Pune Vidyarthi Griha's

COLLEGE OF ENGINEERING, NASHIK – 3.

" Searching & Sorting "

By

Prof. Anand N. Gharu

(Assistant Professor)

PVGCOE Computer Dept.

10 September 2019

Introduction

Searching :

“ Searching is a techniques of finding an element in a given list of elements.”

List of element could be represented using an

1. Array
2. Linked list
3. Binary tree
4. B-tree
5. Heap

Why do we need searching?

✓Searching is one of the core computer science algorithms.

✓We know that today's computers store a lot of information.

✓To retrieve this information proficiently we need very efficient searching algorithms.

Types of Searching

- *Linear search*
- *Binary search*
- *Sentinel search*

SEARCH TECHNIQUES

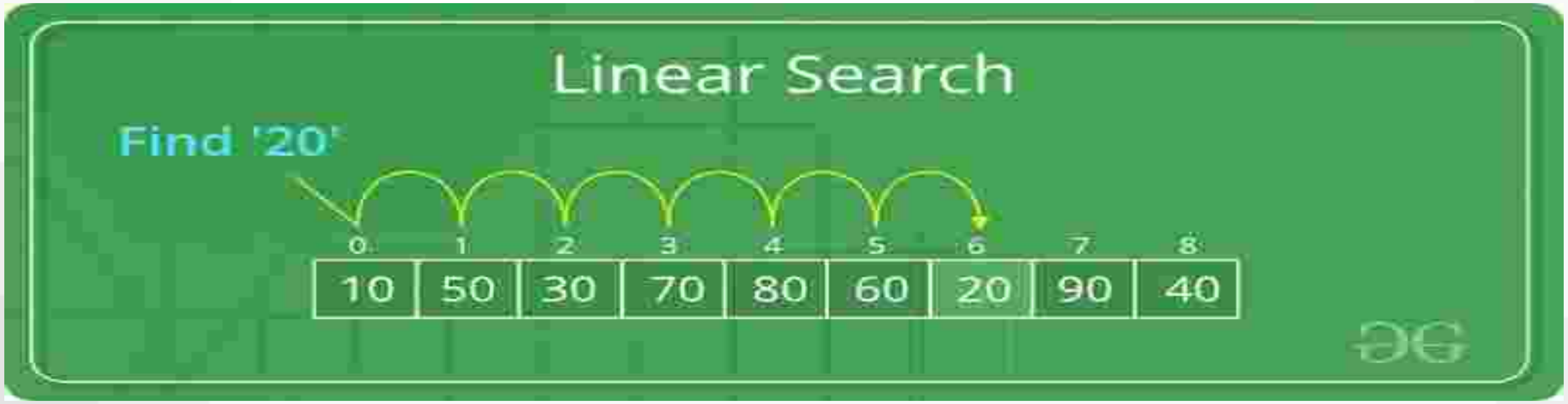
- ***Sequential search***
- ***Binary search***
- ***Fibonacci search***
- ***Hashed search***
- ***Index sequential search***

Linear Search

- The linear search is a sequential search, which uses a loop to step through an array, starting with the first element.
- It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered.
- If the value being searched is not in the array, the algorithm will unsuccessfully search to the end of the array.

Linear Search

- Since the array elements are stored in linear order searching the element in the linear order make it easy and efficient.
- The search may be successful or unsuccessfully. That is, if the required element is found them the search is successful other wise it is unsuccessfully.



Advantages Linear Search

- easy to understand.
- It operates on both sorted and unsorted list
- It doest not require array to be in order
- Easy to implement
- Time complexity O(n)

Disadvantages Linear Search

- Linear search is not efficient when list is large
- Maximum no. of comparision are N(n Element).
- Not suitable for large problem.
- You need to search whole list.
- Linear search is slower.

Linear Search Algorithm

Consider an integer type array A with size n. So list of elements from that array are,

A[0], A[1], A[2], A[3],………………, A[n-1]

1. Declare and initialize one variable which contents the number to be search in an array A.

   (variable **key** is declared)

2. Start Comparing each element from array A with the key

   LOOP: A[size] == key

   Repeat step no 2 while A[size]   key

3. if key is found, display the location of element(index+1)

   or else display message KEY NOT FOUND

4. Terminate the program successfully

Linear Search Algorithm

```
printf("accept  number to search");
scanf( key );
 for( i=0 ;i<n ;i++)
   {
        if( A [ i ] == key )
          {
    printf( key is FOUND);
    break;
    }
    }
if( i==n )
  {
     printf(NOT FOUND);
  }
```

Analysis of Linear Search

Complexity of linear search :

1. Best Case = O(1)
2. Average Case = O(n)
3. Worst case = O(n)

Binary Search

“Binary search is an searching algorithm which is used to find element from the sorted list”

Concepts :

- Algorithm can be applied only on sorted data
- Mid = lower/upper - formula used to find mid
- Given element is compared with middle element of the list.
- If key=mid then element is found
- Otherwise list divide into two part.(key <mid) (>mid)
- First to mid-1 or mid+1 to last.

Binary Search

Concepts :

- If given element is less than middle element then continue searching in first part (first+mid-1) otherwise searching is second part(mid+1 to last).
- Repeat above step still element is found.

Binary Search

Assume that two variables are declared, variable first and last, they denotes beginning and ending indices of the list under consideration respectively.

Step 1. Algorithm compares key with middle element from list (A[middle] == key), if true go to step 4 or else go to next.

Step 2. if key < A[middle], search in left half of the list or else go to step 3

Step 3. if key > A[middle], search in right half of the list or go to step 1

Step 4. display the position of key else display message "NOT FOUND"

Binary Search algorithm

```
int i, first=0, last=n-1, middle;
 while( last>=first )
   {
    middle = (first + last)/2;

     if( key > A[middle] )
       { first = middle + 1; }
          else if ( key < A[middle] )
       { last= middle - 1; }
          else
           { printf( FOUND ) }
    }
if( last < first )
  {
    printf( NOT FOUND );
  }
```

Advatages Binary Search

1. Binary search is optimal searching algorithms
2. Excellent time efficiency
3. Suitable for large list.
4. Faster because no need to check all element.
5. Most suitable for sorted array
6. It can be search quickly
7. Time complexity O(log n)

Disadvatages Binary Search

1. Element must be sorted

2. Need to find mid element

3. Bit more complicated to implement and test

4. It does not support random access.

5. Key element require to compare with middle.

Linear Search Vs Binary Search

- Element is searched by scanning the entire list from first element to the last
- Many times entire list is search
- Simple to implementation
- Time complexity is O(n)
- Less efficient sort

- First list is divided into two sub-lists. Then middle element is compared with key element and then accordingly left or right sub-list is searched
- Only sub-list is search
- Complex to implement, since it involves computation for finding the middle element
- Time complexity is $O(\log_2 n)$
- More efficient sort

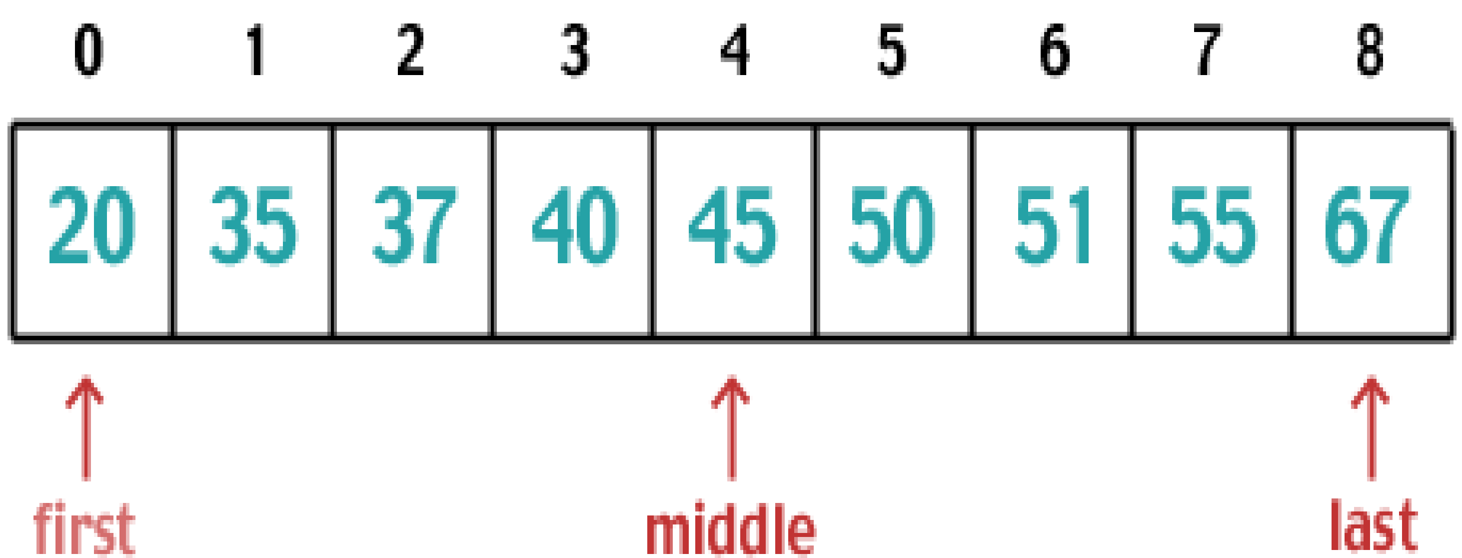
Binary Search

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Binary Search

2. Calculate middle = (low + high) / 2.

= (0 + 8) / 2 = 4

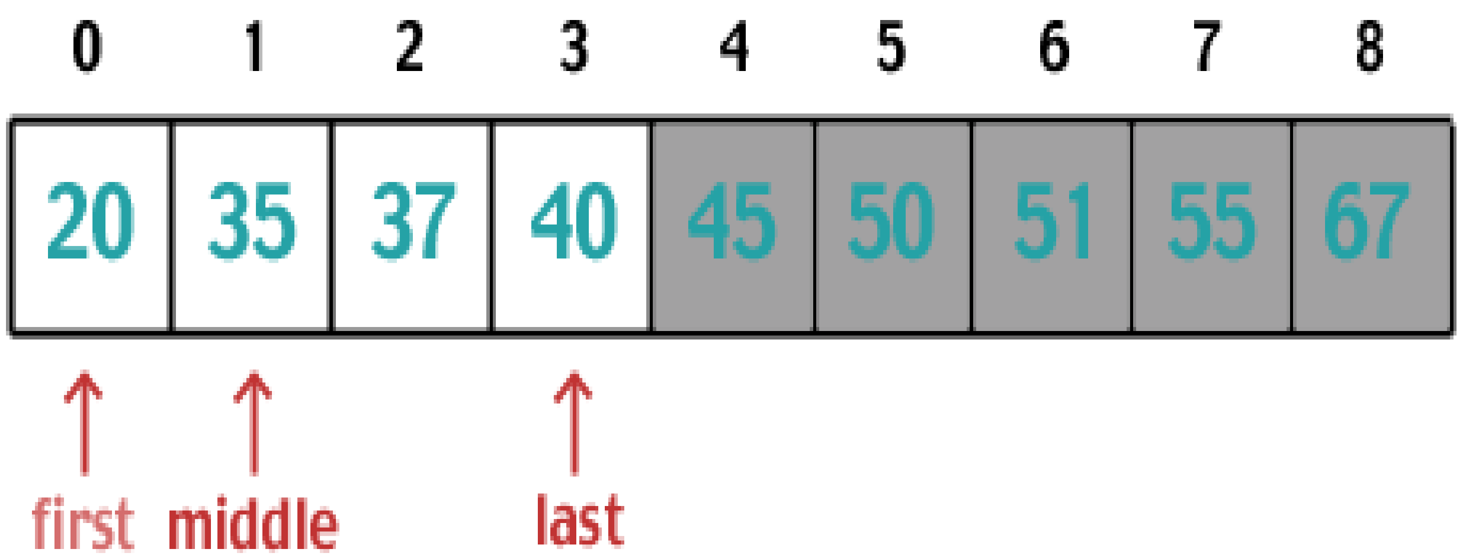


If 37 == array[middle] □ return middle
Else if 37 < array[middle] □ high = middle -1
Else if 37 > array[middle] □ low = middle +1

Binary Search

Repeat 2. Calculate middle = (low + high) / 2.

= (0 + 3) / 2 =



If 37 == array[middle] □ return middle

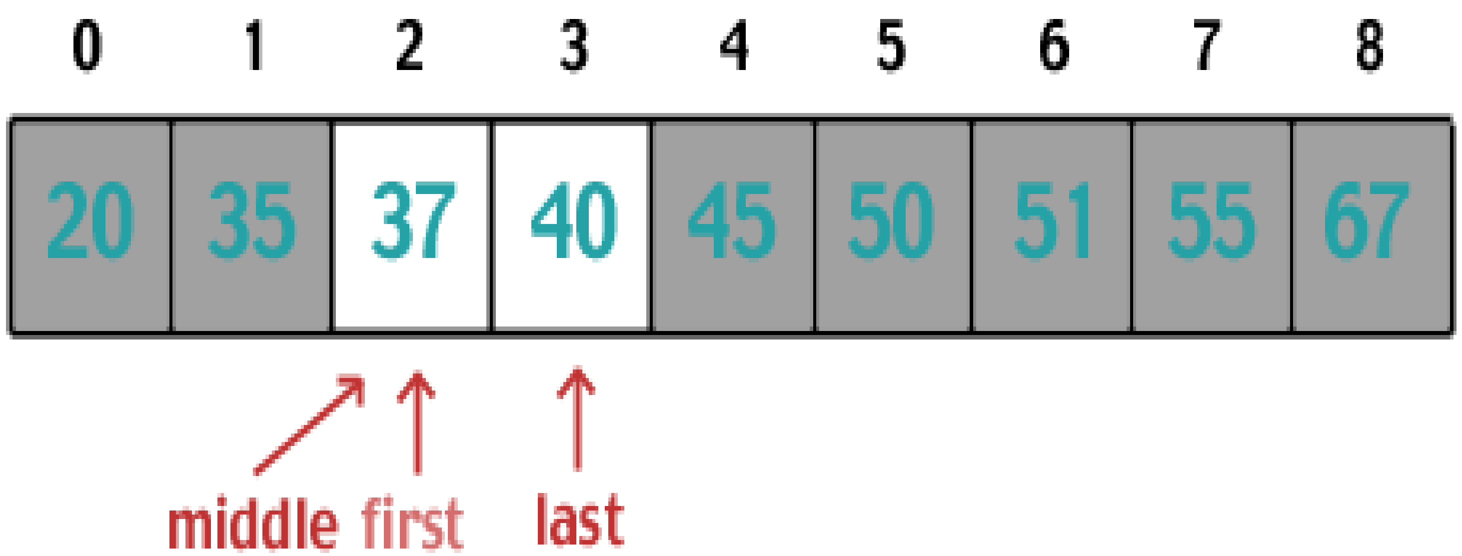
Else if 37 < array[middle] □ high = middle -1

Else if 37 > array[middle] □ low = middle +1

Binary Search

Repeat 2. Calculate middle = (low + high) / 2.

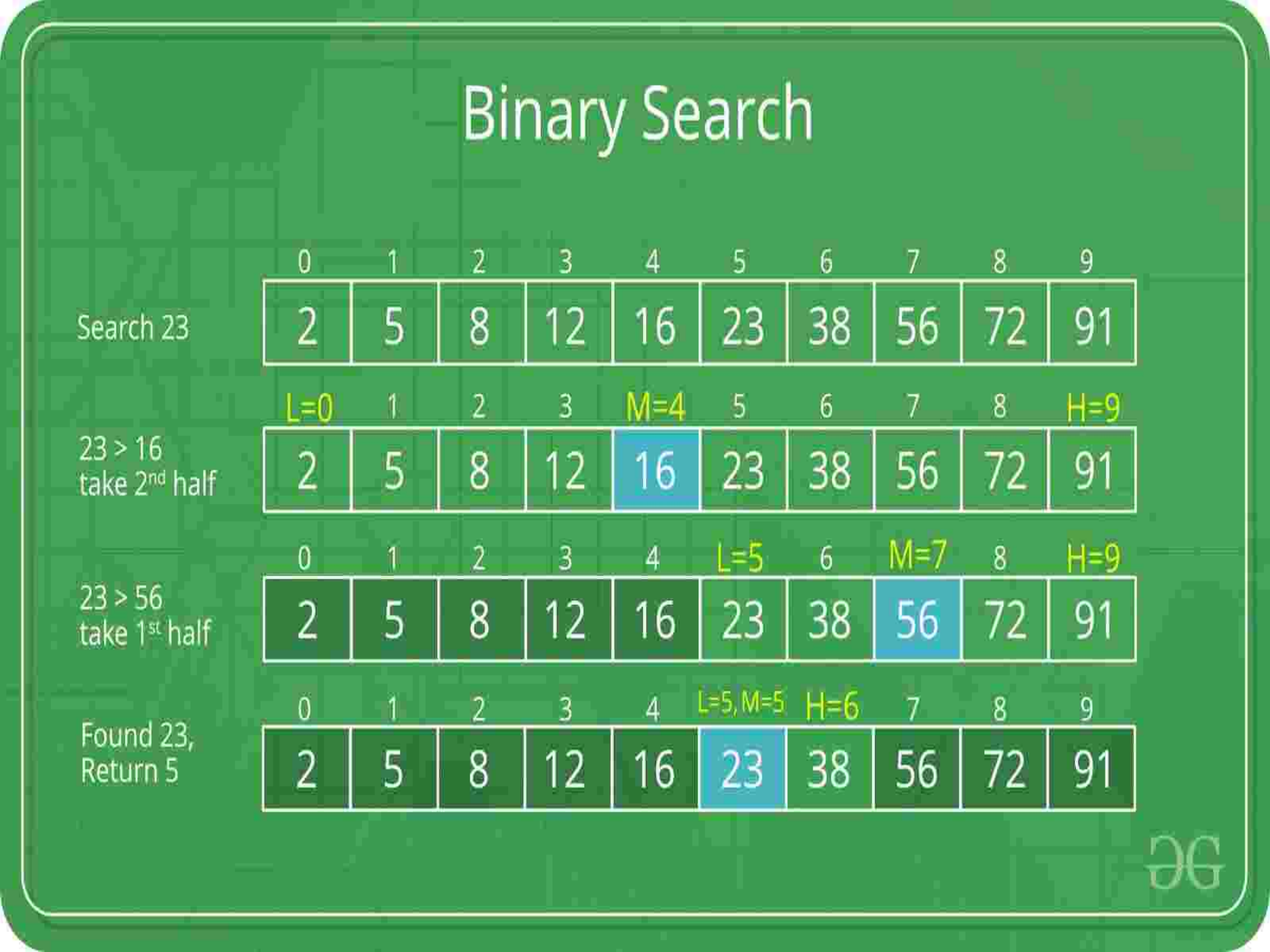
= (2 + 3) / 2 =



If 37 == array[middle] □ return middle

Else if 37 < array[middle] □ high = middle -1

Else if 37 > array[middle] □ low = middle +1


Binary Search
0 1 2 3 4 5 6 7 8 9
Search 23
2 5 8 12 16 23 38 56 72 91
L=0 1 2 3 M=4 5 6 7 8 H=9
23 > 16
take 2nd half
2 5 8 12 16 23 38 56 72 91
0 1 2 3 4 L=5 6 M=7 8 H=9
23 > 56
take 1st half
2 5 8 12 16 23 38 56 72 91
0 1 2 3 4 L=5, M=5 H=6 7 8 9
Found 23,
Return 5
2 5 8 12 16 23 38 56 72 91

Example 7.3.1

[Searching the element 29 in the given array]

Index	0	1	2	3	4	5	6	7	8
Elements	5	9	11	15	25	29	30	35	40
	↑				↑				↑
	$i = 0$				$c = \frac{i+j}{2} = \frac{0+8}{2} = 4$				$j = 8$

Solution :

Element to be searched, key = 29

Step 1 : Since key > a[c] (29>25) right half is selected.

index	5	6	7	8
elements	29	30	35	40
	↑	↑		↑
	$i = 5$	$c = 6$		$j = 8$

Step 2 : Since key < a[c] (29 < 30) left half is selected.

Index	5	element is found and search is successful.
element	29	
	↑	
	$i = 5$, $c = 5$, $j = 5$	

Example 7.3.3

Apply binary search on the following numbers stored in array from

A [0] to A [10]

9, 17, 23, 38, 45, 50, 57, 76, 79, 90, 100 to search numbers

~ 10 to 100.

Solution :

Searching 10

0	1	2	3	4	5	6	7	8	9	10						
9	17	23	38	45	50	57	76	79	90	100				i	j	k
↑					↑					↑				0	10	5
i					k					j						

Step 1 : Since $10 < A[5]$, $j = K - 1 = 4$

0	1	2	3	4												i	j	k
9	17	23	38	45												0	4	2
↑		↑		↑														
i		k		J														

Step 2 : Since $10 < A[2]$, $j = K - 1 = 1$

0	1									
9	17							i	j	k
↑ ↑	↑							0	1	0
i k	j									

Step 3 : Since $10 > A[0]$, $j = K + 1 = 1$

				i	j	k
1				1	1	1
17						
↑ ↑ ↑						
i j k						

Step 4 : Since $10 < A[1]$, $j = K - 1 = 0$

As i becomes less than j, element 10 is not in the array A [ ]

Searching 100

0	1	2	3	4	5	6	7	8	9	10				
9	17	23	38	45	50	57	76	79	90	100		i	j	K
↑					↑					↑		0	10	5
i					k					j				

Step 1 : Since 100 > A [5], i = K + 1 = 6

	7	8	9	10										
57	76	79	90	100								i	j	K
↑		↑		↑								6	10	8
i		k		j										

Step 2 : Since 100 > A [8], i = K + 1 = 9

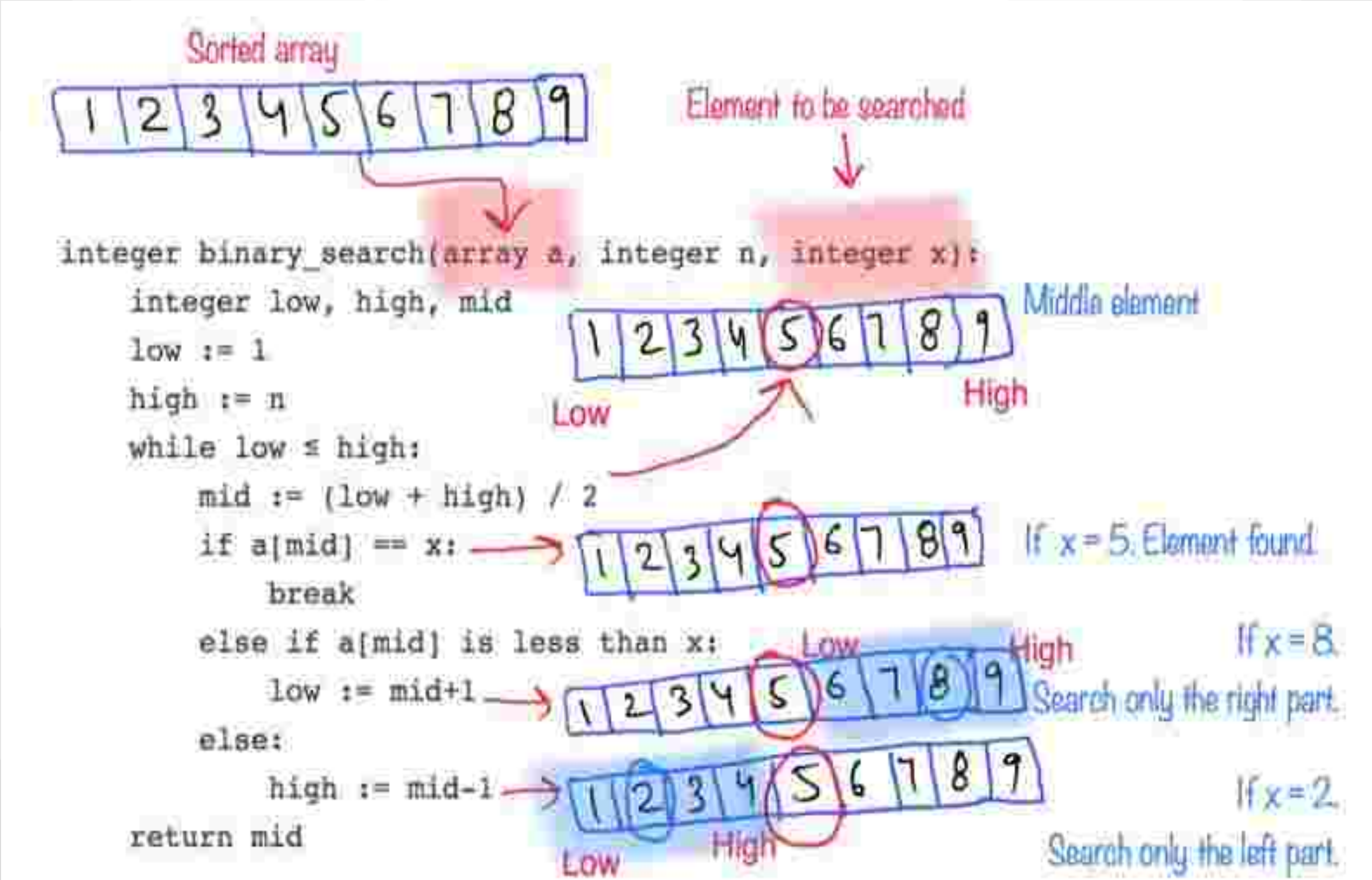
9	10													
90	100											i	j	K
↑ ↑	↑											9	10	9
i k	j													

Step 3 : Since 100 > A [9], i = K + 1 = 10

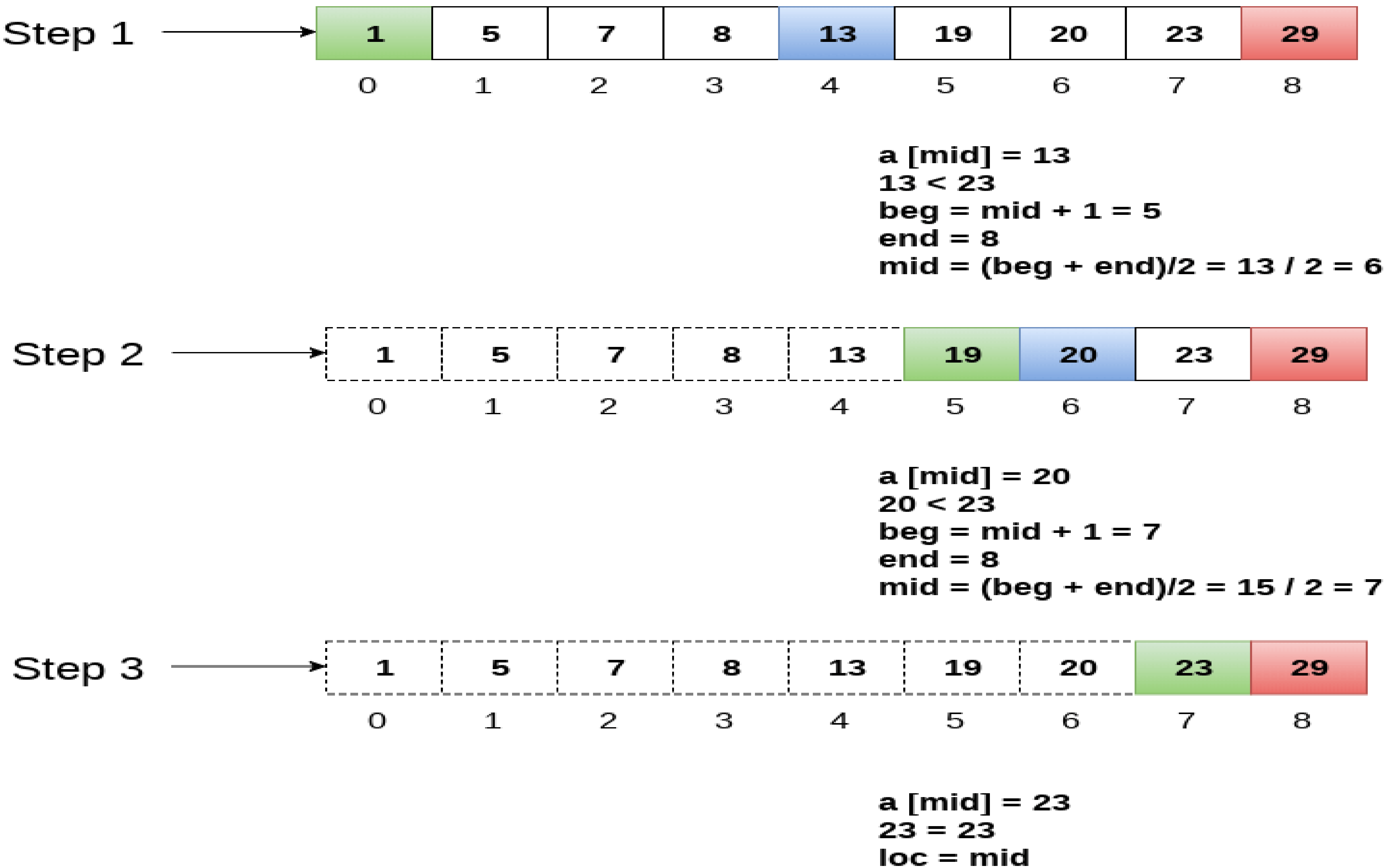
10														
100												i	j	K
↑ ↑ ↑												10	10	10
i j k														

Since, the element to be searched is found at A [10], search terminates with a success.

