UNIT 2

Syllabus: Sorting and Searching Brute force approach: General method -Sorting (bubble, selection, insertion) –Searching (Sequential/Linear) Divide and Conquer approach: General method - Sorting (merge, quick) – Searching (Binary Search).

Sorting Approach

Systematic arrangement of data is called SORTING. For example – in telephone directory the person surname are arranged in increasing or decreasing order. The **sorting is a technique** by which the elements are arranged in some particular order. Usually the sorting order is of two types-

Ascending order: It is the sorting order in which the elements are arranged from low value to high value. In other words elements are in increasing order. **For example: 10, 50, 40, 20, 30** can be arranged in awarding order after applying some sorting technique as **10, 20. 30, 40, 50**

Descending Order: It is the sorting order in which the elements are arranged from high value to low value. In other words elements are in decreasing order. It is reverse of the ascending order. **For example: 10, 50, 40, 20, 30 can** be arranged in descending order after applying some sorting technique as **50,40,30,20,10**

While sorting the elements, we always consider a specific order and expect our data to be arranged in that order.

Need for sorting

Sorting is useful for arranging the data in desired order. After sorting the required element can be located easily. Sorting is needed due to following reasons

1. The sorting is useful in database applications for **arranging the data** in desired order.
2. In the **dictionary like applications** the data is arranged in sorted order.
3. For **searching the element** from list of elements, the sorting is required.
4. For **checking the uniqueness** of the element the sorting is required.
5. For **finding the closest pair** from the list of elements the sorting is required.

Types of sorting algorithm - Internal & External sorting algo.

There are two types of sorting techniques - **Internal sorting** and **External sorting**.

Internal sorting – In the internal sorting **data resides on main memory** of the computer. It is used to **sort small amount** of data. This type of sorting is **faster** in comparison to external sorting and required low memory for sorting. Various internal sorting techniques are - bubble sort, insertion sort, selection sort.

External sorting - In external sorting, the **data stored on secondary memory** is part by part loaded into main memory, sorting can be done over there. The **sorted data** can be then **stored in the intermediate files**. Finally these intermediate files can be merged repeatedly to get sorted data. Thus **huge amount of data** can be

sorted using this technique. The external merge sort is a technique in which the data is loaded in intermediate files. Each **intermediate file** is **sorted independently** and then **combined or merged** to get the sorted data.

For example: Consider that there are 10,000 records that has to be sorted- Clearly we need to apply external sorting method. Suppose main memory has a capacity to store 500 records in blocks, with each block size of 100 records.

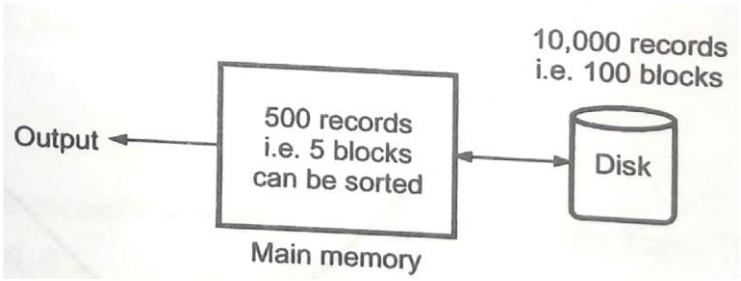


Fig. 1 External Sorting

The sorted 5 blocks (i.e. 500 records) are stored in intermediate file. This process will be repeated 20 times to get all the records sorted in chunks. In the second step, we start merging a pair of intermediate files in the main memory to get output file.

Difference between internal and external sorting

Sr.	Internal sorting	External sorting
1	The internal sorting can be performed for sorting small amount of data	The internal sorting can be performed for sorting large amount of data
2	In internal sorting all the data lies on same main memory.	In external sorting all the data cannot be accommodated on the single memory.
3	It does not require secondary memory for sorting the data.	Some amount of secondary memory needs to be kept all the data for sorting such as hard disk compact disk and so on
4	No intermediate file is required to store data	Intermediate file is required to sort and store data individually. Then we combine all files to get final sorted data

Stable sorting Algorithm

A sorting algorithm is said to be stable **if two objects with equal keys appear in the same order** in sorted output as they appear in the input unsorted array. **For example:** Consider the list of strings **(mango, apple, grapse, guava, pomegranate).** If the sorting is based on number of characters in a string then, the sorted list will be **(mango, apple, guava, grapse, pomegranate)**

The given strings "mango", "apple", guava" all are of length 5. Hence they are arranged in the order in which they appear in the input list. Various sorting techniques such as **merge sort, radix sort** are the **example of stable sorting** algorithm

Outline and offline algorithm

Online/outline Algorithm: The online algorithms are the algorithms in which the algorithm **gets one element at time to process**. **For example** – heap sort is a online algorithm in which each input element is retrieved and processed at a time.

Offline Algorithm: The offline algorithms are a kind of algorithm in which the **entire list of elements is available** at the **time of processing**. **For example** - selection sort is a sorting technique in which entire list is scanned for finding minimum element each time. Thus entire list is available at the time of processing in case of selection sort

General method - Sorting (bubble, selection, insertion)

Sorting Techniques

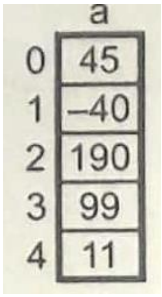
Sorting is an **important activity** and every time we insert or delete the data at the time of sorting, the remaining data retain in queue to sort. Various algorithms that are developed for sorting are as follows -

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

Bubble Sorting

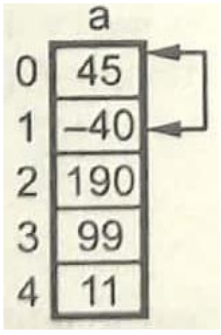
This is the simplest kind of sorting method. In **bubble sort procedure** we perform several **iterations in groups** which are **called passes**. In this procedure first each element is filled in array. Now in each pass we compare a[0] element of array with next element i.e. a[1]. If a[0] is greater than a[1] we interchange the value and if it is not greater then value remain same and we move to next step and compare a[1] with a[2]. Similarly if a[1] is greater than a[2] we interchange the value otherwise not and move to next step. Like this when we reach at last position, largest element comes at last position in array and 1st pass is over. Now list is sorted upto some extend. Similarly passes is repeated from a[0] to a[n] till all element get sort.

Example: Consider 5 unsorted elements are 45, -40, 190, 99, 11.

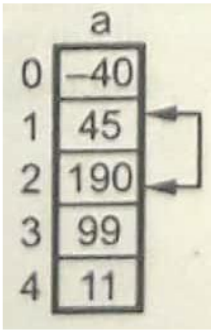


First store those elements in the **array a**

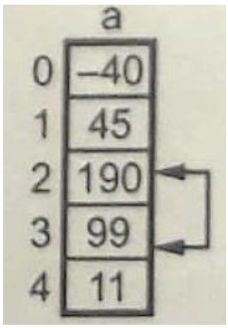
Pass 1: In this pass each element will be compared with its neighbouring element.



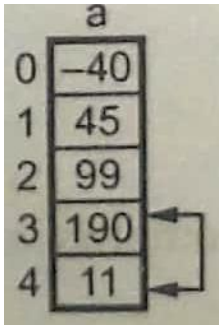
Compare a[0] and a[l]
i.e. compare 45 and -40
if a[0] is greater than a[1]
Interchange them
Therefore a[0] = - 40 & a[l] = 45



Compare a[1] and a[2]
i.e. compare 45 and 190
if a[1] is smaller than a[2]
No Interchange will done
Therefore a[1] = 45 & a[2] = 190



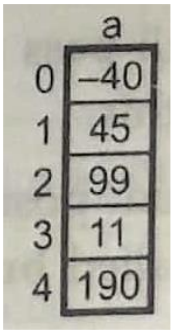
Compare a[2] and a[3]
i.e. compare 190 and 99
if a[2] is greater than a[3]
Interchange them
Therefore a[2] = 99 & a[3] = 190

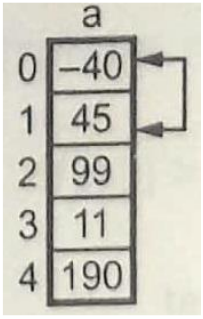


Compare a[3] and a[4]
i.e. compare 190 and 11
if a[3] is greater than a[4]
Interchange them
Therefore a[3] = 190 & a[4] = 11

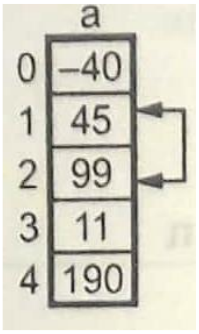
After first pass the array will hold the elements which are sorted to some extent.

Pass2:

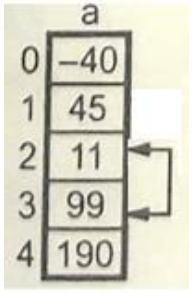




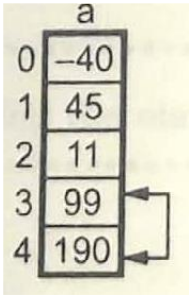
Compare a[0] and a[1]
no interchange



Compare a[1] and a[2]
no interchange.

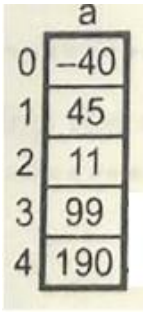


Compare a[2] and a[3]
Since 99 > 11 Interchange
a[2] = 11 & a[3] = 99

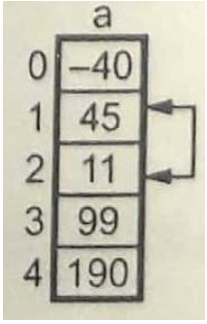


Compare a[3] and a[4]
no interchange.

