Linear Search Algorithm

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.

### Algorithm

1. Linear\_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
2. Step 1: set pos = -1
3. Step 2: set i = 1
4. Step 3: repeat step 4 while i **<**= n
5. Step 4: if a[i] == val
6. set pos = i
7. print pos
8. go to step 6
9. [end of if]
10. set ii = i + 1
11. [end of loop]
12. Step 5: if pos = -1
13. print "value is not present in the array "
14. [end of if]
15. Step 6: exit

## Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(1) |
| **Average Case** | O(n) |
| **Worst Case** | O(n) |

* **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
* **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
* **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

## Implementation of Linear Search

Now, let's see the programs of linear search in different programming languages.

**Program:** Write a program to implement linear search in C language.

1. #include <stdio.h>
2. **int** linearSearch(**int** a[], **int** n, **int** val) {
3. // Going through array sequencially
4. **for** (**int** i = 0; i < n; i++)
5. {
6. **if** (a[i] == val)
7. **return** i+1;
8. }
9. **return** -1;
10. }
11. **int** main() {
12. **int** a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array
13. **int** val = 41; // value to be searched
14. **int** n = **sizeof**(a) / **sizeof**(a[0]); // size of array
15. **int** res = linearSearch(a, n, val); // Store result
16. printf("The elements of the array are - ");
17. **for** (**int** i = 0; i < n; i++)
18. printf("%d ", a[i]);
19. printf("\nElement to be searched is - %d", val);
20. **if** (res == -1)
21. printf("\nElement is not present in the array");
22. **else**
23. printf("\nElement is present at %d position of array", res);
24. **return** 0;
25. }

# Binary Search Algorithm

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

## Algorithm

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

## Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(1) |
| **Average Case** | O(logn) |
| **Worst Case** | O(logn) |

* **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
* **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
* **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

### 2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |

* The space complexity of binary search is O(1).

## Implementation of Binary Search

Now, let's see the programs of Binary search in different programming languages.

**Program:** Write a program to implement Binary search in C language.

1. #include <stdio.h>
2. **int** binarySearch(**int** a[], **int** beg, **int** end, **int** val)
3. {
4. **int** mid;
5. **if**(end >= beg)
6. {        mid = (beg + end)/2;
7. /\* if the item to be searched is present at middle \*/
8. **if**(a[mid] == val)
9. {
10. **return** mid+1;
11. }
12. /\* if the item to be searched is smaller than middle, then it can only be in left subarray \*/
13. **else** **if**(a[mid] < val)
14. {
15. **return** binarySearch(a, mid+1, end, val);
16. }
17. /\* if the item to be searched is greater than middle, then it can only be in right subarray \*/
18. **else**
19. {
20. **return** binarySearch(a, beg, mid-1, val);
21. }
22. }
23. **return** -1;
24. }
25. **int** main() {
26. **int** a[] = {11, 14, 25, 30, 40, 41, 52, 57, 70}; // given array
27. **int** val = 40; // value to be searched
28. **int** n = **sizeof**(a) / **sizeof**(a[0]); // size of array
29. **int** res = binarySearch(a, 0, n-1, val); // Store result
30. printf("The elements of the array are - ");
31. **for** (**int** i = 0; i < n; i++)
32. printf("%d ", a[i]);
33. printf("\nElement to be searched is - %d", val);
34. **if** (res == -1)
35. printf("\nElement is not present in the array");
36. **else**
37. printf("\nElement is present at %d position of array", res);
38. **return** 0;
39. }

# Bubble sort Algorithm

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Bubble short is majorly used where -

* complexity does not matter
* simple and shortcode is preferred

## Algorithm

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

1. begin BubbleSort(arr)
2. **for** all array elements
3. **if** arr[i] > arr[i+1]
4. swap(arr[i], arr[i+1])
5. end **if**
6. end **for**
7. **return** arr
8. end BubbleSort

## Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **O(n2).**
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **O(n2).**

AD

### 2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | YES |

* The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
* The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

## mplementation of Bubble sort

Now, let's see the programs of Bubble sort in different programming languages.