Binary Search Routine



Item to be searched = 23



Return location 7

Sentinel search

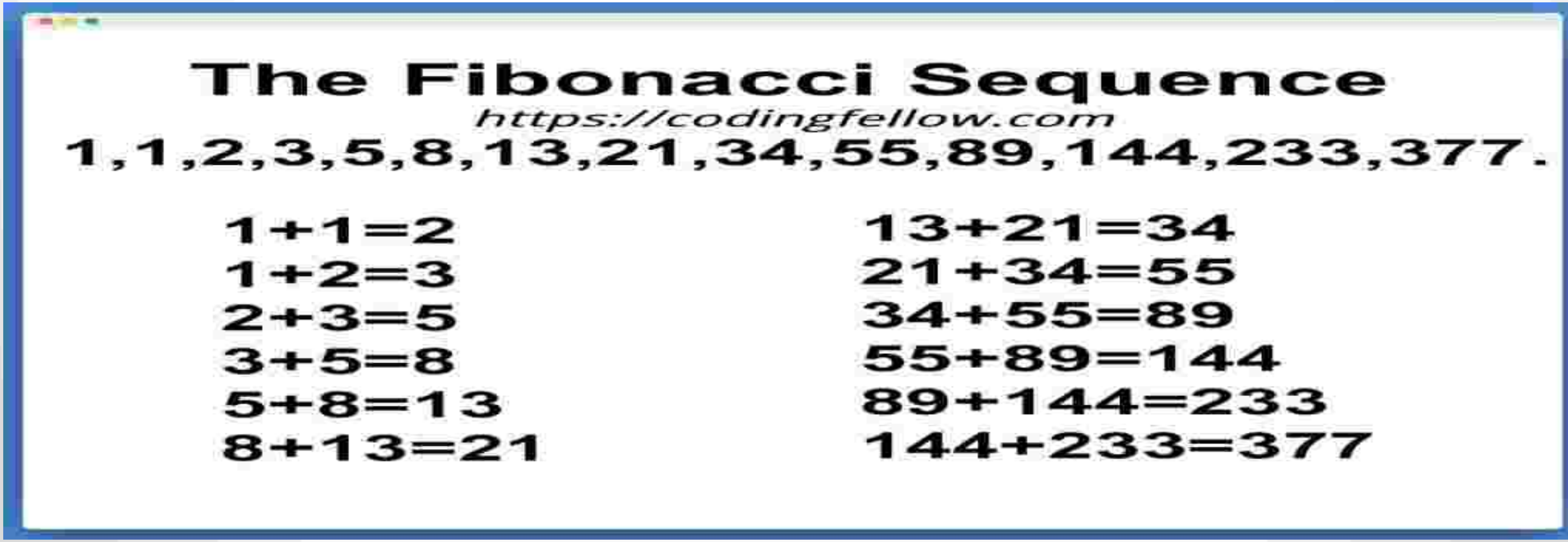
- This additional entry at the end of the list is called as Sentinel.
- The speed of sequential search can be improved by storing the key being searched at end of the array.
- This will eliminate extra comparision inside the loop for number od element in the array.

The example is given below.

```
int sentinel search (int a[], int n , int key)
    {
            int i;
            a[n] = key;
                 i = 0;
            while (a[i] ! = key)
                 i++;
            return(i);
    }
```

Fibonacci search

- Fibonacci search technique is a method of searching a sorted array using a divide and conquer algorithm that narrows down possible locations with the aid of Fibonacci numbers. Compared to binary search where the sorted array is divided into two equal-sized parts, one of which is examined further, Fibonacci search divides the array into two parts that have sizes that are consecutive Fibonacci numbers.



Fibonacci search

- Fibonacci search changes the binary search algorithm slightly
- Instead of halving the index for a search, a Fibonacci number is subtracted from it
- The Fibonacci number to be subtracted decreases as the
  - size of the list decreases
- Note that Fibonacci search sorts a list in a non decreasing order
- Fibonacci search starts searching the target by comparing
  - it with the element at Fkth location

Cases in Fibonacci search

Case 1: if equal the search terminates;

Case 2: if the target is greater and F_1 is 1, then the search terminates with an unsuccessful search;

else the search continues at the right of list with new values of low, high, and mid as

$\mathtt{mid} = \mathtt{mid} + F_2$, $F_1 = F_{k-4}$ and $F_2 = F_{k-5}$

Case 3: if the target is smaller and F_2 is 0, then the search terminates with unsuccessful search;

else the search continues at the left of list with new values of low, high, and mid as

$\mathtt{mid} = \mathtt{mid} - F_2$, $F_1 = F_{k-3}$ and $F_2 = F_{k-4}$

The search continues by either searching at the left of mid or at the right of mid in the list.

Fibonacci search

Algorithm

Given a table of records R1, R2, …, RN whose keys are in increasing order K1 < K2 < … < KN, the algorithm searches for a given argument K. Assume N+1 = Fk+1

Step 1. [Initialize] i ← Fk, p ← Fk-1, q ← Fk-2 (throughout the algorithm, p and q will be consecutive Fibonacci numbers)

Step 2. [Compare] If K < Ki, go to Step 3; if K > Ki go to Step 4; and if K = Ki, the algorithm terminates successfully.

Step 3. [Decrease i] If q=0, the algorithm terminates unsuccessfully. Otherwise set (i, p, q) ← (p, q, p - q) (which moves p and q one position back in the Fibonacci sequence); then return to Step 2

Step 4. [Increase i] If p=1, the algorithm terminates unsuccessfully. Otherwise set (i,p,q) ← (i + q, p - q, 2q - p) (which moves p and q two positions back in the Fibonacci sequence); and return to Step 2

SORTING

- "Sorting is the process ordering a list of element in either ascending or descending order."

- Sorting is the operation of arranging the records of a table according to the key value of each record, or it can be defined as the process of converting an unordered set of elements to an ordered set of elements
- Sorting is a process of organizing data in a certain order to help retrieve it more efficiently

INTERNAL SORTING(types)

- Any sort algorithm that uses main memory exclusively during the sorting is called as internal sort algorithm
- Internal sorting is faster than external sorting

Internal Sorting techniques :

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort
5. Shell sort
6. Heap sort
7. Radix sort
8. Bucket sort

EXTERNAL SORTING

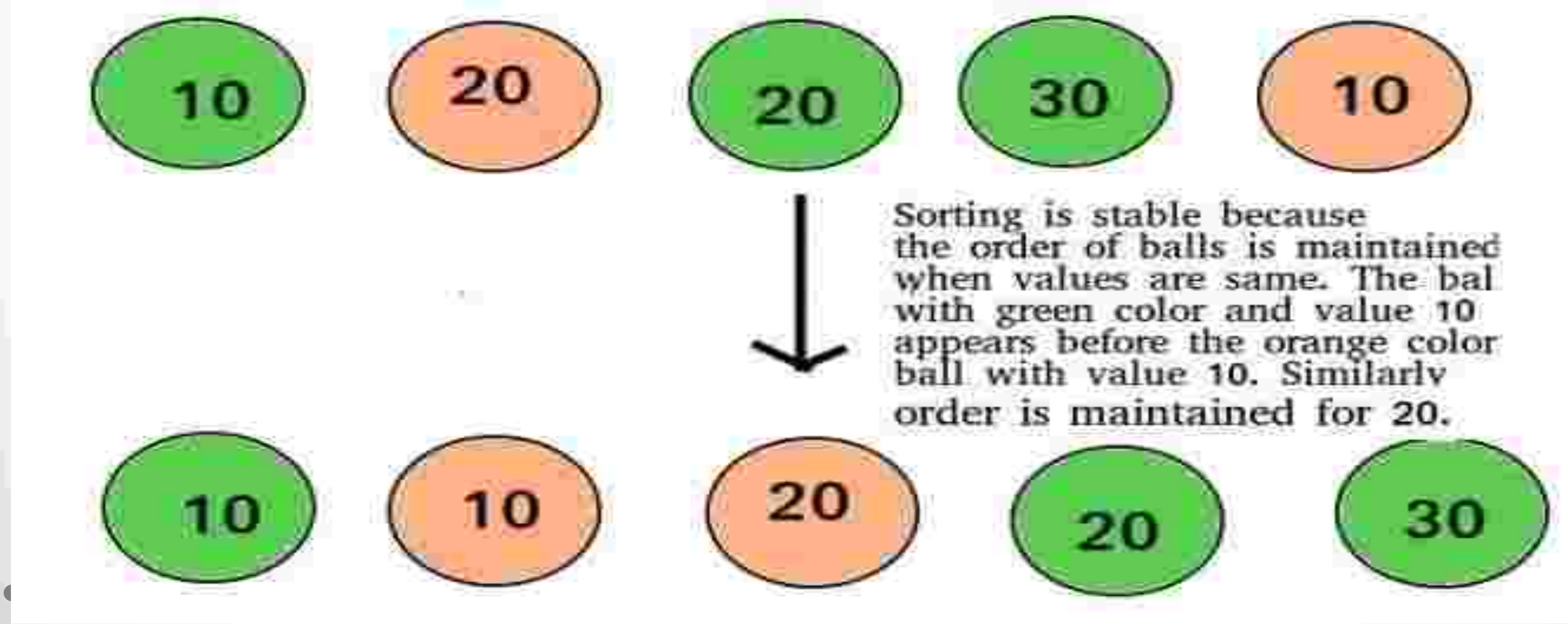
- Any sort algorithm that uses external memory, such as tape or disk, during the sorting is called as **external sort algorithm**

- **Merge sort** is used in external sorting

STABILITY OF SORTING

- A sorting method is said to be stable if at the end of the method, identical elements occur in the same relative order as in the original unsorted set

1. EXAMPLE :



SORT EFFICIENCY

- Sort efficiency is a measure of the relative efficiency of a sort
- It is usually an estimate of the number of comparisons and data movement required to sort the data

PASSES IN SORTING

- During the sorted process, the data is traversed many times
- Each traversal of the data is referred to as a sort pass
- In addition, the characteristic of a sort pass is the placement of one or more elements in a sorted list

BUBBLE SORTING

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

- We take an unsorted array for our example. Bubble sort takes

  $O(n^2)$ time so we're keeping it short and precise.

  Bubble sort start with first two element, compare them to check which one is greater. And swap it.

Algorithm Bubble sorting

1. In Bubble sort pairs of adjacent elements (start from 0^{th} and 1^{st} locations) is compared and then swapping is performed when first element is greater than another element in pair.
2. Repeat step 1 until (n - 2) position element is compared with (n - 1) position element.
3. Here first iteration is completed and largest value in list is stored at (n - 1) location.
4. Now start second iteration, Repeat step 1 until (n - 3) position element is compared with (n - 2) position element.
5. If there are n elements, then (n - 1) passes are required.

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

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

The new array should look like this –

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

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

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

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

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

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

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

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

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Implementation in C

Try it

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

#define MAX 10

int list[MAX] = {1,8,4,6,0,3,5,2,7,9};

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0; i < MAX; i++) {
      printf("%d ",list[i]);
   }

   printf("]\n");
}

void bubbleSort() {
   int temp;
   int i,j;

   bool swapped = false;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++) {
      swapped = false;
```

```
    // loop through numbers falling ahead
    for(j = 0; j < MAX-1-i; j++) {
       printf("     Items compared: [ %d, %d ] ", list[j],list[j+1]);

       // check if next number is lesser than current no
       //   swap the numbers.
       //  (Bubble up the highest number)

       if(list[j] > list[j+1]) {
          temp = list[j];
          list[j] = list[j+1];
          list[j+1] = temp;

          swapped = true;
          printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
       }else {
          printf(" => not swapped\n");
       }

    }

    // if no number was swapped that means
    //   array is sorted now, break the loop.
    if(!swapped) {
       break;
    }

    printf("Iteration %d#: ",(i+1));
    display();
  }

}
```

```
main() {
   printf("Input Array: ");
   display();
   printf("\n");

   bubbleSort();
   printf("\nOutput Array: ");
   display();
}
```

If we compile and run the above program, it will produce the following result −

Output

```
Input Array: [1 8 4 6 0 3 5 2 7 9 ]


     Items compared: [ 1, 8 ]  => not swapped
     Items compared: [ 8, 4 ]  => swapped [4, 8]
     Items compared: [ 8, 6 ]  => swapped [6, 8]
     Items compared: [ 8, 0 ]  => swapped [0, 8]
     Items compared: [ 8, 3 ]  => swapped [3, 8]
     Items compared: [ 8, 5 ]  => swapped [5, 8]
     Items compared: [ 8, 2 ]  => swapped [2, 8]
     Items compared: [ 8, 7 ]  => swapped [7, 8]
     Items compared: [ 8, 9 ]  => not swapped
```

```
Iteration 1#: [1 4 6 0 3 5 2 7 8 9 ]
      Items compared: [ 1, 4 ]  => not swapped
      Items compared: [ 4, 6 ]  => not swapped
      Items compared: [ 6, 0 ]  => swapped [0, 6]
      Items compared: [ 6, 3 ]  => swapped [3, 6]
      Items compared: [ 6, 5 ]  => swapped [5, 6]
      Items compared: [ 6, 2 ]  => swapped [2, 6]
      Items compared: [ 6, 7 ]  => not swapped
      Items compared: [ 7, 8 ]  => not swapped
Iteration 2#: [1 4 0 3 5 2 6 7 8 9 ]
      Items compared: [ 1, 4 ]  => not swapped
      Items compared: [ 4, 0 ]  => swapped [0, 4]
      Items compared: [ 4, 3 ]  => swapped [3, 4]
      Items compared: [ 4, 5 ]  => not swapped
      Items compared: [ 5, 2 ]  => swapped [2, 5]
      Items compared: [ 5, 6 ]  => not swapped
      Items compared: [ 6, 7 ]  => not swapped
Iteration 3#: [1 0 3 4 2 5 6 7 8 9 ]
      Items compared: [ 1, 0 ]  => swapped [0, 1]
      Items compared: [ 1, 3 ]  => not swapped
      Items compared: [ 3, 4 ]  => not swapped
      Items compared: [ 4, 2 ]  => swapped [2, 4]
      Items compared: [ 4, 5 ]  => not swapped
      Items compared: [ 5, 6 ]  => not swapped
```

```
Iteration 4#: [0 1 3 2 4 5 6 7 8 9 ]
      Items compared: [ 0, 1 ]  => not swapped
      Items compared: [ 1, 3 ]  => not swapped
      Items compared: [ 3, 2 ]  => swapped [2, 3]
      Items compared: [ 3, 4 ]  => not swapped
      Items compared: [ 4, 5 ]  => not swapped
Iteration 5#: [0 1 2 3 4 5 6 7 8 9 ]
      Items compared: [ 0, 1 ]  => not swapped
      Items compared: [ 1, 2 ]  => not swapped
      Items compared: [ 2, 3 ]  => not swapped
      Items compared: [ 3, 4 ]  => not swapped


Output Array: [0 1 2 3 4 5 6 7 8 9 ]
```

Original array with n = 6 5 9 6 2 8 1

First pass i = 1

Comparisons							
	j = 0	5	9	6	2	8	1
	j = 1	5	9	6	2	8	1
	j = 2	5	6	9	2	8	1
	j = 3	5	6	2	9	8	1
	j = 4	5	6	2	8	9	1

Second pass i = 2

Comparisons							
	j = 0	5	6	2	8	1	9
	j = 1	5	6	2	8	1	9
	j = 2	5	2	6	8	1	9
	j = 3	5	2	6	8	1	9

Third pass i = 3

Comparisons							
	j = 0	5	2	6	1	8	9
	j = 1	2	5	6	1	8	9
	j = 2	2	5	6	1	8	9

Fourth pass i = 4

Comparisons							
	j = 0	2	5	6	1	8	9
	j = 1	2	5	1	6	8	9

Fifth pass i = 5

Comparisons							
	j = 0	2	1	5	6	8	9

Sorted array

a[ ] → 1 2 5 6 8 9 9

Example 8.3.1

Show output of each pass using bubble sort to arrange the following nos. in ascending order. Write pseudo C code for bubble sort : 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

Solution :

Pass No.	Data at the end of the pass
1.	9, 8, 7, 6, 5, 4, 3, 2, 1, 10
2.	8, 7, 6, 5, 4, 3, 2, 1, 9, 10
3.	7, 6, 5, 4, 3, 2, 1, 8, 9, 10
4.	6, 5, 4, 3, 2, 1, 7, 8, 9, 10
5.	5, 4, 3, 2, 1, 6, 7, 8, 9, 10
6.	4, 3, 2, 1, 5, 6, 7, 8, 9, 10
7.	3, 2, 1, 4, 5, 6, 7, 8, 9, 10
8.	2, 1, 3, 4, 5, 6, 7, 8, 9, 10
9.	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Write pseudo C code to sort a list of integers using bubble sort. Show output of each pass for the following list :

10, 5, 4, 18, 17, 1, 2.

Solution :

Pass – I, I = 1 :

Pass – I , i = 1

j = 0	10	5	4	18	17	1	2
j = 1	5	10	4	18	17	1	2
j = 2	5	4	10	18	17	1	2
j = 3	5	4	10	18	17	1	2
j = 4	5	4	10	17	18	1	2
j = 5	5	4	10	17	1	18	2
	5	4	10	17	1	2	18

Pass – II, I = 2

Pass – II , i = 2

j = 0	5	4	10	17	1	2	18
j = 1	4	5	10	17	1	2	18
j = 2	4	5	10	17	1	2	18
j = 3	4	5	10	17	1	2	18
j = 4	4	5	10	1	17	2	18
	4	5	10	1	2	17	18

Pass – III, I = 3

Pass – III , i = 3

j = 0	4	5	10	1	2	17	18
j = 1	4	5	10	1	2	17	18
j = 2	4	5	10	1	2	17	18
j = 3	4	5	1	10	2	17	18
	4	5	1	2	10	17	18

Pass – IV, I = 4

Pass - IV , i = 4

j = 0	4	5	1	2	10	17	18
j = 1	4	5	1	2	10	17	18
j = 2	4	1	5	2	10	17	18
	4	1	2	5	10	17	18

Pass – V, I =5

Pass - V , i = 5

j = 0	4	1	2	5	10	17	18
j = 1	1	4	2	5	10	17	18
	1	2	4	5	10	17	18

Pass – VI, I = 4

Pass - VI , i = 6

j = 0	1	2	4	5	10	17	18
	1	2	4	5	10	17	18

Sort the following list in ascending order using bubble sort. Show all passes. Analyze time complexity.

9, 7, – 2, 4, 5, 3, – 6, 2, 1, 8

Solution :

Pass 1 :

9	7	–2	4	5	3	–6	2	1	8
7	9	–2	4	5	3	–6	2	1	8
7	–2	9	4	5	3	–6	2	1	8
7	–2	4	9	5	3	–6	2	1	8
7	–2	4	5	9	3	–6	2	1	8
7	–2	4	5	3	9	–6	2	1	8
7	–2	4	5	3	–6	9	2	1	8
7	–2	4	5	3	–6	2	9	1	8
7	–2	4	5	3	–6	2	1	9	8
7	–2	4	5	3	–6	2	1	8	9

Show the output of each pass using bubble sort to arrange the following numbers in ascending order.

90, 87, 76, 65, 43, 32, 19, 7, 0, – 17.

Solution :

Pass-I

90	87	76	65	43	32	19	7	0	–17
87	90	76	65	43	32	19	7	0	–17
87	76	90	65	43	32	19	7	0	–17
87	76	65	90	43	32	19	7	0	–17
87	76	65	43	90	32	19	7	0	–17
87	76	65	43	32	19	7	0	–17	90

Pass-II

87	76	65	43	32	19	7	0	–17	90
76	87	65	43	32	19	7	0	–17	90
76	65	43	32	19	7	0	–17	87	90

Insertion Sort

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

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

14	33	27	10	35	19	42	44

Insertion sort compares the first two elements.

14	33	27	10	35	19	42	44

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

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

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

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

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

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

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

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

ALGORITHM OF INSERTION SORT

1. First iteration starts with comparison of 1^{st} location element with 0^{th} location element in the list, if 1^{st} location element is less then it is inserted at 0^{th} location and at 0^{th} location element is moved one position right with all next elements.

2. Like that, each element in the list is compared with all previous elements, If the element is less than any previous element then the element is inserted at position of previous small element and the position of that previous element shifted one position to right.

3. The same procedure is repeated for all the elements in list.

0		1	2	3	4	5	6	Initial unsorted list
5		0	1	9	2	6	4	

A list of sorted element (a list of single element is always sorted)

a list of unsorted element

1st iteration (place element at location '1' i.e. a[1], at its correct place)

0	1		2	3	4	5	6
0	5		1	9	2	6	4

Sorted unsorted

2nd iteration (place a[2] at its correct place)

0	1	5		9	2	6	4

Sorted unsorted

3rd iteration (place a[3] at its correct place)

0	1	5	9		2	6	4

Sorted unsorted

4th iteration (Place a[4] at its correct place)

0	1	2	5	9		6	4

sorted Unsorted

5th iteration (Place a[5] at its correct place)

0	1	2	5	6	9		4

Sorted unsorted

6th iteration (Place a[6] at its correct place)

0 1 2 4 5 6 9

Fig. 8.2.1 : Sorting of elements using insertion sort

Elements	20	10	8	6	4	2	1	–	initially
								pass	Positions moved
	10	20	8	6	4	2	1	1	1
	8	10	20	6	4	2	1	2	2
	6	8	10	20	4	2	1	3	3
	4	6	8	10	20	2	1	4	4
	2	4	6	8	10	20	1	5	5
	1	2	4	6	8	10	20	6	6

Show all passes to sort the values in descending order using insertion sort.

56,12,84,56,28,0,–13,47,94,31,12,– 2

Solution :

Initial	56	12	84	56	28	0	–13	47	94	31	12	–2
After pass 1	56	12	84	56	28	0	–13	47	94	31	12	–2
After pass 2	84	56	12	56	28	0	–13	47	94	31	12	–2
After pass 3	84	56	56	12	28	0	–13	47	94	31	12	–2
After pass 4	84	56	56	28	12	0	–13	47	94	31	12	–2
After pass 5	84	56	56	28	12	0	–13	47	94	31	12	–2
After pass 6	84	56	56	28	12	0	–13	47	94	31	12	–2
After pass 7	84	56	56	47	28	12	0	–13	94	31	12	–2
After pass 8	94	84	56	56	47	28	12	0	–13	31	12	–2
After pass 9	94	84	56	56	47	31	28	12	0	– 13	12	–2
After pass 10	94	84	56	56	47	31	28	12	12	0	13	–2
After pass 11	94	84	56	56	47	31	28	12	12	0	– 13	–2

Example 8.2.2

Here are five integers 1, 7, 3, 2, 0. Sort them using insertion sort.

Solution :

Pass (i)	Comparisons (j)	List to sort	Remarks
		1 7 3 2 0	original list
i = 1	j = 0	1 7 3 2 0	7 > 1, therefore inner loop terminates
i = 2	j = 1	1 7 3 2 0	7 > 3, move 7 right
	j = 0	1 7 7 2 0	3 > 1, inner loop terminates
		1 3 7 2 0	insert 3
i = 3	j = 2 j = 1 j = 0	1 3 7 7 0 1 3 3 7 0 1 3 3 7 0 1 2 3 7 0	7 > 2, move 7 right 3 > 2, move 2 right 2 > 1, inner loop terminates insert 2
i = 4	j = 3 j = 2 j = 1 j = 0 j = – 1	1 2 3 7 7 1 2 3 3 7 1 2 2 3 7 1 1 2 3 7 1 1 2 3 7 0 1 2 3 7	7 > 0 3 > 0 2 > 0 1 > 0 j = – 1, inner loop terminates Insert 0

Hence the sorted list = {0, 1, 2, 3, 7}

Sort the following nos. using insertion sort. Show all passes :
50, 10, 78, 40, 30, 02, 04, 15.

Solution :

Initial list   50 | 10 78 40 30 02 04 15

sorted   To be sorted

After pass-1   10 50 | 78 40 30 02 04 15

sorted

After pass-2   10 50 78 | 40 30 02 04 15

sorted

After pass-3   10 40 50 78 | 30 02 04 15

Sorted

After pass-4   10 30 40 50 78 | 02 04 15

sorted

After pass-5   02 10 30 40 50 78 | 04 15

sorted

After pass-6   02 04 10 30 40 50 78 | 15

sorted

After pass-7   02 04 10 15 30 40 50 78

sorted

Selection Sort

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

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

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

14	33	27	10	35	19	42	44

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

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

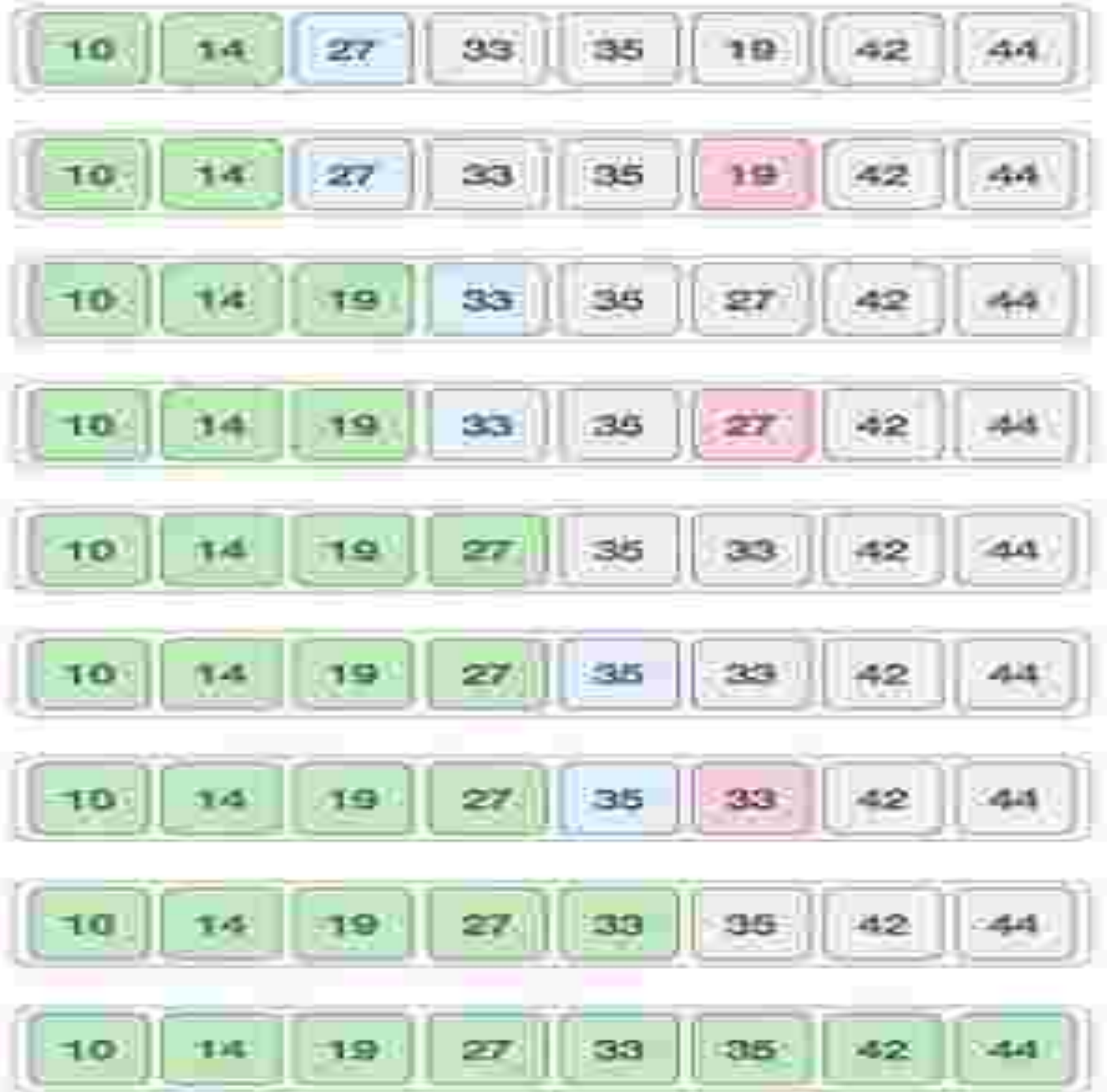
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

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

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

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

Following is a pictorial depiction of the entire sorting process −



Algorithm