Pass 3:



Compare a[0] and a[1]
No interchange



Compare a[1] and a[2]
45 > 11 Interchange
a[1] = 11 & a[2] = 45

Next compare a[2] and a[3]
No interchange

Compare a[3] and a[4]
No interchange

This is end of pass 3. This process will be thus continued till pass 4. As element is sorted so no interchange will happen in pass 4, so diagram is not shown here. Finally at the [end of last pass the array will hold all the sorted elements like this, Since the comparison positions look like bubbles, therefore it is called bubble sort.

Algorithm:

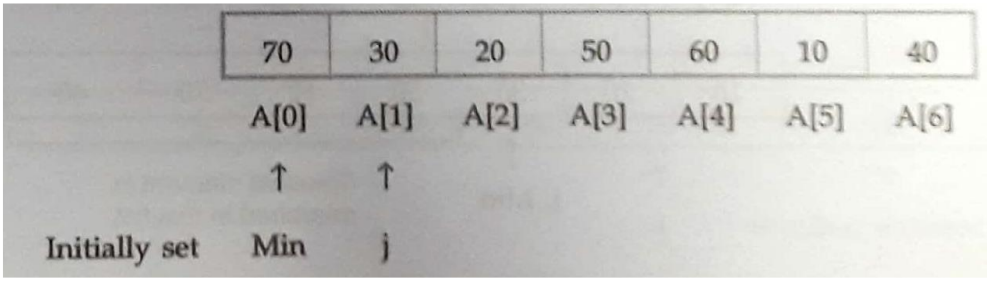
1. Read the total number of n elements.
2. Store the elements in the array.
3. Set the i = 0.
4. Compare the adjacent elements i.e. a[0] with a[1].
   4.1. If a[0] > a[1] interchange elements
        Otherwise no interchange is done
   4.2. Move to next adjacent element i.e. a[1] and a[2]
5. Repeat step 4 for all n elements.
6. Increment the value of i by 1 and repeat step 4, 5 for i < n
7. Print; the sorted list of elements.
8. Stop.

Selection Sorting

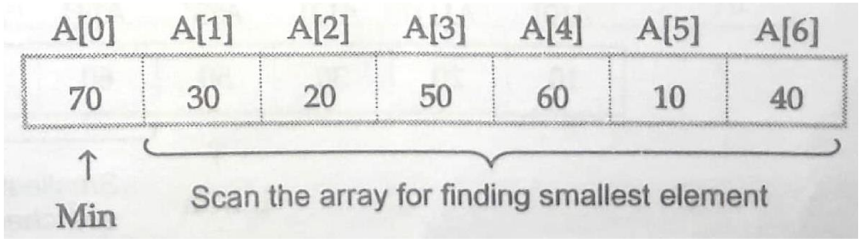
Scan the array to **find its smallest** element and **swap** it with the **first element.** Then, starting with the second element scan the entire list to find the smallest element and swap it with the second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

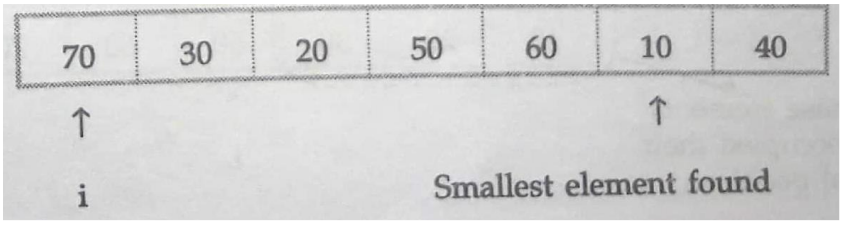
Generally, on pass i (0 <= i <= n-2), the smallest element is searched among last n-i elements and is swapped with A[i]. The list gets sorted after n-l passes.

Example: Consider the elements **70, 30, 20, 50, 60, 10, 40** We can store these elements in array A as :

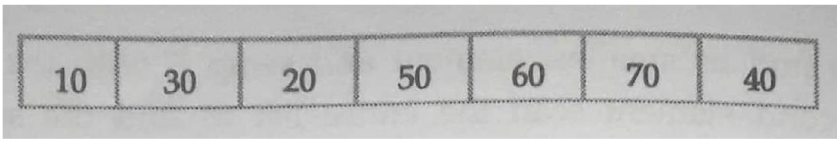


1st Pass:

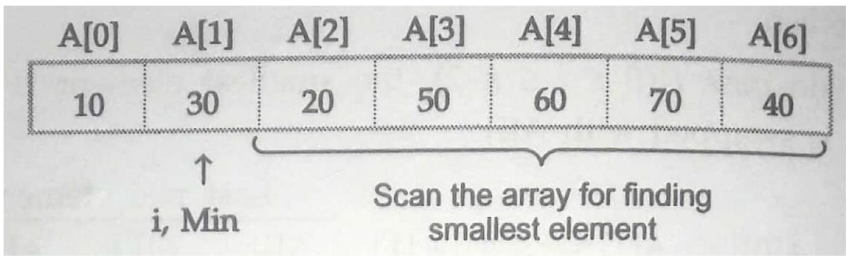


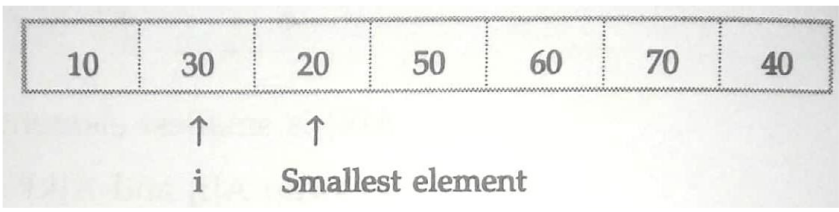


Now Swap **A[i]** with smallest element. The array then becomes,



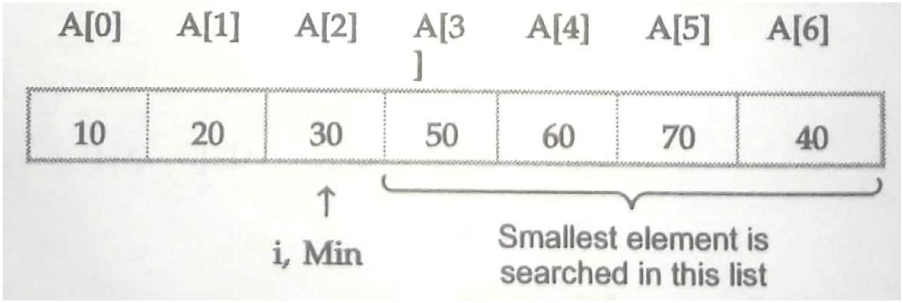
2nd Pass:





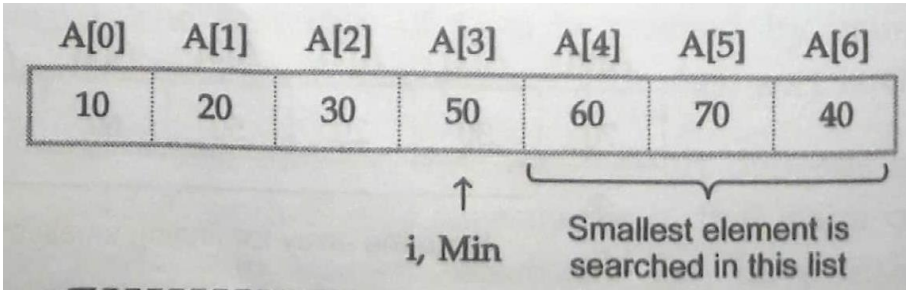
Swap **A[i]** with smallest element. The array then becomes,

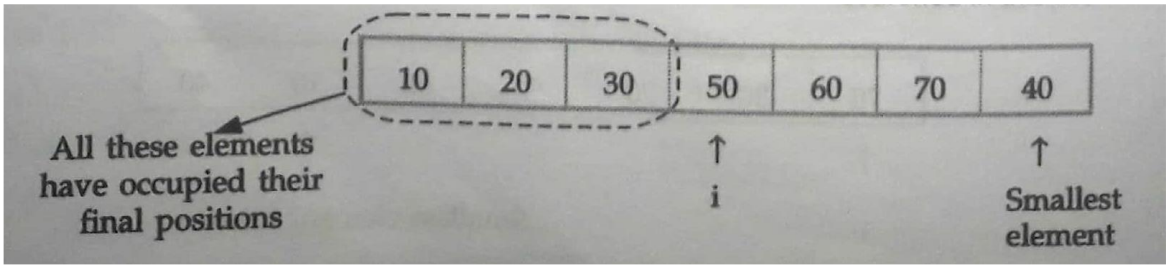
3rd Pass:



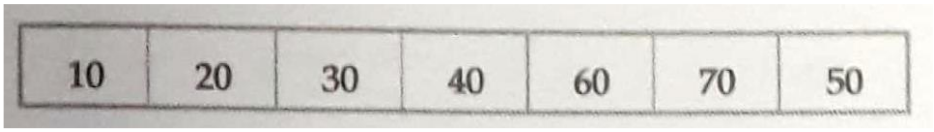
As there is no smallest element than 30 so we will increment i pointer

4th Pass:

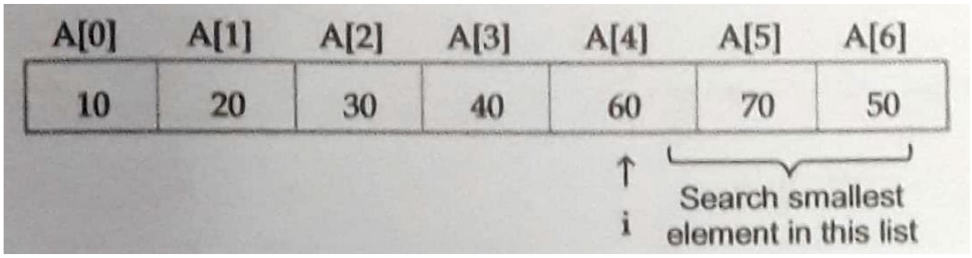


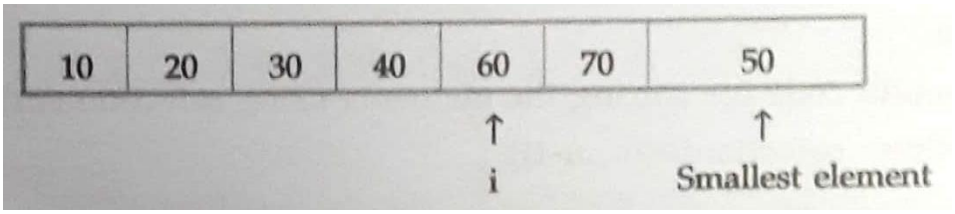


Swap **A[i]** with smallest element. The array then becomes,

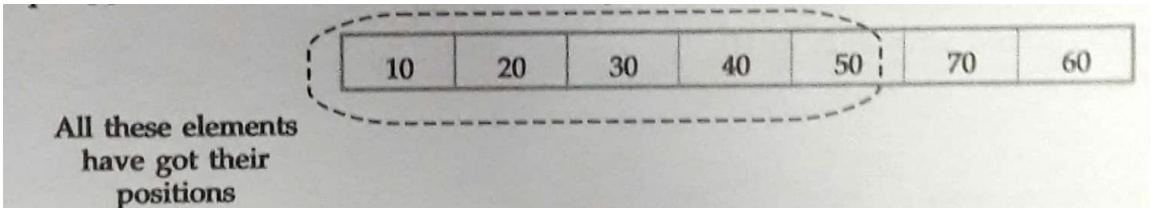


5th Pass:

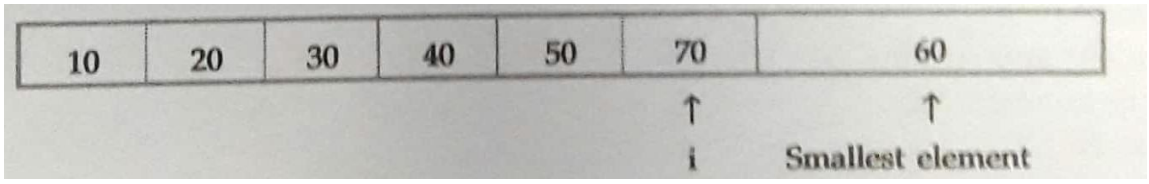




Swap **A[i]** with smallest element. The array then becomes,



6th Pass:



Swap **A[i]** with smallest element. The array then becomes,

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
10	20	30	40	50	60	70

This is a sorted array.

Analysis:

The above algorithm can be analysed mathematically. We will apply a general plan for non recursive mathematical analysis.

1. The input size is n i.e. total number of elements in the list.
2. In the algorithm the basic operation is key comparison. if **A[i] < A[min]**
3. This basic operation depends only on array size n. Hence we can find sum as

C (n) = Outer for loop with variable i x Inner for loop with variable j x Basic operation

Solving this equation we get

$$= [2 (n -1) (n - 1) - (n - 2) (n - 1)]/2$$
$$= [2 (n^2 - 2n + 1) - (n^2 - 3n + 2)]/2$$
$$= (n^2 - n)/2$$
$$= n (n - 1)/2$$
$$= 1/2 * (n^2)$$
$$\epsilon \Theta (n^2)$$

Thus time complexity of selection sort is $\Theta (n^2)$ for all input, but total number of key swaps is only $\Theta (n)$

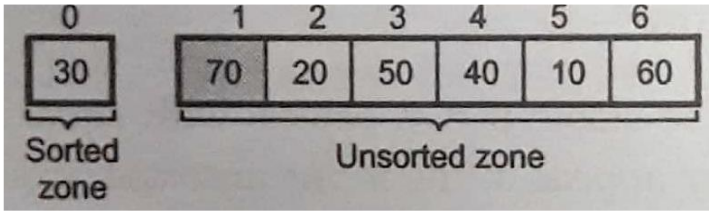
Insertion Sorting

In this method the elements are inserted at their appropriate place. Hence is the name **insertion sort**. Consider the following example to understand insertion sort.

For Example: Consider a list of elements as **30, 70, 20, 50, 40, 10, 60**

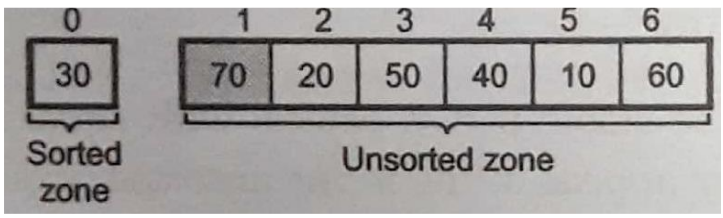
0	1	2	3	4	5	6
30	70	20	50	40	10	60

The process starts with first element



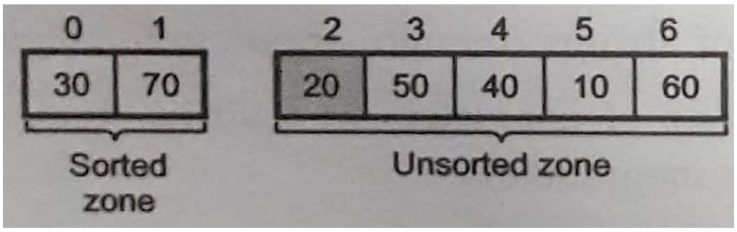
Step 1

Put 1st element i.e. 30 in sorting zone



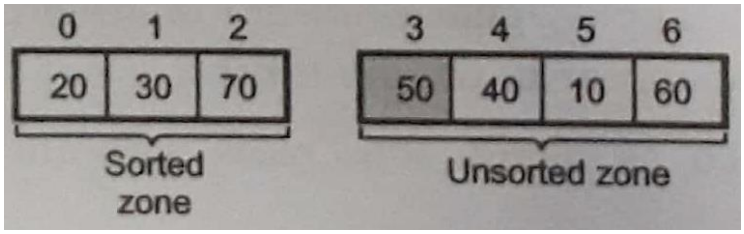
Step 2

Put 1st element i.e. 30 in sorting zone and compare 2nd element i.e. 70 with 1st element and insert it on desired position



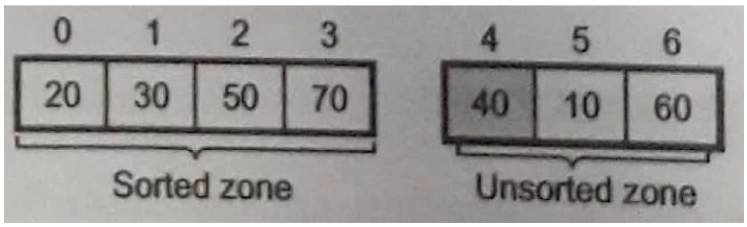
Step 3

Now compare 20 with the element of sorted zone i.e. 30, 70 and insert it on desired position with respect to it



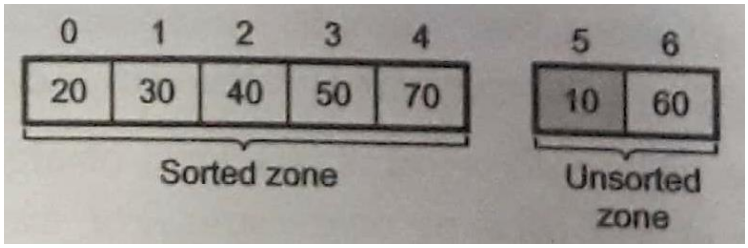
Step 4

Similarly now compare 50 with the element of sorted zone i.e. 20. 30, 70 and insert it on desired position with respect to it



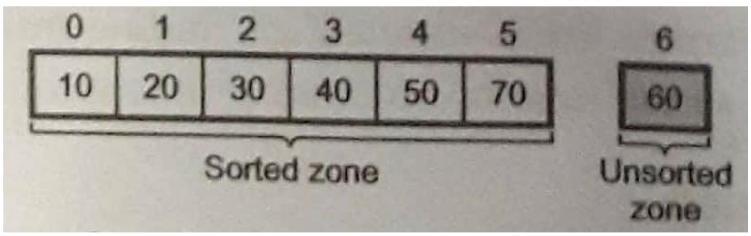
Step 5

Similarly now compare 40 with the element of sorted zone i.e. 20. 30, 50, 70 and insert it on desired position with respect to it



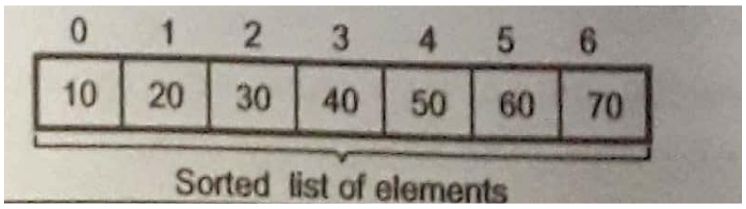
Step 6

Similarly now compare 10 with the element of sorted zone i.e. 20. 30, 40, 50, 70 and insert it on desired position with respect to it



Step 7

Similarly now compare 60 with the element of sorted zone i.e. 10, 20. 30, 40, 50, 70 and insert it on desired position with respect to it



Step 8

Finally we get all elements in sorted zone i.e. 10, 20, 30, 40, 50, 60, 70.

Algorithm - Although it is very natural to implement insertion using recursive (top -down) algorithm but it to very efficient to implement it using bottom up (iterative) approach.

Analysis - When an array of elements is almost sorted then it is best case complexity. The **best case time complexity** of insertion sort is **O(n)**. If an array is randomly distributed then it results in **average case time complexity** which is **$O(n^2)$**. If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it result in **worst case time complexity** which is **$O(n^2)$**.