**Program:** Write a program to implement bubble sort in C language.

1. #include<stdio.h>
2. **void** print(**int** a[], **int** n) //function to print array elements
3. {
4. **int** i;
5. **for**(i = 0; i < n; i++)
6. {
7. printf("%d ",a[i]);
8. }
9. }
10. **void** bubble(**int** a[], **int** n) // function to implement bubble sort
11. {
12. **int** i, j, temp;
13. **for**(i = 0; i < n; i++)
14. {
15. **for**(j = i+1; j < n; j++)
16. {
17. **if**(a[j] < a[i])
18. {
19. temp = a[i];
20. a[i] = a[j];
21. a[j] = temp;
22. }
23. }
24. }
25. }
26. **void** main ()
27. {
28. **int** i, j,temp;
29. **int** a[5] = { 10, 35, 32, 13, 26};
30. **int** n = **sizeof**(a)/**sizeof**(a[0]);
31. printf("Before sorting array elements are - \n");
32. print(a, n);
33. bubble(a, n);
34. printf("\nAfter sorting array elements are - \n");
35. print(a, n);
36. }

# Selection Sort Algorithm

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

Selection sort is generally used when -

* A small array is to be sorted
* Swapping cost doesn't matter
* It is compulsory to check all elements

## Algorithm

1. SELECTION SORT(arr, n)
3. Step 1: Repeat Steps 2 **and** 3 **for** i = 0 to n-1
4. Step 2: CALL SMALLEST(arr, i, n, pos)
5. Step 3: SWAP arr[i] with arr[pos]
6. [END OF LOOP]
7. Step 4: EXIT
9. SMALLEST (arr, i, n, pos)
10. Step 1: [INITIALIZE] SET SMALL = arr[i]
11. Step 2: [INITIALIZE] SET pos = i
12. Step 3: Repeat **for** j = i+1 to n
13. **if** (SMALL > arr[j])
14. SET SMALL = arr[j]
15. SET pos = j
16. [END OF **if**]
17. [END OF LOOP]
18. Step 4: RETURN pos

## Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n2) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n2)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n2)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n2)**.

AD

### 2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | YES |

* The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

## Implementation of selection sort

Now, let's see the programs of selection sort in different programming languages.

**Program:** Write a program to implement selection sort in C language.

1. #include <stdio.h>
3. **void** selection(**int** arr[], **int** n)
4. {
5. **int** i, j, small;
7. **for** (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
8. {
9. small = i; //minimum element in unsorted array
11. **for** (j = i+1; j < n; j++)
12. **if** (arr[j] < arr[small])
13. small = j;
14. // Swap the minimum element with the first element
15. **int** temp = arr[small];
16. arr[small] = arr[i];
17. arr[i] = temp;
18. }
19. }
21. **void** printArr(**int** a[], **int** n) /\* function to print the array \*/
22. {
23. **int** i;
24. **for** (i = 0; i < n; i++)
25. printf("%d ", a[i]);
26. }
28. **int** main()
29. {
30. **int** a[] = { 12, 31, 25, 8, 32, 17 };
31. **int** n = **sizeof**(a) / **sizeof**(a[0]);
32. printf("Before sorting array elements are - \n");
33. printArr(a, n);
34. selection(a, n);
35. printf("\nAfter sorting array elements are - \n");
36. printArr(a, n);
37. **return** 0;
38. }

# Insertion Sort Algorithm

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

Insertion sort has various advantages such as -

* Simple implementation
* Efficient for small data sets
* Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

## Algorithm

The simple steps of achieving the insertion sort are listed as follows -

**Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.

**Step2 -** Pick the next element, and store it separately in a **key.**

**Step3 -** Now, compare the **key** with all elements in the sorted array.

**Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

**Step 5 -** Insert the value.

**Step 6 -** Repeat until the array is sorted.

## Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n2)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n2)**.

AD

### 2. Space Complexity

|  |  |
| --- | --- |
| **Space Complexity** | O(1) |
| **Stable** | YES |

* The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

## Implementation of insertion sort

Now, let's see the programs of insertion sort in different programming languages.

**Program:** Write a program to implement insertion sort in C language.

1. #include <stdio.h>
3. **void** insert(**int** a[], **int** n) /\* function to sort an aay with insertion sort \*/
4. {
5. **int** i, j, temp;
6. **for** (i = 1; i < n; i++) {
7. temp = a[i];
8. j = i - 1;
10. **while**(j>=0 && temp <= a[j])  /\* Move the elements greater than temp to one position ahead from their current position\*/
11. {
12. a[j+1] = a[j];
13. j = j-1;
14. }
15. a[j+1] = temp;
16. }
17. }
19. **void** printArr(**int** a[], **int** n) /\* function to print the array \*/
20. {
21. **int** i;
22. **for** (i = 0; i < n; i++)
23. printf("%d ", a[i]);
24. }
26. **int** main()
27. {
28. **int** a[] = { 12, 31, 25, 8, 32, 17 };
29. **int** n = **sizeof**(a) / **sizeof**(a[0]);
30. printf("Before sorting array elements are - \n");
31. printArr(a, n);
32. insert(a, n);
33. printf("\nAfter sorting array elements are - \n");
34. printArr(a, n);
36. **return** 0;
37. }

# Quick Sort Algorithm

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

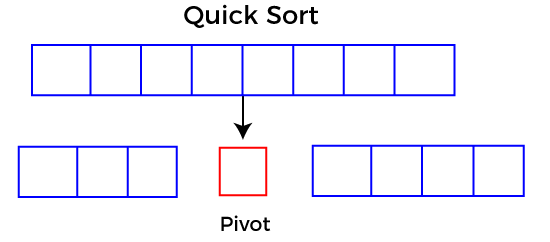
**Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively, sort two subarrays with Quicksort.