```
Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted
```

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if

   end for

end procedure
```

6.12.1 Algorithm of Selection Sort

1. In first iteration first element is compared with rest of elements. If first element is greater than that then they are swapped.

2. After completion of first iteration smallest element is stored at 0^{th} location.

3. In second iteration second element is compared with rest of (3^{rd} to n^{th} location) elements and process of swapping is repeated.

4. If the list contains n elements, then (n - 1) iterations are required.

5	9	1	11	2	4	original array
1	9	5	11	2	4	After first pass
1	2	5	11	9	4	After second pass
1	2	4	11	9	5	After Third pass
1	2	4	5	9	11	After forth pass
1	2	4	5	9	11	After fifth pass

Illustration of selection sort

Show all the passes to sort the values in descending order :

84, 56, 28, 0, −13, 47, 94, 31.

Solution :

Original array	84	56	28	0	-13	47	94	31
After pass 1	94	56	28	0	-13	47	84	31
After pass 2	94	84	28	0	-13	47	56	31
After pass 3	94	84	56	0	-13	47	28	31
After pass 4	94	84	56	47	-13	0	28	31
After pass 5	94	84	56	47	31	0	28	-13
After pass 6	94	84	56	47	31	28	0	-13
After pass 7	94	84	56	47	31	28	0	-13

Example 8.4.4

Consider the following numbers sort them using "Selection sort". Show the output after each pass. 50, 20, 70, 40, 30

Solution :

Original data	50	20	70	40	30
After Pass I	20	50	70	40	30
After Pass II	20	30	70	40	50
After Pass III	20	30	40	70	50
After Pass IV	20	30	40	50	70

Sort the following list using selection sort. Show output each pass and write time complexity.

Data : 10,6,13,7,5,51,27,2,3,15,-3,4.

Solution :

Pass No.	Data											
Initially	10	6	13	7	5	51	27	2	3	15	–3	4
1	–3	6	13	7	5	51	27	2	3	15	10	4
2	–3	2	13	7	5	51	27	6	3	15	10	4
3	–3	2	3	7	5	51	27	6	13	15	10	4
4	–3	2	3	4	5	51	27	6	13	15	10	7
5	–3	2	3	4	5	51	27	6	13	15	10	7
6	–3	2	3	4	5	6	27	51	13	15	10	7
7	–3	2	3	4	5	6	7	51	13	15	10	27
8	–3	2	3	4	5	6	7	10	13	15	51	27
9	–3	2	3	4	5	6	7	10	13	15	51	27
10	–3	2	3	4	5	6	7	10	13	15	51	27
11	–3	2	3	4	5	6	7	10	13	15	27	51

Selection Sort

Pass 1	**75**	35	42	13	87	24	64	57	
Pass 2	13	**35**	42	75	87	**24**	64	57	
Pass 3	13	24	**42**	75	87	**35**	64	57	
Pass 4	13	24	35	**75**	87	**42**	64	57	
Pass 5	13	24	35	42	**87**	75	64	**57**	
Pass 6	13	24	35	42	57	**75**	**64**	87	
Pass 7	13	24	35	42	57	64	**75**	87	
Sorted elements		13	24	35	42	57	64	**75**	87

	Original unsorted array						
Elements	76	67	36	55	23	14	6
Index	0	1	2	3	4	5	6

Index	0	1	2	3	4	5	6	I	minpos
	76	67	36	55	23	14	6	0	6
Pass 1	6	67	36	55	23	14	76	1	5
Pass 2	6	14	36	55	23	67	76	2	4
Pass 3	6	14	23	55	36	67	76	3	4
Pass 4	6	14	23	36	55	67	76	4	4
Pass 5	6	14	23	36	55	67	76	5	5
Sorted array	6	14	23	36	55	67	76		

Quick Sort Algorithm

- Quick sort is based on divide-and-conquer strategy
  - Quick sort is thus in-place, divide-and-conquer based massively recursive sort technique
  - This technique reduces unnecessary swaps and moves the element at great distance in one move

Quick Sort Algorithm

- **The recursive algorithm consists of four steps:**
  - If there is one or less element in the array to be sorted, return immediately
  - Pick an element in the array to serve as a ‘pivot’ usually the left-most element in the list)
  - Partition the array into two parts—one with elements smaller than the pivot and the other with elements larger than the pivot by traversing from both the ends and performing swaps if needed
  - Recursively repeat the algorithm for both partitions

Quick Sort Algorithm

1. Lowest index element set as pivot element.
2. Take two index variable, i and j. i points to 1^{st} location element and j points to $(n-1)^{th}$ location element.
3. Index variable i is in search of element which is greater than pivot element. Here i will incremented by 1 till greater element is not found.
4. Index variable j is in search of element which is less than pivot element. Here j will be decremented by 1 till small element is not found.
5. If these two elements are found, they are swapped.
6. The process ends when these two variables are crossed or meet (In above example they are crossed). Then value at index j is swapped with pivot and list is divided into 2 sublists.
7. Above steps are repeated on these two sub arrays (sublists) until all sub arrays contain only 1 element.

Quick Sort Algorithm

Let the array of number be

0	1	2	3	4	5	6	7	8	9	10	11	12
30	35	10	15	20	34	5	18	6	11	13	26	38

Initially $l = 0$, $j = 12$, $V = a[l]$ {i.e. V = 30}

i is set to $l + 1$ and j is set to $n - 1$ as shown below

30	35	10	15	20	34	5	18	6	11	13	26	38

i = 1

j = 12

Now i moves right, while a[i] < 30. Hence i does not move further to right as a[1] > 30.

Then, j moves left, while a[j] > 30.

30	35	10	15	20	34	5	18	6	11	13	26	38

i=1

j=11

interchange

At this point a[i] and a[j] are interchanged and the movement of i and j resumes.

0	1	2	3	4	5							
30	26	10	15	20	34	5	18	6	11	13	35	38

i=5

j=10

i moves right to a[5] as 34 > 30. j moves left to a[10] as 13 < 30. Once again a[i] and a[j] are swapped.

0	1	2	3	4	5	6	7	8	9	10	11	12
30	26	10	15	20	13	5	18	6	11	34	35	38

j = 9

i = 10

Now, i moves to a[10] as 34 > 30 and j moves to a[9] as 11 < 30.

Since, i > j, the process ends. j gives the location of the pivot element. Pivot element a[l with l = 0] is interchanged with a[i] to partition the array.

11	26	10	15	20	13	5	18	6	30	34	35	38

left partition

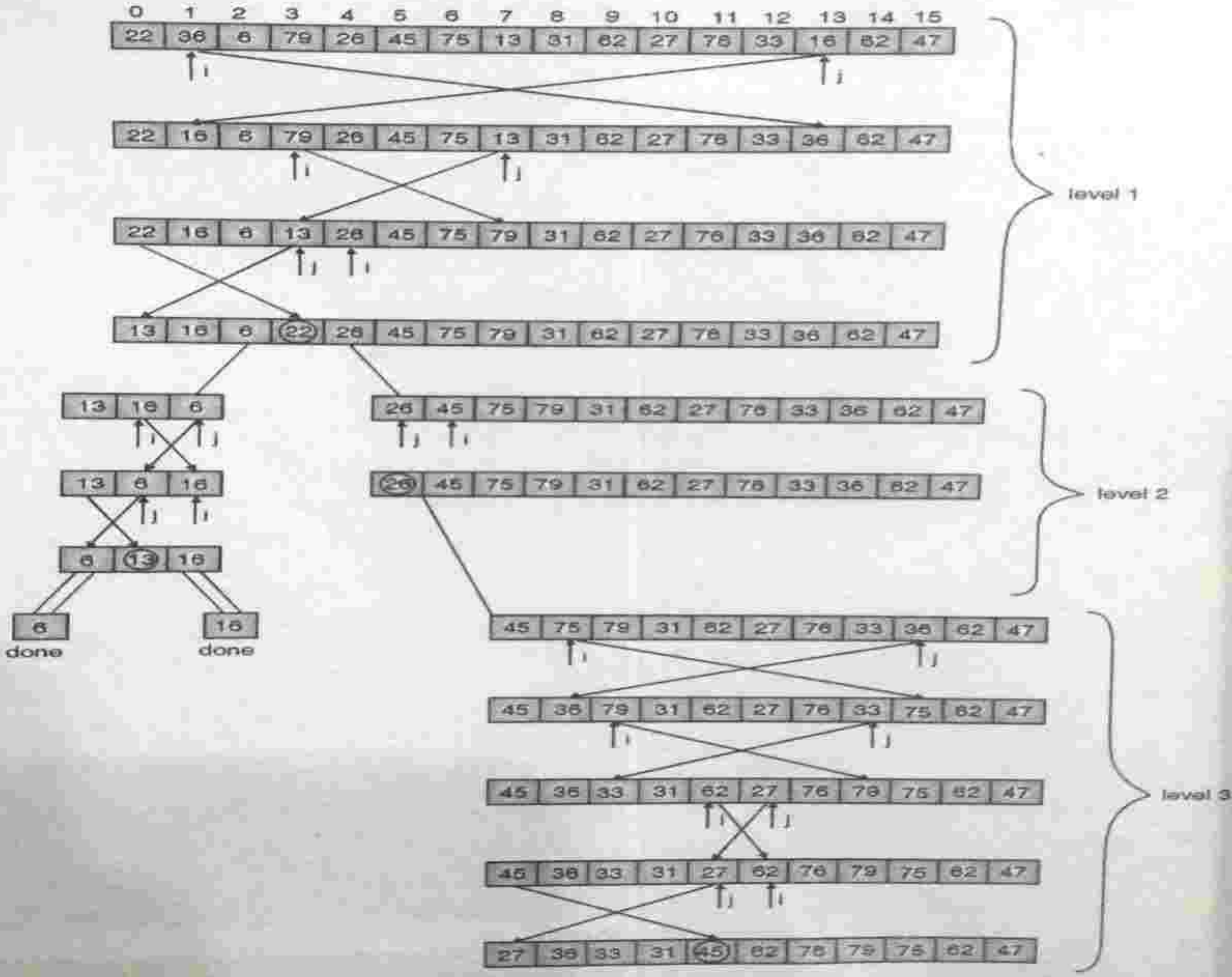
Pivot

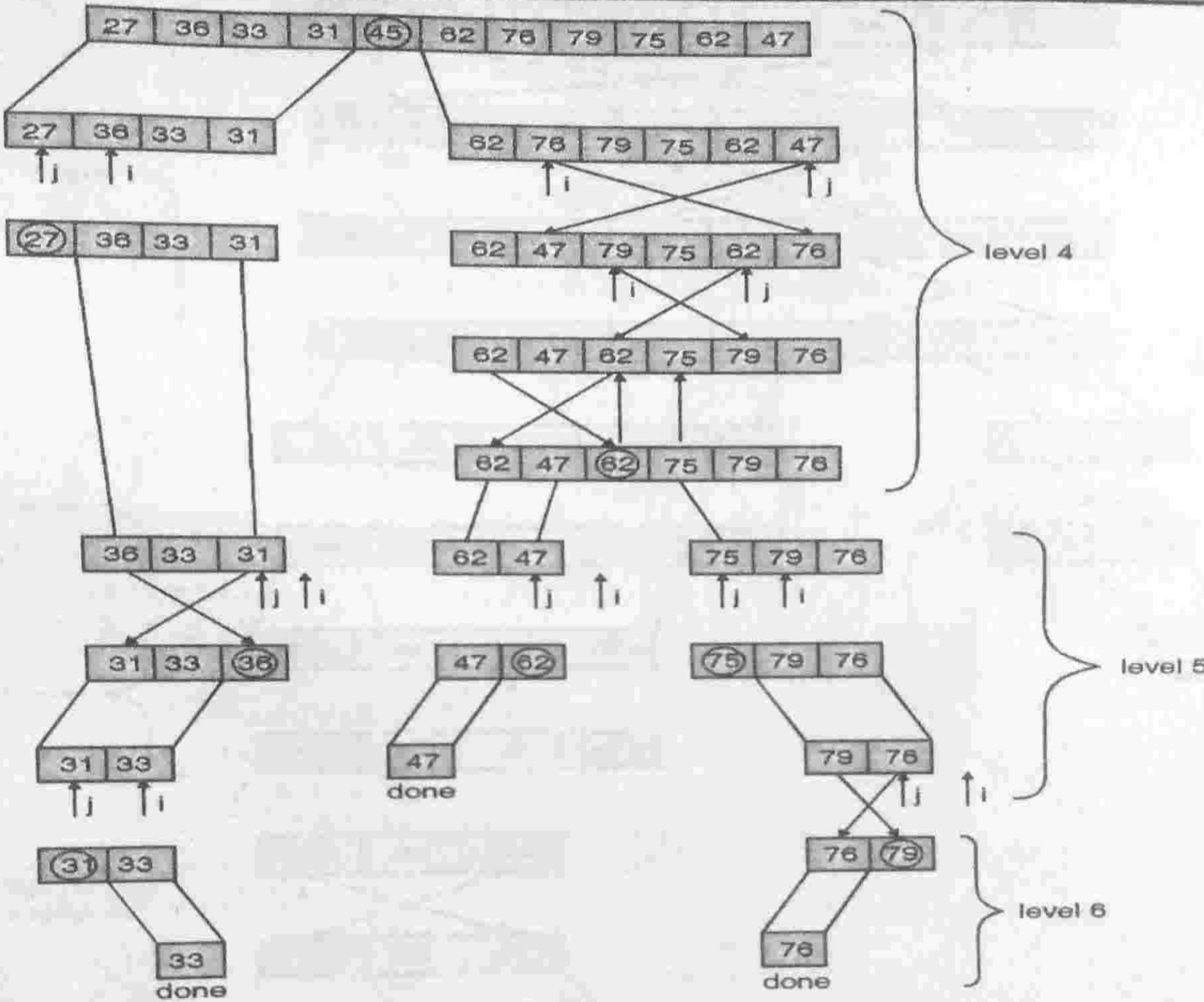
right partition

Please note that all element a[i] with $l \le i < j$ are less than 30 and all elements a[j] with $u \ge i > j$ are greater than 30.

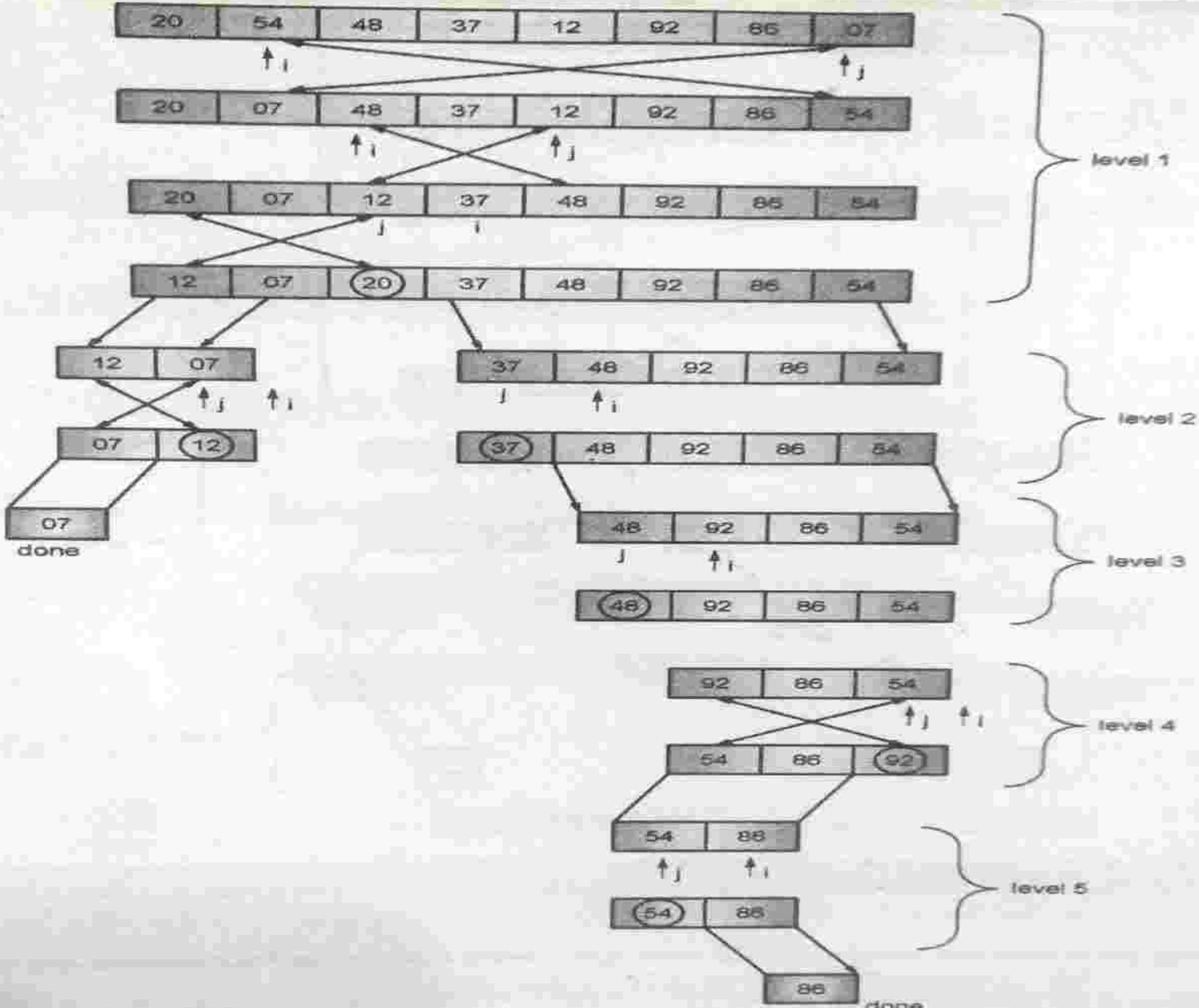
Here are sixteen integers : 22, 36, 6, 7, 9, 26, 45, 75, 13, 31,62, 27, 76, 33, 16, 62, 49. Sort them using quick sort.