The worst case occurs for insertion sort when the list of elements is in descending order and we have to sort it in ascending order. In such a case the total number of key comparisons will be

= (n-1) + (n-2) + ... + 2 + 1
= n (n-l)/2
= O(n^2)

Hence **worst case time complexity** is **O(n^2)**

Advantages of insertion sort

1. Simple to implement.
2. This method is efficient when we want to sort small number of elements and this method has excellent performance on almost sorted list of elements.
3. More efficient than most other simple 0(n2) algorithm: such as selection sort or bubble sort.
4. This is a stable (does not change the relative order of equal elements).
5. It is called in-place sorting algorithm. An algorithm in which the input is overwritten by output and to execute sorting it does not requires any more additional space is called in-place sorting algorithm.

Searching Brute force approach
Sequential/Linear Searching

Searching is the process of **finding a given value** position in a list of values or we can say that Searching is the process of **finding some particular** element in the list. It decides whether a **search key is present** in the data **or not.** It is the algorithmic process of **finding a particular item** in a collection of items. It can be done on internal data structure or on external data structure. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Searching Techniques - To search an element in a given array, it can be done in following ways:
1. Sequential Search
2. Binary Search

Linear Searching

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we **simply traverse the list completely** and **match each element** of the list with the item whose location is to be found. If the **match is found,** then the location of the **item is returned;** otherwise, the algorithm **returns NULL.** It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The **worst-case time complexity** of linear search is **O(n).**

In linear search, we access each element of an array **one by one sequentially** and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found. In the worst case, the number of average case we may have to scan half of the size of the array (n / 2). Therefore, linear search can be defined as the technique which traverses the array sequentially to locate the given item.

Efficiency of sequential searching - The **time taken** or the **number of comparisons made** in searching a record in a search table determines the efficiency of the technique. if the desired record is present in the first position of the search table, then only one comparison is made. If the desired record is the last one, then n comparisons have to be made. If the record is present somewhere in the search table, on an average, the number of comparisons will be (n+1)/2. The worst-case efficiency of this technique is 0(n) stands for the order of execution

Here **a** is a linear array with n elements. and item is a given item of information. This algorithm finds the location be of item in c. or sets loc = 0 if the search is unsuccessful.

Linear Search Algorithm

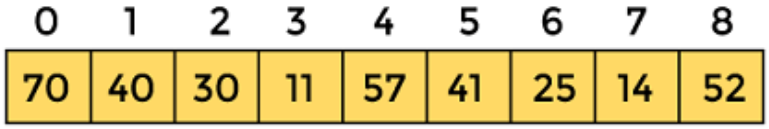
1. [Insert item at the end of data] set data [n+1] = item.
2. [Initialize counter] set loc = 0.
3. [search for item]
   Repeat while data[loc] =! item
   Set loc = loc+1.
4. [successfu1] if loc = n+1, then set loc = 0 **else** print element not found.
5. Exit.

Working of Linear search

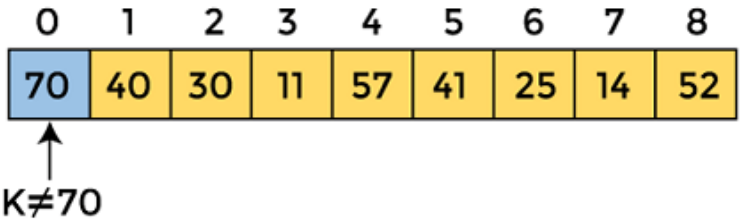
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

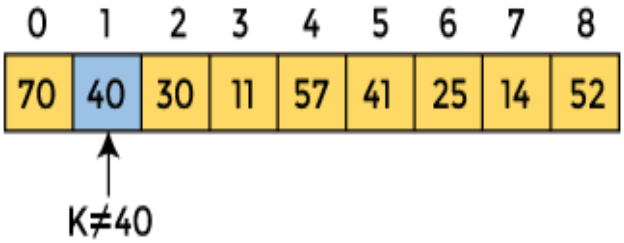
Let the elements of array are and the element to be searched is **K = 41**

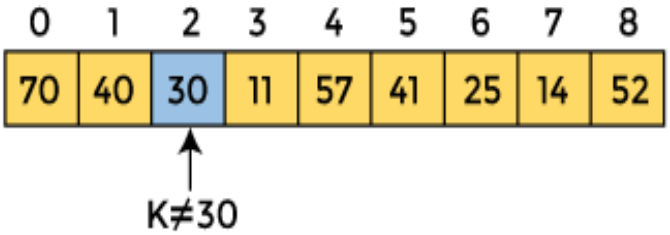
Now, start from the first element and compare **K** with each element of the array.

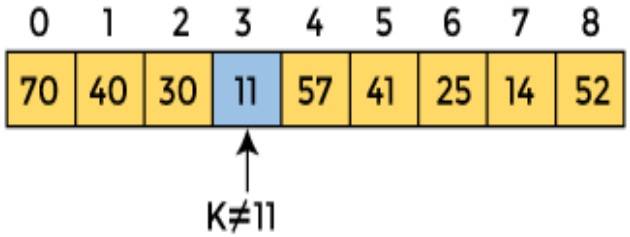


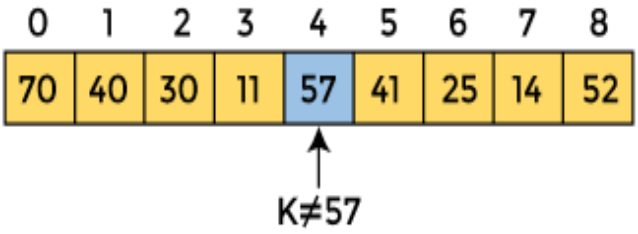
The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



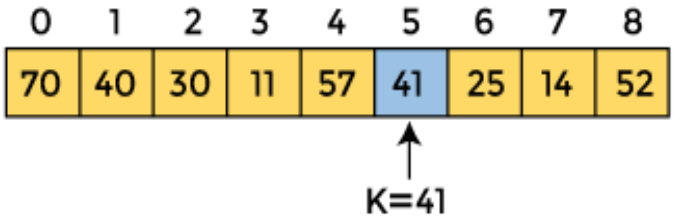








Now, the element to be searched is found. So algorithm will return the index of the element matched.



Binary Search

Binary search is an **extremely efficient** algorithm. This search technique searches the given **item in minimum possible comparisons.** To do the binary search, **first we had to sort the array** elements. The logic behind this technique is given below :

1. First **find the middle** element of the array.
2. **Compare** the **mid element** with an item.
3. There are **three cases :**
   (a) If it is a **desired element** then search is successful.
   (b) If it is **less than** desired item then search only the **first half** of the array.
   (c) If it is **greater than** the desired element search in the **second half** of the array.

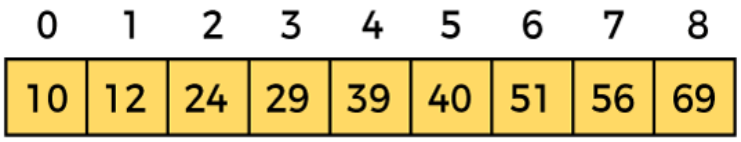
Repeat the same steps until an element is found or exhausts in the search area. In this algorithm every time we are reducing the search area. 50 number of comparisons keep on decreasing. In **worst case** the number of comparisons is atmost **log(N + 1)**. So it is an efficient algorithm when compared to linear search but the array has to be sorted before doing binary Search.

Here **a** is sorted array with lower bound LB and upper bound UB and Item is a given Item Information. The variables **beg, end** and **mid** denoted, respectively, the beginning and middle location of a segment of element of **a**. This algorithm finds the location loc of item in or sets loc = NULL.

Binary Search Algorithm

1. [Initialize segment variables]
   set beg = LB. end = UB and mid = int (beg + end)/2
2. If [mid] = item element found and exit, otherwise repeat steps 3 and 4 while beg <= end
3. If item < a [mid] then
   Set end = mid -1
   else
   Set beg = mid + 1 [End of if structure]
4. set mid = int (beg + end)/2 [End of step 2 loop]
5. If a[mid]= item then
   Set loc = mid
   else
   Set loc = NULL [End of if structure}
6. Exit

Working of Binary search



To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example. There are two methods to implement the binary search algorithm –

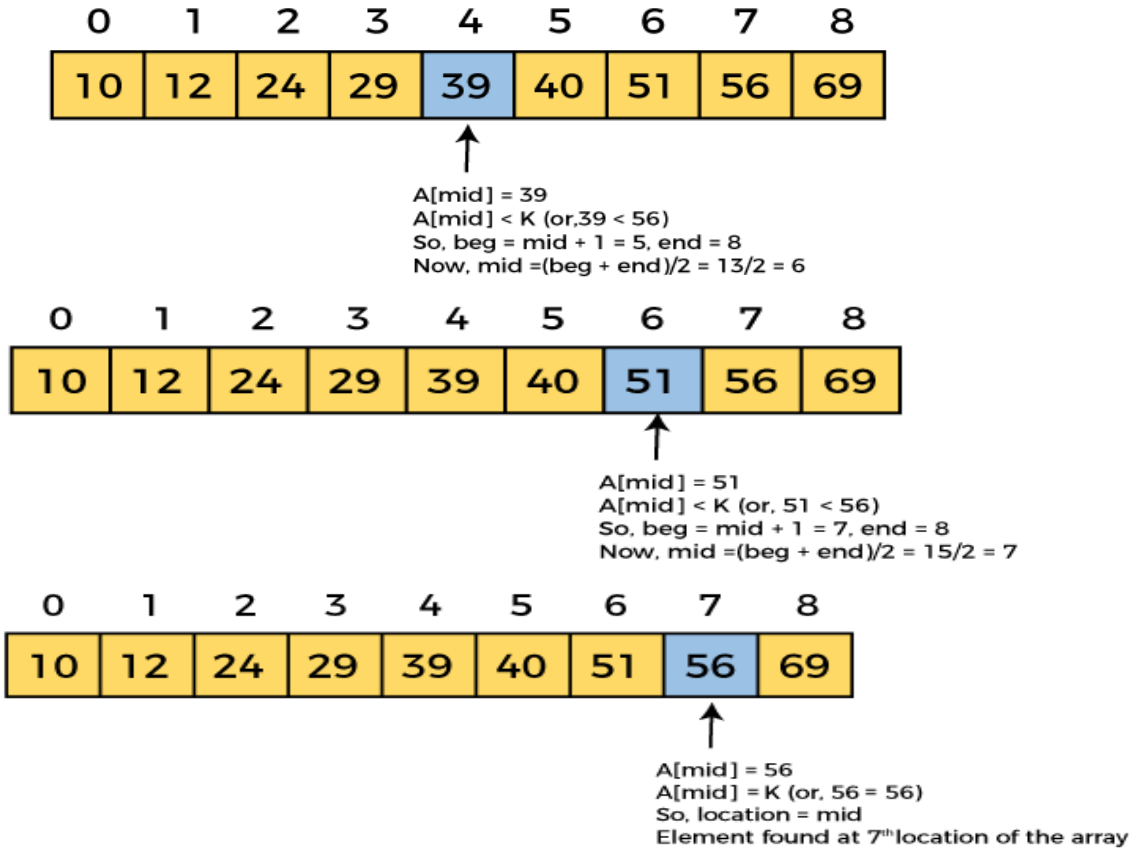
1. Iterative method
2. Recursive method

The **recursive method** of binary search follows the **divide and conquer approach.**

Let the elements of array are given and the element to search is, **K = 56**

We have to use the below formula to calculate the mid of the array - **mid = (beg + end)/2**

So, in the given array - beg = 0, end = 8, mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.



