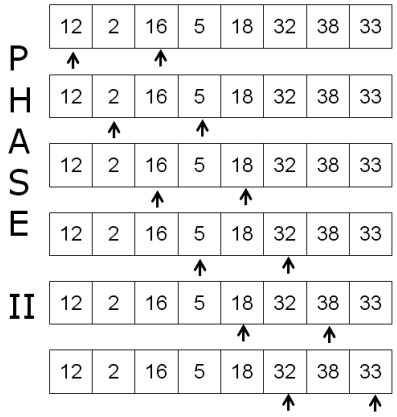
**18** and **38** stays at its current position because they are in order

**32** and **33** stays at its current position because they are in order

**Change is occurred so continue with increment 2 again**

- **Shell Sort Example (6/9) :**

**Increment Remains = 4**




**12** and **16** stays at its current position because they are in order

**2** and **5** stays at its current position because they are in order

**16** and **18** stays at its current position because they are in order

**5** and **32** stays at its current position because they are in order

**18** and **38** stays at its current position because they are in order

**32** and **33** stays at its current position because they are in order

**No Change is occurred**

**Now Phase II Completed**

## Shell Sort Example (7/9) :

Now **increment= Floor(2/2) = 1**

PHASE III

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 12 | 2 | 16 | 5 | 18 | 32 | 38 | 33 | **12** and **2** need to be switched to be in order |
| 2 | 12 | 16 | 5 | 18 | 32 | 38 | 33 | **12** and **16** stays at its current position because they are in order |
| 2 | 12 | 16 | 5 | 18 | 32 | 38 | 33 | **16** and **5** need to be switched to be in order |
| 2 | 12 | 5 | 16 | 18 | 32 | 38 | 33 | **16** and **18** stays at its current position because they are in order |
| 2 | 12 | 5 | 16 | 18 | 32 | 38 | 33 | **18** and **32** stays at its current position because they are in order |
| 2 | 12 | 5 | 16 | 18 | 32 | 38 | 33 | **32** and **33** stays at its current position because they are in order |
| 2 | 12 | 5 | 16 | 18 | 32 | 38 | 33 | **38** and **33** need to be switched to be in order |
| 2 | 12 | 5 | 16 | 18 | 32 | 33 | 38 | **Change is occurred so continue with increment 1 again** |

