

7 Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. Sorting algorithms often have additional properties that are of interest, depending on the application.

Sorting algorithm: An algorithm that rearranges records in lists so that they follow some well-defined ordering relation on values of keys in each record.

Let $a[n]$ be an array of n elements $a_0, a_1, a_2, a_3, \dots, a_{n-1}$ in memory. The sorting of the array $a[n]$ means arranging the content of $a[n]$ in either increasing or decreasing order.

i.e. $a_0 \leq a_1 \leq a_2 \leq a_3 \leq \dots \leq a_{n-1}$

consider a list of values: 2, 4, 6, 8, 9, 1, 22, 4, 77, 8, 9

After sorting the values: 1, 2, 4, 4, 6, 8, 8, 9, 9, 22, 77

In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

- It is very difficult to select a sorting algorithm over another. And there is no sorting algorithm better than all others in all circumstances. Some sorting algorithm will perform well in some situations, so it is important to have a selection of sorting algorithms. Some factors that play an important role in selection processes are the time complexity of the algorithm (use of computer time), the size of the data structures (for Eg: an array) to be sorted (use of storage space), and the time it takes for a programmer to implement the algorithms (programming effort).

7.1 Bubble Sort

- Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts. This procedure of sorting is called **bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).**
- If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.
- Consider Following Example

Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need $n - 1$ passes to sort n elements.

Algorithm

Let A be a linear array of n numbers. **temp** is a temporary variable for swapping (or interchange) the position of the numbers.

1. Input n numbers of an array A
2. Initialize $i = 0$ and repeat through step 4 if $(i < n)$
3. Initialize $j = 0$ and repeat through step 4 if $(j < n - i - 1)$
4. If $(A[j] > A[j + 1])$
 - i. $\text{temp} = A[j]$
 - ii. $A[j] = A[j + 1]$
 - iii. $A[j + 1] = \text{temp}$
5. Display the sorted numbers of array A
6. Exit

Source Code

```
Void BubbleSort(int A[],int n)
{
    int i, j, temp;
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Time Complexity:

Inner loop executes for $(n-1)$ times when $i=0$, $(n-2)$ times when $i=1$ and so on:

$$\text{Time complexity} = (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ = O(n^2)$$

There is no best-case linear time complexity for this algorithm.

Space Complexity:

Since no extra space besides 3 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

Bubble Sort Optimization

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all $n-1$ passes. We may even have an array that will be sorted in 2 or 3 passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to FALSE before each pass and is made TRUE when a swapping is performed. The code for the optimized bubble sort can be given as:

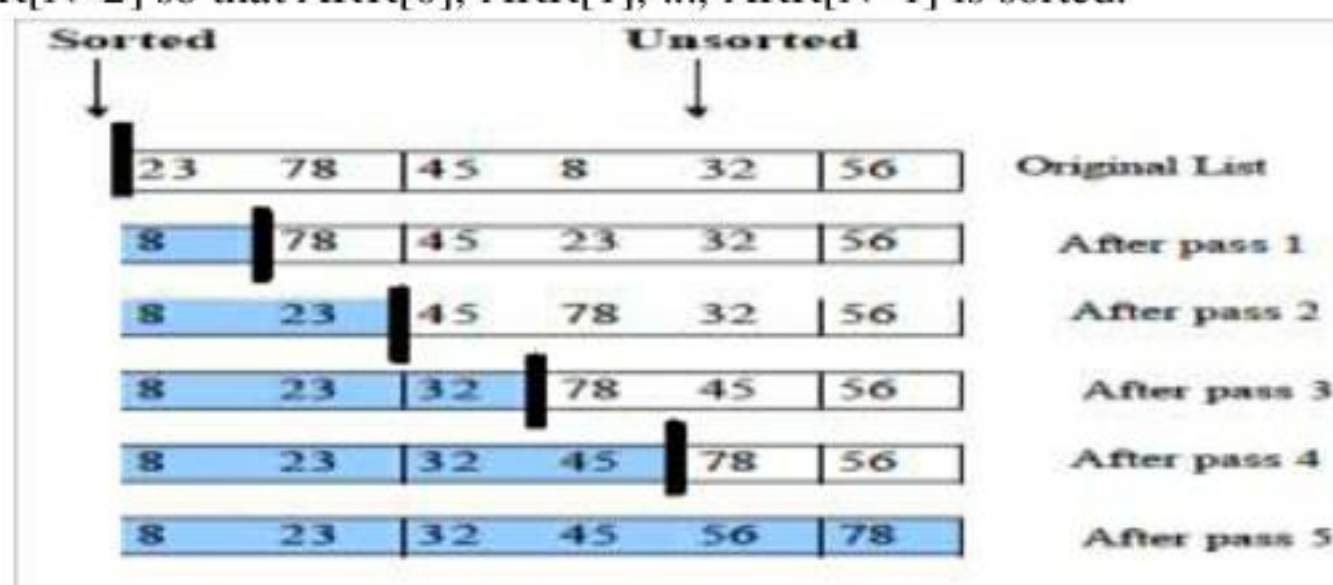
```
void BubbleSort(int A[],int n)
{
    int i, j, temp, flag = 0;
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                flag = 1;
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
        if(flag == 0)           // array is sorted
            return;
    }
}
```

Complexity of Optimized Bubble Sort Algorithm

In the best case, when the array is already sorted, the optimized bubble sort will take $O(n)$ time. In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In average case also, the performance will see an improvement. Compare it with the complexity of original bubble sort algorithm which takes $O(n^2)$ in all the cases.

7.2 Selection Sort

- **Idea:** Find the least (or greatest) value in the array, swap it into the leftmost (or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly. Let $a[n]$ be a linear array of n elements.
- First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,
 - In Pass 1, find the position POS of the smallest value in the array and then swap $ARR[POS]$ and $ARR[0]$. Thus, $ARR[0]$ is sorted.
 - In Pass 2, find the position POS of the smallest value in sub-array of $N-1$ elements. Swap $ARR[POS]$ with $ARR[1]$. Now, $ARR[0]$ and $ARR[1]$ is sorted.
 - In Pass $N-1$, find the position POS of the smaller of the elements $ARR[N-2]$ and $ARR[N-1]$. Swap $ARR[POS]$ and $ARR[N-2]$ so that $ARR[0]$, $ARR[1]$, ..., $ARR[N-1]$ is sorted.



Algorithm

Let A be a linear array of n numbers. **temp** be a temporary variable for swapping (or interchanging). **pos** is the variable to store the location of smallest number

1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step 5 if (i < n - 1)
 - pos = i
3. Initialize j = i + 1 and repeat through step 4 if (j < n)
4. if (A[pos] > A[j])
 - pos = j
5.
 - i. temp = A[pos]
 - ii. A[pos] = A[i]
 - iii. A[i] = temp
6. Display "the sorted numbers of array A"
7. Exit

Source Code

```
void selectionsort(int a[],int n)
{
    int i,j,temp,pos;
    for(i=0;i<n-1;i++)
    {
        pos = i;
        for(j=i+1;j<n;j++)
        {
            if(a[pos] > a[j])
                pos = j;
        }
        temp= a[pos];
        a[pos]=a[i];
        a[i]=temp;
    }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

$$\text{Time complexity} = (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ = O(n^2)$$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

7.3 Insertion Sort

- Insertion sort works by repeatedly taking an element from the unsorted portion of a list and inserting it into the sorted portion of the list until every element has been inserted.
- We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.
- The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Technique

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.