Solution :

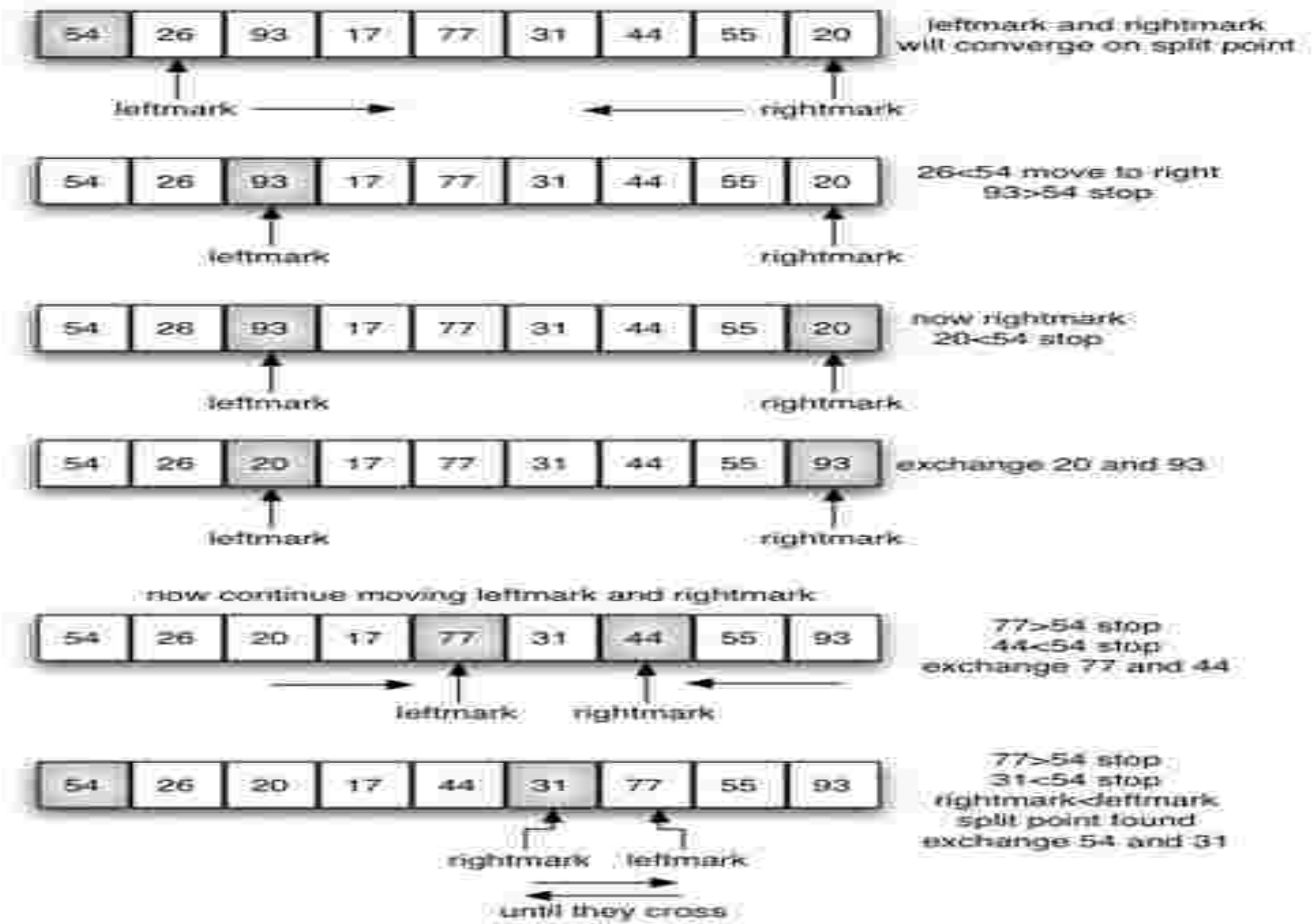




level 4
level 5
level 6
done
done
done


20 54 48 37 12 92 86 07
i j
20 07 48 37 12 92 86 54
i j
20 07 12 37 48 92 86 54
j i
12 07 20 37 48 92 86 54
level 1
12 07
j i
07 12
07
done
37 48 92 86 54
j i
37 48 92 86 54
level 2
48 92 86 54
j i
48 92 86 54
level 3
92 86 54
j i
54 86 92
level 4
54 86
j i
54 86
level 5
86
done

Quick Sort Algorithm



Quick Sort Algorithm

initial array	6	4	5	8	2	3	1	9	5
choose pivot	6	4	5	8	2	3	1	9	5
arrange values	2	4	5	8	9	3	1	5	6
increment	2	4	5	8	9	3	1	5	6
swap values	2	4	5	1	9	3	8	5	6
increment	2	4	5	1	9	3	8	5	6
swap values	2	4	5	1	3	9	8	5	6
increment, cross over	2	4	5	1	3	9	8	5	6
move pivot to final position	2	4	5	1	3	5	8	9	6

Quick Sort Algorithm

Step 1
Determine pivot

4	2	6	5	3	9

Step 2
Start pointers at left and right

4	2	6	5	3	9
L					R

Step 3
Since 4 < 5, shift left pointer

4	2	6	5	3	9
	L				R

Step 4
Since 2 < 5, shift left pointer
Since 6 > 5, stop

4	2	6	5	3	9
		L			R

Step 5
Since 9 > 5, shift right pointer
Since 3 < 5, stop

4	2	6	5	3	9
		L		R	

Step 6
Swap values at pointers

4	2	3	5	6	9
		L		R	

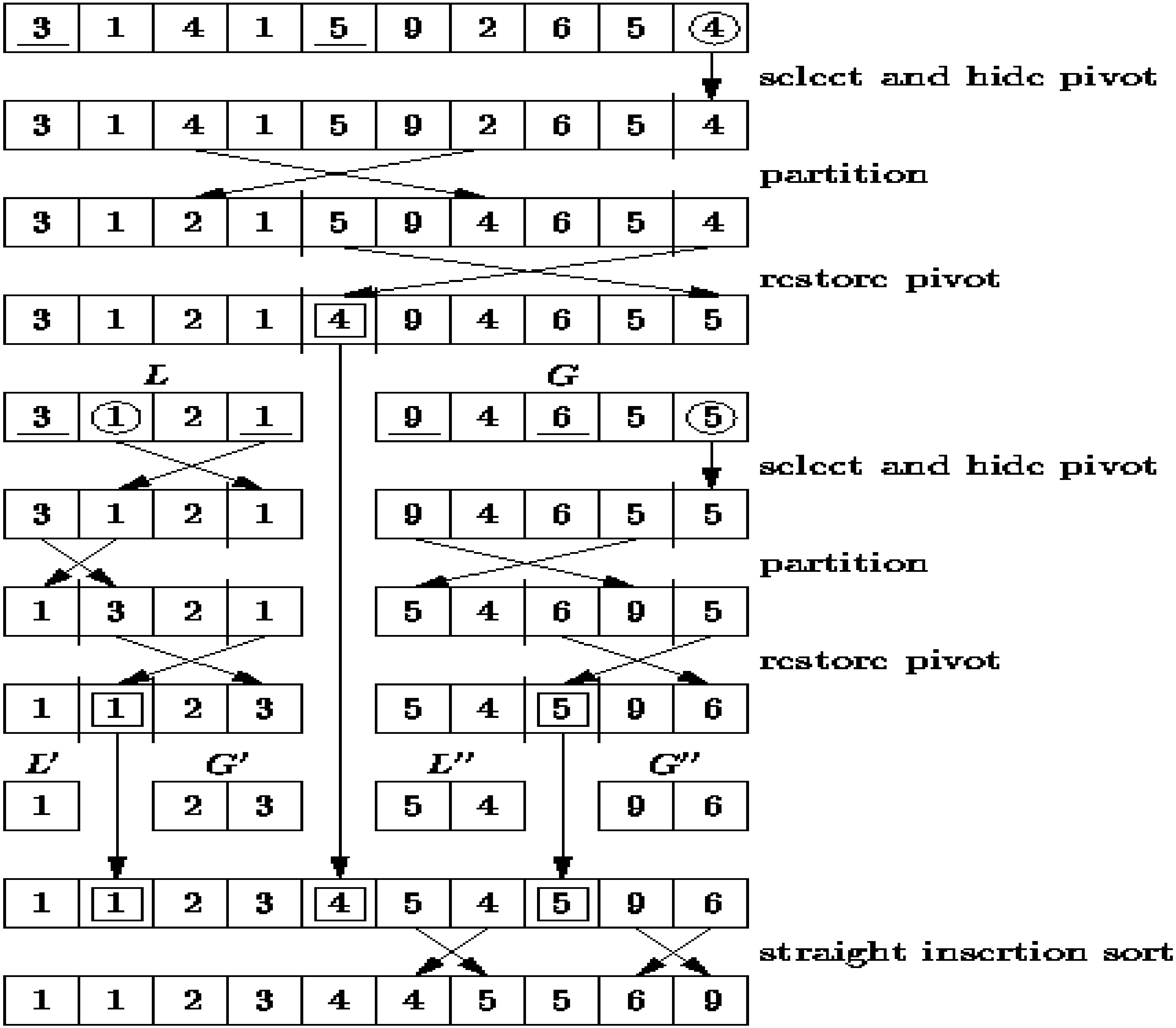
Step 7
Move pointers one more step

4	2	3	5	6	9
			L R		

Step 8
Since 5 == 5,
move pointers one more step
Stop

4	2	3	5	6	9
		R		L	

Quick Sort Algorithm



Merge Sort Algorithm

- *Merge sort is a sorting technique based on divide and conquer technique. With Average case and worst-case time complexity being O(n log n), it is one of the most respected algorithms.*
- *Merge sort first divides the array into equal halves and then combines them in a sorted manner.*

Merge Sort

- The most common algorithm used in external sorting is the merge sort
- Merging is the process of combining two or more sorted files into the third sorted file
- We can use a technique of merging two sorted lists
- Divide and conquer is a general algorithm design paradigm that is used for merge sort

Merge Sort

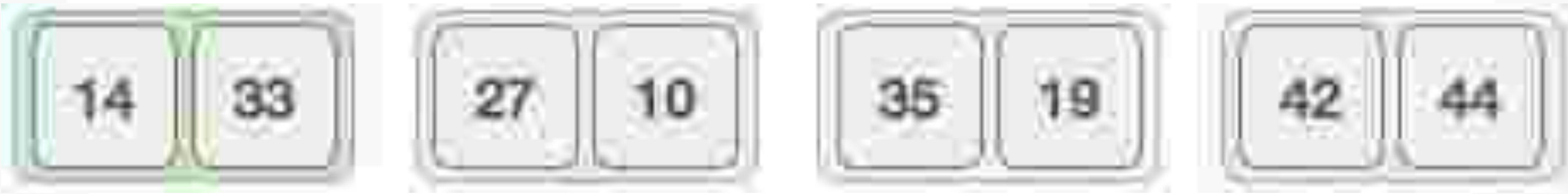
- *Time Complexity* $T(n) = O(n \log n)$

How merge sort works

- *To understand merge sort, we take unsorted array as depicted below – a*

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

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



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

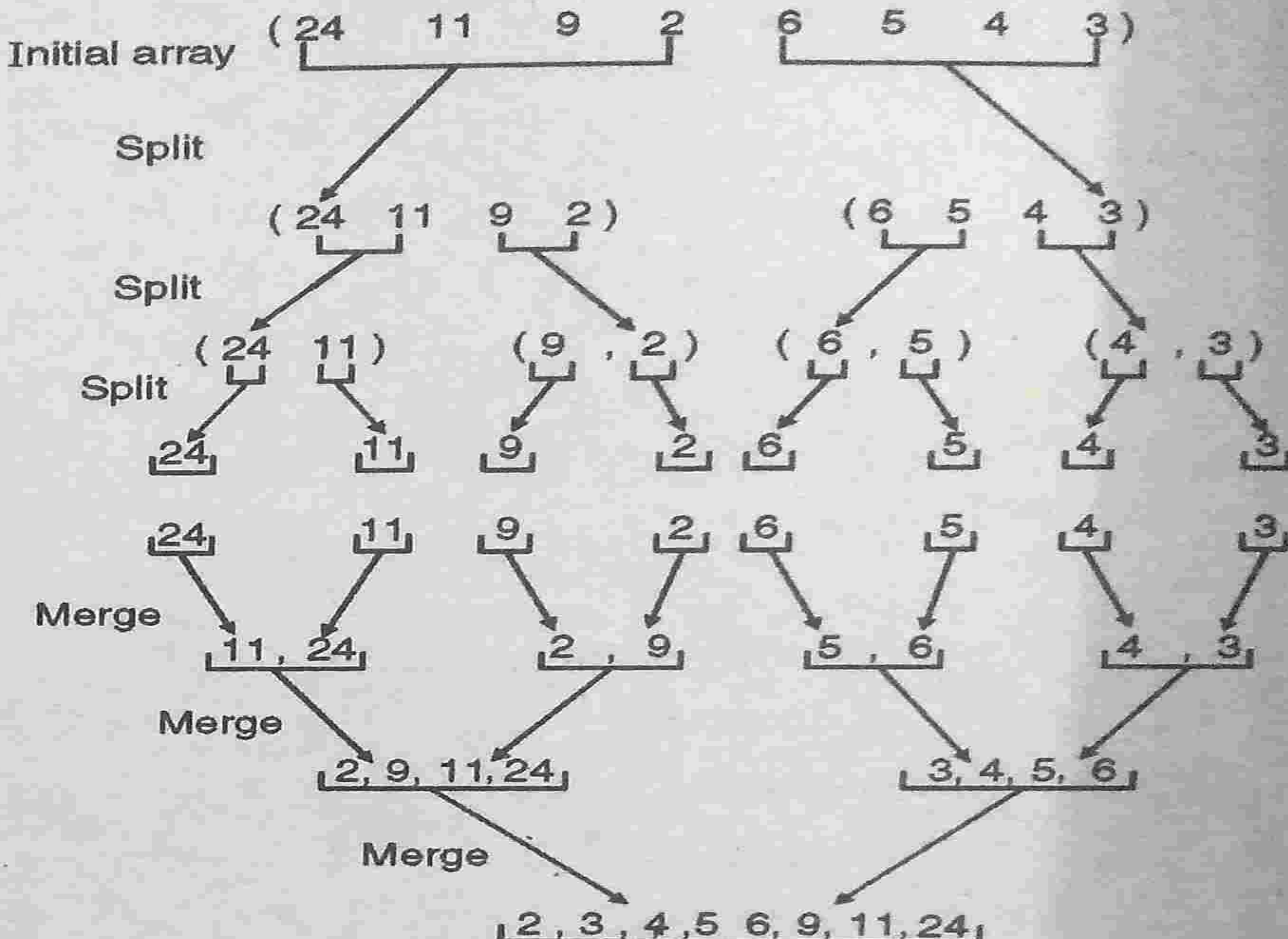
- *Now, we combine them in exactly same manner they were broken down.*
- *We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.*

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

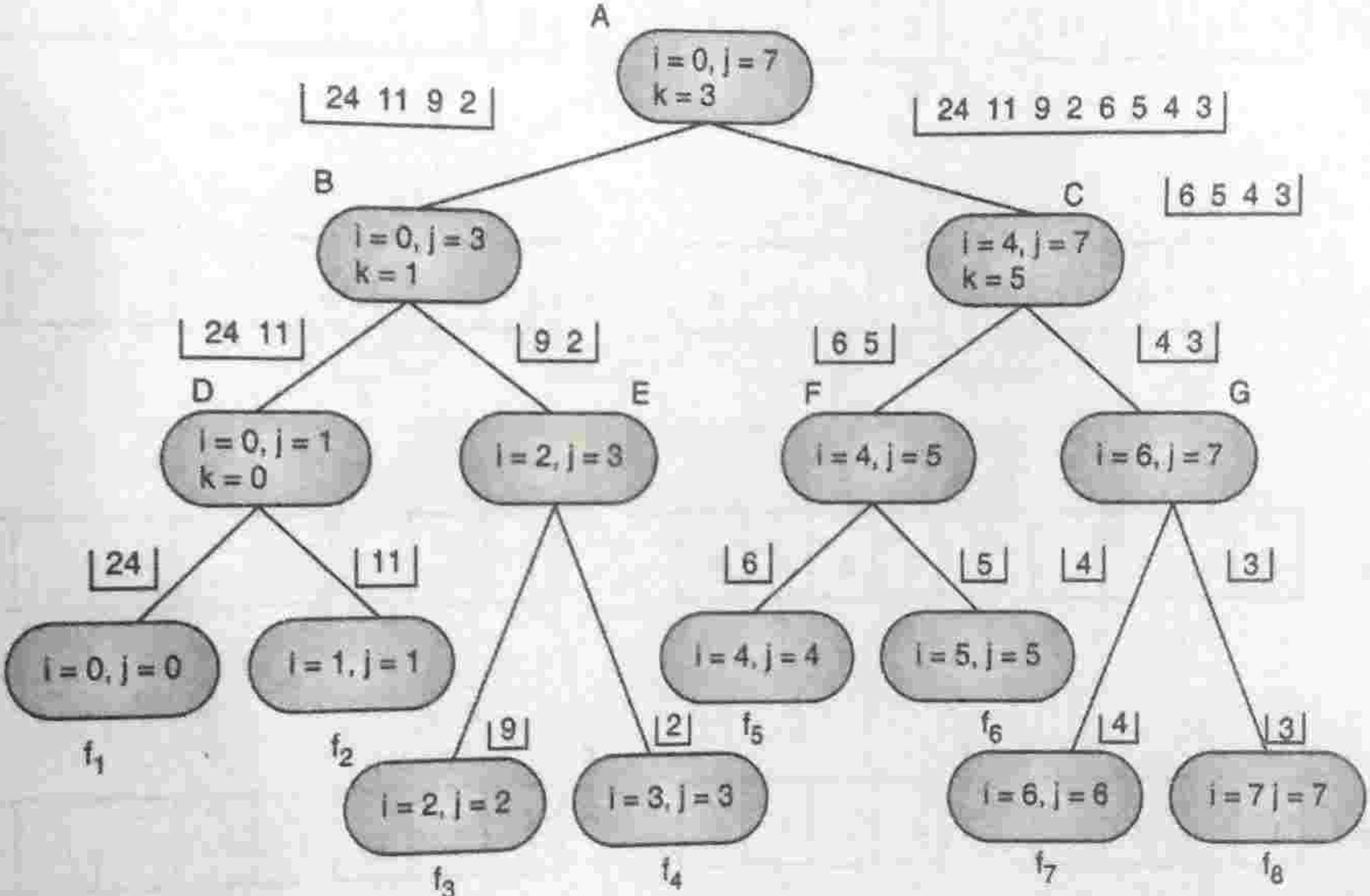
- *After final merging, the list should look like this –*

Algorithm of merge sort

- *Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then merge sort combines smaller sorted lists keeping the new list sorted too.*
  - ***Step 1*** *– divide the list recursively into two halves until it can no more be divided.*
  - ***Step 2*** *– if it is only one element in the list it is already sorted, return.*
  - ***Step 3*** *– merge the smaller lists into new list in sorted order.*

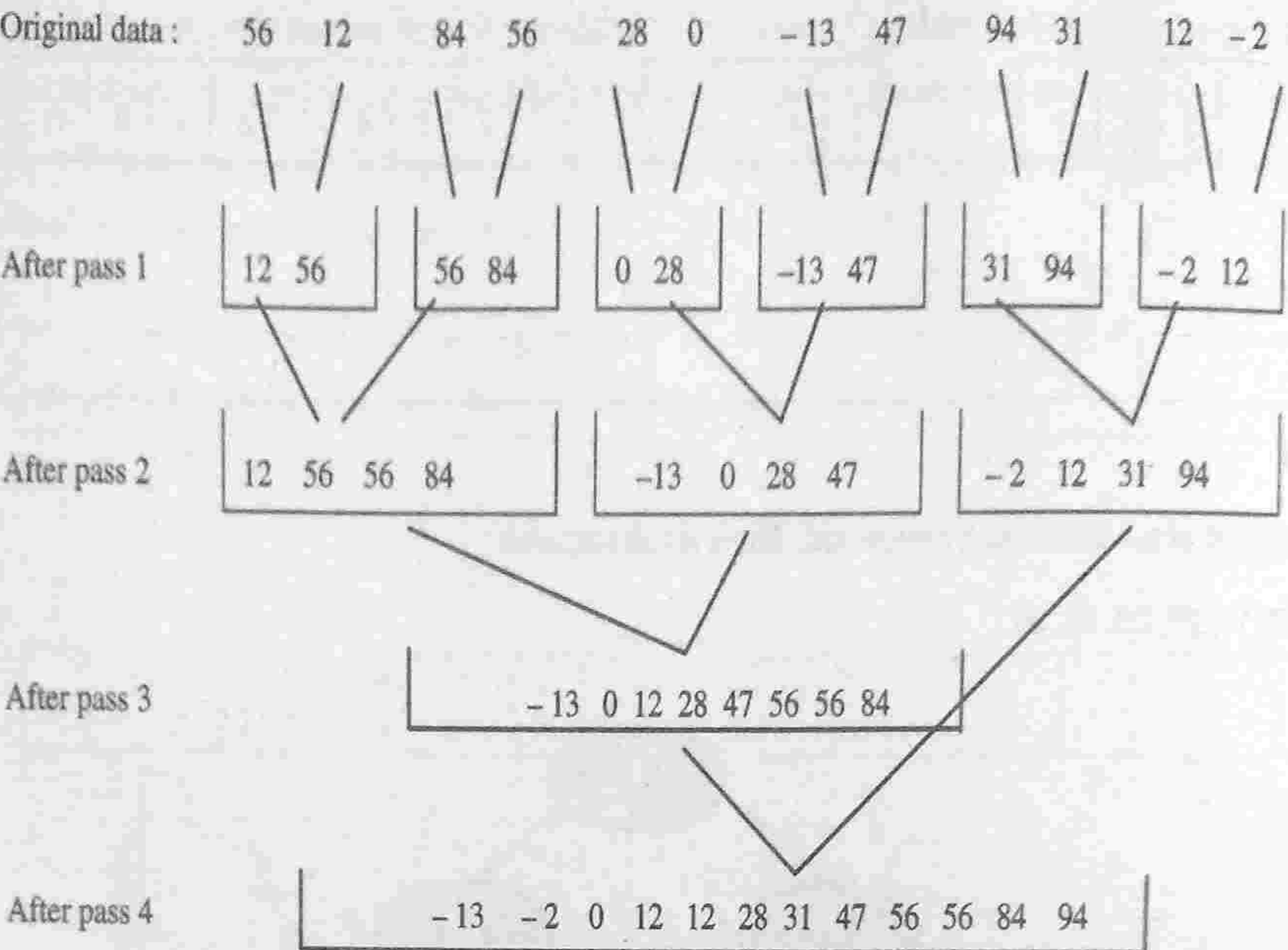


Initial array
(24 11 9 2 6 5 4 3)
Split
(24 11 9 2)
(6 5 4 3)
Split
(24 11)
(9 , 2)
(6 , 5)
(4 , 3)
Split
24
11
9
2
6
5
4
3
24
11
9
2
6
5
4
3
Merge
11 , 24
2 , 9
5 , 6
4 , 3
Merge
2, 9, 11, 24
3, 4, 5, 6
Merge
2 , 3 , 4 ,5 6, 9, 11, 24



A
i = 0, j = 7
k = 3
24 11 9 2
24 11 9 2 6 5 4 3
B
i = 0, j = 3
k = 1
C
6 5 4 3
i = 4, j = 7
k = 5
24 11
9 2
6 5
4 3
D
i = 0, j = 1
k = 0
E
i = 2, j = 3
F
i = 4, j = 5
G
i = 6, j = 7
24
11
6
5
4
3
i = 0, j = 0
i = 1, j = 1
i = 4, j = 4
i = 5, j = 5
f_1
f_2
9
2
f_5
f_6
4
3
i = 2, j = 2
i = 3, j = 3
i = 6, j = 6
i = 7 j = 7
f_3
f_4
f_7
f_8

Fig. 8.6.2

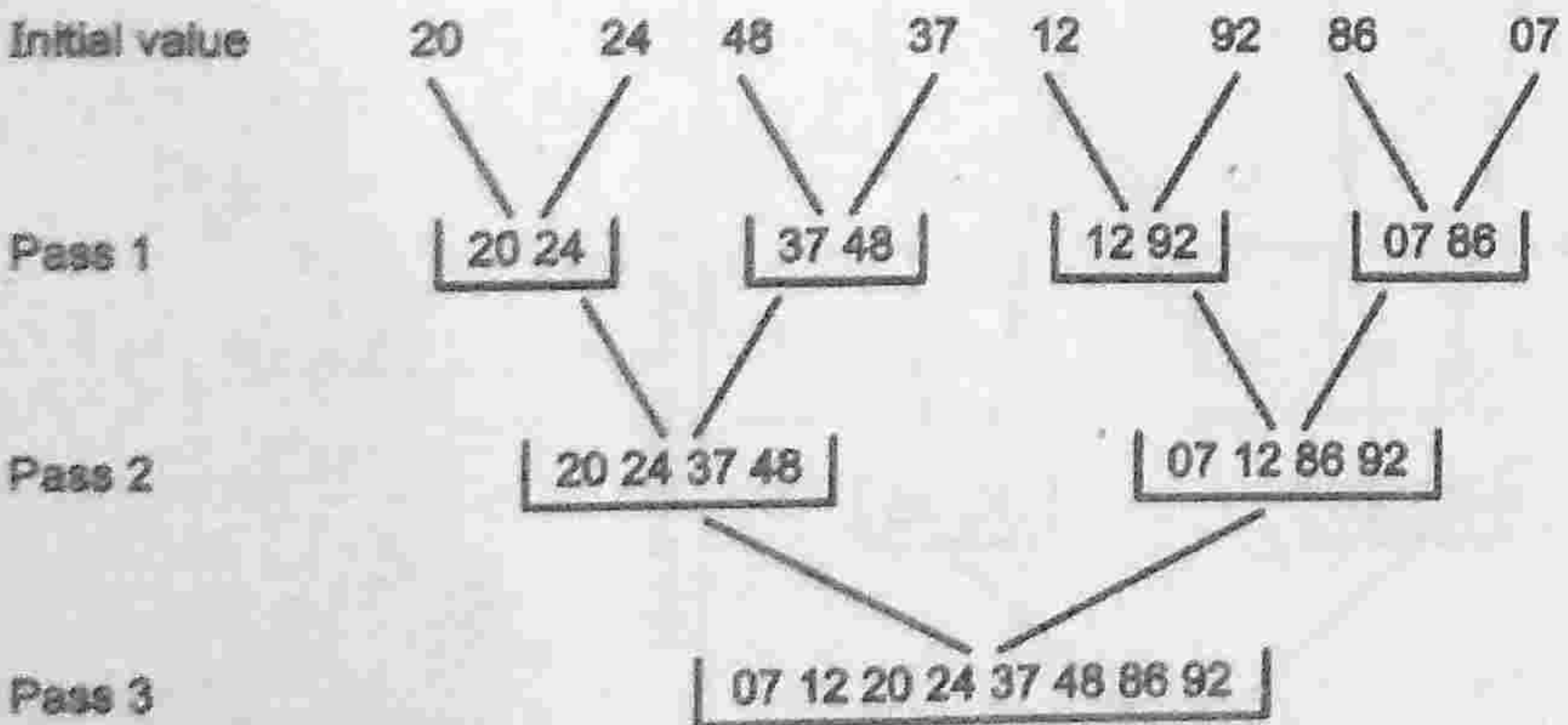


Original data : 56 12 84 56 28 0 – 13 47 94 31 12 – 2
After pass 1
12 56
56 84
0 28
–13 47
31 94
– 2 12
After pass 2
12 56 56 84
–13 0 28 47
– 2 12 31 94
After pass 3
– 13 0 12 28 47 56 56 84
After pass 4
– 13 – 2 0 12 12 28 31 47 56 56 84 94

Consider the following set of numbers, sort them usi

iterative merge sort. Show all passes

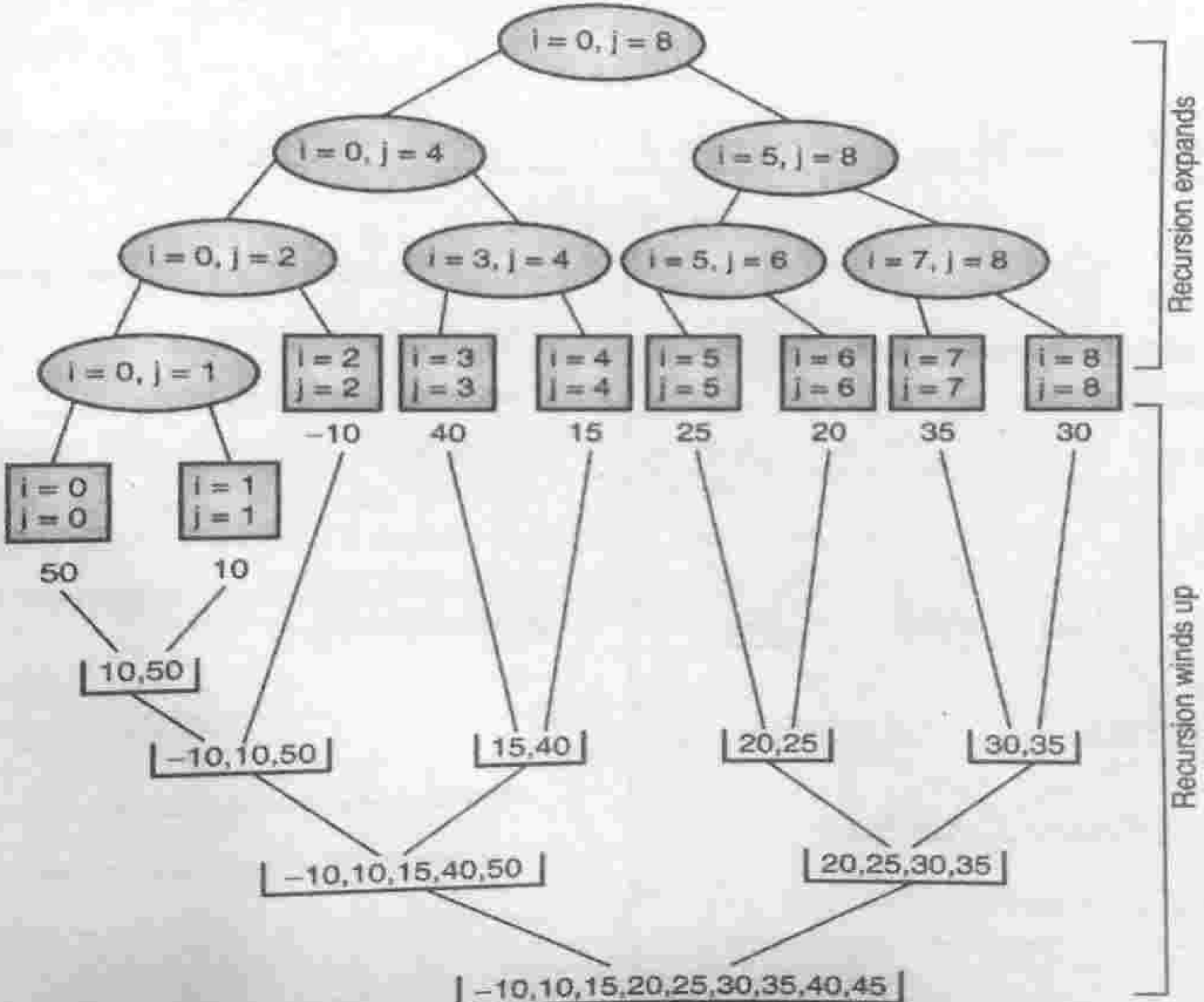
20 24 48 37 12 92 86 07



Sort the following list of numbers using merge sort. Show result stepwise :