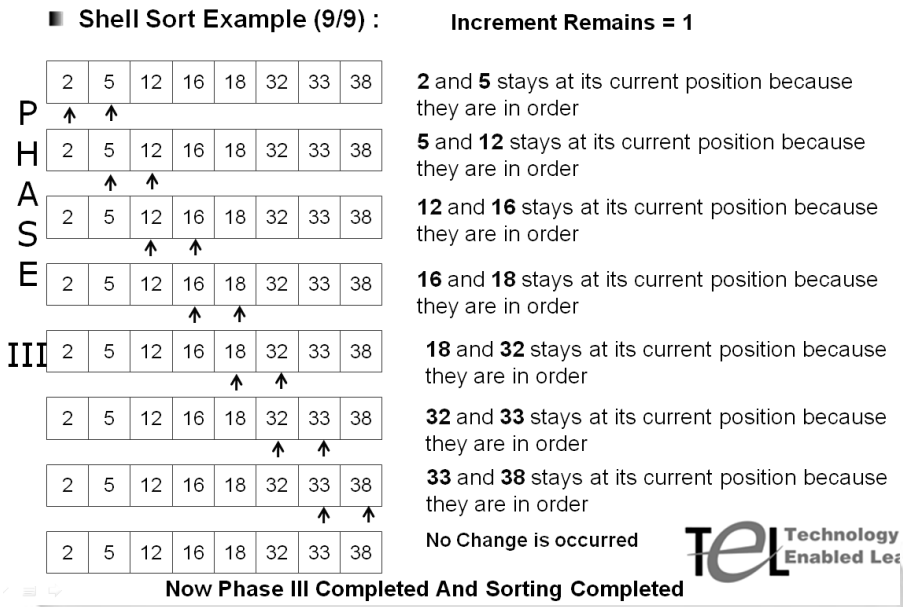


## Shell Sort Example (8/9) :

**Increment Remains = 1**

PHASE III

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 12 | 5 | 16 | 18 | 32 | 33 | 38 | **2** and **12** stays at its current position because they are in order |
| 2 | 12 | 5 | 16 | 18 | 32 | 33 | 38 | **12** and **5** need to be switched to be in order |
| 2 | 5 | 12 | 16 | 18 | 32 | 33 | 38 | **12** and **16** stays at its current position because they are in order |
| 2 | 5 | 12 | 16 | 18 | 32 | 33 | 38 | **16** and **18** stays at its current position because they are in order |
| 2 | 5 | 12 | 16 | 18 | 32 | 33 | 38 | **18** and **32** stays at its current position because they are in order |
| 2 | 5 | 12 | 16 | 18 | 32 | 33 | 38 | **32** and **33** stays at its current position because they are in order |
| 2 | 5 | 12 | 16 | 18 | 32 | 33 | 38 | **33** and **38** stays at its current position because they are in order |
| 2 | 5 | 12 | 16 | 18 | 32 | 33 | 38 | **Change is occurred so continue with increment 1 again** |




**Example:** Write a program to sort elements using shell short algorithm. [File Name: u13q8.c]

```
/********** Program for shell sort of random 20 numbers ********/
#include <stdio.h>
#include <stdlib.h>
void shell_sort(int array[], int size)  /* FUNCTION DECLARATION */
{
   int temp, gap, i, exchange_occurred;
   gap = size / 2;
   do {
  do {
      exchange_occurred = 0;
      for (i = 0; i < size - gap; i++)
       {
         if (array[i] > array[i + gap])
          {
```

```
            temp = array[i];
            array[i] = array[i + gap];
            array[i + gap] = temp;
            exchange_occurred = 1;
          }
        }
    } while (exchange_occurred);
   gap=gap/2;
  } while (gap);
}
/*****************************************************************/
  void main()
  {
  int values[20],i;
  clrscr();
  printf(" ******** SHELL SORT ********\n");
  printf(" Unsorted list is as follows \n");
  for(i=0;i<20;i++)          /* Generating Randam Numbers */
  {
     values[i] = rand() % 100;
     printf("%4d", values[i]);
  }
  shell_sort(values, 20);   /* Calling the function */
  printf("\n Sorted list is as follows \n");
  for(i=0;i<20;i++)
    printf("%4d",values[i]);
   getch();
  }
/*****************************************************************/
```

### 13.4.7 Radix Sort

In computer science, **radix sort** is a sorting algorithm that sorts integers by processing individual digits. Because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers.

- Radix sort is not a comparison sort, therefore it is not subject to the ***O(n log n)*** sorting lower bound
- Each element is represented by *b* bits (i.e. 32 bit integers)
- The algorithm performs a number of iterations; each iteration considers only *r* bits of each element at a time, with the *i*th pass sorting according to the *i*th group of the least significant bits

**Iteration**

| 20 | 30 | 32 | 18 | 06 | 09 | 38 | 49 | 78 | 89 | 67 |
|---|---|---|---|---|---|---|---|---|---|---|

- Number of pass will be the number of digits of the maximum number.
- Here maximum number is 89 and number of digits is 2. So, number of PASS=2
- The distribution will be from LSD (Less Significant Digit)

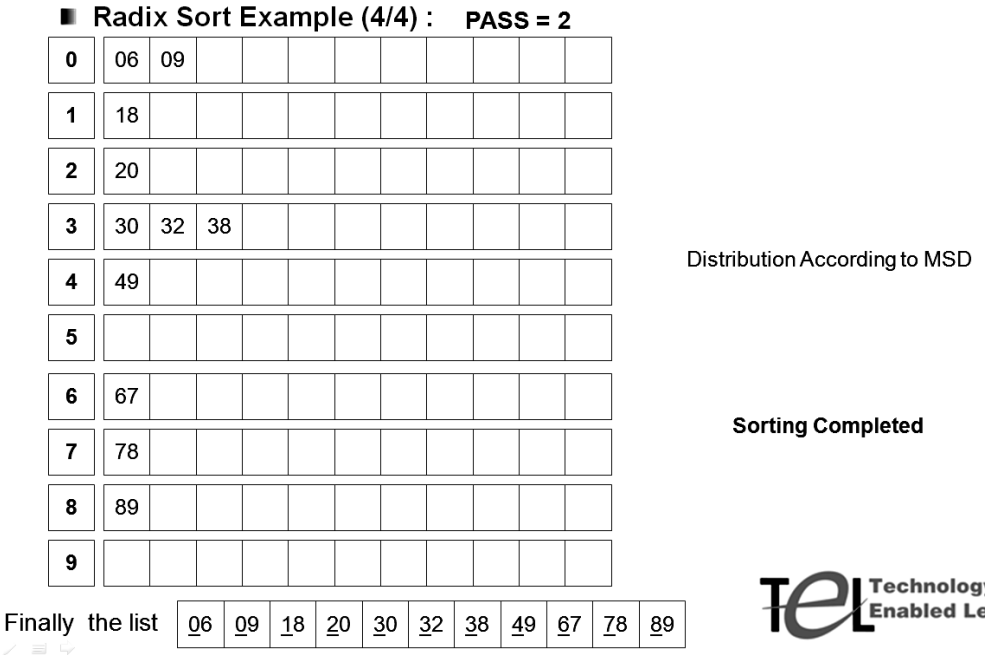
■ **Radix Sort Example (3/4) : PASS = 1**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 20 | 30 | | | | | | | | |
| **1** | | | | | | | | | | |
| **2** | 32 | | | | | | | | | |
| **3** | | | | | | | | | | |
| **4** | | | | | | | | | | |
| **5** | | | | | | | | | | |
| **6** | 06 | | | | | | | | | |
| **7** | 67 | | | | | | | | | |
| **8** | 18 | 38 | 78 | | | | | | | |
| **9** | 09 | 49 | 89 | | | | | | | |