- The first element of the unsorted partition has array index 1 (if $LB = 0$).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Suppose an array $a[n]$ with n elements. The insertion sort works as follows:

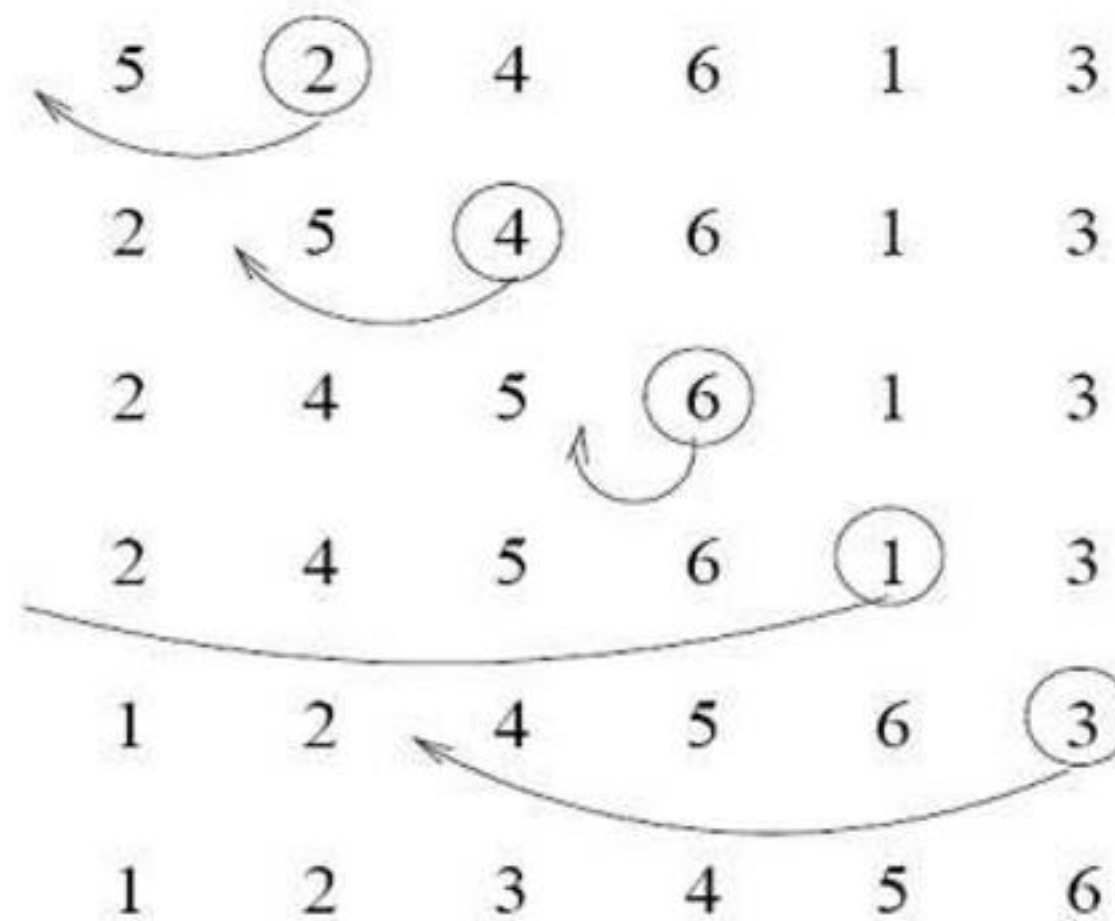
Pass 1: $a[0]$ by itself is trivially sorted.

Pass 2: $a[1]$ is inserted either before or after $a[0]$ so that $a[0], a[1]$ is sorted.

Pass 3: $a[2]$ is inserted into its proper place in $a[0], a[1]$ that is before $a[0]$, between $a[0]$ and $a[1]$, or after $a[1]$ so that $a[0], a[1], a[2]$ is sorted.

.....

Pass N: $a[n-1]$ is inserted into its proper place in $a[0], a[1], a[2], \dots, a[n-2]$ so that $a[0], a[1], a[2], \dots, a[n-1]$ is sorted with n elements.



Algorithm

Let A be a linear array of n numbers $A[1], A[2], A[3], \dots, A[n]$. Swap be a temporary variable to interchange the two values. Pos is the control variable to hold the position of each pass.

1. Input an array A of n numbers
2. Initialize $i = 1$ Repeat Steps 3 to 5 for $i = 1$ to $N - 1$
3. $temp = A[i]$,
 $Pos = i - 1$
4. Repeat the step 3 while($temp < A[Pos]$ and ($Pos \geq 0$))
 $A[Pos+1] = A[Pos]$
 $Pos = Pos - 1$
5. $A[Pos + 1] = temp$
6. Exit

Source Code

```
void insertion_sort(int A[], int n)
{
    int i, pos, temp;
    for(i=1; i<n; i++)
    {
        temp = A[i];           /*temp is to be inserted at proper place*/
        pos = i-1;
        while(temp<A[pos] && pos >= 0)
        {
```



```

        A[pos+1]= A[pos];
        pos = pos -1;
    }
    A[pos+1]= temp;
}

```

✚ Time Complexity

Worst Case Analysis:

Array elements are in reverse sorted order

Inner loop executes for 1 times when $i=1$, 2 times when $i=2$... and $n-1$ times when $i=n-1$:

$$\text{Time complexity} = 1 + 2 + 3 + \dots + (n-2) + (n-1) \\ = O(n^2)$$

Best case Analysis:

Array elements are already sorted

Inner loop executes for 1 times when $i=1$, 1 times when $i=2$... and 1 times when $i=n-1$:

$$\text{Time complexity} = 1 + 2 + 3 + \dots + 1 + 1 \\ = O(n)$$

✚ Space Complexity

Since no extra space besides 5 variables is needed for sorting

Space Complexity = $O(1)$

✚ Advantages of Insertion Sort

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- it requires less memory space (only $O(1)$ of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

7.4 SHELL SORT

Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observed two things:

- First, insertion sort works well when the input data is 'almost sorted'.
- Second, insertion sort is quite inefficient to use as it moves the values just one position at a time.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position. In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already 'almost sorted', and hence it provides good performance. If we take a scenario in which the smallest element is stored in the other end of the array, then sorting such an array with either bubble sort or insertion sort will execute in $O(n^2)$ time and take roughly n comparisons and exchanges to move this value all the way to its correct position. On the other hand, Shell sort first moves small values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.

Technique

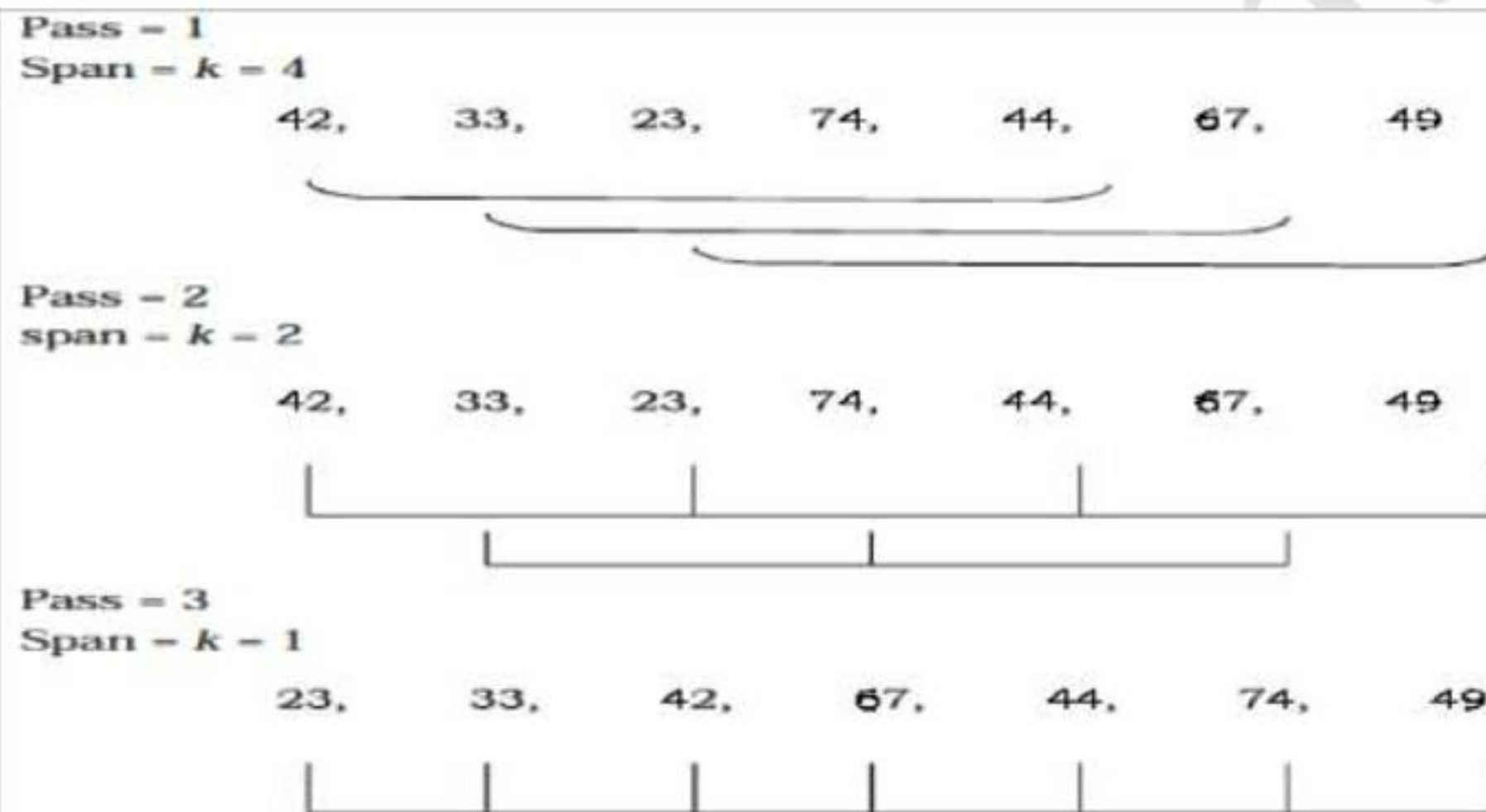
- To visualize the way in which shell sort works, perform the following steps:

Step 1: Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).