50, 10, – 10, 40, 15, 25, 20, 35, 30

Solution :



Shell Sort

- *Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort if smaller value is very far right and have to move to far left.*
- *This algorithm uses insertion sort on widely spread elements first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as –*

- ***h = h * 3 + 1***

  where – h is interval with initial value 1

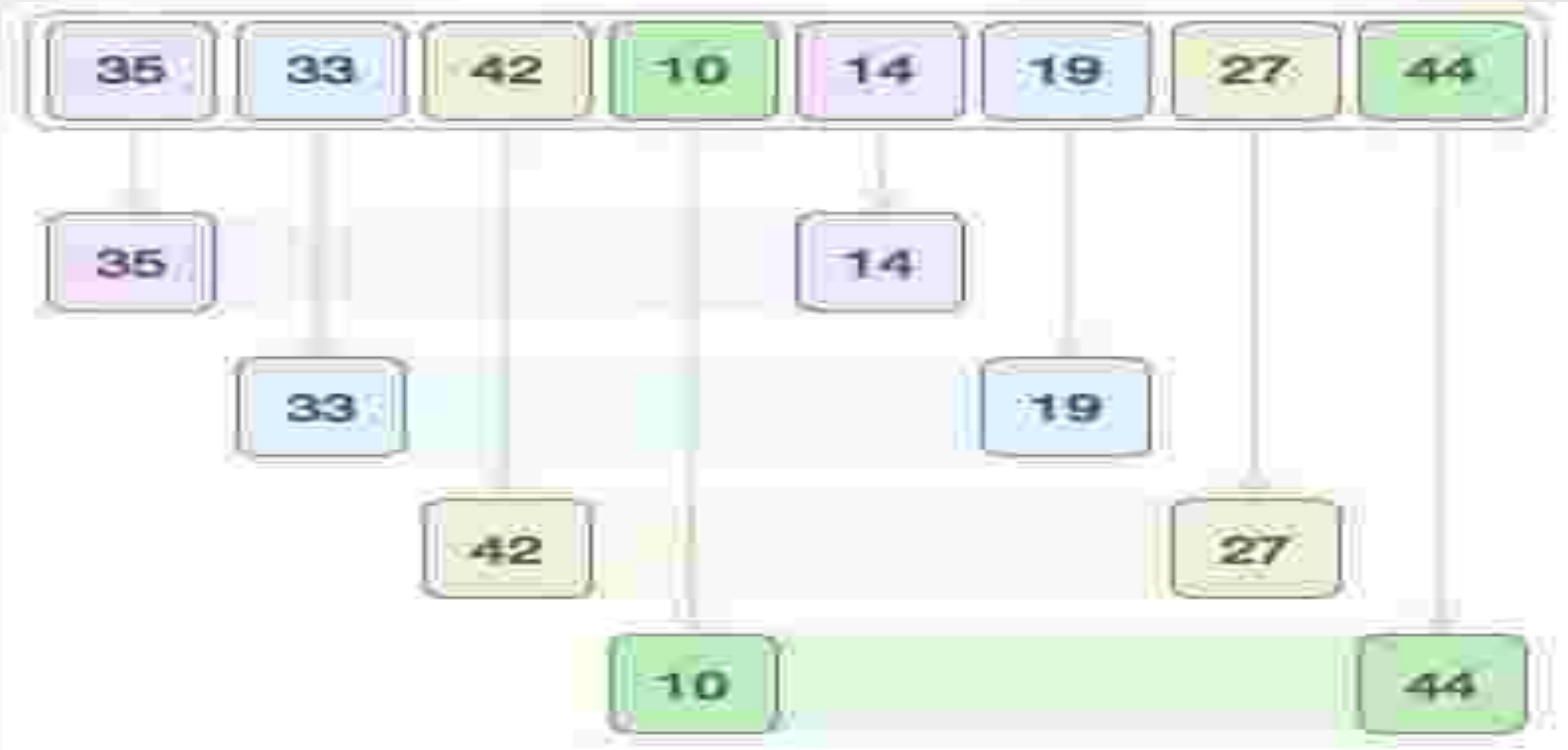
This algorithm is quite efficient for medium sized data sets as its average and worst case complexity are of O(n^2) where n are no. of items.

How shell sort works

- *We take the below example to have an idea, how shell sort works?*

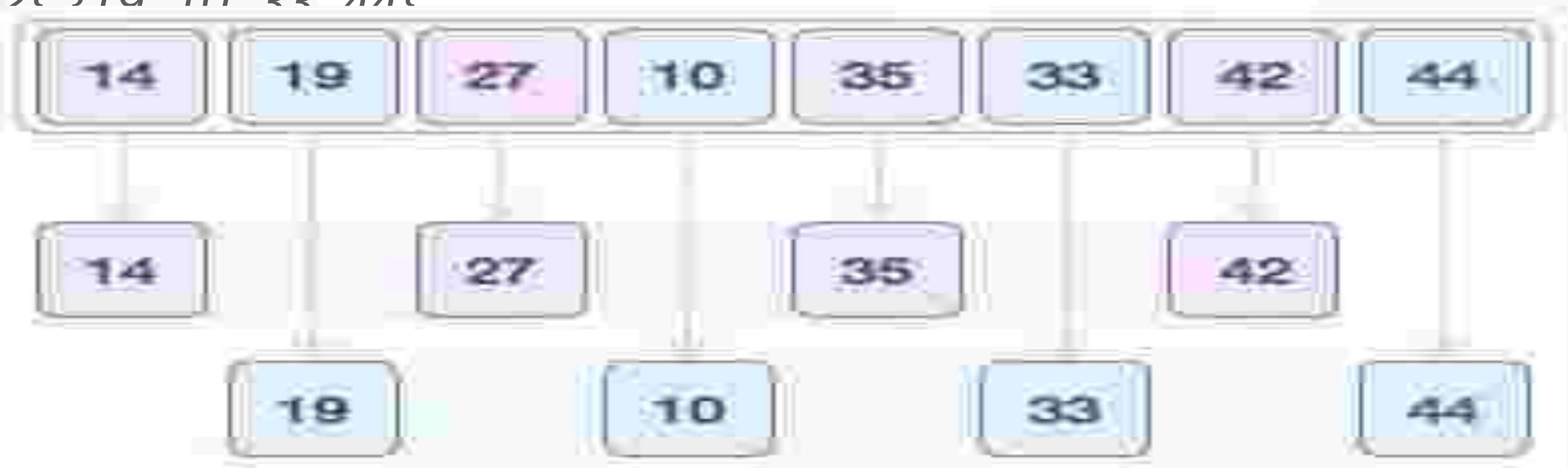
- *We take the same array we have used in our previous examples. {35,33,42,10,14,19,27,44}*
- *For our example and ease of understanding we take the interval of 4.*
- *And make a virtual sublist of all values located at the interval of 4 positions. Here these values are*

  {35, 14}, {33, 19}, {42, 27} and {10, 14}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, new array should look like this −

Then we take interval of 2 and this gap generates two sublists - {14, 27, 35,
42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, this array should look like this −

And finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array. The step by step depiction is shown below −

14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	27	10	35	33	42	44
14	19	10	27	35	33	42	44
14	10	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	33	35	42	44
10	14	19	27	33	35	42	44

Shell sort Algorithm

- *We shall now see the algorithm for shell sort.*
- ***Step 1** – Initialize the value of h*
- ***Step 2** – Divide the list into smaller sub-list of equal interval h*
- ***Step 3** – Sort these sub-lists using **insertion sort***
- ***Step 4** – Repeat until complete list is sorted*

(5 1 9 8 2 4 6 9) using shell sort

Solution :

original array 5 1 9 8 2 4 6 9

there are four groups, each group has 2 elements.

elements of a group are sorted using insertion sort.

Array After 2 1 6 8 | 5 4 9 9

there are 2 groups, each group has 4 elements.

Elements of a group are sorted using insertion sort.

there is only one group, group has 8 elements.

elements of the group are sorted using insertion sort.

Array after 1 2 4 5 6 8 9 9

Original array : 8 3 2 11 5 14 0 9 4 20

First pass : (Step 5)

Array before :

8 3 2 11 5 | 14 0 9 4 20

There are five groups each has 2 elements which are sorted independently using insertion sort.

Array after :

8 0 2 4 5 14 3 9 11 20

Second pass : (Step 2)

Array before :

8 0 | 2 4 | 5 14 | 3 9 | 11 20

There are two groups each has 5 elements sorted using insertion sort.

Array after :

2 0 3 4 5 9 8 14 11 20

Third pass : (Step 1)

Array before :

2 0 3 4 5 9 8 14 11 20

There is only one groups with 10 elements, sorted using insertion sort.

Array after :

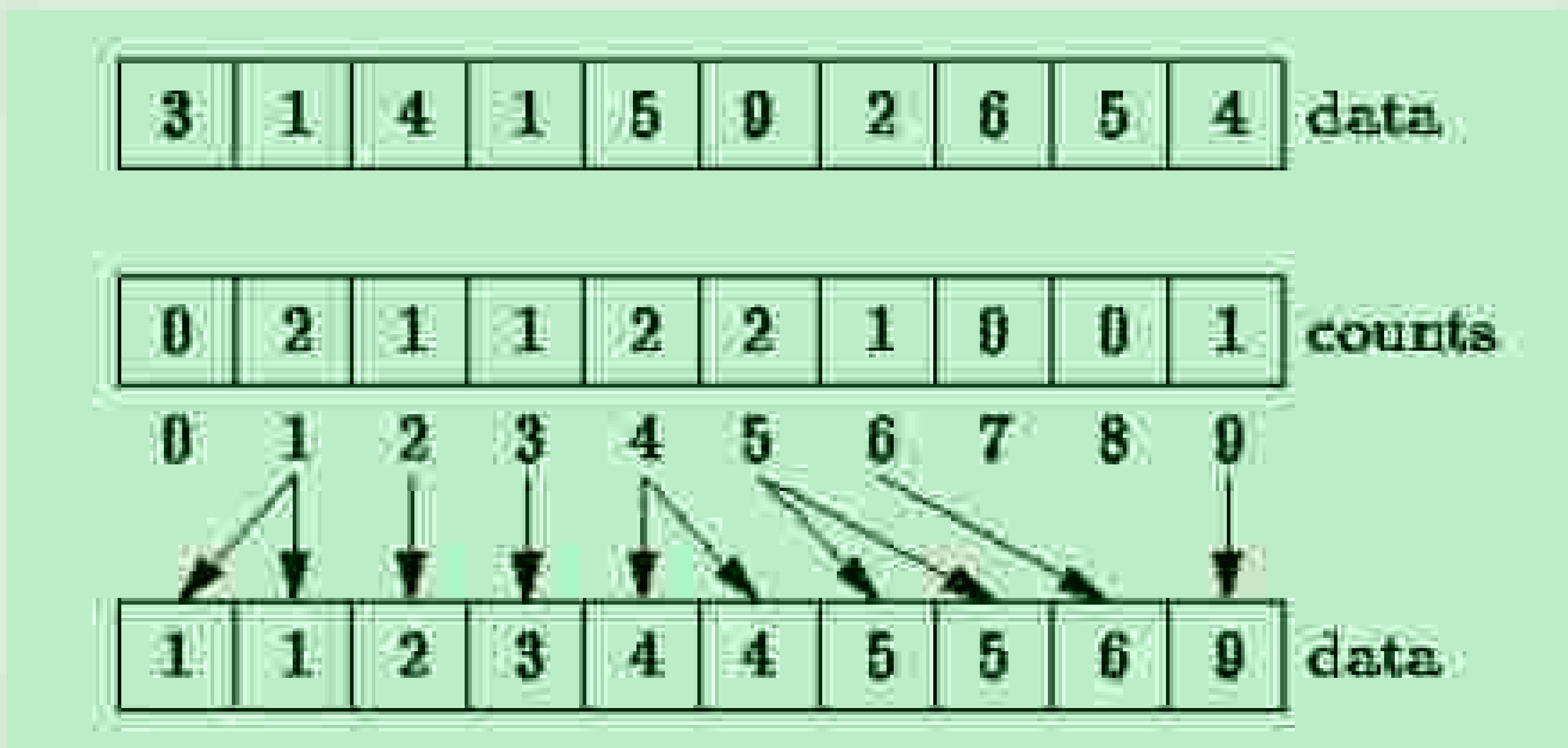
0 2 3 4 5 8 9 11 14 20

Bucket Sort

- *For example, suppose that we are sorting elements from the set of integers in the interval [0, m – 1]. The bucket sort uses m buckets or counters*

- *The i^{th} counter/bucket keeps track of the number of occurrences of the ith element of the list*

Bucket Sort


3 1 4 1 5 9 2 6 5 4
data
0 2 1 1 2 2 1 0 0 1
counts
0 1 2 3 4 5 6 7 8 9
1 1 2 3 4 4 5 5 6 9
data

Bucket Sort

Sort the following elements in ascending order using bucket sort. Show all passes :

121, 235, 55, 973, 327, 179.

Solution :

Numbers are being sorted using radix sort. Radix sort is generalization of bucket sort.

Buckets after 1st pass

		121		973		55 235		327	178	
Bucket Number	0	1	2	3	4	5	6	7	8	9

Fig. Ex. 8.7.2(a)

Merged list : 121 973 235 55 327 178

Buckets after 2nd pass

			327 121	235		55		178 973		
Bucket Number	0	1	2	3	4	5	6	7	8	9

Fig. Ex. 8.7.2(b)

Merged list : 121 327 235 55 973 178

Buckets after 3rd pass

	55	178 121	235	327						973
Bucket Number	0	1	2	3	4	5	6	7	8	9

Fig. Ex. 8.7.2(c)

Merged list : 55 121 178 235 327 973

Radix Sort

- *Radix Sort is generalization of Bucket Sort*
- *To sort Decimal Numbers radix/base will be used as 10. so we need 10 buckets.*
- *Buckets are numbered as 0,1,2,3,…,9*
- *Sorting is Done in the passes*
- *Number of Passes required for sorting is number of digits in the largest number in the list.*

Radix Sort

- Radix sort is a generalization of bucket sorting
- Radix sort works in three steps:

  - Distribute all elements into m buckets
  - Here m is a suitable integer, for example, to sort decimal numbers with radix 10
  - We take 10 buckets numbered as 0, 1, 2, …, 9
  - For sorting strings, we may need 26 buckets, and so on
  - Sort each bucket individually
  - Finally, combine all buckets

Ex.

Range	Passes
0 to 99	2 Passes
0 to 999	3 Passes
0 to 9999	4 Passes

- *In First Pass number sorted based on Least Significant Digit and number will be kept in same bucket.*
- *In 2nd Pass, Numbers are sorted on second least significant bit and process continues.*
- *At the end of every pass, numbers in buckets are merged to produce common list.*

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

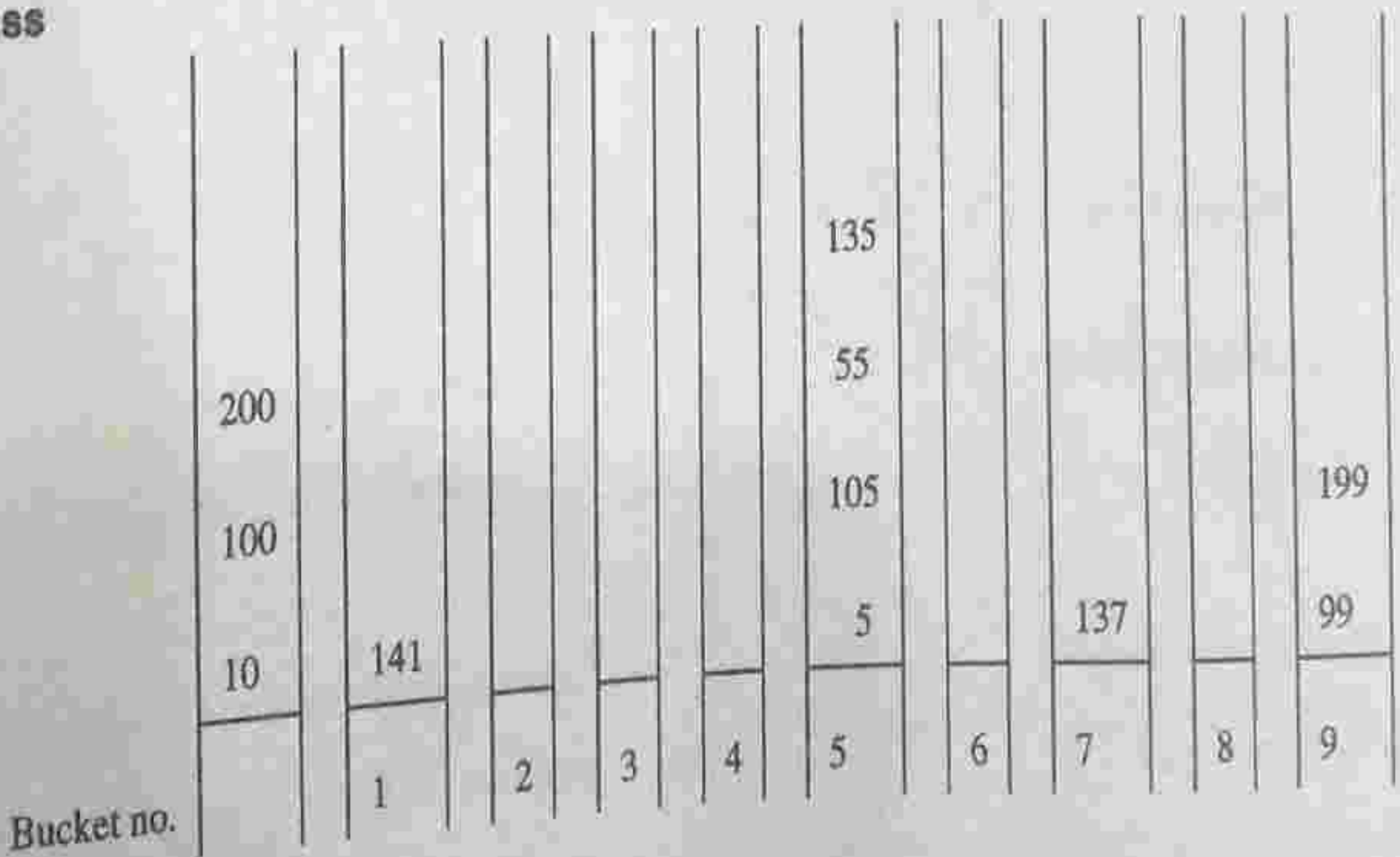
195 340 385 437 493 582 710 715 812

RADIX SORT EXAMPLE

passes.

10 5 99 105 55 100 135 141 137 200 199

Buckets after 1st pass



Merged list = 10 100 200 141 5 105 55 135 137 99 199

Buckets after 2nd pass

0	1	2	3	4	5	6	7	8	9
100									
5									
200			137						199
100	10		135	141	55				99

merged list = 100 200 5 105 10 135 137 141 55 99 199

Buckets after third pass

0	1	2	3	4	5	6	7	8	9
	199								
	141								
99	137								
55	135								
10	105								
5	100	200							

Merged list = 5 10 55 99 100 105 135 137 141 199 200

Sort the following numbers in ascending order using radix sort

14, 1, 66, 74, 22, 36, 41, 59, 64, 54

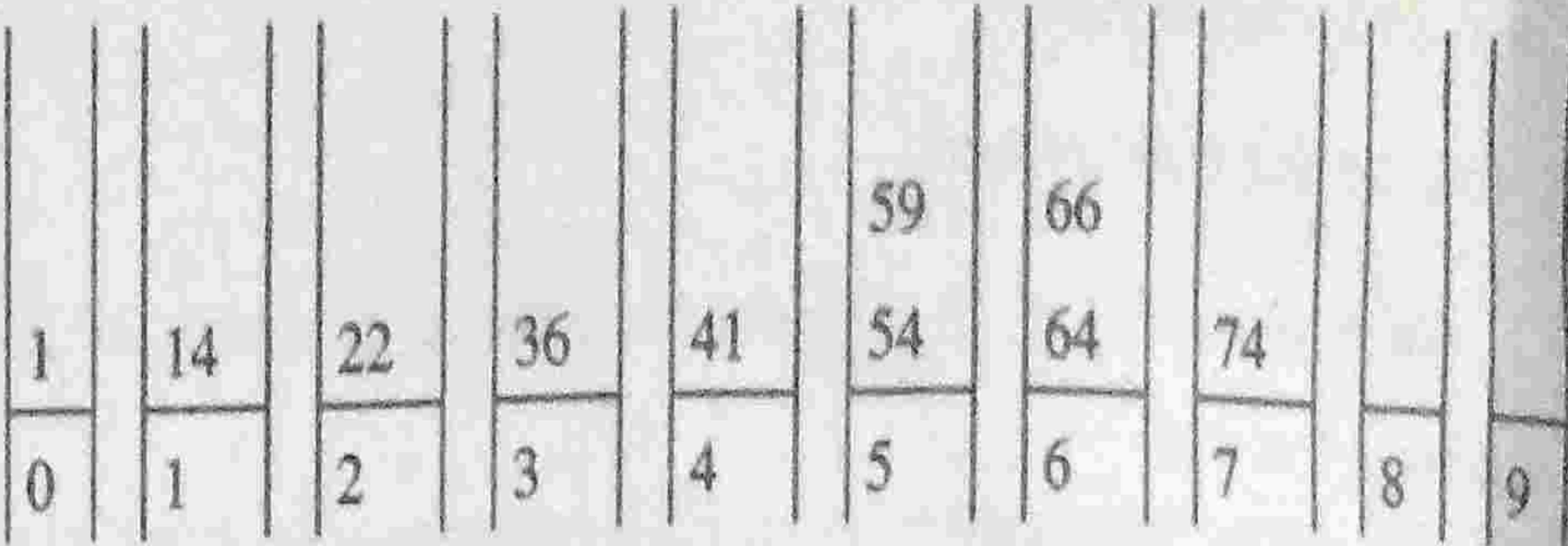
Solution :

Buckets after 1st pass

				54					
				64					
	41			74	36				
	1	22		14	66				59
0	1	2	3	4	5	6	7	8	9

Merged list = 1 41 22 14 74 64 54 66 36 59

Buckets after 2nd pass



Merged list = 14 22 36 41 54 59 64 66 74

56 12 84 56 28 0 − 13 47 94 31 12 − 2

Subtracting − 13 from every number, we get,

69 25 97 69 41 13 0 60 107 44 25 11

Buckets after 1st pass (sorting on least significant digit)

60	11				25		107		69
0	41		13	44	25		97		69
0	1	2	3	4	5	6	7	8	9

Merged list : 0 60 41 11 13 44 25 25 97 107 69 69

Buckets after 2nd pass

						69			
107	13	25		44		69			
0	11	25		41		60			97
0	1	2	3	4	5	6	7	8	9

- *Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.*
- ***Average case and Worst case Complexity - O(n)***

Disadvantages

- *Still, there are some tradeoffs for Radix Sort that can make it less preferable than other sorts.*
- *The speed of Radix Sort largely depends on the inner basic operations, and **if** the operations are not efficient enough, **Radix Sort can be slower than some other algorithms** such as Quick Sort and Merge Sort.*
- *In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.*
- *Radix Sort can also take up more **space** than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sublist for **each** of the possible digits or letters.*

HEAP SORT

- Heap sort is one of the fastest sorting algorithms, which achieves the speed as that of quick sort and merge sort
- The advantages of heap sort are as follows: it does not use recursion, and it is efficient for any data order
- It achieves the worst-case bounds better than those of quick sort
- And for the list, it is better than merge sort since it needs only a small and constant amount of space apart from the list being sorted

Heap Sort

- The steps for building heap sort are as follows:
  - Build the heap tree
  - Start delete heap operation storing each deleted element at the end of the heap array

Heap Sort

- ***ALGORITHM***
- ***1. Build a heap tree with a given set of data***
  - ***(a) Delete root node from heap***
  - ***(b) Rebuild the heap after deletion***
  - ***(c) Place the deleted node in the output***

  Continue with step (2) until the heap tree is empty

Analysis of Heap Sort

- ***The time complexity is stated as follows:***
  - ***Best case O(n logn)***
  - ***Average case O(n logn)***

Create a max heap with following elements :

5, 1, 9, 2, 11, 50, 6, 100, 7

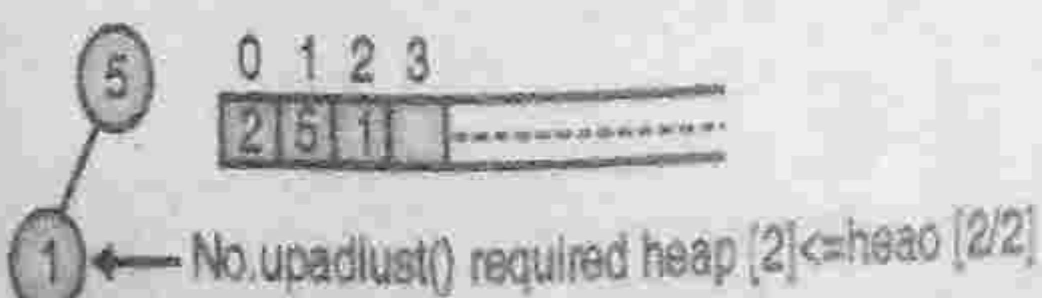
Solution :

Let us assume that heap is represented using an array heap[12].

