



Tutorial Link <https://course.testpad.chitkara.edu.in/tutorials/Divide & Conquer - Quick Sort/5a12ec9446765b2b63e34761>

#### TUTORIAL

## Divide & Conquer - Quick Sort

### Topics

1.2 Implementation of quicksort

1.8 Properties of Quick sort

It is another divide and conquer approach. It is slightly different from merge sort as it will not pick the middle element always. Instead it picks a random element called as pivot element. And then it rearrange the array so that pivot element divide the array in two halves such that all elements lesser than pivot element are on left side of pivot element and all elements greater than the pivot element are on right side of pivot element. It then do the same thing recursively on both these halves. This way it can do the sorting as an in-place sort. Obviously if the division is not proper then complexity may increase. So for quicksort the picking of pivot element is very important and tricky. The best and average case complexity of it is  $O(n \log n)$ , whereas in worst case it go to  $O(n^2)$ . But the space complexity is  $O(\log n)$  which is better in comparison to merge sort.

Let's see an example on quick sort.

Let's **start with** array holding [17, 13, 20, 16, 19, 24, 22, 21] **8 values, for** the sake of simplicity, the **pivot is** picked as the rightmost **element** usually (In implementations we can chose **any element** randomly but deterministically).

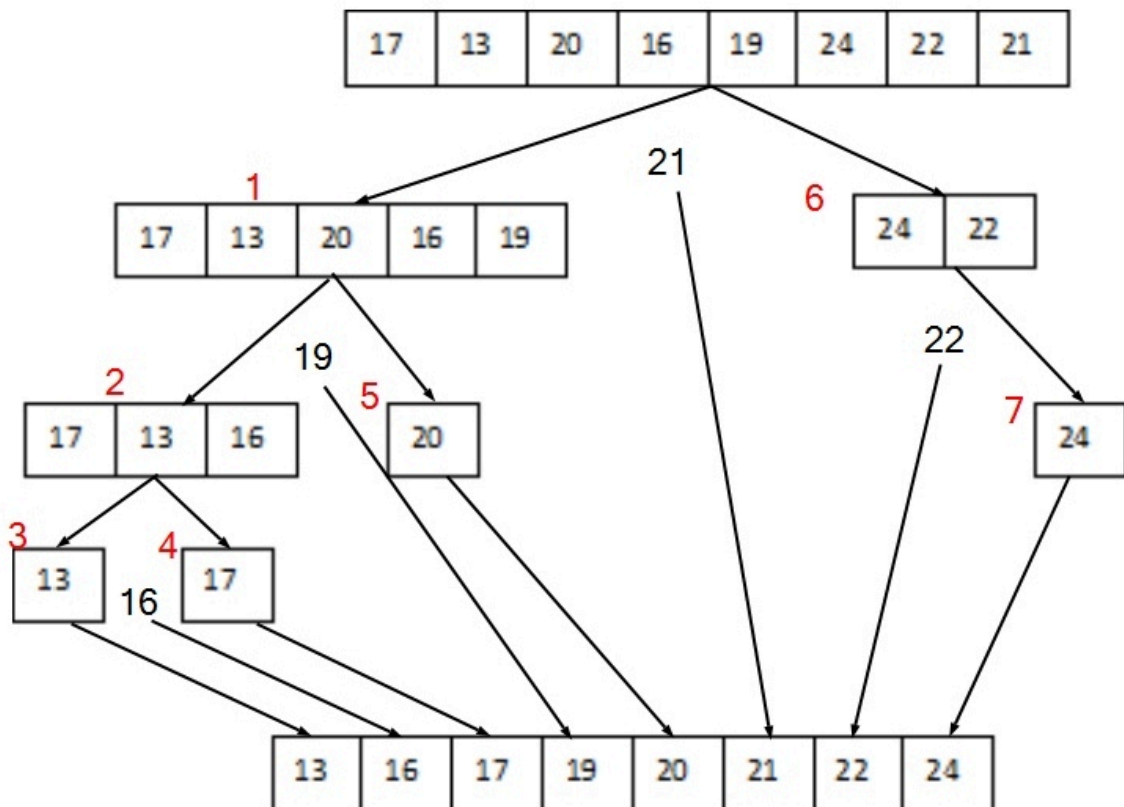
So **in** 1st iteration the **pivot element is** 21, **Now** the array will be rearranged such that 21 will be **at proper position and** all elements lesser will be **on left of 21 and** all elements greater will be **on right of 21**.