Distribution According to LSD

Finally the list

| 20 | 30 | 32 | 06 | 67 | 18 | 38 | 78 | 09 | 49 | 89 |
|---|---|---|---|---|---|---|---|---|---|---|




Example: Write a program to sort elements using Radix sort algorithm.[File Name: u13q9.c]

```
/* RADIX SORT */
# include<stdio.h>
# include<malloc.h>
# include<stdlib.h>
struct node
{
        int data ;
        struct node *next;
};
typedef struct node node1;
node1 *first;
node1 *pocket[100], *pocket1[100];
void create_node(node1 *, int);
```

```
void display(node1 *);
node1 *radix_sort(node1 *);
int large(node1 * );
int numdig(int );
int digit(int, int);
void update(int, node1 *);
node1 *Make_link(int, node1 *);
/* This function create nodes and take input data */
void  create_node(node1 *rec, int n)
{
        int i, j, k;
        for(i = 0 ; i< n; i++)
        {
                rec->next = (node1 *) malloc(sizeof(node1));
                printf("\n First node value: %d: ", i);
                scanf("%d", &rec->data);
                rec = rec->next;
        }
        rec->next = NULL;
}
/* Output Function */
void  display(node1 *rec)
{
        while(rec !=NULL)
        {
                printf(" %d", rec->data);
                rec= rec->next;
        }
}
```

```
/* This radix sort function */
node1 *radix_sort(node1 *rec)
{
        node1 *r, *nex;
        int poc = 0 ;
        int i, j, k;
        int larg = large(rec);
        int m = numdig(larg);
        /* These statements create pockets */
        for(k = 0 ; k < 10; k++)
        {
                pocket[k] = (node1 *)malloc(sizeof(node1));
                pocket1[k] = (node1 *)malloc(9*sizeof(node1));
        }
        /* These statements initialize pockets */
        for(j = 1; j <= m ; j++)
        {
                for(i = 0 ; i < 10 ; i++)
                {
                        pocket[i] = NULL;
                        pocket1[i] = NULL ;
                }
                r = rec ;
                while(r != NULL)
                {
                        int dig = digit(r->data, j);
                        nex = r->next ;
                        update(dig,r);
                        r = nex;
```

```
                }
                if(r!= NULL)
                {
                        int dig = digit(r->data,j);
                        update(dig, r);
                }
                while(pocket1[poc] == NULL)
                        poc ++;
                rec = Make_link(poc, rec);
        }
        return(rec);
}
/* This function finds largest number in the list */
int large(node1 *rec)
{
        node1 *save ;
        int p = 0;
        save = rec ;
        while(save != NULL)
        {
                if(save ->data > p)
                {
                        p = save->data;
                }
                save = save->next ;
        }
        printf("\n Largest element: %d", p);
        return(p);
}
```

```
/* This Function finds number digits in a number */
int numdig(int large)
{
        int temp = large ;
        int num = 0 ;
        while(temp != 0)
        {
                ++num ;
                temp = temp/10 ;
        }
        printf("\n  Number of digits of the number %d is %d", large, num);
        return(num);
}
/* This function scarve a number into digits */
int digit(int num, int j)
{
        int dig, i, k;
        int temp = num ;
        for(i = 0 ; i < j ; i++)
        {
                dig = temp % 10 ;
                temp = temp / 10 ;
        }
        printf("  %d digit of number  %d is %d", j, num, dig);
        return(dig);
}
/* This function updates the pockets value */
void  update(int dig, node1 *r)
{
```

```
        if(pocket[dig] == NULL)
        {
                pocket[dig] = r ;
                pocket1[dig] = r ;
        }
        else
        {
                pocket[dig]->next = r ;
                pocket[dig] = r ;
        }
        r->next = NULL;
}
/* This function create links between the nodes */
node1* Make_link(int poc , node1 *rec)
{
        int i, j, k;
        node1 *pointer;
        rec = pocket1[poc];
        for(i = poc +1 ; i< 10 ; i++)
        {
                pointer = pocket[i-1];
                if(pocket[i] != NULL)
                        pointer->next= pocket1[i];
                else
                        pocket[i] = pointer ;
        }
        return(rec);
}
```

```
/* Main function */
void  main()
{
        node1 *start, *pointer;
        int number;
        printf("\n Input the number of elements in the list:");
        scanf("%d", &number);
        start = (node1 *)malloc(sizeof(node1));
        create_node(start, number);
        printf("\n Given list is as follows \n");
        display(start);
        start = radix_sort(start);
        printf("\n Sorted list is as follows:\n");
        display (start);
}
```