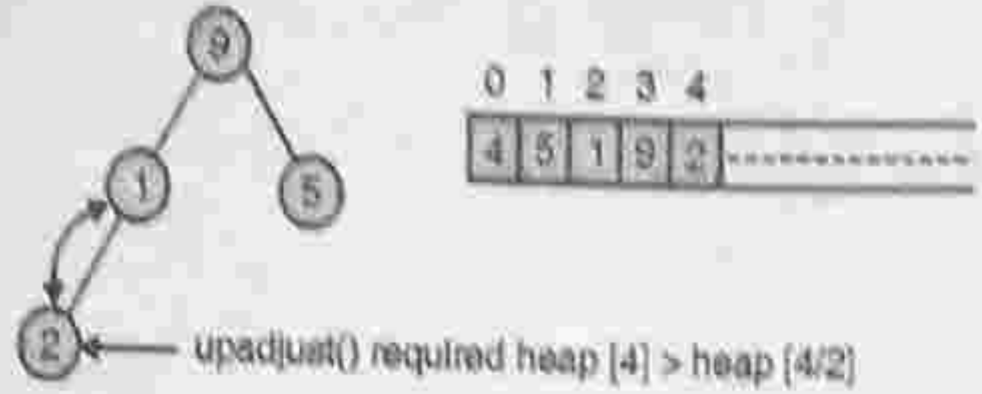
Initially

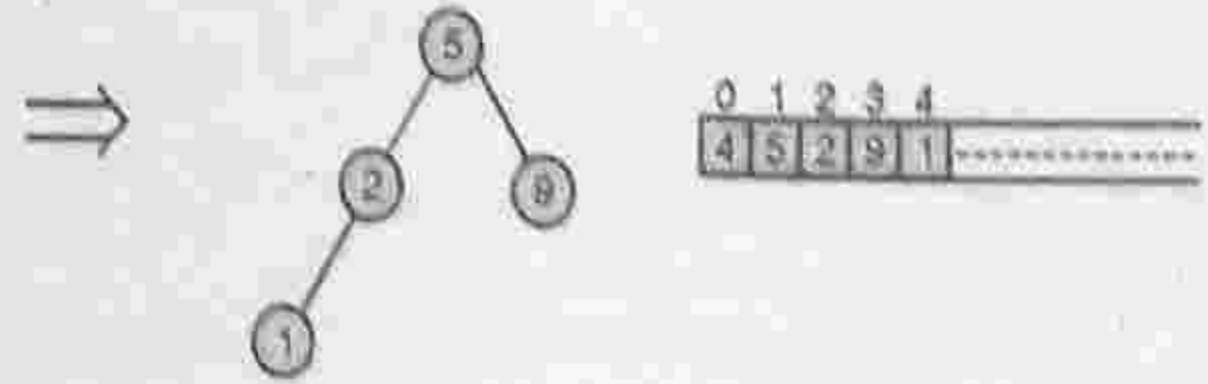
Insert 5

Insert 1

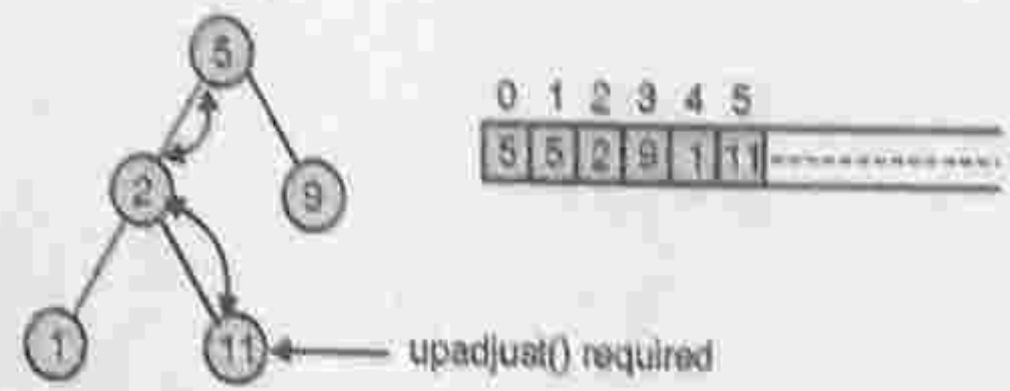


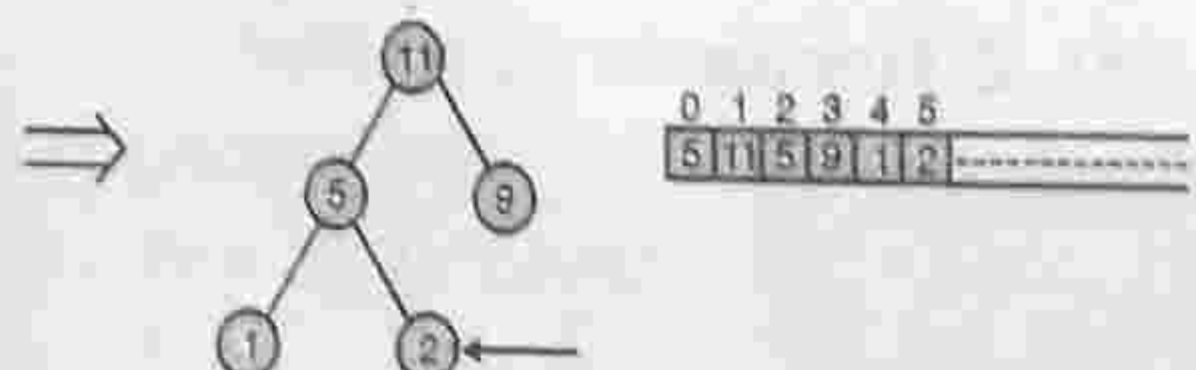
Insert 2

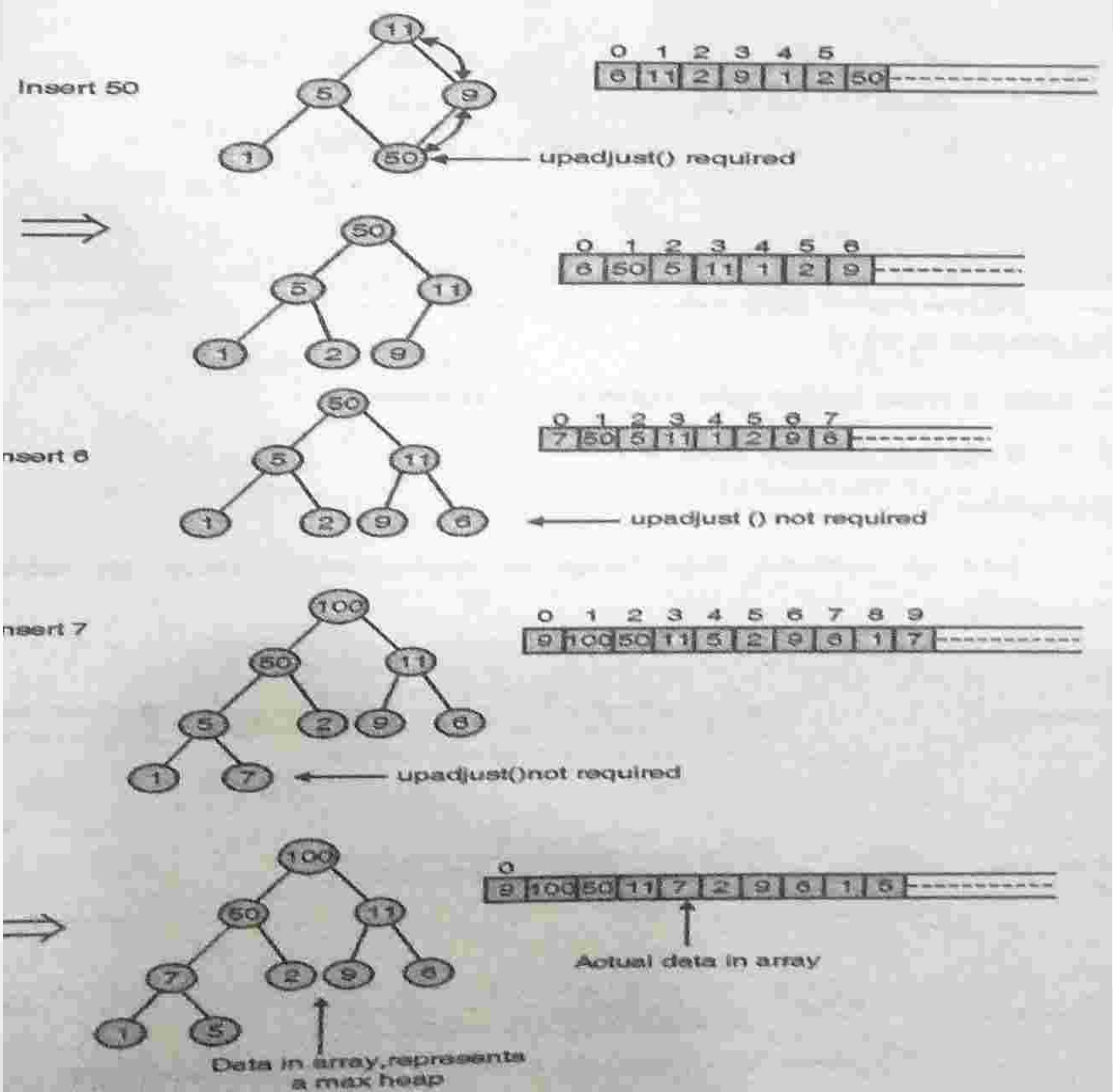




Insert 11

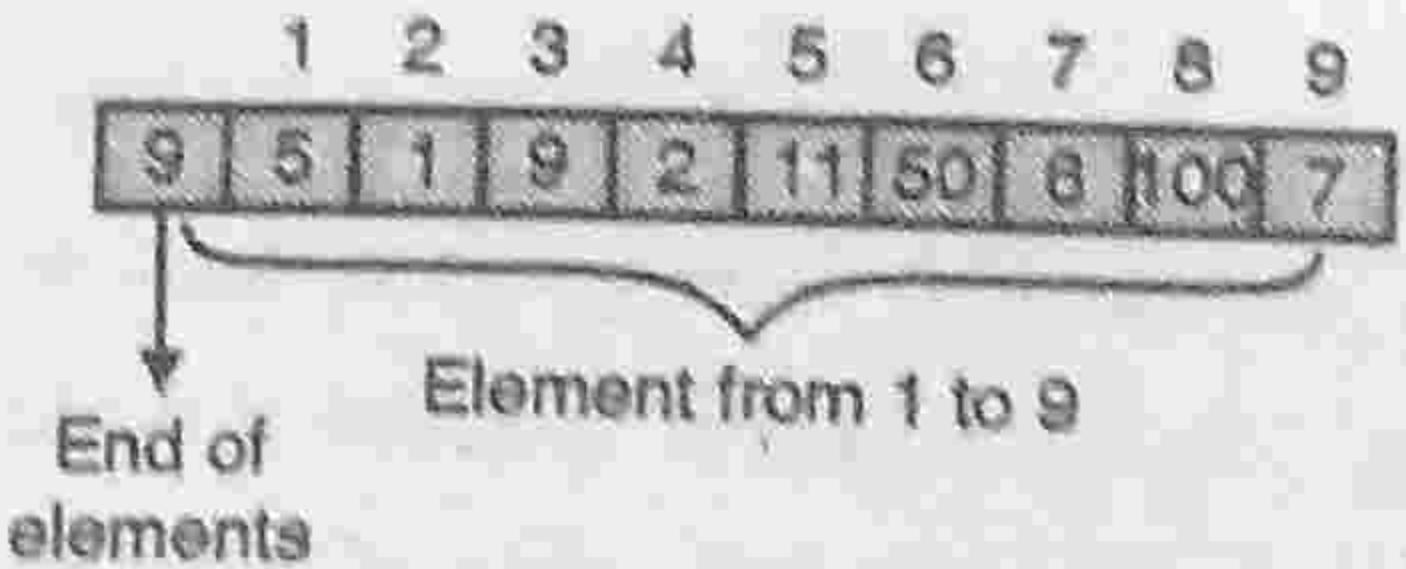




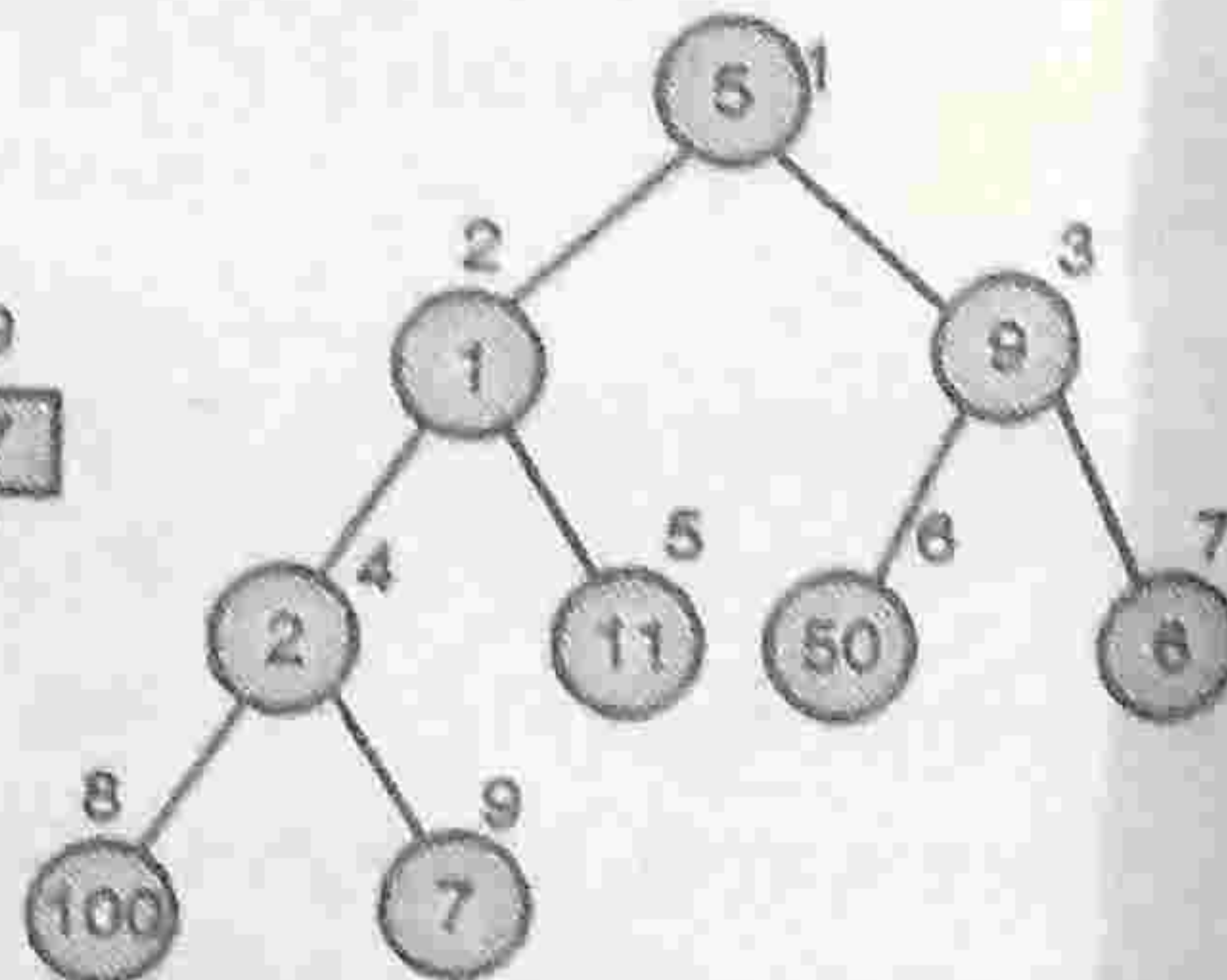

Insert 50
upadjust() required
nsert 6
upadjust () not required
nsert 7
upadjust()not required
Actual data in array
Data in array,represents
a max heap

8.11.3(G) Heap Creation - A Better Approach

Suppose that n elements are stored in an array from index 1 to n. These elements represent a complete binary tree. A tree, thus represented may not satisfy the heap property.



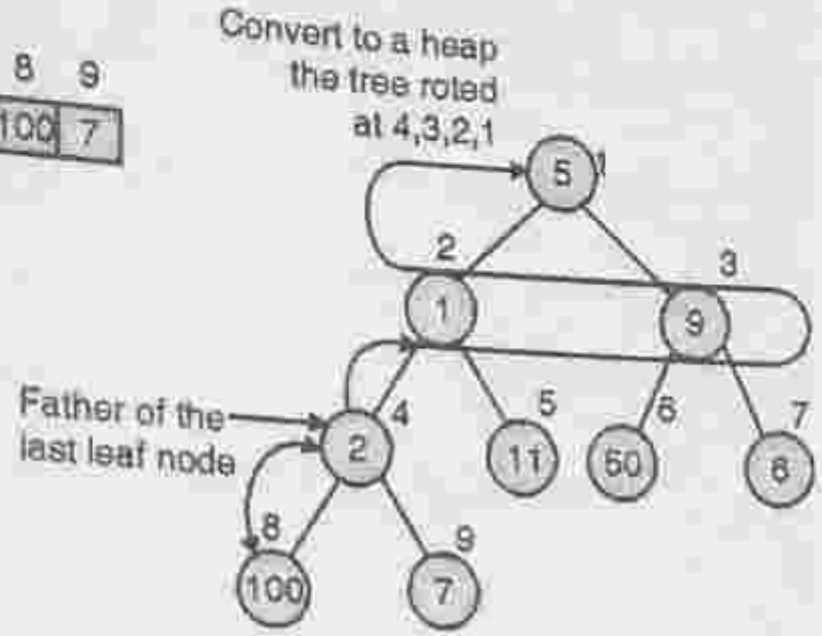
(a) An array of elements



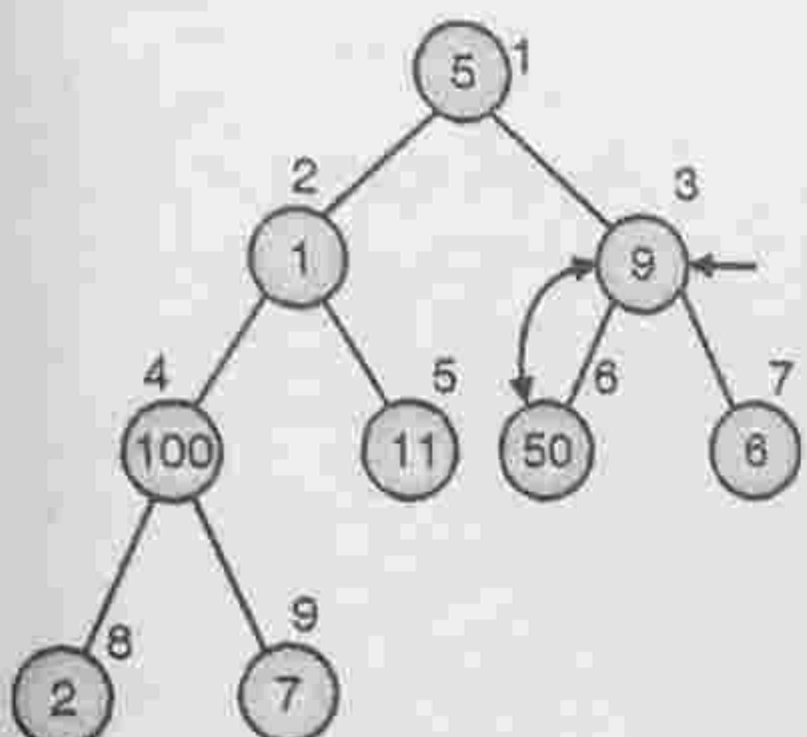
(b) Tree representation of array of

- Subtree rooted at node number 4 is taken up and it is converted to a heap through downadjust().
- Subsequently subtrees rooted at node number 3, 2, 1 are converted to a heap through downadjust().

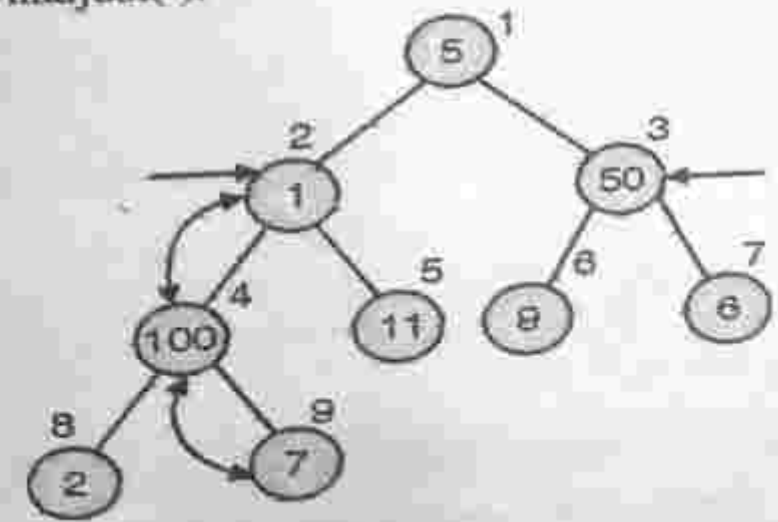
	1	2	3	4	5	6	7	8	9
9	5	1	9	2	11	50	6	100	7



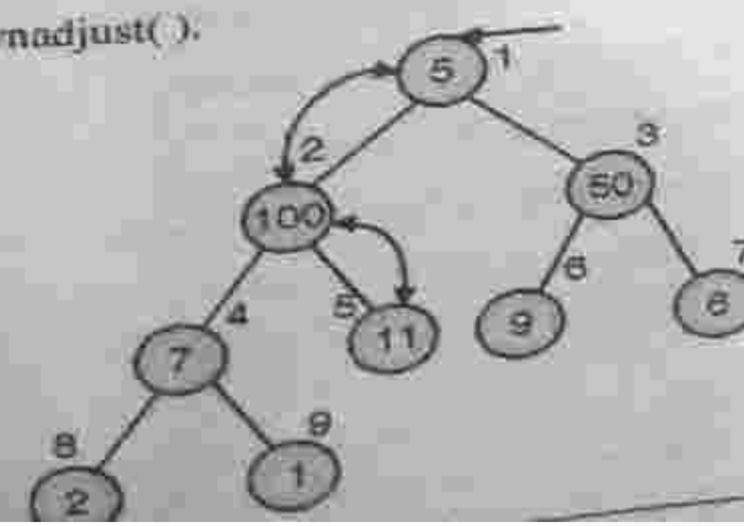
Step 1 : Tree rooted at node no. 4 is converted to a heap through downadjust( )



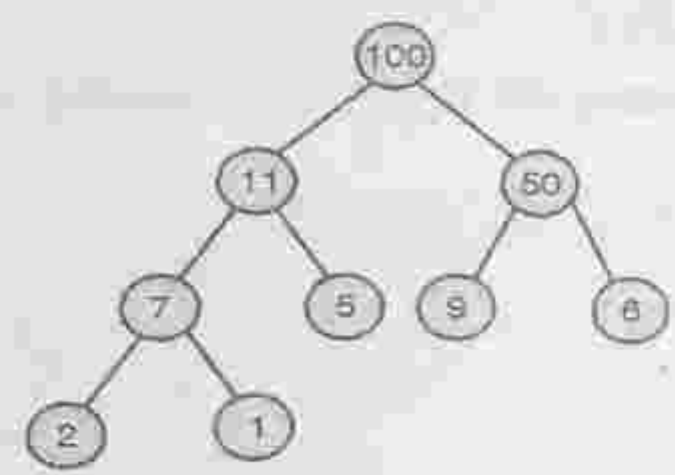
Step 2 : Tree rooted at node no. 3 is converted to a heap through downadjust( ).



Step 3 : Tree rooted at node no. 2 is converted to a heap through downadjust( ).



Step 4 : Tree rooted at node no. 1 is converted to a heap through downadjust( ).



Now, a heap

Example 8.11.2

Sort the following number using heap sort. 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

Create the heap first and then sort it. Show each step separately.