Now, the element to search is found. So algorithm will return the index of the element matched.

Divide and Conquer approach

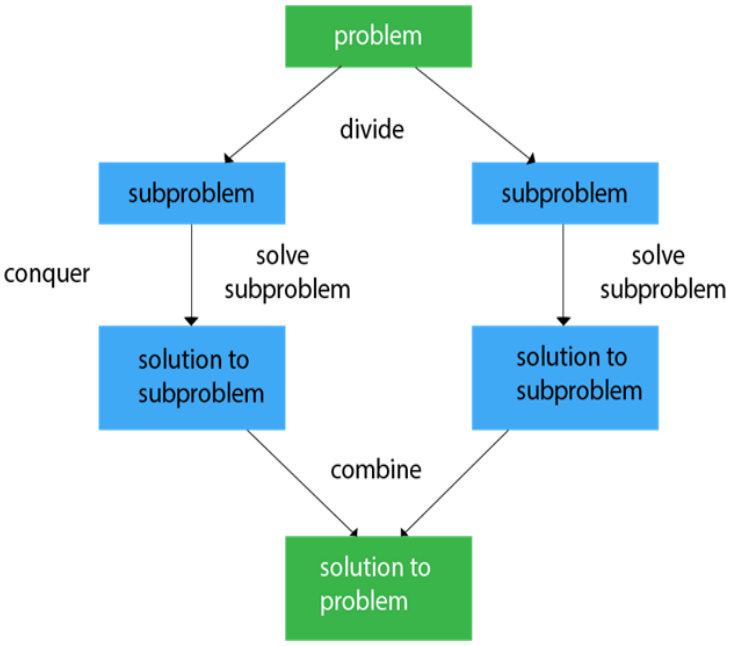
Divide and Conquer is a popular algorithmic technique in computer science that involves **breaking down a problem** into **smaller sub-problems**, **solving** each sub-problem **independently**, and then **combining the solutions** to the sub-problems to solve the original problem. The basic idea behind this technique is to divide a problem into **smaller**, more **manageable** sub-problems that can be solved more easily. This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into smaller sub-problems.
2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

Fig. 2



1. **Relational Formula:**
   It is the formula that we generate from the given technique. After generation of Formula, we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. **Stopping Condition:**
   When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

The following are some standard algorithms **application that follow Divide and Conquer** algorithm.

1. **Quicksort** is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
2. **Merge Sort** is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.
3. **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in the x-y plane.
4. **Strassen's Algorithm** is an efficient algorithm to multiply two matrices.
5. **Cooley–Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT.
6. **Karatsuba algorithm for fast multiplication** does the multiplication of two *n*-digit numbers in at most

Time Complexity: The time complexity of the divide and conquer algorithm to find the maximum and minimum element in an array is O(n). This is because each time we divide the array in half, so we will have a total of **log(n)** divisions. In each division, we compare two elements to find the maximum and minimum element, which takes constant time. Therefore, the total **time complexity is O(n*log(n)).**

Space Complexity: The space complexity of the divide and conquer algorithm to find the maximum and minimum element in an array is **O(log(n))**. This is because we are using recursion to divide the array into smaller parts, and each recursive call takes up space on the call stack. The maximum depth of the recursion tree is log(n), which is the number of times we can divide the array in half. Therefore, the **space complexity is O(log(n)).**

Divide and Conquer (D & C) vs Dynamic Programming (DP) Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. **How do choose one of them** for a given problem? Divide and Conquer should be used when the same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used. For example, Quicksort is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for calculating the nth Fibonacci number, Dynamic Programming should be preferred.

Advantages of Divide and Conquer Algorithm:

- The difficult problem can be solved easily.
- It divides the entire problem into subproblems thus it can be solved parallelly ensuring multiprocessing
- Efficiently uses cache memory without occupying much space
- Reduces time complexity of the problem
- Helps in the discovery of efficient algorithms.

Disadvantages of Divide and Conquer Algorithm:

- It involves recursion which is sometimes slow
- Efficiency depends on the implementation of logic
- It may crash the system if the recursion is performed rigorously.
- **Overhead:** The process of dividing the problem into subproblems and then combining the solutions can require additional time and resources.
- **Complexity:** Dividing a problem into smaller subproblems can increase the complexity of the overall solution.
- **Difficulty of implementation:** Some problems are difficult to divide into smaller subproblems or require a complex algorithm to do so.
- **Memory limitations:** When working with large data sets, the memory requirements for storing the intermediate results of the subproblems can become a limiting factor.
- **Suboptimal solutions:** Depending on how the subproblems are defined and how they are combined, a divide and conquer solution may not always produce the most optimal solution.
- **Difficulty in parallelization:** In some cases, dividing the problem into subproblems and solving them independently may not be easily parallelizable,

General method - Sorting (merge, quick)
External merge sort

In external sorting, the data stored on secondary memory is part by part loaded into main memory, sorting can be done over there. The sorted data can be then stored in the intermediate files. Finally these intermediate files can be merged repeatedly to get sorted data. Thus huge amount of data can be sorted using this technique.

The external merge sort is a technique in which the data is loaded in intermediate files. Each intermediate file is sorted independently and then combined or merged to get the sorted data. **For example:** Consider that there are 10,000 records that as to be sorted. Clearly we need to apply external sorting method. Suppose main memory has a capacity to store 500 records in blocks, with each block size of 100 records.

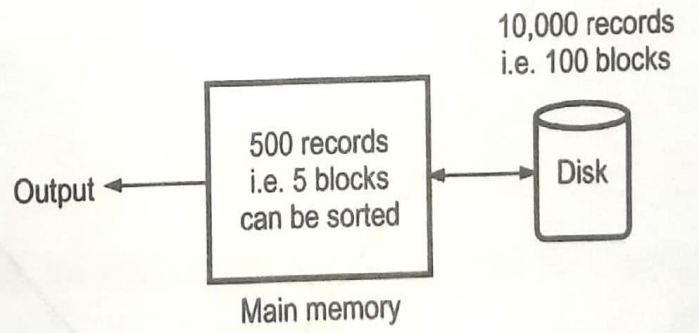


Fig. 3

The sorted 5 blocks (i.e. 500 records) are stored in intermediate file. This process will i be repeated 20 times to get all the records sorted in chunks. In the second step, we start merging a pair of intermediate files in the main memory to get output file.

Merge Sorting

The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. Merge sort on an input array with n elements consists of three steps:

1. **Divide:** Partition array into two sublists s1 and s2 with n/2 elements each
2. **Conquer:** Then sort sub list s1 and sublist s2.
3. **Combine:** Merge 51 and 52 into a unique sorted group.

Merge sort Algoritham

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 the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

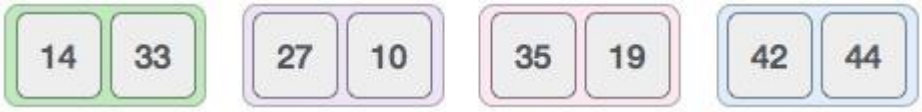
Step 3 − merge the smaller lists into new list in sorted order.

How Merge Sort Works?

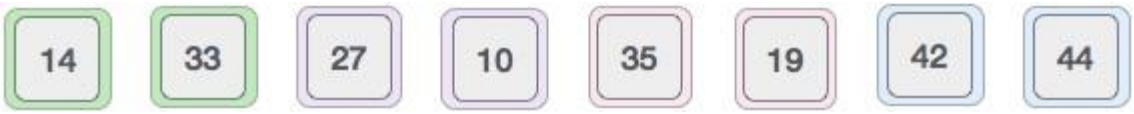
To understand merge sort, we take an unsorted array as the following –

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 the same manner as they were broken down. We first compare the element for each list and then combine them into another list in a 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 of 19 and 35 whereas 42 and 44 are placed sequentially.

