**Batch: C1 Roll No.: 16010122221**

**Experiment No.\_\_\_\_\_\_\_**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of Quick sort/Merge sort algorithm** |

**Objective:** To learn the divide and conquer strategy of solving the problems of different types

**CO to be achieved:**

|  |  |
| --- | --- |
| CO 2 | Describe various algorithm design strategies to solve different problems and analyze Complexity. |

**Books/ Journals/ Websites referred:**

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algortihtms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Quicksort**
4. **https://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort.html**
5. **http://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf**
6. **http://www.sorting-algorithms.com/quick-sort**
7. **http://www.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf**
8. **http://en.wikipedia.org/wiki/Merge\_sort**
9. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm**
10. **http://www.sorting-algorithms.com/merge-sort**
11. **http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge\_sort.html**

**Pre Lab/ Prior Concepts:**

Data structures, various sorting techniques

**Historical Profile:**

**Quicksort and merge sort are** divide**-**and-conquer sorting algorithm in which division is dynamically carried out. They are one the most efficient sorting algorithms.

**New Concepts to be learned:**

Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem solving vs Divide-and-Conquer problem solving.

**Algorithm** **Recursive Quick Sort:**

**void** quicksort( Integer A[ ], Integer left, Integer right)

**//**sorts A[left.. right] by using partition() to partition A[left.. right], and then //calling itself // twice to sort the two subarrays.

{ **IF** ( left < right ) then

{ q = partition( A, left, right);

quicksort( A, left, q–1);

quicksort( A, q+1, right);

}

}

**Integer *partition(integer A*T[], Integer *left*, Integer *right*)**

*//This function*rarranges *A*[*left***..***right*] and finds and returns an integer *q*, such that *A*[*left*], ..., //*A*[*q*–1] **<**∼*pivot*, *A*[*q*] = *pivot*, *A*[*q*+1], ..., *A*[*right*] > *pivot*, where *pivot* is the first element of //a[left…right], before partitioning**.**

{

pivot = A[left]; lo = left+1; hi = right;

**WHILE** ( lo ≤ hi)

{ **WHILE** (A[hi] > pivot) hi = hi – 1;

**WHILE** ( lo ≤ hi and A[lo] <∼pivot) lo = lo + 1;

**IF** ( lo ≤ hi) then swap( A[lo], A[hi]);

}

swap(pivot, A[hi]);

**RETURN** hi;

}

**The space complexity of Quick Sort:**

**Derivation of best case and worst-case time complexity (Quick Sort)**

**Algorithm Merge Sort**

MERGE-SORT (*A*, *p*, *r*)

// To sort the entire sequence A[1 .. n], make the initial call  to the procedure MERGE-SORT (*A*, //1, *n*). Array *A* and indices *p*, *q*, *r* such that *p* ≤ *q* ≤ r and sub array *A*[*p* .. *q*] is sorted and sub array //*A*[*q* + 1 .. *r*] is sorted. By restrictions on *p*, *q*, *r*, neither sub array is empty.

**//OUTPUT**: The two sub arrays are merged into a single sorted sub array in *A*[*p* .. *r*].

**IF** *p* < *r*                                                    // Check for base case  
         **THEN** *q* = FLOOR [(*p* + *r*)/2]                 // Divide step  
                 **MERGE** (A, *p*, *q*)                          // Conquer step.  
                 MERGE (A, *q* + 1, *r*)                     // Conquer step.  
                 MERGE (A, *p*, *q*, *r*)                       // Conquer step.

MERGE (*A*, *p*, *q*, *r*)

{

*n*1 ← *q* − *p* + 1  
      *n*2 ← *r* − *q*  
      Create arrays L[1 . . *n*1 + 1] and R[1 . . *n*2 + 1]  
      **FOR** *i* ← 1 **TO** *n*1  
            **DO** L[*i*] ← A[*p* + *i* − 1]  
      **FOR** *j* ← 1 **TO** *n*2  
            **DO** R[*j*] ← A[*q* + *j* ]  
      L[*n*1 + 1] ← ∞  
      R[*n*2 + 1] ← ∞  
    *i* ← 1  
    *j* ← 1  
    **FOR** *k* ← *p* **TO** *r*  
         **DO IF** L[*i* ] ≤ R[ *j*]  
                **THEN** A[*k*] ← L[*i*]  
                        *i* ← *i* + 1  
                **ELSE** A[k] ← R[j]  
                        *j* ← *j* + 1

}

**The space complexity of Merge sort:**

Derivation of best case and worst-case time complexity (Merge Sort)