Step 1 : Creation of max heap [through repeated insertion]

Insert **Make it the last node** **Heap after adjustment**

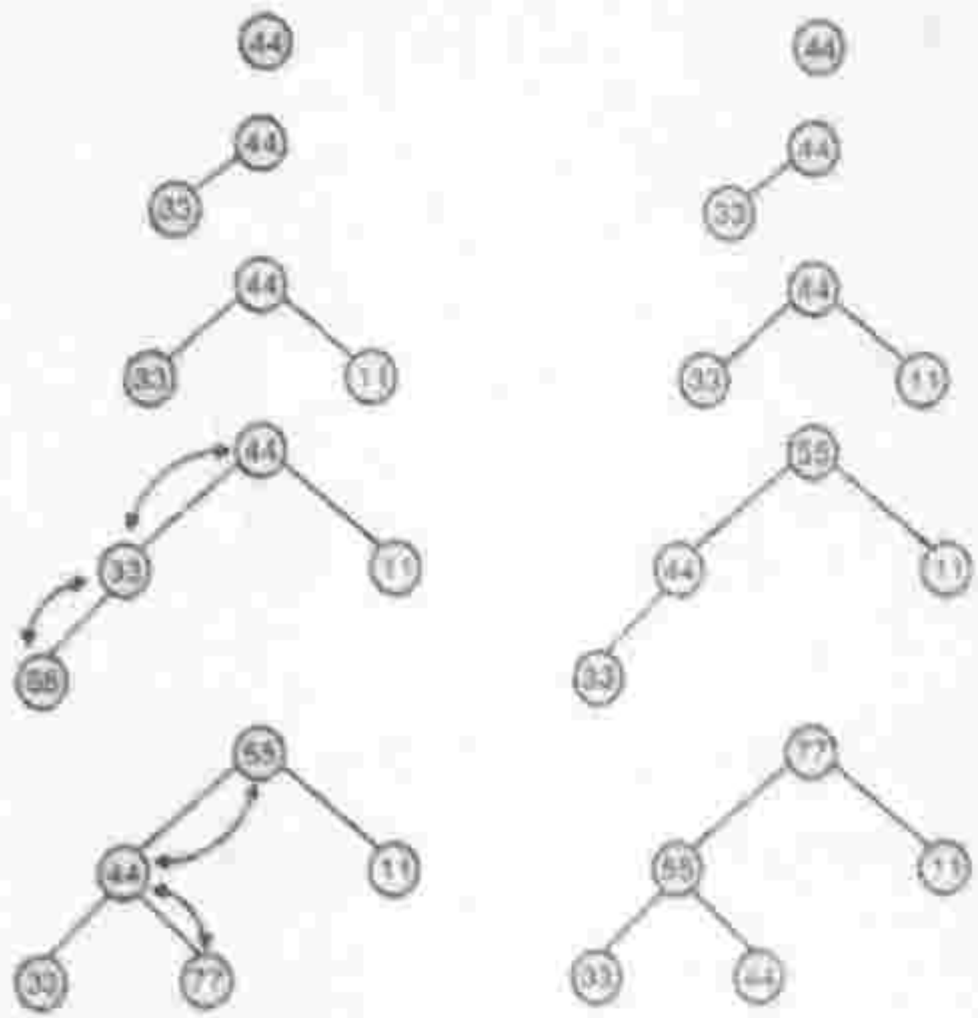
44

33

11

55

77



Insert **Make it the last node** **Heap after adjustment**

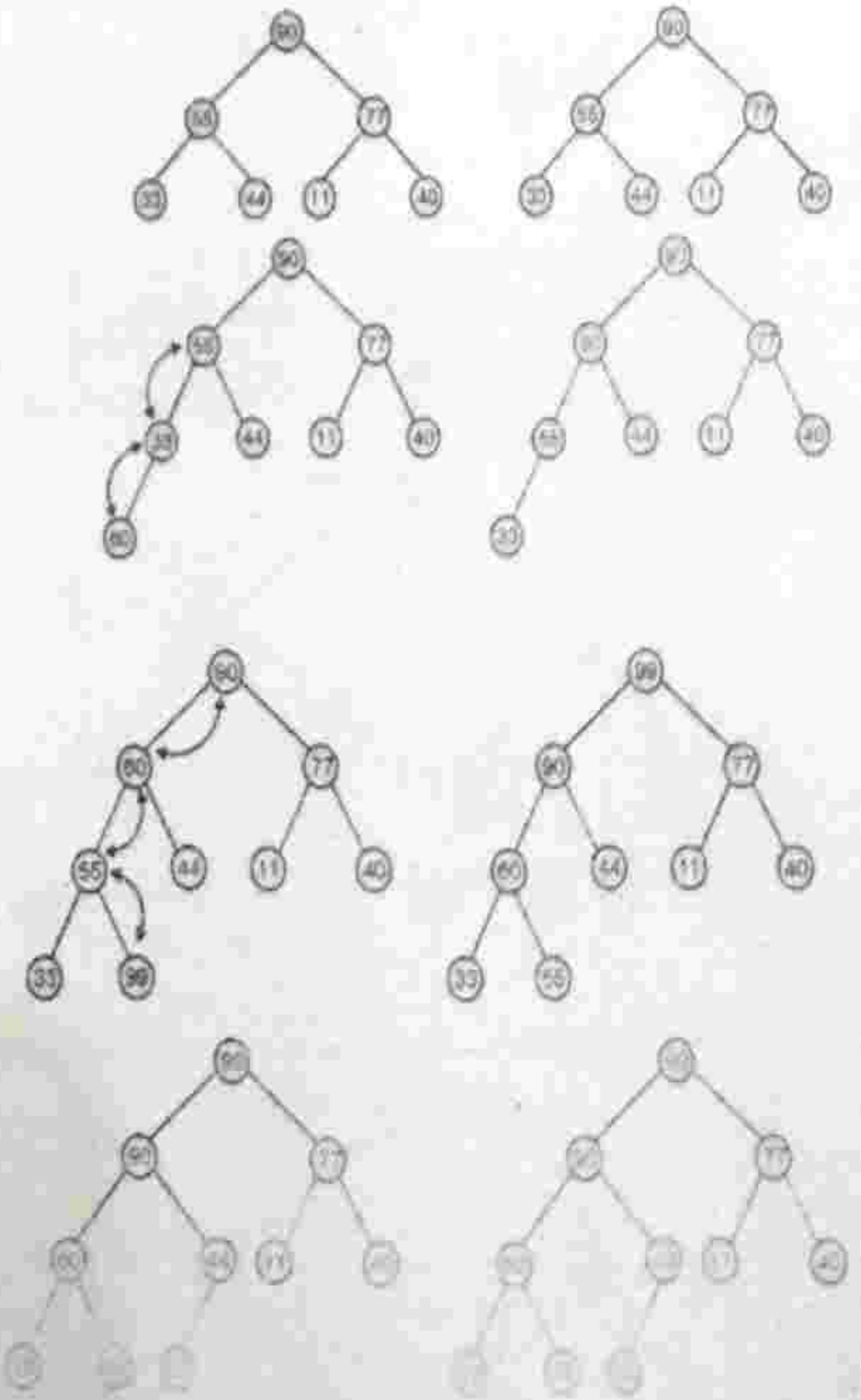
90

90

60

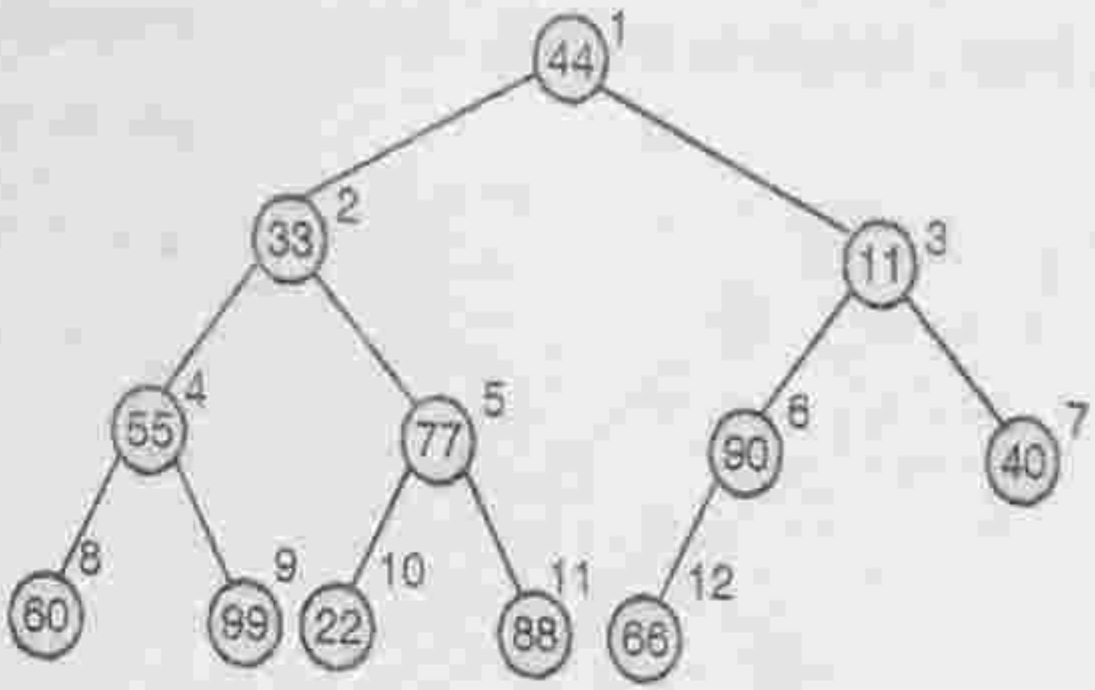
99

22

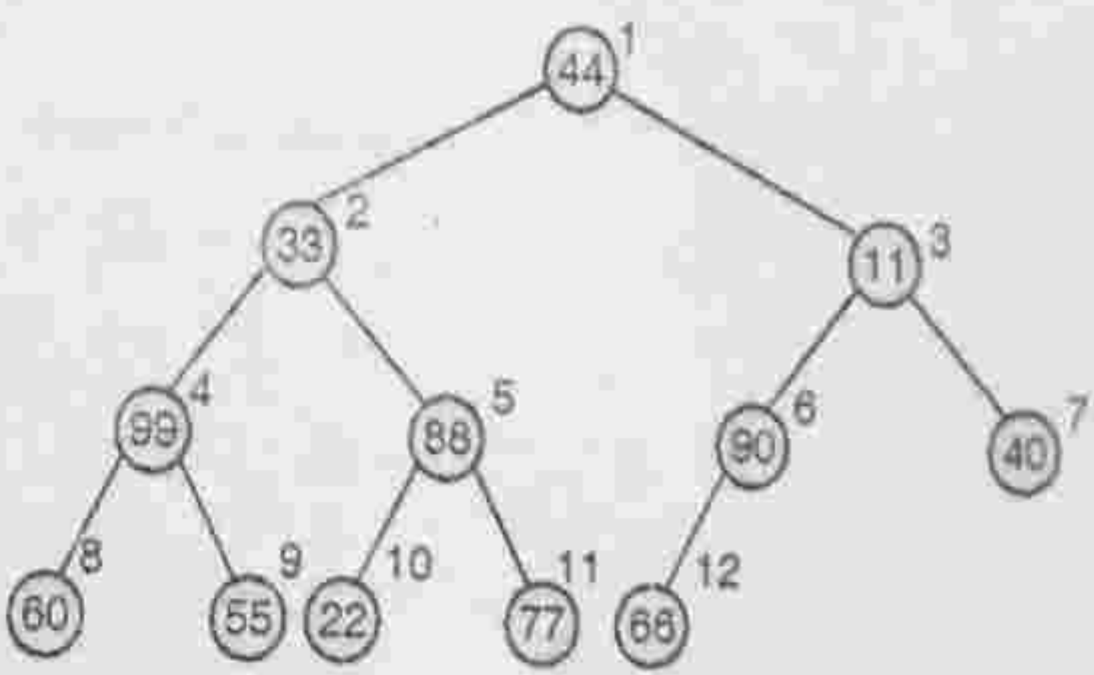


Step 1 : Creation of heap [using a better technique]

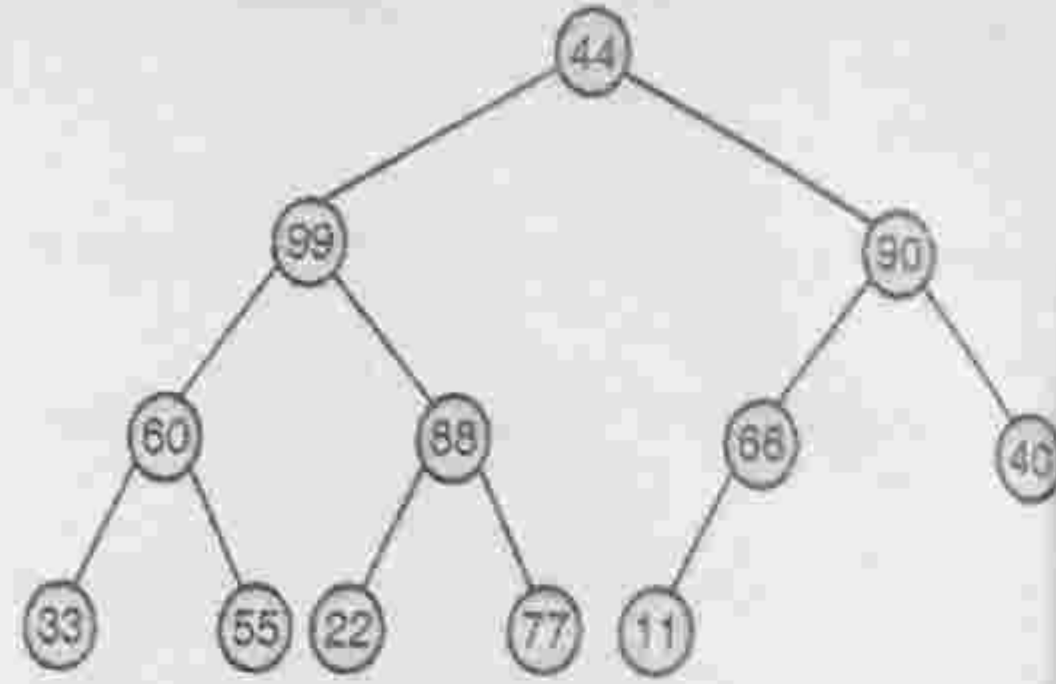
Given data represent the binary tree :



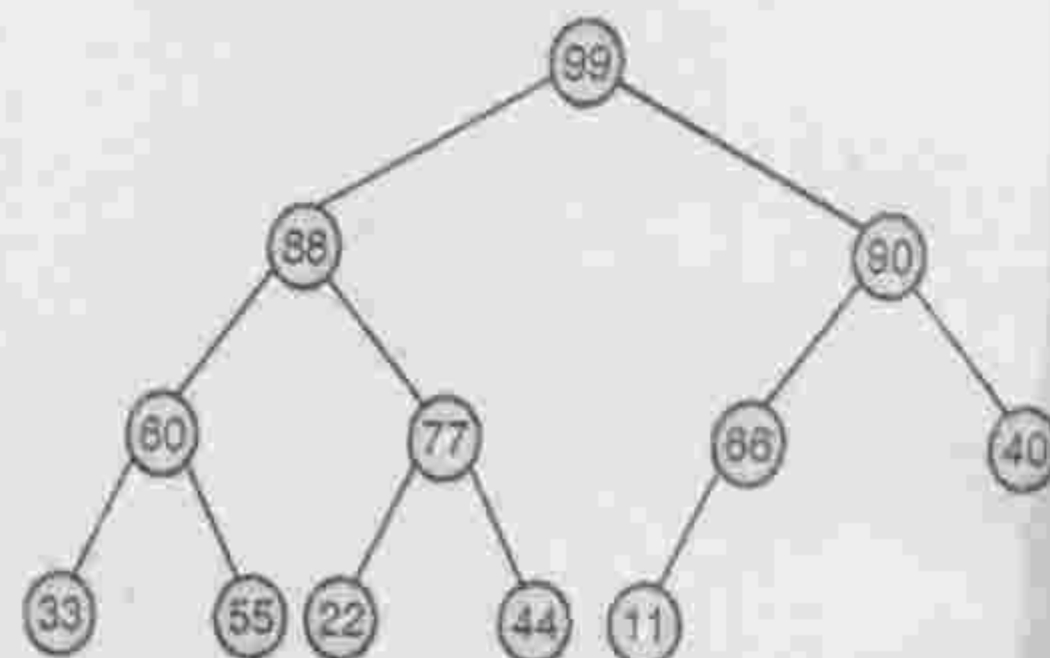
The above binary tree can be converted into a heap by down-adjusting nodes 6, 5, 4, 3, 2, 1. Down-adjusting nodes 6, 5 and 4, we get



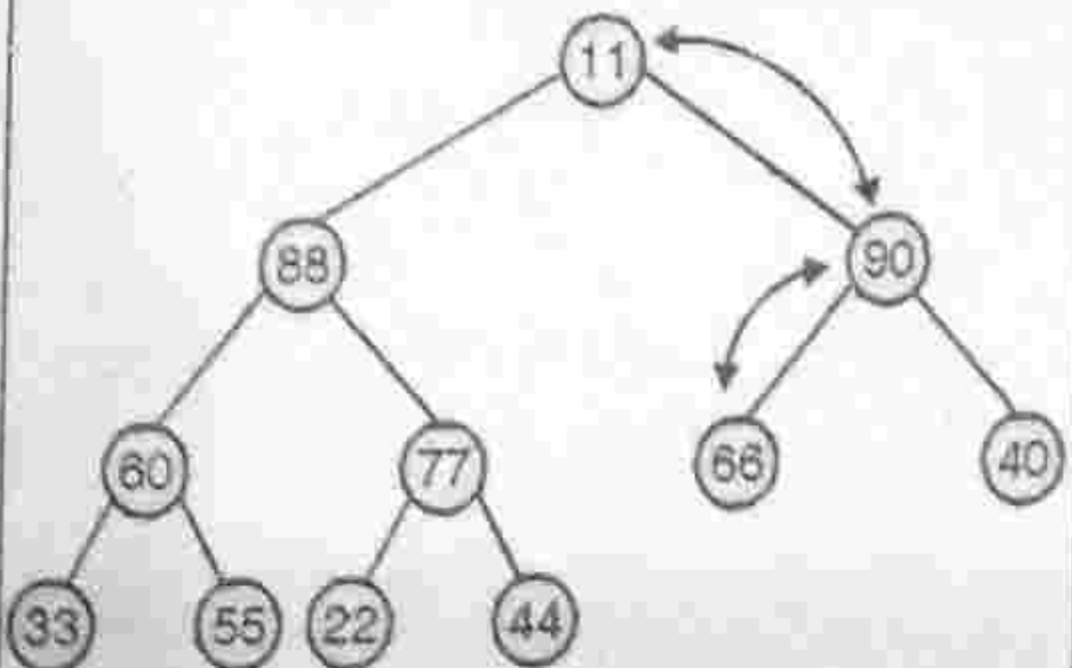
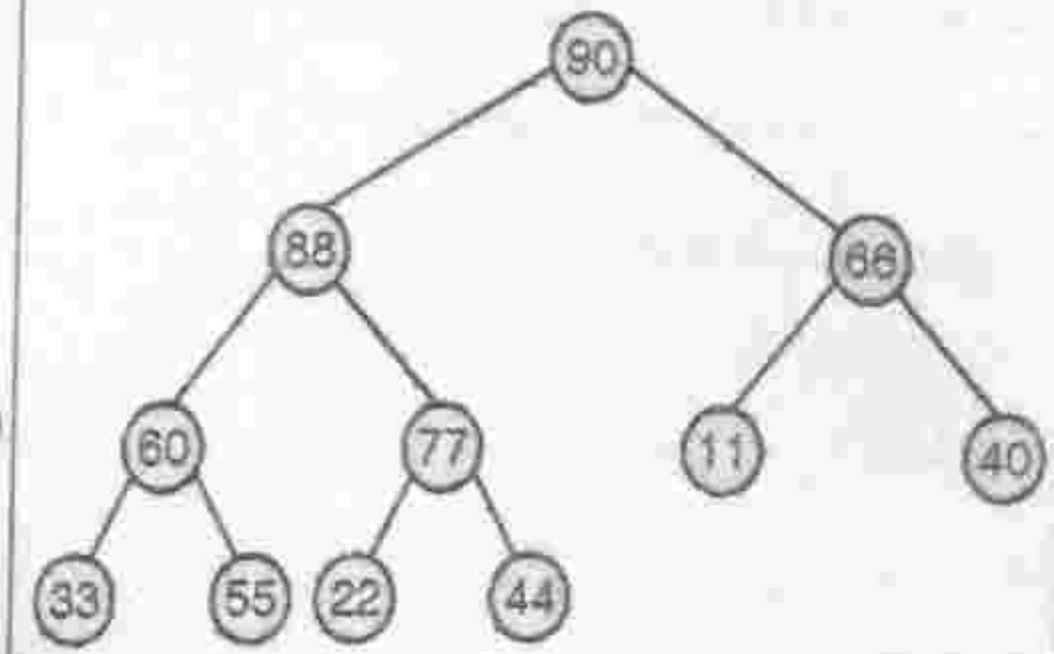
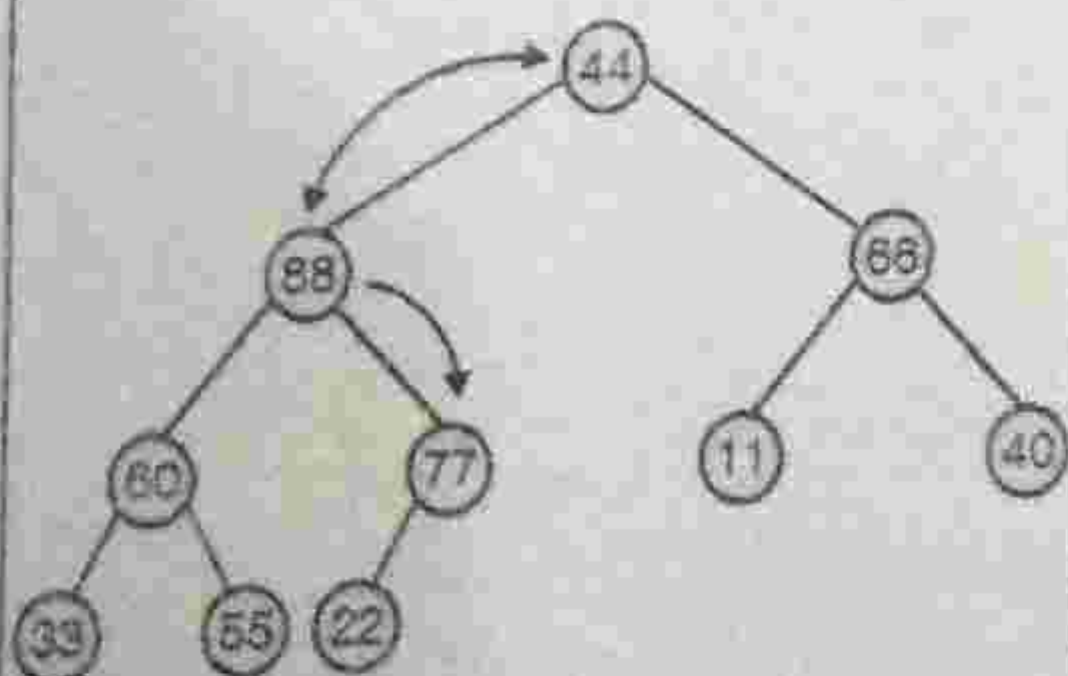
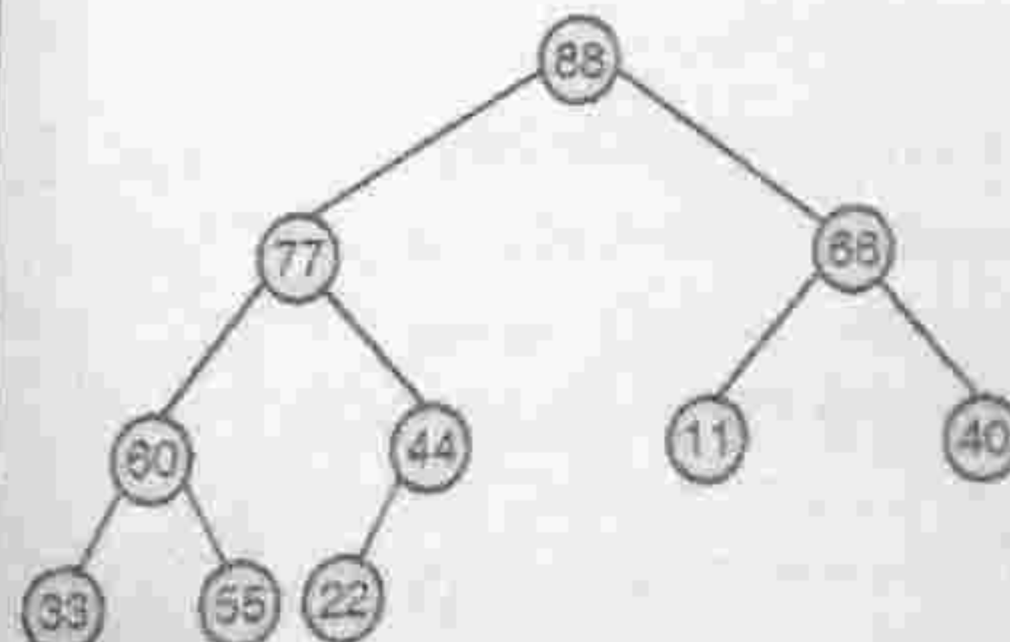
Down-adjusting nodes 3 and 2, we get