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

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

Analysis: In merge sort algorithm two recursive calls are made. Each recursive call focuses on n/ 2 elements of the list. After two recursive calls one call is made to combine two sublists i.e. to merge all the elements. We can write it as –

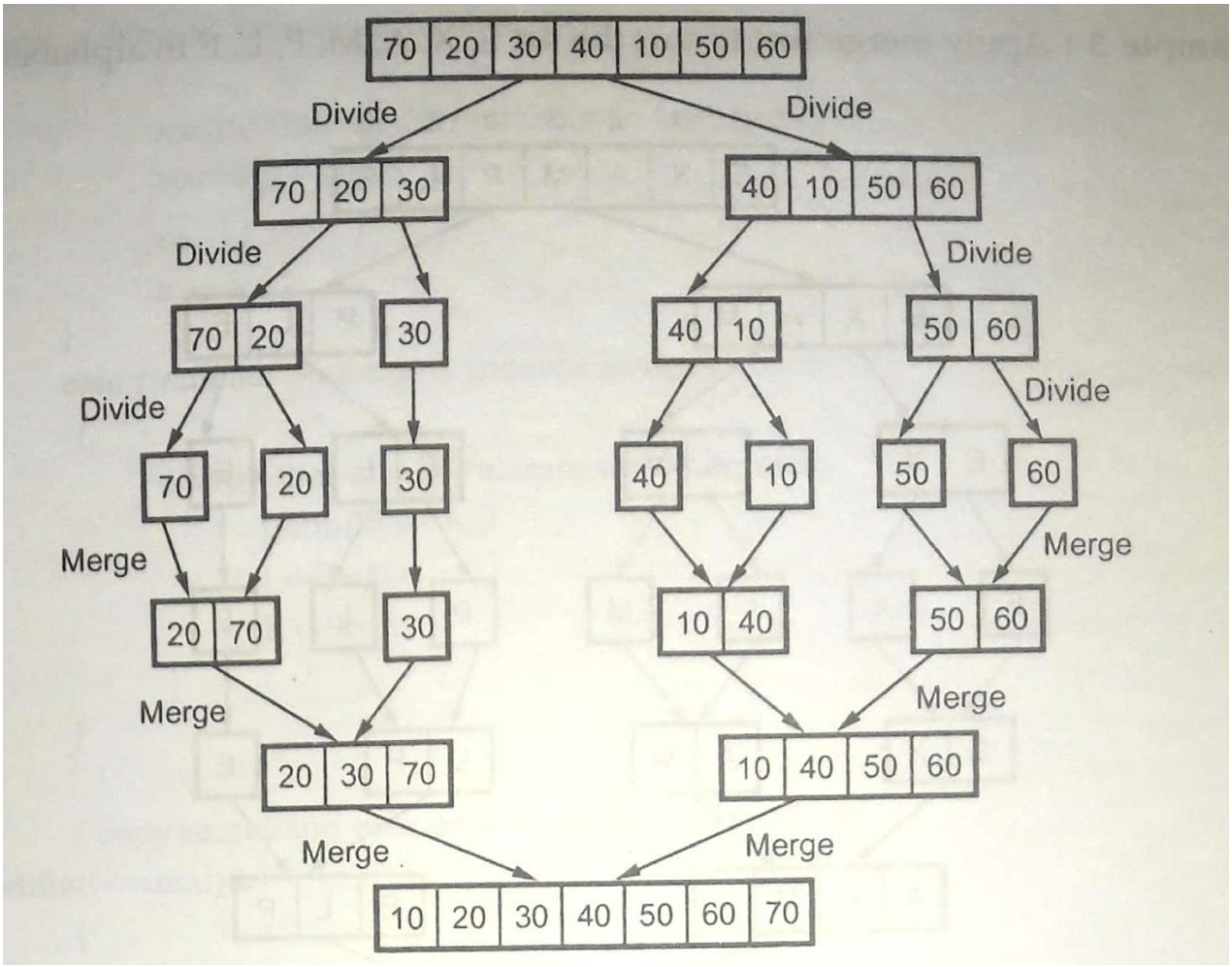
T(n) =	**T(n/2)**	**+**	**T(n/2)**	**+**	**Cn**
	Time taken by left sublist to get sorted		Time taken by right sublist to get sorted		Time for combining two soblists

$T(n) = O(n\log_2 n)$

The **average and worst case time complexity** of merge sort is **$O(n\log_2 n)$**

Example 2 : Consider the elements as **70, 20, 30, 40, 10, 50, 60**

Now we will split this list into two sublists.



Quick Sorting

Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows:

1. **Divide:** Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. The splitting of the array into two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.
2. **Conquer:** Recursively sort the two sub arrays. A
3. **Combine:** Combine all the sorted elements in a group to form a list of sorted elements.

In merge sort the division of array is based on the positions of array elements, but quick sort this division is based on actual value of the element. Consider an array a[i] where i is ranging from 0 to n - 1 then we can formulize the division of array elements as

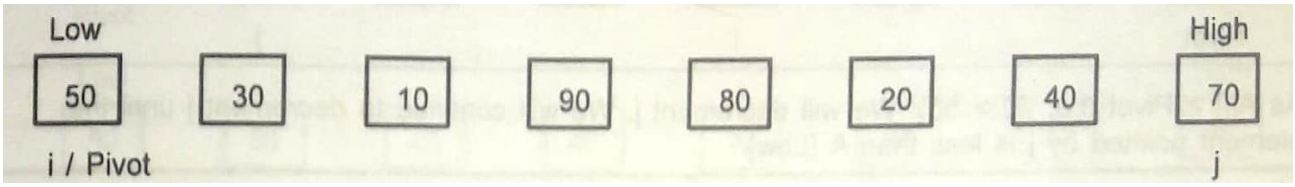
A[0],..... A[m—1] , A[m] , A[m+1],.....A[n—1]

These elements are smaller than A[m] — Mid — These elements are greater than A[m]

Example - let us understand this algorithm with the help of some example.

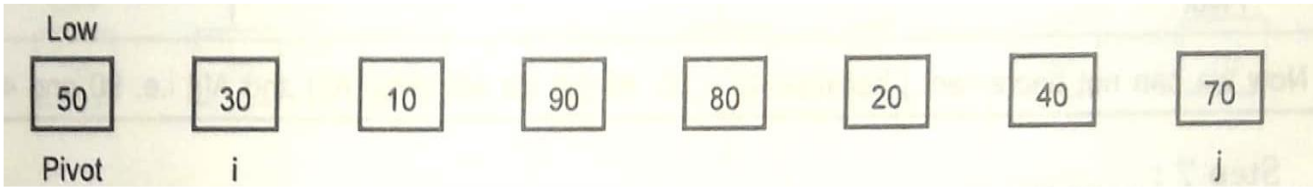
50 30 10 90 80 20 40 70

Step 1:



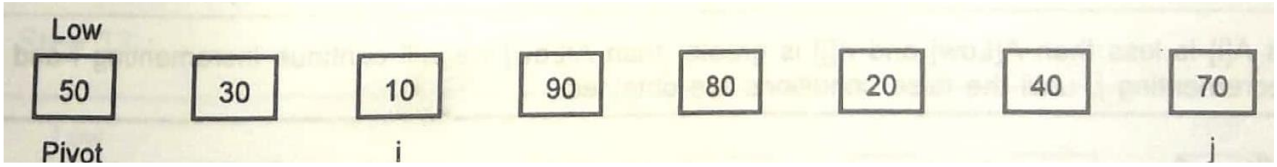
We will now split the array in two parts. The left sublist will contain the elements less than Pivot (i.e. 50) and right sublist contains elements greater than pivot.

Step 2:



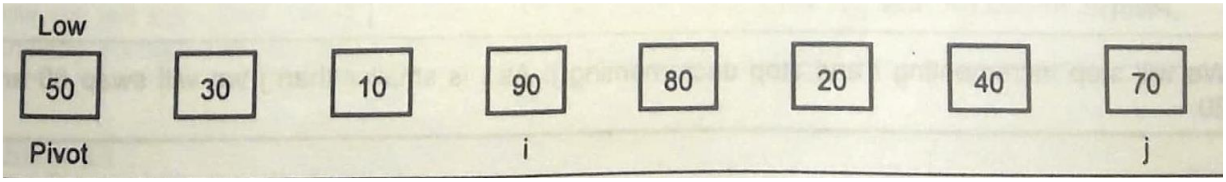
We will increment i. If A[i] <= Pivot, we will continue to increment it until the element pointed by i is greater than A[Low].

Step 3:



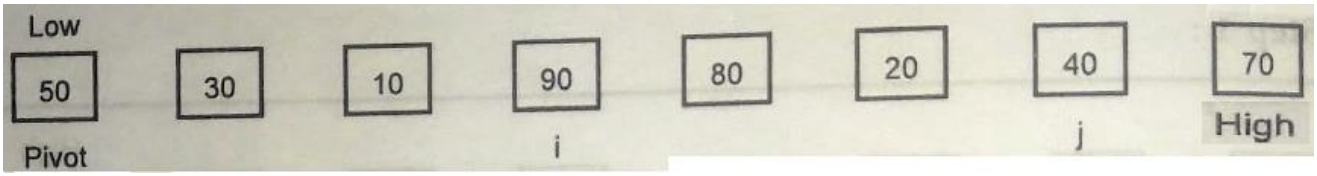
increment i as A[i] <= A[Low].

Step 4:



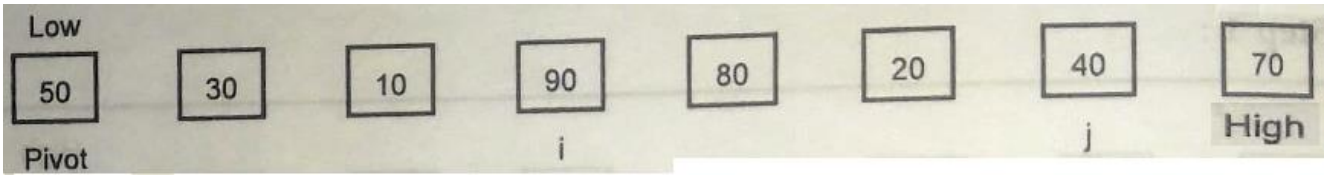
As A[i] > A [Low], we will stop incrementing i.