There **are** many ways **to do it**, so we can **choose any method and get** it done as below: -

[17, 13, 20, 16, 19, 21, 24, 22]

**Now** array **is** splitted **into** two halves, **and** quicksort will **repeat** the same thing **on** these two halves recursively, till the whole array **is** sorted.

Following figure shows the step by step procedure for quick sort on this array: -



The array itself is rearranged to shift all the elements at proper positions

The numbers in red color are showing the sequence of steps the quick sort procedure will take to sort this array. The algorithm for quick-sort is as below: -

```

If left < right
    Pick the pivot element from the list
    Split the list into two halves according to the pivot element.
    Sort the left half repeatedly using this approach.
    Sort the right half repeatedly using this approach.
End if

```

Following is the implementation of quicksort in languages: -

## Implementation of quicksort

```

1  #include<stdio.h>
2
3  // To print array of length=size
4  void printArray(int array[], int size)
5  {
6      int i;
7      for (i=0; i < size; i++)
8          printf("%d\t", array[i]);
9      printf("\n");
10 }
11

```

C

```
12 // To swap two elements
13 void swap(int* a, int* b)
14 {
15     int t = *a;
16     *a = *b;
17     *b = t;
18 }
19
20 /* This function takes last element as pivot, and places all
21    smaller
22    elements to left of pivot and all greater elements to right of
23    pivot */
24 int partition (int array[], int low, int high)
25 {
26     int pivot = array[high];    // pivot
27     int i = (low - 1);    // Index of smaller element
28     int j;
29
30     for (j = low; j <= high- 1; j++)
31     {
32         if (array[j] <= pivot)
33         {
34             i++;    // increment index of smaller element
35             swap(&array[i], &array[j]);    // shift all lesser elements
36             // in left half
37         }
38     }
39     swap(&array[i + 1], &array[high]);    // place pivot element at
40     // end of smaller elements
41     return (i + 1);    // the index of pivot element
42 }
43
44 // Recursive function to sort array with quicksort
45 void quickSort(int array[], int low, int high)
46 {
47     if (low < high)
48     {
49         int pivot_index;
50         pivot_index = partition(array, low, high);
51         printf("Pivot element is %d \n",array[pivot_index]);
52         printf("Array after pivot partitioning : \n");
53         printArray(array,8);
54
55         // Call quicksort() on left half and right half excluding
56         // pivot element as it is already at proper position i.e.
57         // between lesser and greater elements.
58         quickSort(array, low, pivot_index - 1);
59         quickSort(array, pivot_index + 1, high);
60     }
61 }
62
63 int main()
64 {
65     int array[] = {17, 13, 20, 16, 19, 24, 22, 21};
```

```

61     int n = sizeof(array)/sizeof(array[0]);
62     printf("Given array: \n");
63     printArray(array, n);
64     quickSort(array, 0, n-1);
65     printf("Sorted array: \n");
66     printArray(array, n);
67     return 0;
68 }
69

```

```

1  import java.util.Scanner;
2  // Other imports go here
3  // Do NOT change the class name
4  class Main{
5      static void printArray(int array[], int size){
6          int i;
7          for (i=0; i < size; i++)
8              System.out.printf("%d\t", array[i]);
9              System.out.printf("\n");
10     }
11
12     /* This function takes last element as pivot, and places all
13     smaller
14     elements to left of pivot and all greater elements to right
15     of pivot */
16     static int partition (int array[], int low, int high){
17         int pivot = array[high];    // pivot
18         int i = (low - 1); // Index of smaller element
19         int j;
20
21         for (j = low; j <= high- 1; j++)
22         {
23             if (array[j] <= pivot)
24             {
25                 i++;           // increment index of smaller
26                 element
27                 // shift all lesser elements in left half
28                 int temp = array[i];
29                 array[i] = array[j];
30                 array[j] = temp;
31             }
32         }
33         // place pivot element at end of smaller elements
34         int temp = array[i+1];
35         array[i+1] = array[high];
36         array[high] = temp;
37         return (i + 1);    // the index of pivot element
38     }
39
40     // Recursive function to sort array with quicksort
41     static void quickSort(int array[], int low, int high){
42         if (low < high){
43             int pivot_index;