**Combine:** Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



## Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

* Pivot can be random, i.e. select the random pivot from the given array.
* Pivot can either be the rightmost element of the leftmost element of the given array.
* Select median as the pivot element.

## Algorithm

**Algorithm:**

1. QUICKSORT (array A, start, end)
2. {
3. 1 **if** (start < end)
4. 2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }

**Partition Algorithm:**

The partition algorithm rearranges the sub-arrays in a place.

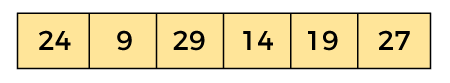
1. PARTITION (array A, start, end)
2. {
3. 1 pivot ? A[end]
4. 2 i ? start-1
5. 3 **for** j ? start to end -1 {
6. 4 **do** **if** (A[j] < pivot) {
7. 5 then i ? i + 1
8. 6 swap A[i] with A[j]
9. 7  }}
10. 8 swap A[i+1] with A[end]
11. 9 **return** i+1
12. }

## Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

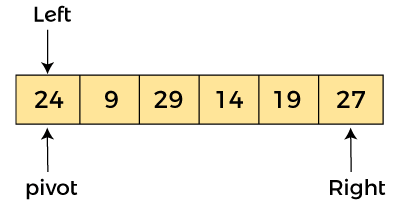
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

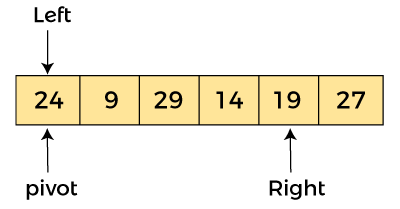


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

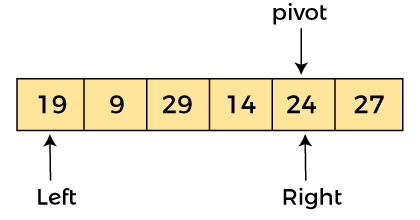


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



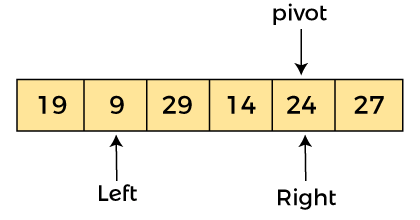
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -



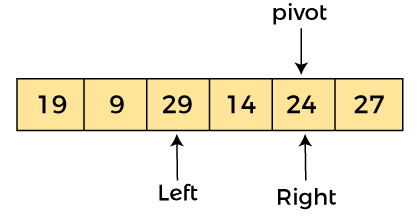
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

As a[pivot] > a[left], so algorithm moves one position to right as -

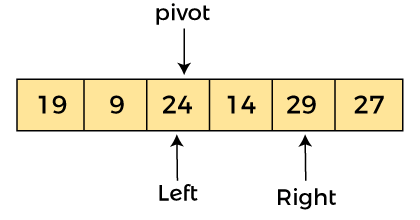


Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -

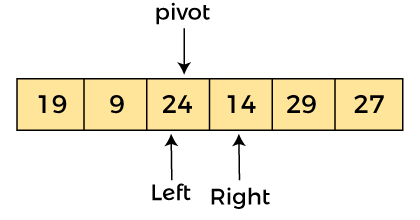
AD



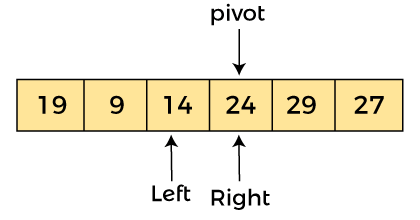
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



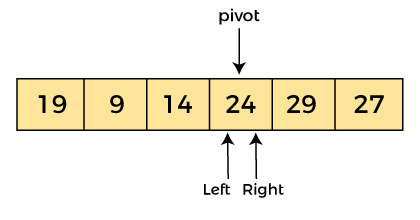
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



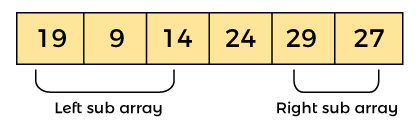
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



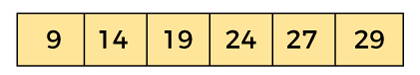
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



## Implementation of quicksort

Now, let's see the programs of quicksort in different programming languages.