Step 5:



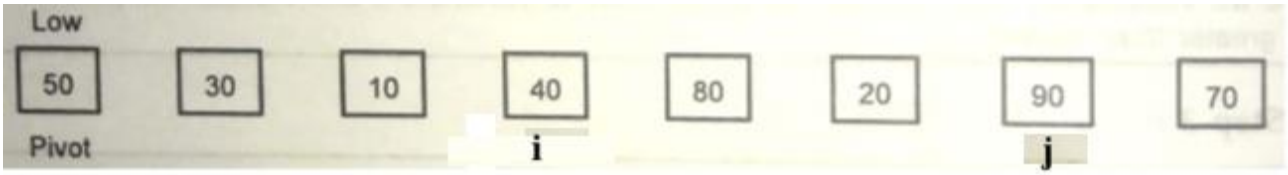
As A[j] > Pivot (i.e. 70 > 50). We will decrement j. We will continue to decrement j until the element pointed by j is less than A [Low].

Step 6:



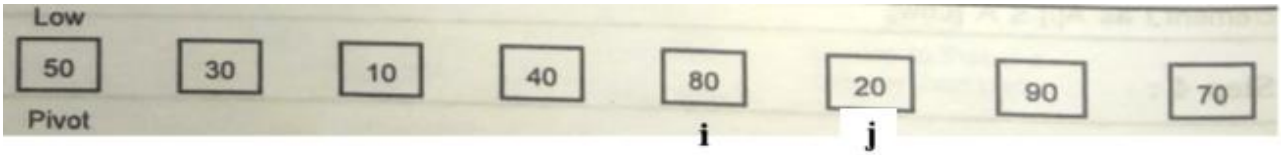
Now we cannot decrement j because 40 < 50. Hence we will swap A[i] and A[j] i.e. 90 and 40.

Step 7:



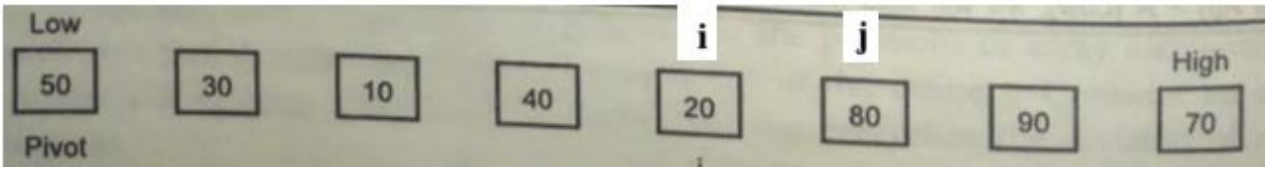
As A[i] is less than A[Low] and A[j] is greater than A[Low] we will continue incrementing i and decrementing j. until the false conditions are obtained.

Step 8:



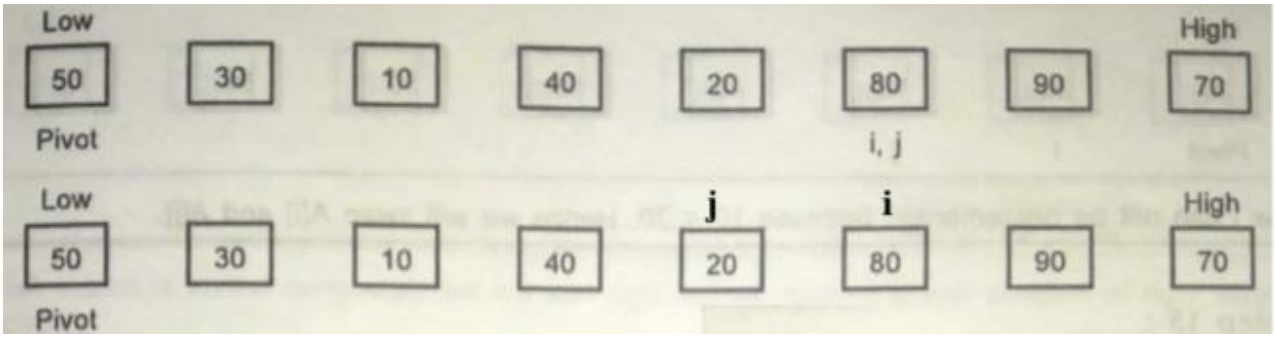
We will stop incrementing i and stop decrementing j. As i is smaller than j we will swap 80 and 20.

Step 9:



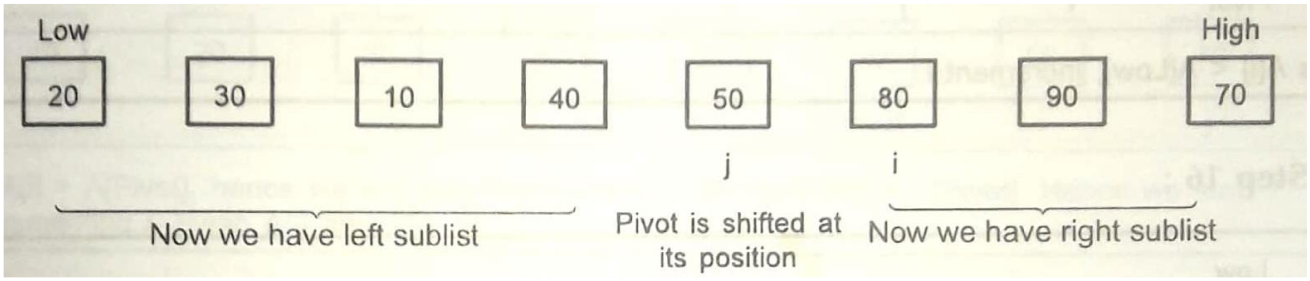
As A[i] < A[Low] and A[j] > A[Low], we will continue incrementing i and decrementing j.

Step 10:



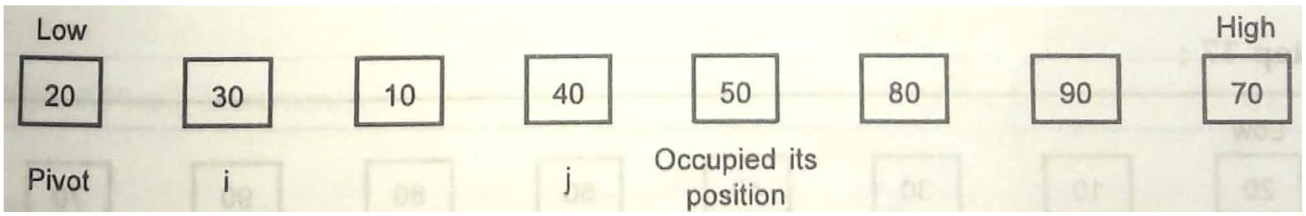
As A[j] < A[Low] and j has crossed i that is j < i, we will swap A[low] and A[j].

Step 11:



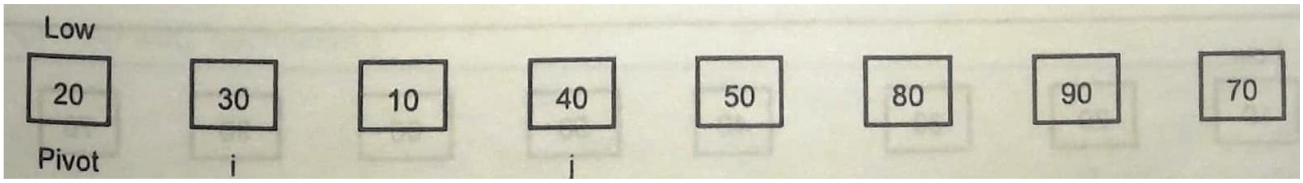
We will now start sorting left sublist, assuming the first element of left sublist as pivot element. Thus now new pivot = 20.

Step 12:



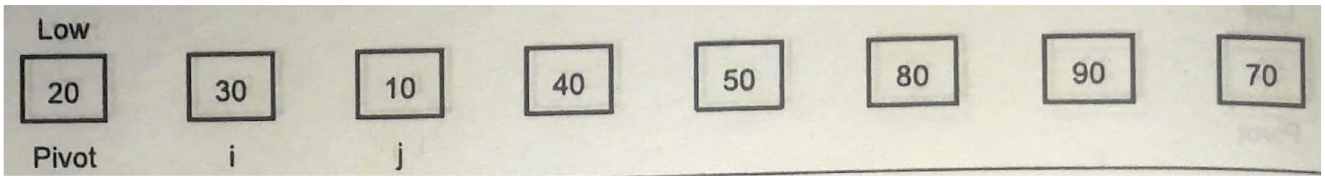
Now we will set i and j pointer and then we will start comparing A[i] with A[Low] or A[Pivot]. Similarly comparison with A[j] and A[Pivot].

Step 13:



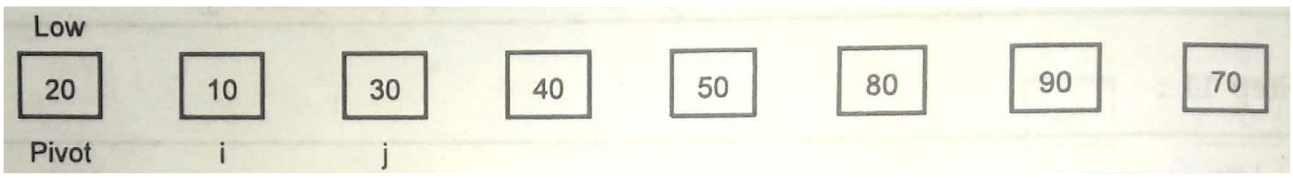
As A[i] > A[Pivot]. hence stop incrementing i. Now as A[i] > A[Pivot]. hence decrement j.

Step 14:



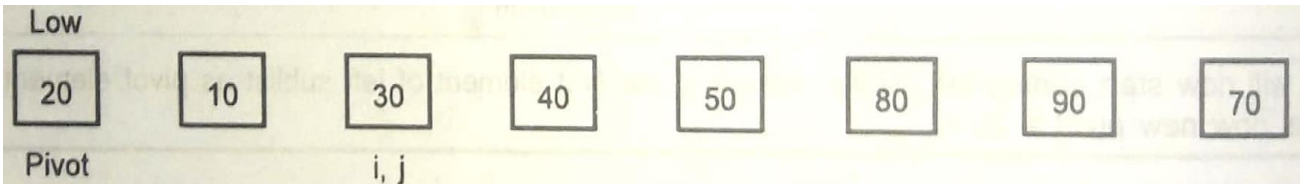
Now j can not be decremented because 10 < 20. Hence we will swap A[i] and A[j].