```

Java

```

41         pivot_index = partition(array, low, high);
42         System.out.printf("Pivot element is %d\n", array[pivot_index]);
43         System.out.printf("Array after pivot partitioning : \n");
44         printArray(array, 8);
45
46         // Call quicksort() on left half and right half
47         // pivot element as it is already at proper position
48         // between lesser and greater elements.
49         quickSort(array, low, pivot_index - 1);
50         quickSort(array, pivot_index + 1, high);
51     }
52 }
53
54 public static void main(String[] args){
55     int array[] = {17, 13, 20, 16, 19, 24, 22, 21};
56     int n = array.length;
57     System.out.printf("Given array: \n");
58     printArray(array, n);
59     quickSort(array, 0, n-1);
60     System.out.printf("Sorted array: \n");
61     printArray(array, n);
62 }
63 }

```

```

1  #include<iostream>
2  #include<cstdio>
3  #include<cmath>
4  using namespace std;
5
6  // To print array of length=size
7  void printArray(int array[], int size)
8  {
9      int i;
10     for (i=0; i < size; i++)
11         cout<<array[i]<<"\t";
12     cout<<endl;
13 }
14
15 // To swap two elements
16 void swap(int* a, int* b)
17 {
18     int t = *a;
19     *a = *b;
20     *b = t;
21 }
22
23 /* This function takes last element as pivot, and places all
24    smaller elements to left of pivot and all greater elements to right of
25    pivot */

```

C++

```
25 int partition (int array[], int low, int high)
26 {
27     int pivot = array[high];    // pivot
28     int i = (low - 1);  // Index of smaller element
29     int j;
30
31     for (j = low; j <= high- 1; j++)
32     {
33         if (array[j] <= pivot)
34         {
35             i++;           // increment index of smaller element
36             swap(&array[i], &array[j]);    // shift all lesser elements
in left half
37         }
38     }
39     swap(&array[i + 1], &array[high]);    // place pivot element at
end of smaller elements
40     return (i + 1);    // the index of pivot element
41 }
42
43 // Recursive function to sort array with quicksort
44 void quickSort(int array[], int low, int high)
45 {
46     if (low < high)
47     {
48         int pivot_index;
49         pivot_index = partition(array, low, high);
50         cout<<"Pivot element is "<<array[pivot_index]<<endl;
51         cout<<"Array after pivot partitioning : \n";
52         printArray(array,8);
53
54         // Call quicksort() on left half and right half excluding
55         // pivot element as it is already at proper position i.e.
56         // between lesser and greater elements.
57         quickSort(array, low, pivot_index - 1);
58         quickSort(array, pivot_index + 1, high);
59     }
60 }
61 int main()
62 {
63     int array[] = {17, 13, 20, 16, 19, 24, 22, 21};
64     int n = sizeof(array)/sizeof(array[0]);
65     cout<<"Given array: \n";
66     printArray(array, n);
67     quickSort(array, 0, n-1);
68     cout<<"Sorted array: \n";
69     printArray(array, n);
70     return 0;
71 }
72
73 /* add tail for c if necessary*/
```

Output: -

```

Given array:
17  13  20  16  19  24  22  21
Pivot element is 21
Array after pivot partitioning :
17  13  20  16  19  21  22  24
Pivot element is 19
Array after pivot partitioning :
17  13  16  19  20  21  22  24
Pivot element is 16
Array after pivot partitioning :
13  16  17  19  20  21  22  24
Pivot element is 24
Array after pivot partitioning :
13  16  17  19  20  21  22  24
Sorted array:
13  16  17  19  20  21  22  24