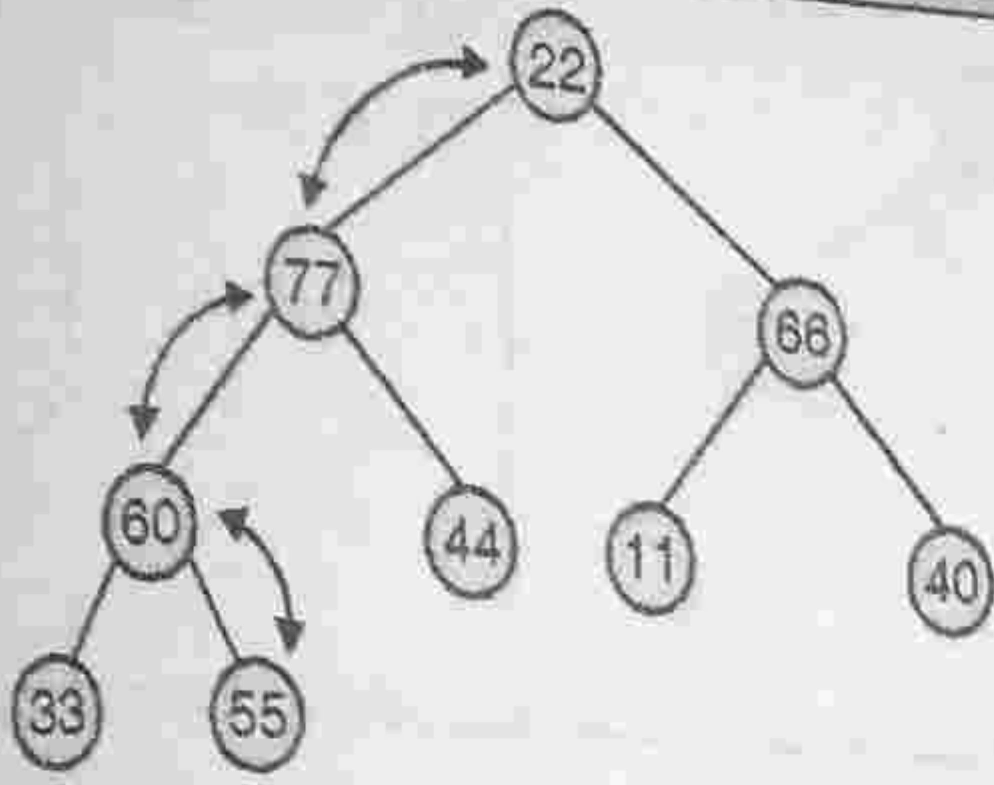
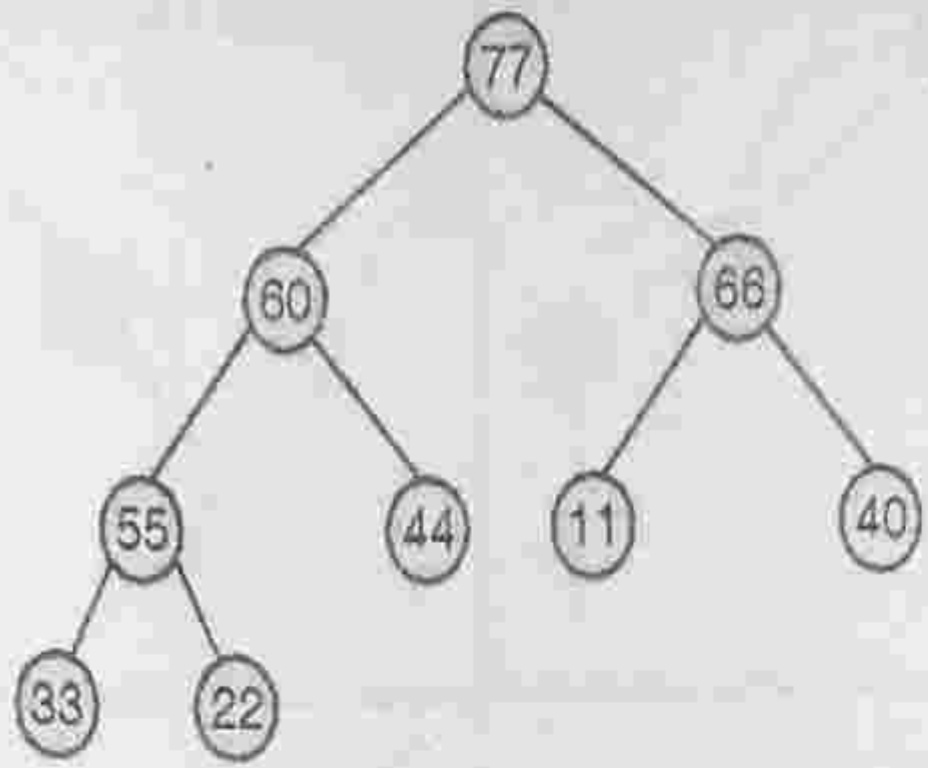
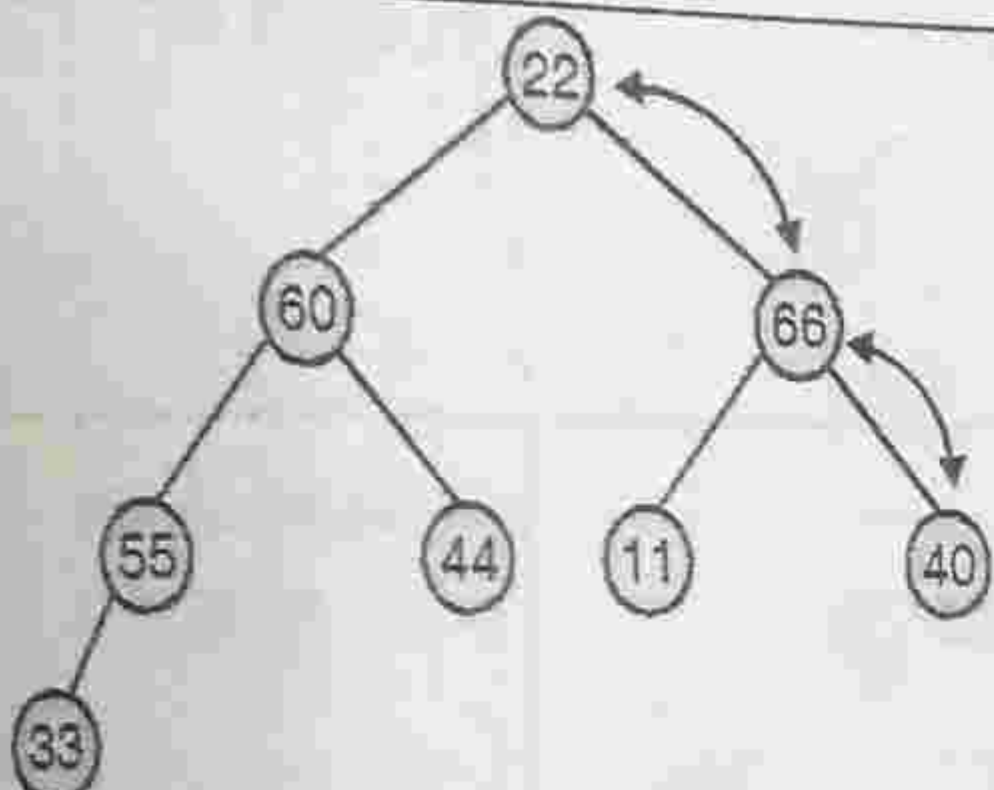
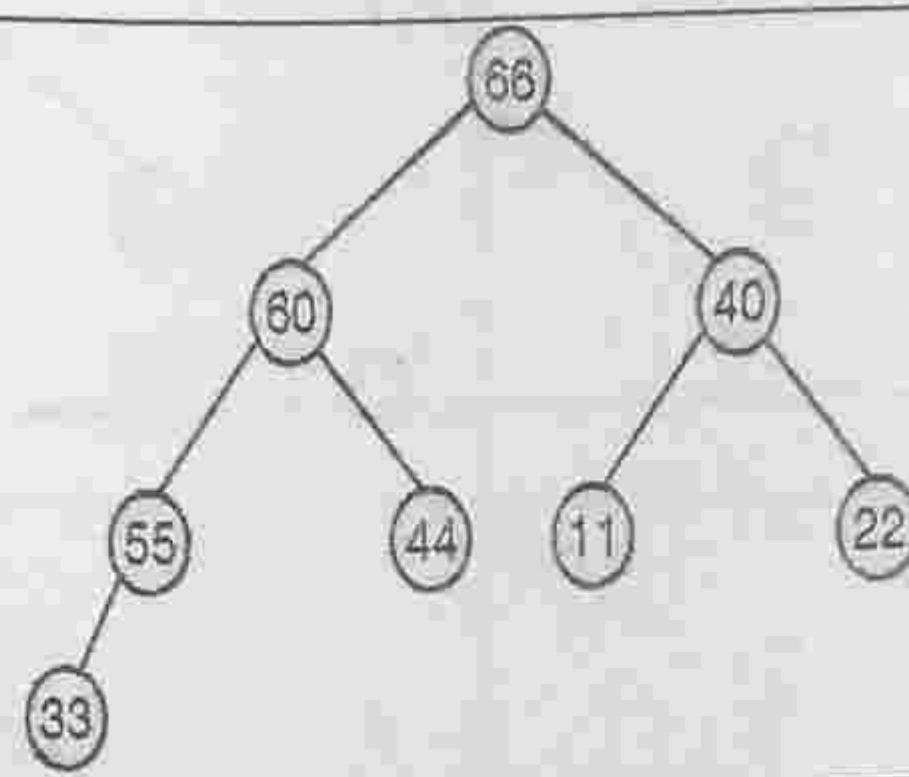


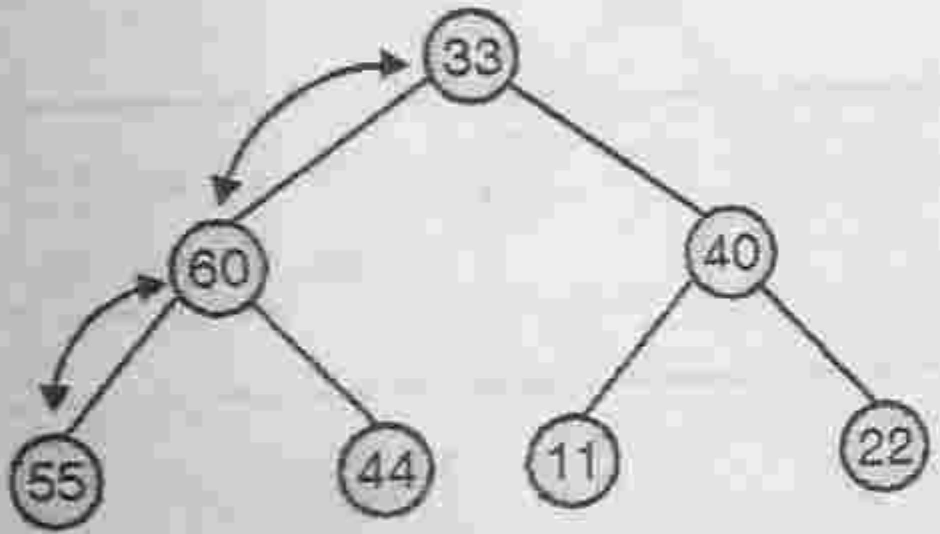
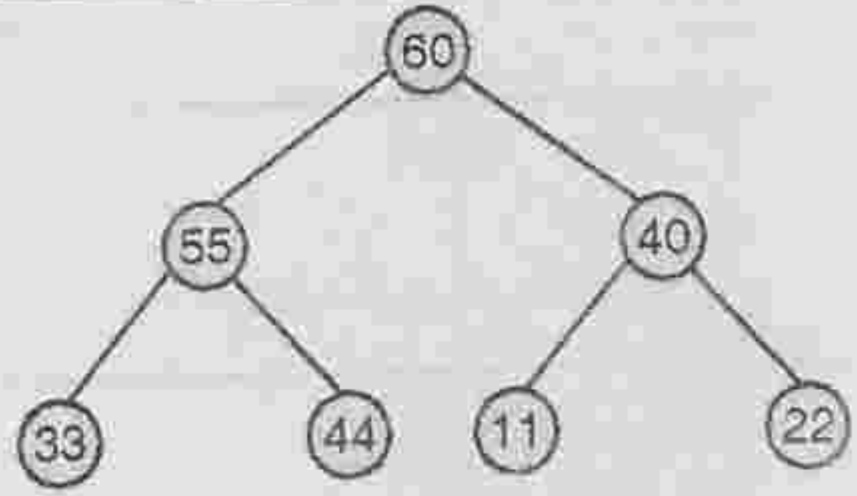
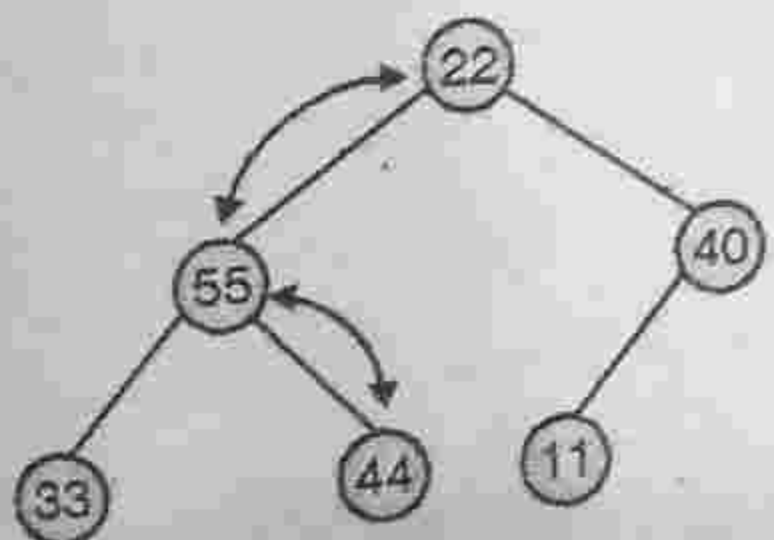
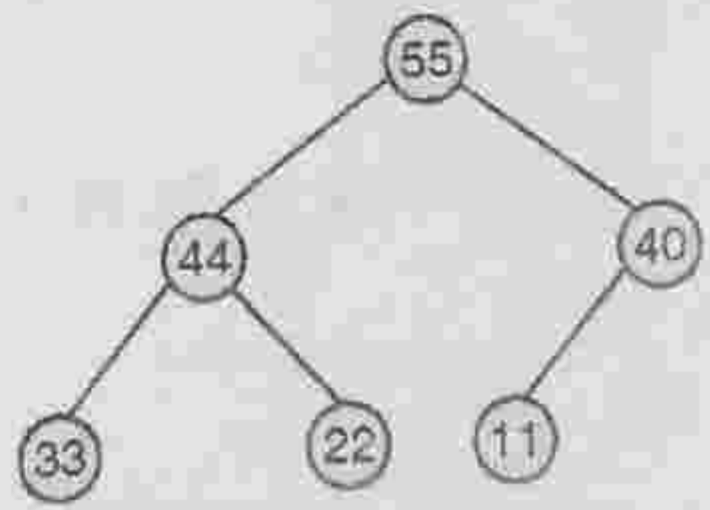
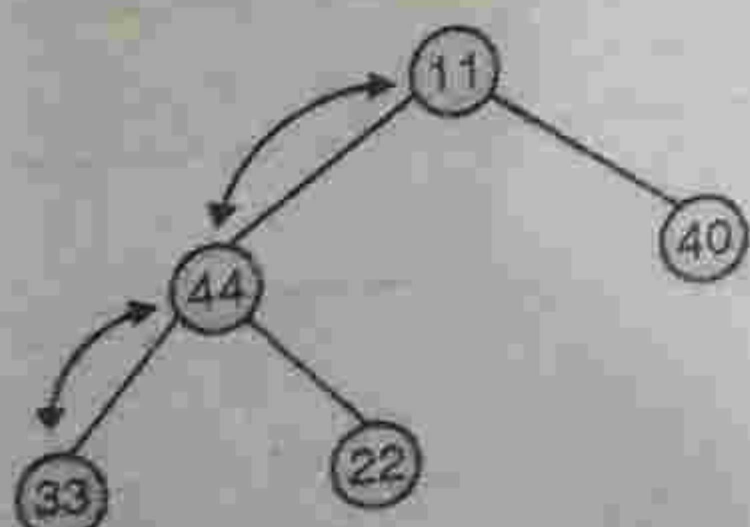
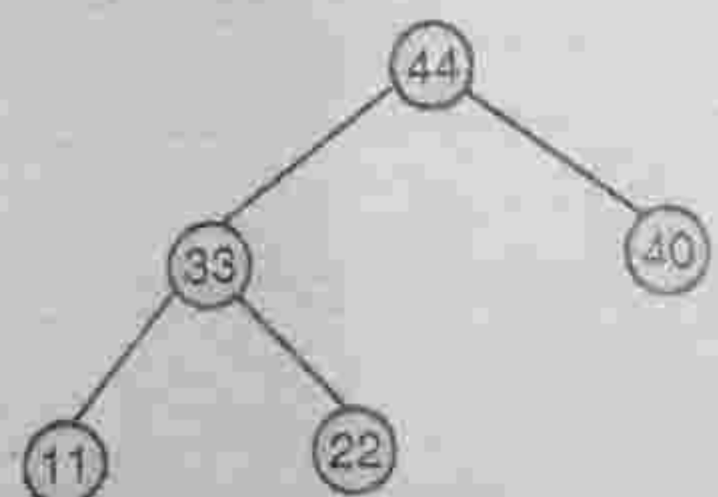
Down-adjusting node 1, we get

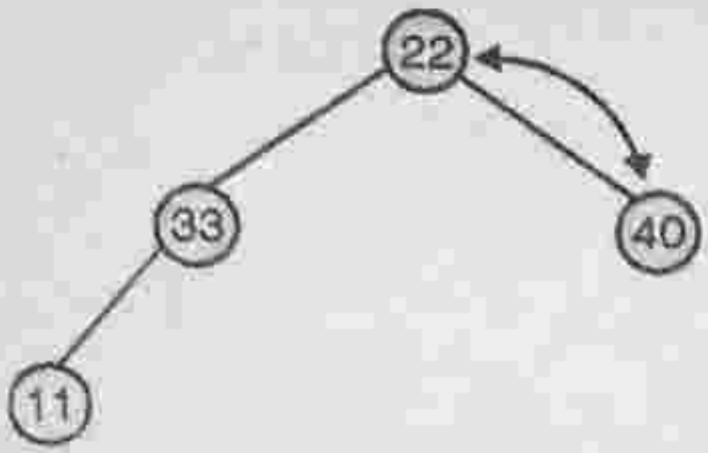
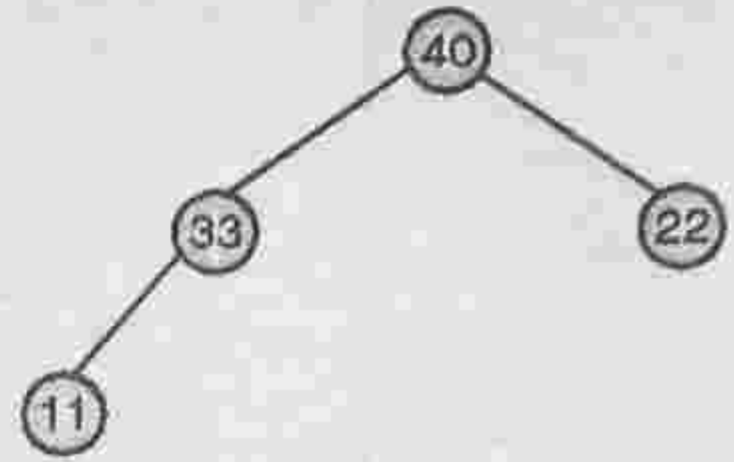
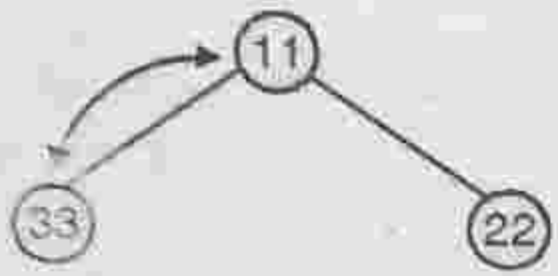
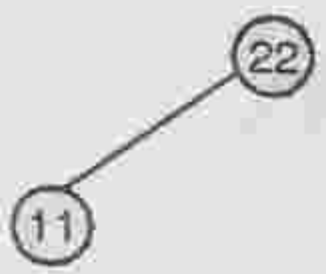
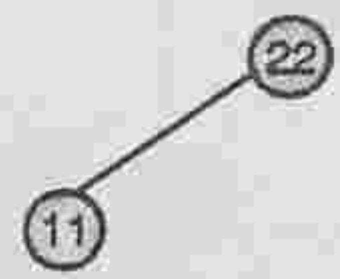


Step 2 : Sorting

<table>
<tr><th>Interchange 1st and the last element and delete the last element</th><th>Down-adjust the root</th><th>Sorted data</th></tr>
<tr><td>
</td><td>
</td><td>99</td></tr>
<tr><td>
</td><td>
</td><td>90, 99</td></tr>
</table>

Interchange 1st and the last element and delete the last element	Down-adjust the root	Sorted data
		88, 90, 99
		77, 88, 90, 99

		66, 77, 88, 90, 99
		60, 66, 77, 88, 90, 99
		55, 60, 66, 77, 88, 90, 99

Interchange 1st and the last element and delete the last element	Down-adjust the root	Sorted data
		44, 55, 60, 66, 77, 88, 90, 99
		40, 44, 55, 60, 66, 77, 88, 90, 99
		33, 40, 44, 55, 60, 66, 77, 88, 90, 99
11	11	22, 33, 40, 44, 55, 60, 66, 77, 88, 90, 99
—	—	11, 22, 33, 44, 55, 60, 66, 77, 88, 90, 99

Comparisoin of sorting

	Best-case	Worst-case	Average-case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix sort	$O(n)$	$O(n)$	$O(n)$

Comparisoin of sorting

III. COMPARISON OF SORTING ALGORITHM IN TABULAR FORM

Sort	*Time Complexity*	*Advantages & disadvantages*
Insertion Sort	O(n)	The advantage of insertion sort is its simplicity. It is also good performance for smallest array. The disadvantage of insertion sort is that it is not useful for large elements array.
Selection Sort	O(n^2)	The advantage of selection sort is that it performs well on small array. The disadvantage of selection is that it is poor efficiency for large elements array.
Bubble Sort	O(n^2)	The advantage of bubble sort is that it is easily implemented. In bubble sort, the elements are swapped without additional temporary storage, so space requirement is minimum. The disadvantage of bubble sort is same as a selection sort.
Quick Sort	O(n log n)	The advantage of Quick sort is that it is used for small elements of array as well as large elements of array. Disadvantage of Quick sort is that the worst case of quick sort is same as a bubble sort or selection sort.

Comparisoin of sorting

BASIS FOR COMPARISON	BUBBLE SORT	SELECTION SORT
Basic	Adjacent element is compared and swapped	Largest element is selected and swapped with the last element (in case of ascending order).
Best case time complexity	O(n)	$O(n^2)$
Efficiency	Inefficient	Improved efficiency as compared to bubble sort
Stable	Yes	No
Method	Exchanging	Selection
Speed	Slow	Fast as compared to bubble sort

COMPARISON OF ALL SORTING METHODS

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Bubble sort	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. A simple and easy method 2. Efficient for small lists $n > 100$	Highly inefficient for large data

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Selection sort	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps for the remainder of the list (starting at the second position and advancing each time)	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. Recommended for small files 2. Good for partially sorted data	Inefficient for large lists

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Insertion sort	Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already sorted list until no input elements remain. The choice of which element to remove from the input is arbitrary	$O(n)$	$O(n^2)$	No extra space needed	Yes	1. Relatively simple and easy to implement 2. Good for almost sorted data	Inefficient for large lists

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Quick sort	Picks an element, called a pivot, from the list. Reorders the list so that all elements with values less than the pivot come before the pivot, whereas all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation. Recursively sorts the sub-list of the lesser elements and the sub-list of the greater elements	$O(n\log_2 n)$	$O(n^2)$	No extra space needed	Yes	1. Extremely fast 2. Inherently recursive	Very complex algorithm

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Shell Sort	It is a generalization of insertion sort, which exploits the fact that insertion sort works efficiently on input that is already almost sorted. It improves on insertion sort by allowing the comparison and	$O(n^{1.5})$	$O(n\log^2 n)$	No extra space needed	No	1. It is faster than a quick sort for small arrays 2. Its speed and simplicity makes it a good choice in practice	Slower for sufficiently big arrays

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Radix sort (most significant digit)	Numbers are placed at proper location by processing individual digits and by comparing individual digits that share the same significant position	$O(n)$	$O(n)$	Extra space proportional to n is needed	Yes	1. Radix sort is very simple, and fast 2. In-Place, Recursive and one of the fastest sorting algorithms for numbers or strings of letters	Radix sort can also take more space than other sorting algorithms since in addition to the array that will be sorted, there needs to be a sub-list for each of the possible digits or letters

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Merge sort	Conceptually, a merge sort works as follows: If the list is of length 0 or 1, then it is already sorted. Otherwise, the algorithm divides the unsorted list into two sub-lists of about half the size Then, it sorts each sub-list recursively by reapplying the merge sort and then merges the two sub-lists back into one sorted list	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra space proportional to n is needed	Yes	1. Good for external file sorting 2.Can be applied to files of any size	1. It requires twice the memory of the heap sort because of the second array used to store the sorted list. 2. It is recursive, which can make it a bad choice for applications that run on machines with limited memory

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Heap sort	Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full	$O(n\log_2 n)$	$O(n\log_2 n)$	No extra space needed	No	1. Advantageous as it does not use recursion and that heap sort works just as fast for any data order. That is, there is basically no worst-case scenario 2. Heaps work well for small tables and the tables where changes are infrequent	Do not work well for most large tables

BUBBLE SORT

```c
#include <stdio.h>
int main()
{
 int arr[100], n, i, j, temp;
 printf("Enter number of elements you want to sort :");
 scanf("%d", &n);
 printf("Enter %d elements : ", n);
 for (i = 0; i < n; i++)
  scanf("%d", &arr[i]);
 for (i = 0 ; i < ( n - 1 ); i++)
 {
  for (j = 0 ; j < n - i ; j++)
  {
   if (arr[j] > arr[j+1])
   {
    temp    = arr[j];
    arr[j]  = arr[j+1];
    arr[j+1] = temp;
   }
  }
 }
printf("Sorted elements are : ");
for ( i = 0 ; i < n ; i++ )
{
  printf("%d ", arr[i]);
}
  printf("\n\n");
 return 0;
}
```

Accepts array elements

In unsorted array, if the first position element is greater than its next element then they get swapped.

Prints sorted elements

Output

```
Enter number of elements you want to sort :5
Enter 5 elements : 31 55 27 16 49
Sorted elements are : 16 27 31 49 55
```

INSERTION SORT

```
#include<stdio.h>
int main()

{

    int arr[100],n,temp,k,j;

    printf("Enter number of elements you want to be sort : ");
    scanf("%d",&n);
    printf("Enter %d elements: ",n);
    for(k=0;k<n;k++)
    {
    scanf("%d",&arr[k]);
    }

    for(k=1;k<n;k++)
    {
        temp = arr[k];
        j=k-1;
    while(temp<arr[j] && j>=0)
        {
            arr[j+1] = arr[j];
            --j;
        }
        arr[j+1]=temp;
    }

    printf("Sorted elements are : ");
    for(k=0; k<n; k++)
    {
    printf("%d\t",arr[k]);
    }
    printf("\n\n");
    return 0;
}
```

Accepts elements from user

If element is less than any previous element then element is inserted at position of previous element and the position of that previous element shifted one position to right.

Prints sorted elements

BUCKET SORT

```
#include<stdio.h>
void main()
{
        int frequency[1000],n,i,x;
        printf("\n Enter no. of elements :");
        scanf("%d",&n);
        /* Initialise the array frequency */
        for(i=0;i<1000;i++)
                frequency[i]=0;
        printf("\n Enter the data to be sorted :");
        for(i=0;i<n;i++)
        {
                scanf("%d",&x);
                frequency[x]++;
        }
        /* print the result */
        printf("\nresult is :");
        for(i=0;i<=n;i++)
                if(frequency[i]>0)
                        while(frequency[i]>0)
                        {
                                printf("%d\t",i);
                                frequency[i]--;
```

Thanks ……..!!!!

Prof. ANAND GHARU

ASSISTANT PROFESSOR

Blog : anandgharu.wordpress.com