Step 15:



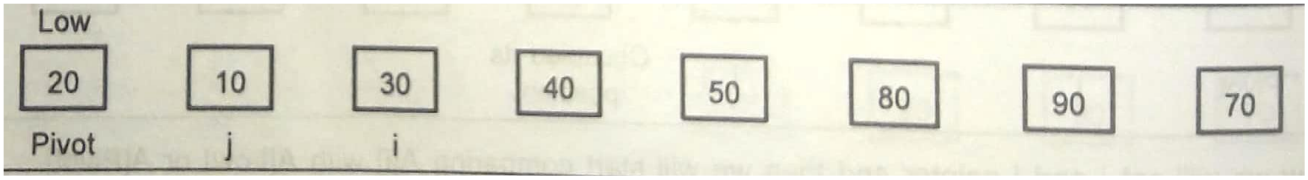
As A[i] < A[Low] increment i.

Step 16:



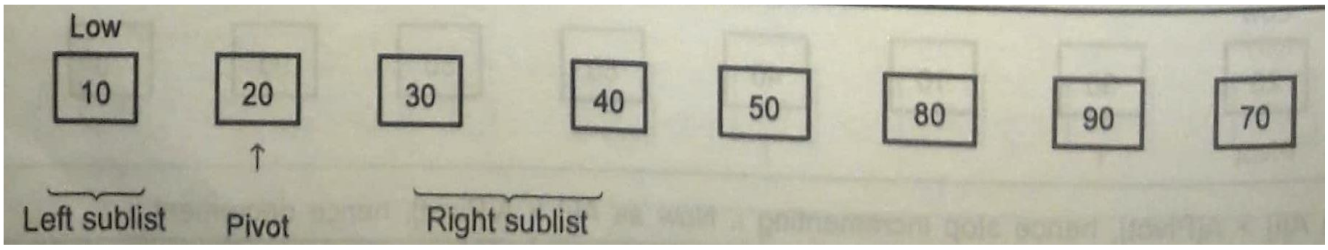
Now as A[i] > A[Low]. or A[i] > A[Pivot] decrement j.

Step 17:



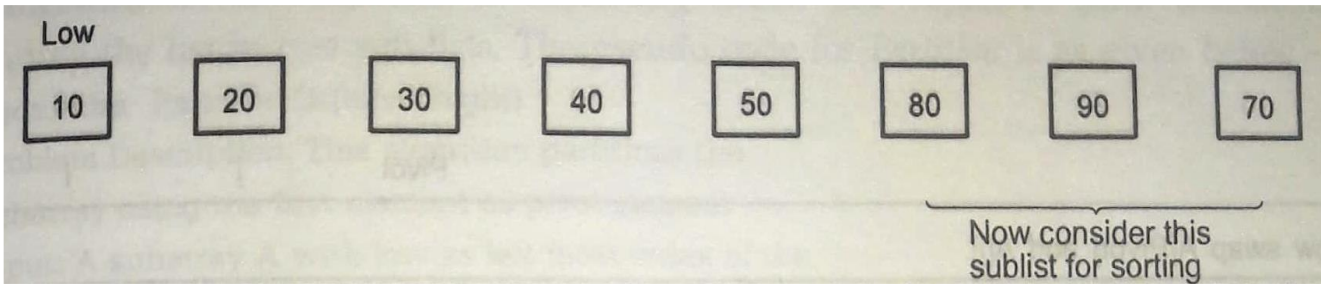
As A[j] < A[Low] we cannot decrement j now. We will now swap A[Low] and A[i] as 1 has crossed i and i > j

Step 18:



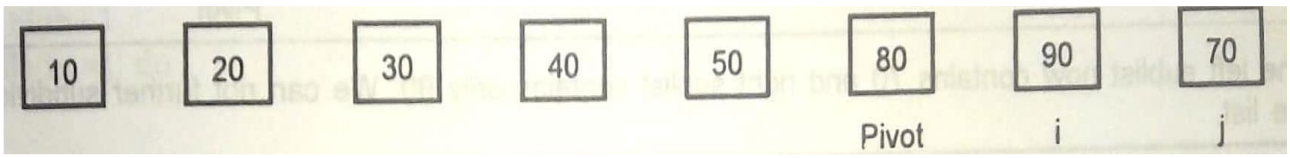
As there is only one element in left sublist hence we will sort right sublist.

Step 19:



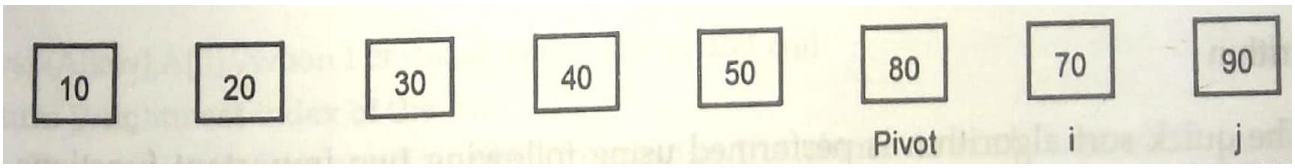
As left sublist is sorted completely we will sort right sublist, assuming first element of right sublist as pivot.

Step 20:



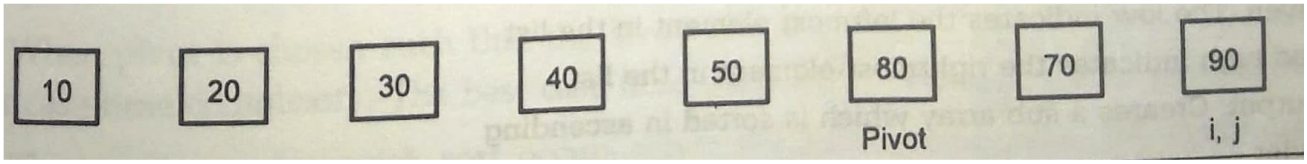
As A[i] > A[Pivot]. hence we will stop incrementing i. Similarly A[j] < A[Pivot]. Hence we stop decrementing j. Swap A[i] and A[j].

Step 21:



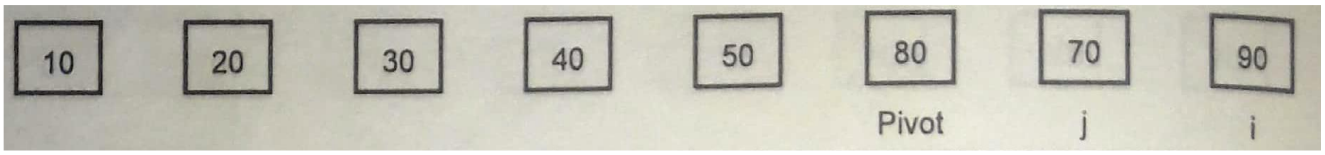
As A[i] < A[Pivot]. increment i.

Step 22:



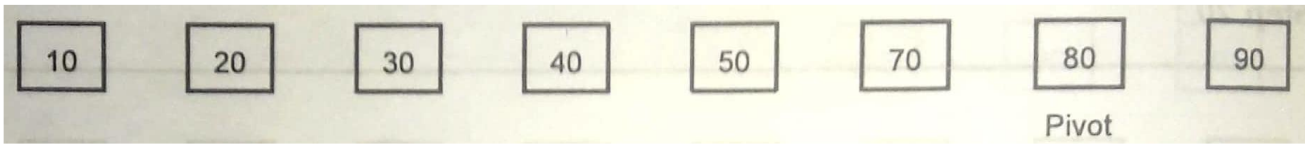
As A[i] > A[Pivot], decrement j.

Step 23:



Now swap A[pivot] and A[j]

Step 24:



The left sublist now contain 70 and right sublist contain only 90. We can not further subdivide the list

Hence list is

This is a sorted list.

Quick Sort Algorithm:

Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point **left i** and **right j** of the list excluding pivot
Step 3 – **left i** points to the low index and **right j** points to the high
Step 4 – while value at **left i** is less than pivot move toward right
Step 5 – while value at **right j** is greater than pivot move toward left
Step 6 – When both step 5 and step 6 matches swap the value of **left i** and **right j**.

6.1 Continue the process till **left i** and **right j** cross each other.

Step 7 – When **left i** and **right j** cross each other, **right j** is replaced with pivot point and **right j** become new Pivot.

7.1. Pivot point is position is fixed. Now array is divided in two parts.

Step 8 – Repeat step 1 to 7 till both array part is sorted and exit

The partition function is called to arrange the elements such that all the elements that are less than pivot are at the left side of pivot and all the elements that are greater than pivot are all at the right of pivot. In other words pivot is occupying its proper position and the partitioned list is obtained in an ordered manner.

Analysis: When pivot is chosen such that the array gets divided at the mid then it gives the best case time complexity. The **best case time complexity** of quick sort is **$O(n\log_2 n)$**. The **worst case** for quick sort occurs when the pivot is minimum or maximum of all the elements in the list. This can be graphically represented as - This ultimately results in 0(n2) time complexity. When array elements are randomly distributed then it results in average case time complexity, and it is **$O(n\log_2 n)$**.

Name	Average Case	Worst Case	Auxiliary Memory	Is Stable?	Other Notes
Bubble	$O(n^2)$	$O(n^2)$	1	yes	Small code
Selection	$O(n^2)$	$O(n^2)$	1	no	Stability depends on the implementation.
Insertion	$O(n^2)$	$O(n^2)$	1	yes	Average case is also $O(n + d)$, where d is the number of inversions
Shell	-	$O(n\log^2 n)$	1	no	
Merge	$O(n\log n)$	$O(n\log n)$	depends	yes	
Heap	$O(n\log n)$	$O(n\log n)$	1	no	
Quick Sort	$O(n\log n)$	$O(n^2)$	$O(\log n)$	depends	Can be implemented as a stable sort depending on how the pivot is handled.