Step 2: Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Note that we are only visualizing the elements being arranged in a table, the algorithm does its sorting in-place.

- In this method, sub-arrays, containing kth element of the original array, are sorted. Let A be a linear array of n numbers A [1], A [2], A [3], A [n].
 - **Step 1:** The array is divided into k sub-arrays consisting of every kth element. Say k= 5, then five sub- array, each containing one fifth of the elements of the original array.
 - Sub array 1 → A[0] A[5] A[10]
 - Sub array 2 → A[1] A[6] A[11]
 - Sub array 3 → A[2] A[7] A[12]
 - Sub array 4 → A[3] A[8] A[13]
 - Sub array 5 → A[4] A[9] A[14]
 - Note : The ith element of the jth sub array is located as A [(i-1) * k+j-1]
 - **Step 2:** After the first k sub array are sorted (usually by insertion sort) , a new smaller value of k is chosen and the array is again partitioned into a new set of sub arrays.
 - **Step 3:** And the process is repeated with an even smaller value of k, so that A [1], A [2], A [3],A [n] is sorted.
- To illustrate the shell sort, consider the following array with 7 elements 42, 33, 23, 74, 44, 67, 49 and the sequence K = 4, 2, 1 is chosen.



Algorithm for shell sort

Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N

Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1

Step 3: SET FLAG = 0

Step 4: SET GAP_SIZE = (GAP_SIZE + 1) / 2

Step 5: Repeat Step 6 for I = 0 to I < (N - GAP_SIZE)

Step 6: IF Arr[I + GAP_SIZE] > Arr[I]

SWAP Arr[I + GAP_SIZE], Arr[I]

SET FLAG = 1

Step 7: END

In the algorithm, we sort the elements of the array Arr in multiple passes. In each pass, we reduce the gap_size (visualize it as the number of columns) by a factor of half as done in Step 4. In each iteration of the for loop in Step 5, we compare the values of the array and interchange them if we have a larger value preceding the smaller one.

Example: Sort the elements given below using shell sort.

63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

Solution

Arrange the elements of the array in the form of a table and sort the columns.

63 19 7 90 81 36 54 45
72 27 22 9 41 59 33

Result:

63 19 7 9 41 36 33 45
72 27 22 90 81 59 54

The elements of the array can be given as:

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

Repeat Step 1 with smaller number of long columns.

63 19 7 9 41
36 33 45 72 27
22 90 81 59 54

Result:

22 19 7 9 27
36 33 45 59 41
63 90 81 72 54

The elements of the array can be given as:

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Repeat Step 1 with smaller number of long columns.

22 19 7
9 27 36
33 45 59
41 63 90
81 72 54

Result:

9 19 7
22 27 36
33 45 54
41 63 59
81 72 90

The elements of the array can be given as:

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

Finally, arrange the elements of the array in a single column and sort the column.

Result:

9
19
7
22
27
36
33
45
54
41
63
59
81
72
90

7
9
19
22
27
33
36
41
45
54
59
63
72
81
90

Finally, the elements of the array can be given as:

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

Program to implement Shell Sort

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int arr[10];
    int i, j, n, flag = 1, gap_size, temp;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter %d numbers: ", n); // n was added
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);
    gap_size = n;
    while(flag == 1 || gap_size > 1)
    {
        flag = 0;
        gap_size = (gap_size + 1) / 2;
```



```

    for(i=0; i< (n - gap_size); i++)
    {
        if( arr[i+gap_size] < arr[i])
        {
            temp = arr[i+gap_size];
            arr[i+gap_size] = arr[i];
            arr[i] = temp;
            flag = 0;
        }
    }
}
printf("\n The sorted array is: \n");
for(i=0;i<n;i++)
    printf(" %d\t", arr[i]);
getch();
return 0;
}