## 13.5 External Sorts

**External sorting** is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted does not fit into the main memory of a computing device (usually RAM) and a slower kind of memory (usually a hard drive) needs to be used.

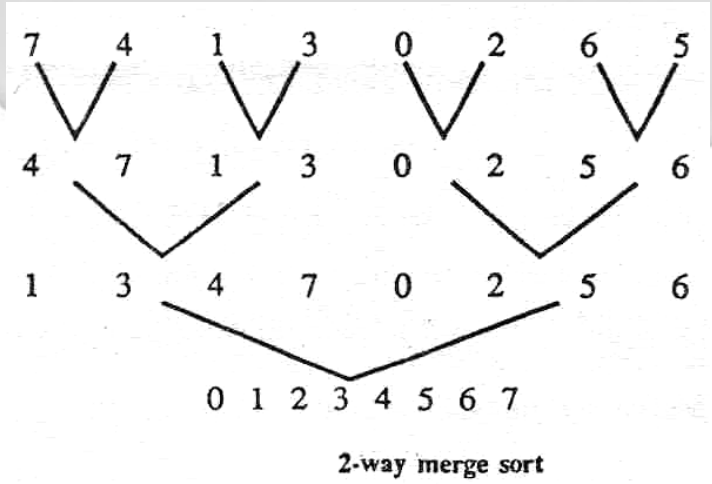
**Why External Sorts?**

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- Sorting is first step in *bulk loading* B+ tree index.
- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- *Sort-merge* join algorithm involves sorting.

❖ Problem: sort 1GB of data with 1MB of RAM.

### 13.5.1 Merge Sort

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea into this is to divide the list into a number of sublists, sort each of these sublists and merge them to get a single sorted list. The recursive implementation of 2- way merge sort divides the list into 2 sorts the sublists and then merges them to get the sorted list. The illustrative implementation of 2 way merge sort sees the input initially as n lists of size 1. These are merged to get 0/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This can be better understood by the following example. This is also called CONCATENATE SORT.