**Program:** Write a program to implement quicksort in C language.

1. #include <stdio.h>
2. /\* function that consider last element as pivot,
3. place the pivot at its exact position, and place
4. smaller elements to left of pivot and greater
5. elements to right of pivot.  \*/
6. **int** partition (**int** a[], **int** start, **int** end)
7. {
8. **int** pivot = a[end]; // pivot element
9. **int** i = (start - 1);
11. **for** (**int** j = start; j <= end - 1; j++)
12. {
13. // If current element is smaller than the pivot
14. **if** (a[j] < pivot)
15. {
16. i++; // increment index of smaller element
17. **int** t = a[i];
18. a[i] = a[j];
19. a[j] = t;
20. }
21. }
22. **int** t = a[i+1];
23. a[i+1] = a[end];
24. a[end] = t;
25. **return** (i + 1);
26. }
28. /\* function to implement quick sort \*/
29. **void** quick(**int** a[], **int** start, **int** end) /\* a[] = array to be sorted, start = Starting index, end = Ending index \*/
30. {
31. **if** (start < end)
32. {
33. **int** p = partition(a, start, end); //p is the partitioning index
34. quick(a, start, p - 1);
35. quick(a, p + 1, end);
36. }
37. }
39. /\* function to print an array \*/
40. **void** printArr(**int** a[], **int** n)
41. {
42. **int** i;
43. **for** (i = 0; i < n; i++)
44. printf("%d ", a[i]);
45. }
46. **int** main()
47. {
48. **int** a[] = { 24, 9, 29, 14, 19, 27 };
49. **int** n = **sizeof**(a) / **sizeof**(a[0]);
50. printf("Before sorting array elements are - \n");
51. printArr(a, n);
52. quick(a, 0, n - 1);
53. printf("\nAfter sorting array elements are - \n");
54. printArr(a, n);
56. **return** 0;
57. }

# Merge Sort Algorithm

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

## Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

1. MERGE\_SORT(arr, beg, end)
3. **if** beg < end
4. set mid = (beg + end)/2
5. MERGE\_SORT(arr, beg, mid)
6. MERGE\_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
10. END MERGE\_SORT

## Implementation of merge sort

Now, let's see the programs of merge sort in different programming languages.

**Program:** Write a program to implement merge sort in C language.

1. #include <stdio.h>
3. /\* Function to merge the subarrays of a[] \*/
4. **void** merge(**int** a[], **int** beg, **int** mid, **int** end)
5. {
6. **int** i, j, k;
7. **int** n1 = mid - beg + 1;
8. **int** n2 = end - mid;
10. **int** LeftArray[n1], RightArray[n2]; //temporary arrays
12. /\* copy data to temp arrays \*/
13. **for** (**int** i = 0; i < n1; i++)
14. LeftArray[i] = a[beg + i];
15. **for** (**int** j = 0; j < n2; j++)
16. RightArray[j] = a[mid + 1 + j];
18. i = 0; /\* initial index of first sub-array \*/
19. j = 0; /\* initial index of second sub-array \*/
20. k = beg;  /\* initial index of merged sub-array \*/
22. **while** (i < n1 && j < n2)
23. {
24. **if**(LeftArray[i] <= RightArray[j])
25. {
26. a[k] = LeftArray[i];
27. i++;
28. }
29. **else**
30. {
31. a[k] = RightArray[j];
32. j++;
33. }
34. k++;
35. }
36. **while** (i<n1)
37. {
38. a[k] = LeftArray[i];
39. i++;
40. k++;
41. }
43. **while** (j<n2)
44. {
45. a[k] = RightArray[j];
46. j++;
47. k++;
48. }
49. }
51. **void** mergeSort(**int** a[], **int** beg, **int** end)
52. {
53. **if** (beg < end)
54. {
55. **int** mid = (beg + end) / 2;
56. mergeSort(a, beg, mid);
57. mergeSort(a, mid + 1, end);
58. merge(a, beg, mid, end);
59. }
60. }
62. /\* Function to print the array \*/
63. **void** printArray(**int** a[], **int** n)
64. {
65. **int** i;
66. **for** (i = 0; i < n; i++)
67. printf("%d ", a[i]);
68. printf("\n");
69. }
71. **int** main()
72. {
73. **int** a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
74. **int** n = **sizeof**(a) / **sizeof**(a[0]);
75. printf("Before sorting array elements are - \n");
76. printArray(a, n);
77. mergeSort(a, 0, n - 1);
78. printf("After sorting array elements are - \n");
79. printArray(a, n);
80. **return** 0;
81. }