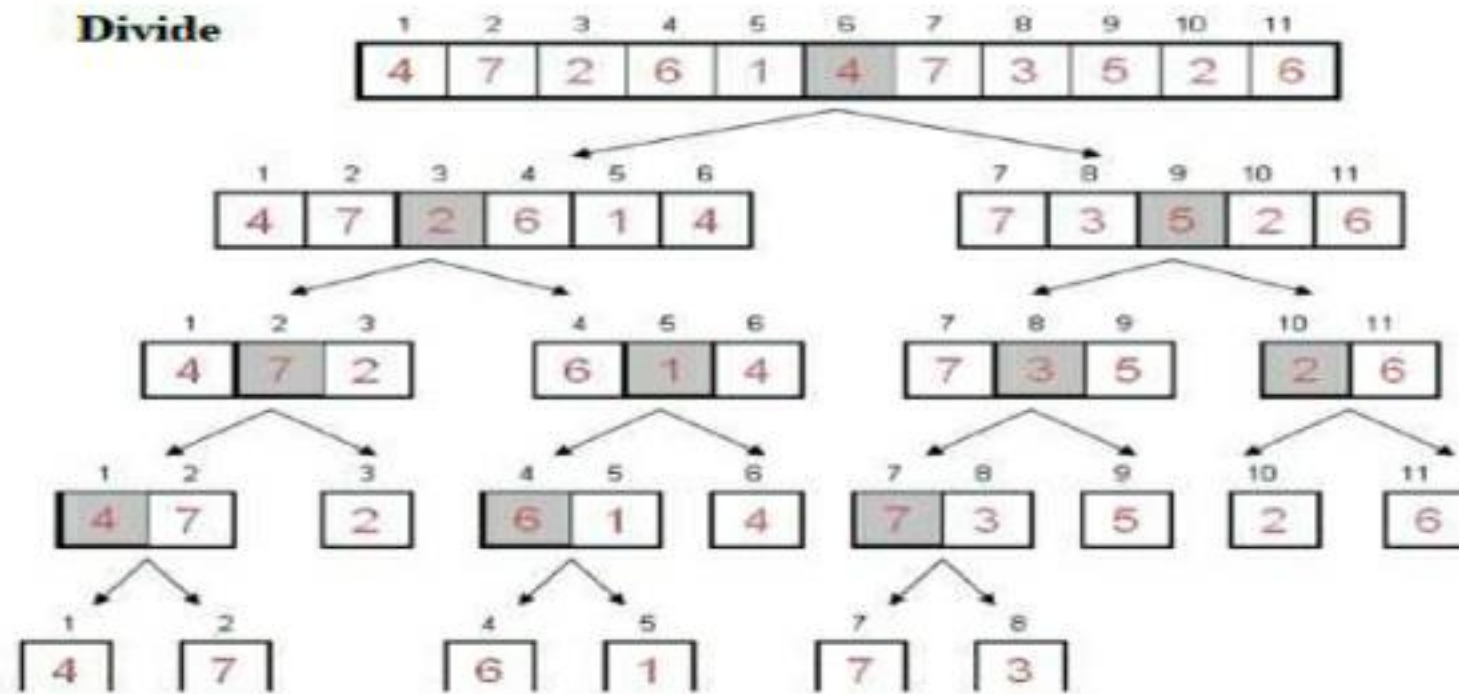
```

7.5 MERGE SORT

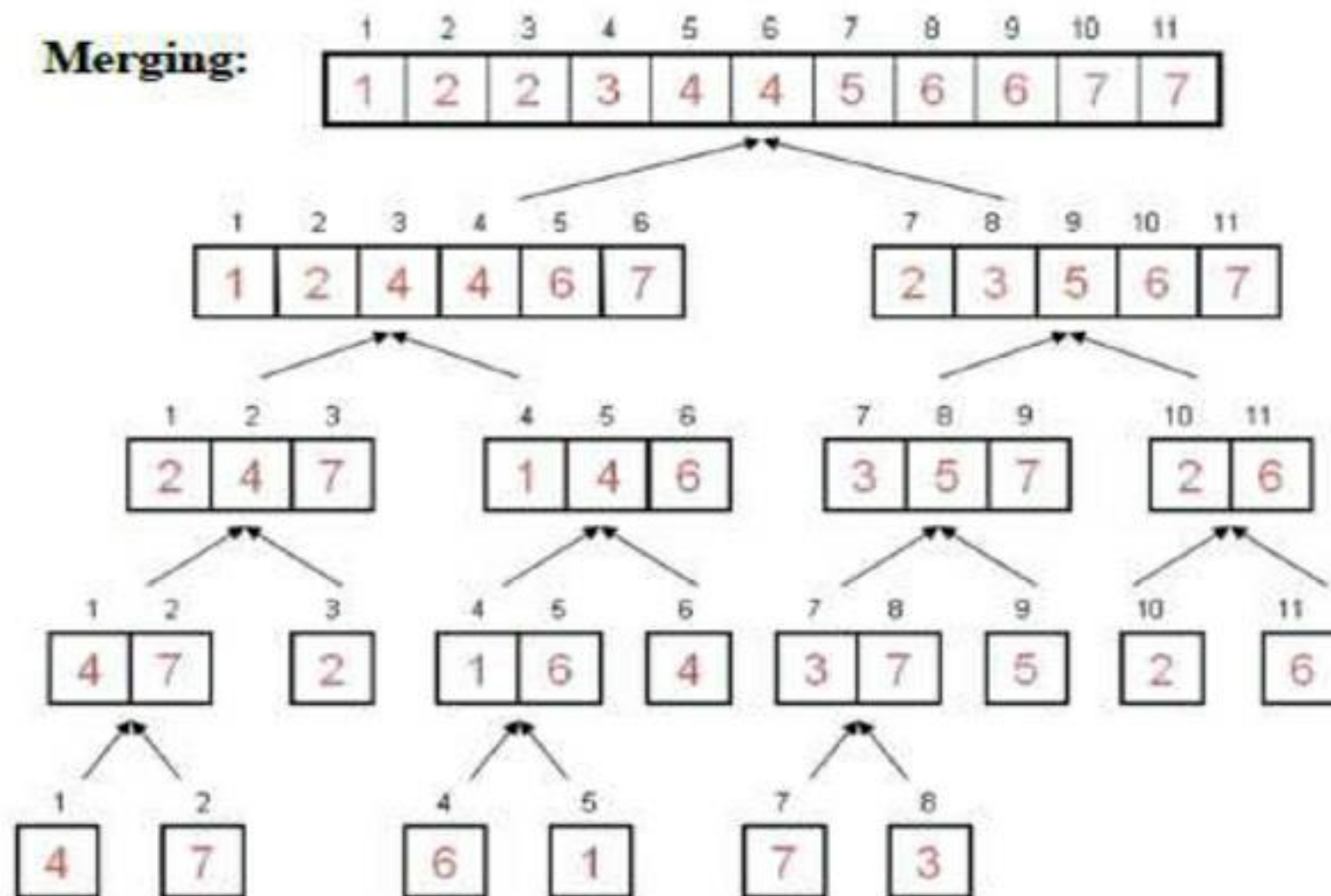
- Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.
 - Divide** means partitioning the n-element array to be sorted into two sub-arrays of $n/2$ elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
 - Conquer** means sorting the two sub-arrays recursively using merge sort.
 - Combine** means merging the two sorted sub-arrays of size $n/2$ to produce the sorted array of n elements.
- Merge sort algorithm focuses on two main concepts to improve its performance (running time):
 - A smaller list takes fewer steps and thus less time to sort than a large list.
 - As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.
- The basic steps of a merge sort algorithm are as follows:
 - If the array is of length 0 or 1, then it is already sorted.
 - Otherwise, divide the unsorted array into two sub-arrays of about half the size.
 - Use merge sort algorithm recursively to sort each sub-array.
 - Merge the two sub-arrays to form a single sorted list.

Example: $a[] = \{4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6\}$

Divide



Merging:



Algorithm for merge sort

```

MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
    SET MID = (BEG + END)/2
    CALL MERGE_SORT (ARR, BEG, MID)
    CALL MERGE_SORT (ARR, MID + 1, END)
    MERGE (ARR, BEG, MID, END)
[END OF IF]
Step 2: END
  
```

MERGE (ARR, BEG, MID, END)

```

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J <= END)
    IF ARR[I] < ARR[J]
        SET TEMP[INDEX] = ARR[I]
        SET I = I + 1
    ELSE
        SET TEMP[INDEX] = ARR[J]
        SET J = J + 1
    [END OF IF]
    SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
    IF I > MID
        Repeat while J <= END
            SET TEMP[INDEX] = ARR[J]
            SET INDEX = INDEX + 1, SET J = J + 1
        [END OF LOOP]
    [Copy the remaining elements of left sub-array, if any]
    ELSE
        Repeat while I <= MID
            SET TEMP[INDEX] = ARR[I]
            SET INDEX = INDEX + 1, SET I = I + 1
        [END OF LOOP]
    [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
    SET ARR[K] = TEMP[K]
    SET K = K + 1
[END OF LOOP]
Step 6: END
  
```


Program to implement merge sort

```

#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[], int, int);
void main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array : ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);
    merge_sort(arr, 0, n-1);
    printf("\n The sorted array is: \n");
    for(i=0; i<n; i++)
        printf(" %d\t", arr[i]);
    getch();
}

void merge(int arr[], int beg, int mid, int end)
{
    int i=beg, j=mid+1, index=beg, temp[size], k;
    while((i<=mid) && (j<=end))
    {
        if(arr[i] < arr[j])
        {
            temp[index] = arr[i];
            i++;
        }
        else
        {
            temp[index] = arr[j];
            j++;
        }
        index++;
    }
    if(i>mid)
    {
        while(j<=end)
        {
            temp[index] = arr[j];
            j++;
            index++;
        }
    }
    else
    {
        while(i<=mid)
        {
            temp[index] = arr[i];
            i++;
            index++;
        }
    }
}

```