```

The output above shows the pivot element and splitting of array in each iteration. The algorithm divide the original array into two halves. The recurrence equation for quicksort is as below: -

$$T(N) = T(i) + T(N - i - 1) + cN$$

The time to sort the file is equal to

the **time** to **sort** the left partition with  $i$  elements, plus  
the **time** to **sort** the right partition with  $N-i-1$  elements, plus  
the **time** to build the partitions

So in worst case the pivot is smallest or largest element of the array so the recurrence equation becomes

$T(N) = T(N-1) + N$  which belongs to  $O(n^2)$  class of complexity, whereas **in** best and average **case** the equation becomes: -  
 $T(N) = 2.T(N/2) + N$  which **is** same **as** Merge sort and belongs to  $O(n \log n)$  class.

The selection of pivot element and adjusting the array according to pivot element is a key part in quicksort. The selection of pivot element should be random in implementations to avoid the worst case partition. Also we can implement the partition() function in multiple ways. The main objective of partition() function is to place the pivot element at a position such that all lesser elements will be on left side of pivot and all greater elements must be on right side of pivot element. So that the pivot should be at its proper position in final ordering. We can achieve it in multiple ways as described below: -

a. We can start from the leftmost element and keep track of index of smaller (or equal to) elements as pivot say small\_idx. While traversing, if we find a smaller than pivot element, we increment small\_idx and swap the element with arr[small\_idx]. Otherwise we ignore current element. At last we can swap the pivot element with element at array[pivot\_idx]. It can be explained as:

```

array[] = {30, 80, 20, 40}
Indexes: 0  1  2  3
low = 0, high = 3, pivot = arr[high] = 40
Initialize index of smaller element, small_idx = -1
Traverse elements from j = low to high-1  // j will loop from 0 to 2
j = 0 : Since array[j] <= pivot, increment small_idx and swap(arr[small_idx],
arr[j])
small_idx = 0
array[] = {30, 80, 20, 40} // No change as 10 is already the first element
j = 1 : Since array[j] > pivot, do nothing.      // No change in i and arr[]
j = 2 : Since array[j] <= pivot, increment small_idx and swap(arr[small_idx],
arr[j])
small_idx = 1
array[] = {30, 20, 80, 40}    // We swap 80 and 20
j = 3 : We come out of loop because j is greater than high-1. Finally we
place pivot at correct position by swapping array[small_idx+1] and pivot
array[] = {30, 20, 40, 80}    // 80 and 40 Swapped
Now 40 is at its correct place. All elements smaller than 40 are before it
and all elements greater than 40 are after it.

```

b. Another way is keep the left pointer to first index of array and right pointer to last index of the array. Now keep incrementing left till we get elements lesser than pivot. And keep decrementing right till we get elements greater than pivot. Now either the two indexes will cross each other. If they do then stop, otherwise swap the elements at these two index and continue. Following is an sample implementation of it:

```

pivot = array[end]
left = start
right = end - 1
completed = False
while not completed:
    while left <= right and array[left] <= pivot:
        left = left + 1
    while array[right] >= pivot and right >= left:
        right = right - 1
    if right < left:
        Completed = True
    Else:
        swap (array[left], array[right])
end
swap(array[left], pivot)

```

Similarly, there can be more ways to implement the partition() function. Each implementation is correct if it achieves the goal of partition() function. You can try any other approach also.

## Properties of Quick sort

**Worst Case Time Complexity:**  $O(n^2)$ . If pivot element is smallest or largest element of the array.

**Best and Average Case Time Complexity:**  $O(n \log n)$ . When the pivot element is median of the array, which split the array from middle.



**Auxiliary Space:**  $O(\log n)$ , as array will be partitioned in the same but space is required for recursive calls only.

**Sorting In Place:** Yes

**Stable:** Generally not but stable version exist, that depends on implementation.

One of the fastest algorithms on average. Commercial applications use Quicksort whereas generally quicksort is not used in applications which require guaranteed response time. Because of its popularity, most programming languages providing a default sort function implements quicksort in best possible way like `qsort()` function C language.



# CodeQuotient

Tutorial by codequotient.com | All rights reserved, CodeQuotient 2025

codequotient.com