2-way merge sort

**Example:** Write a program to sort elements using merge sort algorithm. [File Name : u13q10.c]

```
/* MERGE SORT */
# include<stdio.h>
int merge_sort( int , int *, int  , int *, int * );
int bubble_sort(int , int *);
/* Definition of function */
```

```
bubble_sort(int n, int l[])
{
        int flag = 1;
        int i, j, k, temp;
        for(j = 0 ; j< n - 1; j++)
        {
             for(k = 0; k<  n - j - 1; k++)
            {
                if(l[k] > l[k+1])
               {
                   temp = l[k];
                   l[k] = l[k+1];
                   l[k+1] = temp ;
                   flag = 0;
                }
            }
  if(flag)
   break;
else
  flag = 1;
}
   printf("\n Entered  list as  follows in");
  printf("\n Ascending order: ");
  for(i = 0; i < n; i++)
  printf(" %d", l[i]);
  return 0;
}
/* Definition of function */
merge_sort( int n, int list_a[], int m ,
```

```
int list_b[], int result_list[] )
{
        int i = 0;
        int j = 0;
        int k = 0;
       int ch, l;
/* Repeat the process until the elements of list_a and list_b are exhausted */
 while((i < n) &&(j<m))
  {
      if(list_a[i] < list_b[j])
     {
          result_list[k] = list_a[i];
          i++;
          k++;
     } /*  end of if statement */
   else
   if(list_a[i] > list_b[j])
   {
        result_list[k] = list_b[j];
        j++;
       k++;
  } /* end of if statement */
 else
{
          result_list[k] = list_a[i];
          i++;
          j++;
          k++;
}  /* end of else statement */
```

```
printf("\n");
for (ch = 0; ch < k; ch++)
   printf(" %d", result_list[ch]);
} /* end of while statement */
   /* checks if size of list_a larger than list_b
         copy rest of the elements of list_a into result_list */
if (i <n )
 {
       for(l = i; l <n; l++)
       {
            result_list[k] = list_a[i];
            i++;
            k++;
           printf("\n");
           for(ch = 0; ch < k; ch++)
           printf(" %d", result_list[ch]);
     }
}
else
/* checks if size of list_b larger than list_a_
        copy rest of the elements of list_b into result_list */
if(j < m)
{
   for(l = j; l<m; l++)
  {
   result_list[k] = list_b[j];
   j++;
   k++;
   printf("\n");
   for(ch = 0; ch < k; ch++)
   printf(" %d", result_list[ch]);
```

```
}
}
return (k);
} /* function main */
void main()
{
    int list_a[100], list_b[100];
    int result[200];
    int n, m, k, i;
    printf("\n Input the number of elements of list_a: ");
    scanf("%d", &n);
    for(i = 0; i<n; i++)
    {
        printf("\n Input the element: %d: ",  i+1);
       scanf("%d", &list_a[i]);
  }
    /* Sort the elements of list_a */
bubble_sort(n, list_a);
/* End of sorting */
printf("\n Input the number of elements of list_b:");
scanf("%d", &m);
for(i = 0 ; i< m ; i++)
{
   printf("\n Input the element: %d: ", i+1);
   scanf("%d", &list_b[i]);
}

/* Sort the elements of list_b */
bubble_sort(m, list_b);
/* End of sorting */
 k = merge_sort( n, list_a, m , list_b, result);
```

```
printf("\n Duplicates are : %d", m+n-k);
printf("\n");
printf("\n Sorted list is as follows:\n");
for(i = 0 ; i< k ; i++)
{
  printf(" %d", result[i]);
}
}
```

## 13.6 Summary

Search algorithm, broadly speaking, is an algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions. Most of the algorithms studied by computer scientists that solve problems are kinds of search algorithms. The set of all possible solutions to a problem is called the search space. Searching methods are designed to take advantage of the file organization and optimize the search for a particular record or to establish its absence. The file organization and searching method chosen can make a substantial difference to an application's performance. This unit covers searching and sorting algorithms such as Binary Search, Quick Sort, Heap Sort, Merge Sort.

### Self Assessment Questions

1. Binary Search can be done over sorted data only. (True /False)
2. Internal sorting does operations over main memory only. (True / False)
3. Selection sort and bubble sort having same time complexity. (True / False)
4. In quick sort, any element can be considered as pivot. (True / False)
5. In radix sort, number of pass has calculated as number of digits of maximum value. (True / False)

## 13.7 Terminal Questions

1. What do you mean by sorting?
2. Describe various sorting algorithms.
3. Explain where elementary algorithms are more appropriate.
4. Discuss the Insertion sort with a suitable example.
5. Write C program to arrange the numbers in ascending order using bubble sort technique.
6. Write algorithm for Selection sort.
7. Write the advantages of Quick Sort
8. Discuss the different ideas which lead to Sorting Algorithms.
9. Explain what are the criteria to be used in evaluating a Sorting Algorithm?
10. Explain any two Internal Sorting with algorithm
11. Write a 'C' program to sort 'N" numbers using insertion sort.
12. Write the algorithm and 'C' program for sorting the numbers in ascending order using quick sort technique.
13. What is the limitation of Radix Sort and how can it be managed?

## 13.8 Answers to Self Assessment and Terminal Questions

### Self Assessment Questions

1. True
2. True
3. True
4. True
5. True

**Terminal Questions**

1. Section 13.3
2. Section 13.4
3. Section 13.3
4. Section 13.4.3
5. Section 13.4.2
6. Section 13.4.1
7. Section 13.4.4
8. Section 13.3
9. Section 13.3
10. Section 13.4
11. Section 13.4.3
12. Section 13.4.4
13. Section 13.5.7
14. Sections 13.4.5 , 13.5.1 and 13.4.7