```

    }
    for(k=beg;k<index;k++)
        arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
    int mid;
    if(beg<end)
    {
        mid = (beg+end)/2;
        merge_sort(arr, beg, mid);
        merge_sort(arr, mid+1, end);
        merge(arr, beg, mid, end);
    }
}

```

✚ **Time Complexity:**

Recurrence Relation for Merge sort:

$T(n) = 1$ if $n=1$

$T(n) = 2 T(n/2) + O(n)$ if $n>1$

Solving this recurrence we get

$T(n) = O(n \log n)$

✚ **Space Complexity:**

It uses one extra array and some extra variables during sorting, therefore

Space Complexity = $2n + c = O(n)$

7.6 QUICK SORT

- Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes $O(n \log n)$ comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as $O(n^2)$. Basically, the quick sort algorithm is faster than other $O(n \log n)$ algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.
- Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.
- The quick sort algorithm works as follows:
 1. Select an element pivot from the array elements.
 2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the partition operation.
 3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)
- Like merge sort, the base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.
- Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

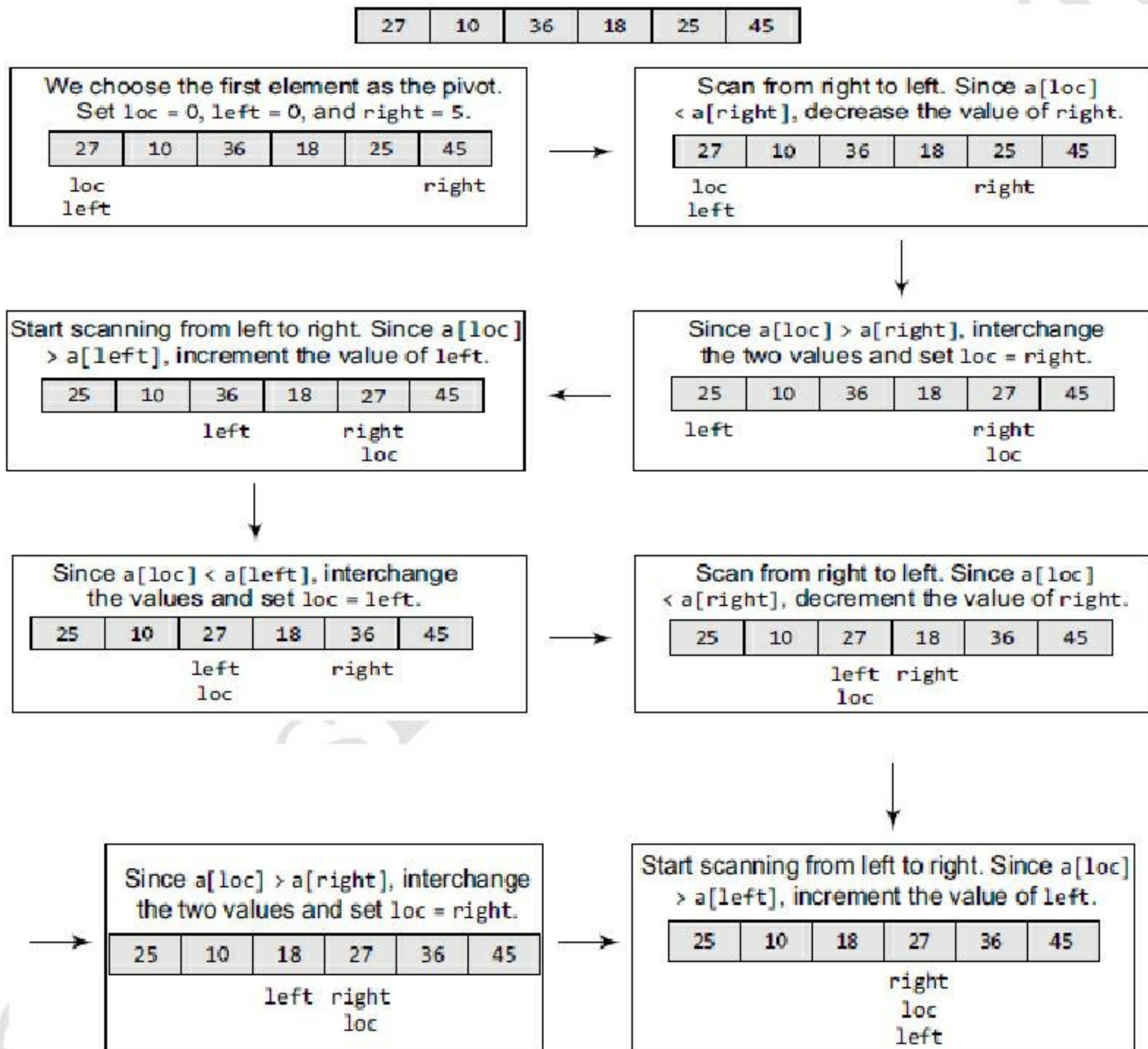
✚ **Technique**

Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, $loc = 0$, $left = 0$, and $right = n-1$ (where n is the number of elements in the array)
2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, $a[loc]$ should be less than $a[right]$.

- a) If that is the case, then simply continue comparing until right becomes equal to loc. Once $\text{right} = \text{loc}$, it means the pivot has been placed in its correct position.
 - b) However, if at any point, we have $a[\text{loc}] > a[\text{right}]$, then interchange the two values and jump to Step 3.
 - c) Set $\text{loc} = \text{right}$
3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, $a[\text{loc}]$ should be greater than $a[\text{left}]$.
 - a) If that is the case, then simply continue comparing until left becomes equal to loc. Once $\text{left} = \text{loc}$, it means the pivot has been placed in its correct position.
 - b) However, if at any point, we have $a[\text{loc}] < a[\text{left}]$, then interchange the two values and jump to Step 2.
 - c) Set $\text{loc} = \text{left}$.

Example: Sort the elements given in the following array using quick sort algorithm



Now $left = loc$, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Algorithm for Quick sort

PARTITION (ARR, BEG, END, LOC)

```

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
        SET RIGHT = RIGHT - 1
    [END OF LOOP]
Step 4: IF LOC = RIGHT
        SET FLAG = 1
    ELSE IF ARR[LOC] > ARR[RIGHT]
        SWAP ARR[LOC] with ARR[RIGHT]
        SET LOC = RIGHT
    [END OF IF]
Step 5: IF FLAG = 0
        Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
        SET LEFT = LEFT + 1
    [END OF LOOP]
Step 6: IF LOC = LEFT
        SET FLAG = 1
    ELSE IF ARR[LOC] < ARR[LEFT]
        SWAP ARR[LOC] with ARR[LEFT]
        SET LOC = LEFT
    [END OF IF]
    [END OF IF]
Step 7: [END OF LOOP]
Step 8: END

```

QUICK_SORT (ARR, BEG, END)

```

Step 1: IF (BEG < END)
        CALL PARTITION (ARR, BEG, END, LOC)
        CALL QUICKSORT(ARR, BEG, LOC - 1)
        CALL QUICKSORT(ARR, LOC + 1, END)
    [END OF IF]
Step 2: END

```

Time Complexity:

Best Case:

Divides the array into two partitions of equal size, therefore
 $T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get,
 $T(n) = O(n \log n)$

Worst case:

when one partition contains the $n-1$ elements and another partition contains only one element.
 Therefore its recurrence relation is:
 $T(n) = T(n-1) + O(n)$, Solving this recurrence we get
 $T(n) = O(n^2)$

Average case:

Good and bad splits are randomly distributed across throughout the tree
 $T_1(n) = 2T'(n/2) + O(n)$ Balanced
 $T'(n) = T(n-1) + O(n)$ Unbalanced
 Solving:
 $B(n) = 2(B(n/2 - 1) + \Theta(n/2)) + \Theta(n)$
 $= 2B(n/2 - 1) + \Theta(n)$
 $= O(n \log n)$
 $\Rightarrow T(n) = O(n \log n)$

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Program to implement quick sort algorithm

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
int main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
    quick_sort(arr, 0, n-1);
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
    getch();
    return 0;
}

int partition(int a[], int beg, int end)
{
    int left, right, temp, loc, flag;
    loc = left = beg;
    right = end;
    flag = 0;
    while(flag != 1)
    {
        while((a[loc] <= a[right]) && (loc!=right))
            right--;
        if(loc==right)
            flag = 1;
        else if(a[loc]>a[right])
        {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
        }
        if(flag!=1)
        {
            while((a[loc] >= a[left]) && (loc!=left))
                left++;
            if(loc==left)
                flag = 1;
            else if(a[loc] <a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
}
```



```

    }
    }
    return loc;
}
void quick_sort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quick_sort(a, beg, loc-1);
        quick_sort(a, loc+1, end);
    }
}

```

7.7 RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on. During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number of letters. After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on. When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, ..., 9) and the number of passes will depend on the length of the number having maximum number of digits.

Algorithm for Radix Sort

Algorithm for RadixSort (ARR, N)

```

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:     SET I = 0 and INITIALIZE buckets
Step 6:     Repeat Steps 7 to 9 while I<N-1
Step 7:         SET DIGIT = digit at PASSth place in A[I]
Step 8:         Add A[I] to the bucket numbered DIGIT
Step 9:         INCREMENT bucket count for bucket numbered DIGIT
                [END OF LOOP]
Step 10:    Collect the numbers in the bucket
                [END OF LOOP]
Step 11: END

```

Example: Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as
123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes $O(kn)$ time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in $O(n)$ asymptotic time.

Pros and Cons of Radix Sort

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters. But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters. Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task. Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Program to implement radix sort algorithm

```

#include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
int main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
    radix_sort(arr, n);
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
    getch();
}
int largest(int arr[], int n)
{
    int large=arr[0],i;
    for(i=1;i<n;i++)
    {
        if(arr[i]>large)
            large = arr[i];
    }
    return large;
}
void radix_sort(int arr[], int n)
{
    int bucket[size][size], bucket_count[size];
    int i, j, k, remainder, NOP=0, divisor=1, large, pass;
    large = largest(arr, n);
    while(large>0)
    {
        NOP++;
        large/=size;
    }
    for(pass=0;pass<NOP;pass++) // Initialize the buckets
    {
        for(i=0;i<size;i++)
            bucket_count[i]=0;
        for(i=0;i<n;i++)
        {
            // sort the numbers according to the digit at passth place
            remainder = (arr[i]/divisor)%size;
            bucket[remainder][bucket_count[remainder]] = arr[i];
            bucket_count[remainder] += 1;
        }
        // collect the numbers after PASS pass
        i=0;
        for(k=0;k<size;k++)

```



```

    {
        for(j=0;j<bucket_count[k];j++)
        {
            arr[i] = bucket[k][j];
            i++;
        }
    }
    divisor *= size;
}

```

7.8 Heap

A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father. It can be sequentially represented as $A[j] \leq A[(j-1)/2]$

for $0 \leq [(j-1)/2] < j \leq n-1$

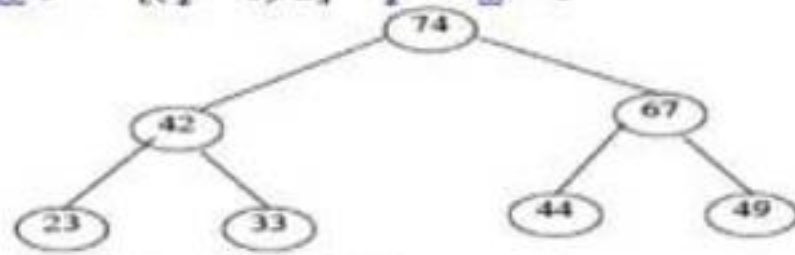


Figure 1: Heap



Figure 2: Array representation

The root of the binary tree (i.e. the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or **max heap**. We can also define an ascending heap as an almost complete binary tree in which the value of each node is greater than or equal to the value of its father. This root node has the smallest element of the heap. This type of heap is also called **min heap**.

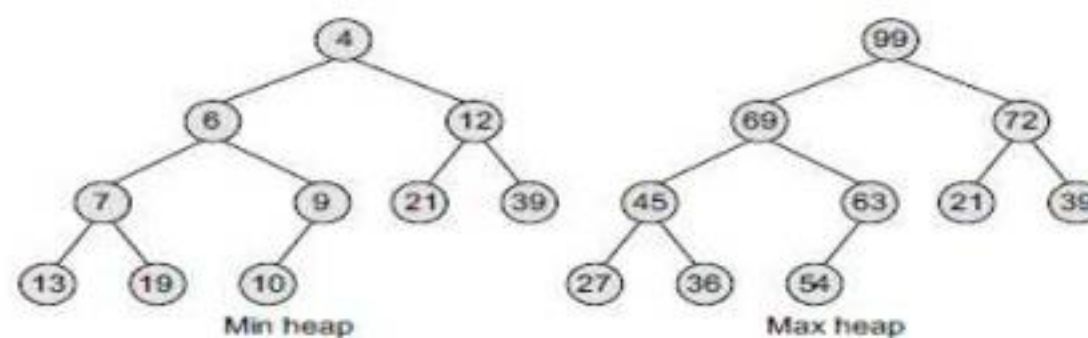


Figure A

7.9 Heap Sort

A heap can be used to sort a set of elements. Let H be a heap with n elements and it can be sequentially represented by an array A . Inset an element data into the heap H as follows:

1. First place data at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then the data be raised to its appropriate place in H so that H is finally a heap.

Inserting an element in to a heap

Consider the heap H in Figure 1. Say we want to add a data = 55 to H .

Step 1: First we adjoin 55 as the next element in the complete tree as shown in Figure 3. Now we have to find the appropriate place for 55 in the heap by rearranging it.

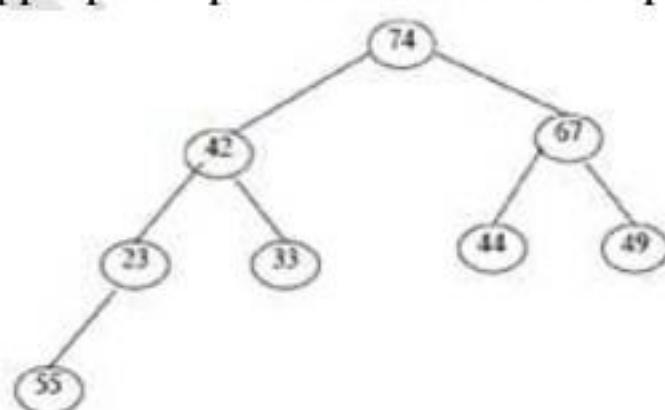


Figure 3

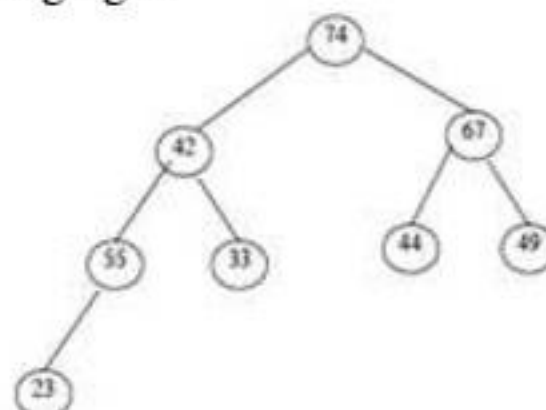


Figure 4

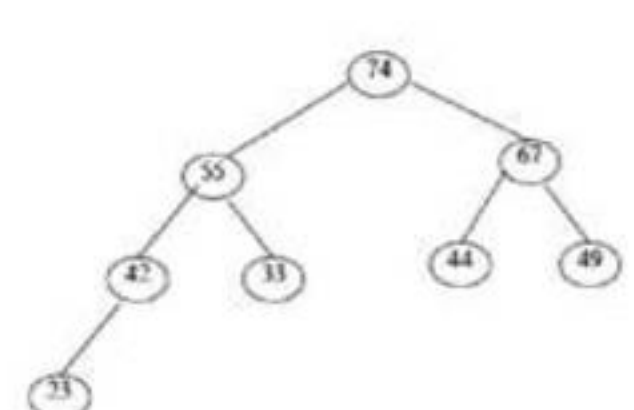


Figure 5

Step 2: Compare 55 with its parent 23. Since 55 is greater than 23, interchange 23 and 55. Now the heap will look like as in Figure 4.

Step 3: Compare 55 with its parent 42. Since 55 is greater than 42, interchange 55 and 42. Now the heap will look like as in Figure 5.

Step 4: Compare 55 with its new parent 74. Since 55 is less than 74, it is the appropriate place of node 55 in the heap H.

Deleting the Root of a Heap

Let H be a heap with n elements. The root R of H can be deleted as follows:

Step1: Assign the root R to some variable data.

Step2: Replace the deleted node R by the last node (or recently added node) L of H so that H is still a complete tree, but not necessarily a heap.

Step3: Now rearrange H in such a way by placing L (new root) at the appropriate place, so that H is finally a heap.

Consider the heap H in Figure 5 where R = 74 is the root and L = 23 is the last node (or recently added node) of the tree. Suppose we want to delete the root node R = 74. Apply the above rules to delete the root. Delete the root node R and assign it to data (i.e., data = 74) as shown in Figure 6.

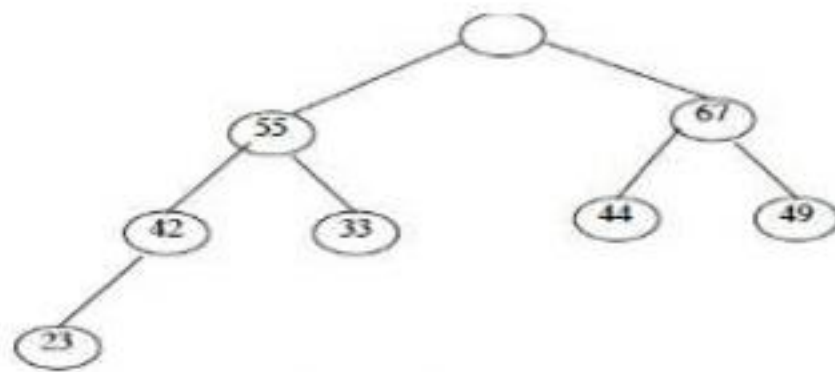


Figure 6

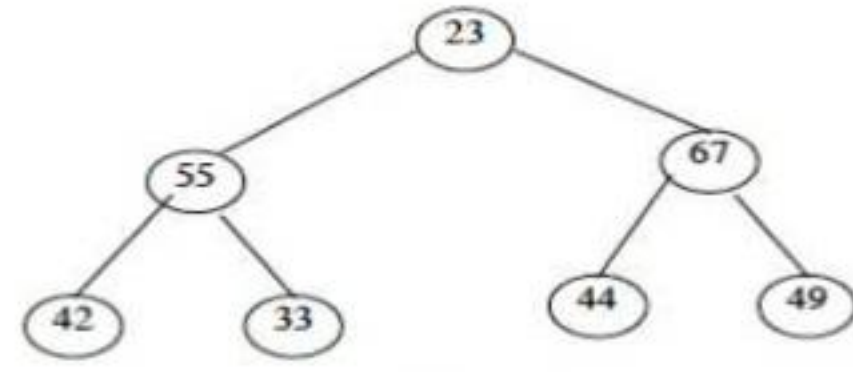


Figure 7

Replace the deleted root node R by the last node L as shown in the Figure 7. Compare 23 with its new two children 55 and 67. Since 23 is less than the largest child 67, interchange 23 and 67. The new tree looks like as in Figure 8. Again compare 23 with its new two children, 44 and 49. Since 23 is less than the largest child 49, interchange 23 and 49 as shown in Figure 9.

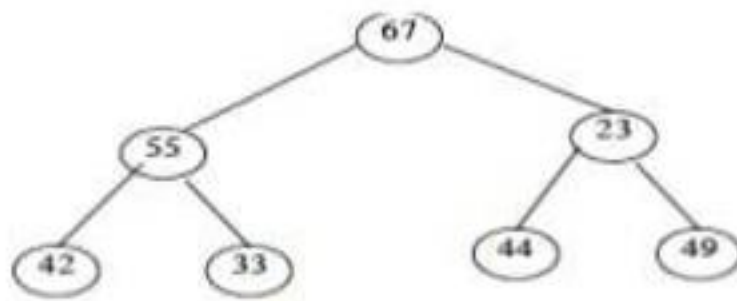


Figure 8

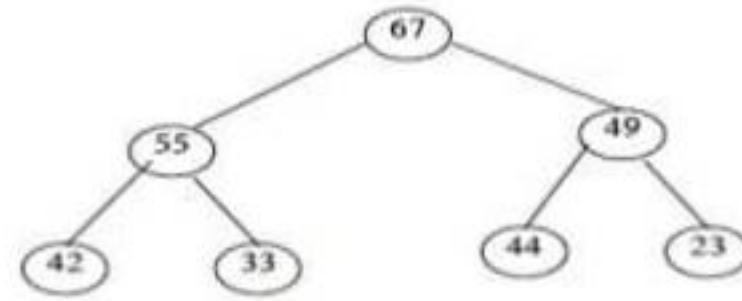


Figure 9

The Figure 9 is the required heap H without its original root R.

Algorithm for heap sort

Let H be a heap with n elements stored in the array HA. Data is the item of the node to be removed. Last gives the information about last node of H. The LOC, left, right gives the location of Last and its left and right children as the Last rearranges in the appropriate place in the tree.

1. Input n elements in the heap H
2. Data = HA[0]; last = HA[n-1] and n = n - 1
3. LOC = 0, left = 2*LOC+1 and right = 2*LOC+2
4. Repeat the steps 5, 6 and 7 while (right <= n)
5. If (last >= HA[left]) and (last >= HA[right])
 - a. HA[LOC] = last
 - b. Exit
6. a) If (HA[right] <= HA[left])
 - (i) HA[LOC] = HA[left]
 - (ii) LOC = left
- (b) Else
 - (i) HA[LOC] = HA[right]

- (ii) LOC = right
- 7. left = 2 * LOC; right = left + 1
- 8. If (left = n) and (last < HA[left])
 - a. LOC = left
- 9. HA [LOC] = last
- 10. Exit

Complexity of Heap Sort

Heap sort uses two heap operations: insertion and root deletion. Each element extracted from the root is placed in the last empty location of the array. In phase 1, when we build a heap, the number of comparisons to find the right location of the new element in H cannot exceed the depth of H. Since H is a complete tree, its depth cannot exceed m , where m is the number of elements in heap H. Thus, the total number of comparisons $g(n)$ to insert n elements of ARR in H is bounded as:

$$g(n) \leq n \log n$$

Hence, the running time of the first phase of the heap sort algorithm is $O(n \log n)$. In phase 2, we have H which is a complete tree with m elements having left and right sub-trees as heaps. Assuming L to be the root of the tree, reheaping the tree would need 4 comparisons to move L one step down the tree H. Since the depth of H cannot exceed $O(\log m)$, reheaping the tree will require a maximum of $4 \log m$ comparisons to find the right location of L in H.

Since n elements will be deleted from heap H, reheaping will be done n times. Therefore, the number of comparisons to delete n elements is bounded as:

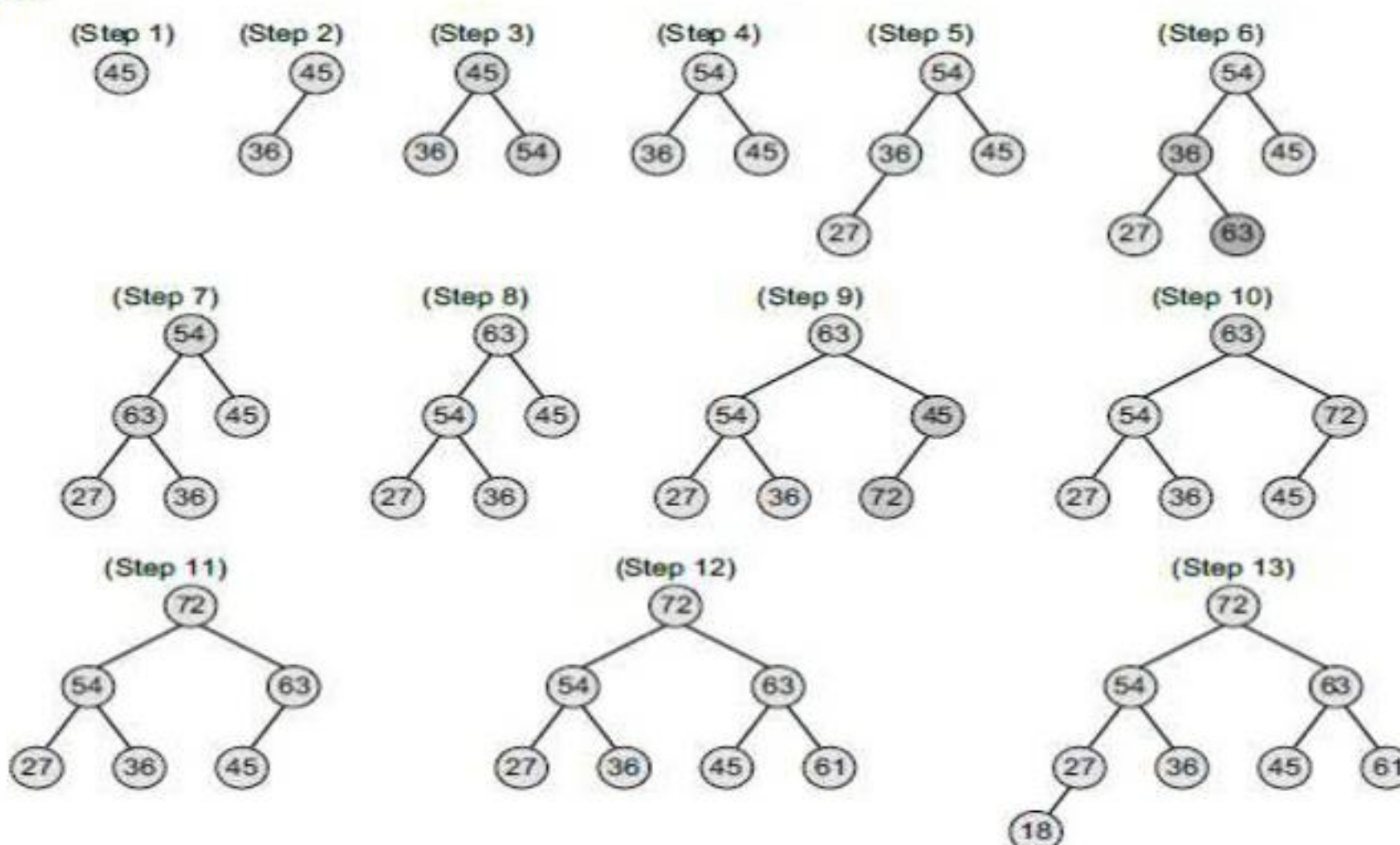
$$h(n) \leq 4n \log n$$

Hence, the running time of the second phase of the heap sort algorithm is $O(n \log n)$. Each phase requires time proportional to $O(n \log n)$. Therefore, the running time to sort an array of n elements in the worst case is proportional to $O(n \log n)$.

Therefore, we can conclude that heap sort is a simple, fast, and stable sorting algorithm that can be used to sort large sets of data efficiently.

Example: Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18.

Solution



Program to implement heap sort algorithm

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
void RestoreHeapUp(int *,int);
void RestoreHeapDown(int*,int,int);
```



```

int main()
{
    int Heap[MAX],n,i,j;
    printf("\n Enter the number of elements : ");
    scanf("%d",&n);
    printf("\n Enter the elements : ");
    for(i=1;i<=n;i++)
    {
        scanf("%d",&Heap[i]);
        RestoreHeapUp(Heap, i); // Heapify
    }
    // Delete the root element and heapify the heap
    j=n;
    for(i=1;i<=j;i++)
    {
        int temp;
        temp=Heap[1];
        Heap[1]= Heap[n];
        Heap[n]=temp;
        n = n-1; // The element Heap[n] is supposed to be deleted
        RestoreHeapDown(Heap,1,n); // Heapify
    }
    n=j;
    printf("\n The sorted elements are: ");
    for(i=1;i<=n;i++)
        printf("%4d",Heap[i]);
    return 0;
}

void RestoreHeapUp(int *Heap,int index)
{
    int val = Heap[index];
    while( (index>1) && (Heap[index/2] < val) ) // Check parent's value
    {
        Heap[index]=Heap[index/2];
        index /= 2;
    }
    Heap[index]=val;
}

void RestoreHeapDown(int *Heap,int index,int n)
{
    int val = Heap[index];
    int j=index*2;
    while(j<=n)
    {
        if( (j<n) && (Heap[j] < Heap[j+1]))// Check sibling's value
            j++;
        if(Heap[j] < Heap[j/2]) // Check parent's value
            break;
        Heap[j/2]=Heap[j];
        j=j*2;
    }
    Heap[j/2]=val;
}

```


7.10 Exchange Sort

The exchange sort is almost similar as the bubble sort. In fact some people refer to the exchange sort as just a different bubble sort. The exchange sort compares each element of an array and swap those elements that are not in their proper position, just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements.

The exchange sort compares the first element with each element of the array, making a swap where is necessary. In some situations the exchange sort is slightly more efficient than its counterpart the bubble sort. The bubble sort needs a final pass to **determine that it is finished, thus is slightly less efficient than the exchange sort, because the exchange sort doesn't need a final pass.**

Assume the following sequence of number: 45 67 78 12

45	67	78	12	9
12	67	78	45	9
9	67	78	45	12
9	67	78	45	12
9	45	78	67	12
9	12	78	67	45
9	12	67	78	45
9	12	45	78	67
9	12	45	67	78

Algorithm

1. Compare the first pair of numbers (positions 0 and 1) and reverse them if they are not in the correct order.
2. Repeat for the next pair (positions 1 and 2).
3. Continue the process until all pairs have been checked.
4. Repeat steps 1 through 3 for positions 0 through n - 1 to i (for i = 1, 2, 3, ...) until no pairs remain to be checked.
5. The list is now sorted.
6. Display the sorted list
7. Exit

Program to implement Exchange sort

```
#include<stdio.h>
#include<conio.h>
void display(int a[],int n);
int main(void)
{
    int array[50]; // An array of integers.
    int i, j;
    int temp;
    int n;
    printf("Enter n");
    scanf("%d",&n); //Some input
    for (i = 0; i < n; i++)
    {
        printf("Enter a number: ");
        scanf("%d", &array[i]);
    }
    //Algorithm
    for(i = 0; i < (n -1); i++)
    {
        for (j=(i + 1); j < n; j++)
        {
            if (array[i] > array[j])
            {
                temp = array[i];
```



```

        array[i] = array[j];
        array[j] = temp;
    }
    display(array,n);
}
}
printf("\nSorted Array is: \n");
display(array,n);
}
void display(int array[],int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d\t",array[i]);

    printf("\n");
}

```

7.11 COMPARISON OF Sorting ALGORITHMS

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	Requires Memory
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	*Small constants

Algorithm	Average Case	Worst Case
Bubble sort	$O(n^2)$	$O(n^2)$
Bucket sort	$O(n.k)$	$O(n^2.k)$
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Shell sort	–	$O(n \log^2 n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n^2)$

7.12 External Sort

External sorting is a sorting technique that can handle massive amounts of data. It is usually applied when the data being sorted does not fit into the main memory (RAM) and, therefore, a slower memory (usually a magnetic disk or even a magnetic tape) needs to be used.

Example: Let us consider we need to sort 700 MB of data using only 100 MB of RAM.

The steps for sorting are given below.

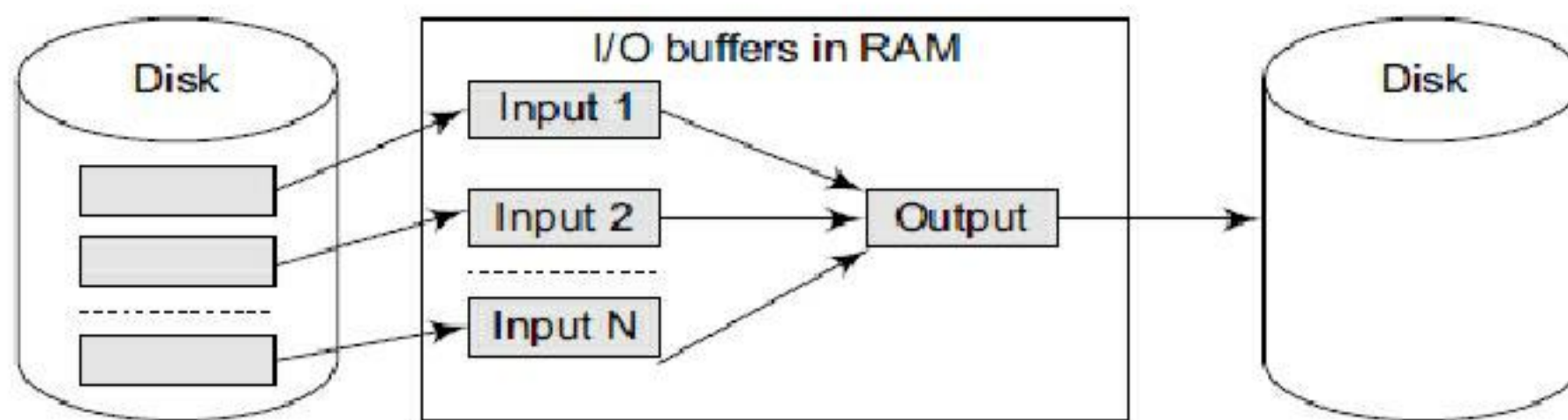
Step 1: Read 100 MB of the data in RAM and sort this data using any conventional sorting algorithm like quick sort.

Step 2: Write the sorted data back to the magnetic disk.

Step 3: Repeat Steps 1 and 2 until all the data (in 100 MB chunks) is sorted. All these seven chunks that are sorted need to be merged into one single output file.

Step 4: Read the first 10 MB of each of the sorted chunks and call them input buffers. So, now we have 70 MB of data in the RAM. Allocate the remaining RAM for output buffer.

Step 5: Perform seven-way merging and store the result in the output buffer. If at any point of time, the output buffer becomes full, then write its contents to the final sorted file. However, if any of the 7 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk or else mark the input buffer (sorted chunk) as exhausted if it does not have any more left with it. Make sure that this chunk is not used for further merging of data. The external merge sorting can be visualized as given in Figure.



Generalized External Merge Sort Algorithm

From the example above, we can now present a generalized merge sort algorithm for external sorting. If the amount of data to be sorted exceeds the available memory by a factor of K , then K chunks (also known as K run lists) of data are created. These K chunks are sorted and then a K -way merge is performed. If the amount of RAM available is given as X , then there will be K input buffers and 1 output buffer.

In the above example, a single-pass merge was used. But if the ratio of data to be sorted and available RAM is particularly large, a multi-pass sorting is used. We can first merge only the first half of the sorted chunks, then the other half, and finally merge the two sorted chunks. The exact number of passes depends on the following factors:

- Size of the data to be sorted when compared with the available RAM
- Physical characteristics of the magnetic disk such as transfer rate, seek time, etc.

Applications of External Sorting

External sorting is used to update a master file from a transaction file. For example, updating the EMPLOYEES file based on new hires, promotions, appraisals, and dismissals.

It is also used in database applications for performing operations like Projection and Join. Projection means selecting a subset of fields and join means joining two files on a common field to create a new file whose fields are the union of the fields of the two files. External sorting is also used to remove duplicate records.