T(n) = 2T(n/2) + n

a = b = 2

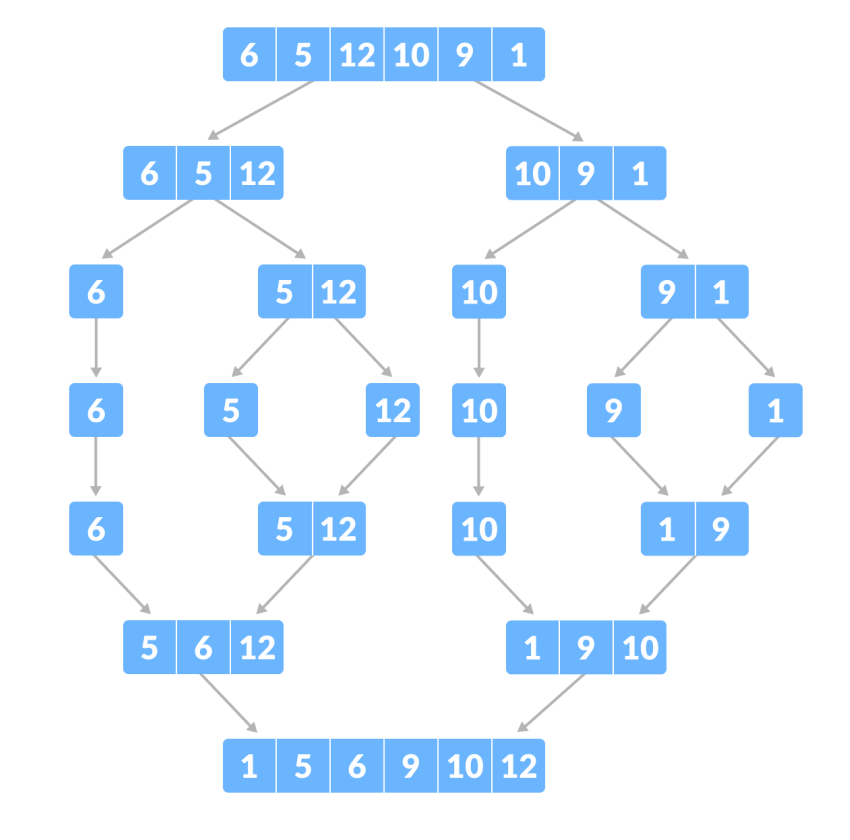
logb a = 1 , c=1

Hence, θ( nclog n) = θ(n log n)

Best Case = Worst Case = θ(n log n)

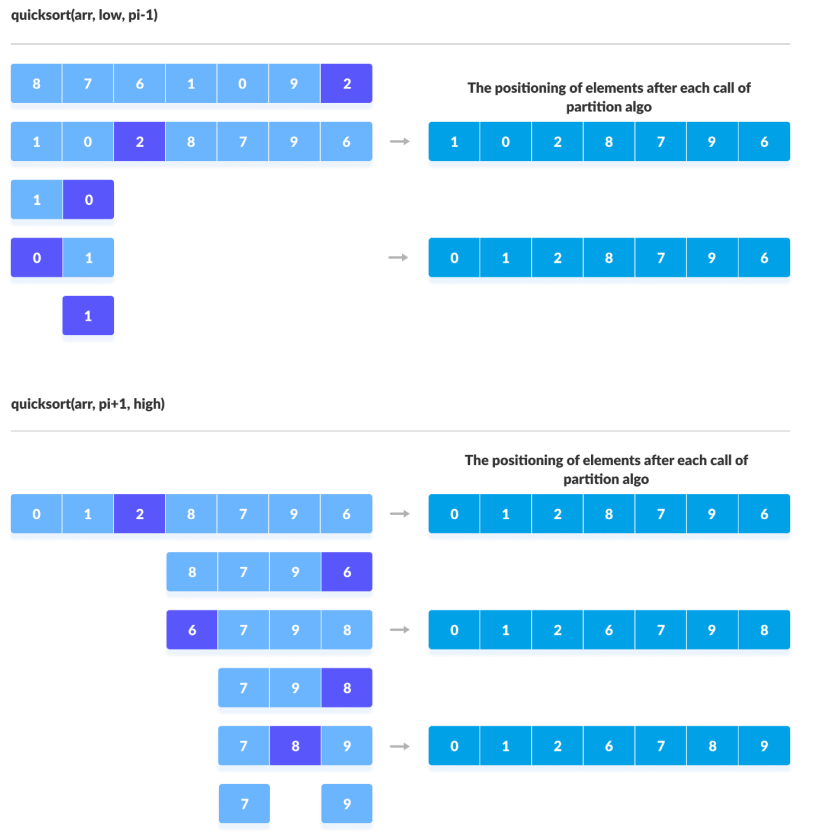
Example for quicksort/Merge tree for merge sort:

**Merge Sort**



**Example for quicksort/Merge tree for merge sort:**

**Quick Sort:**



**Code:**

#include <bits/stdc++.h>

using namespace std;

void merge(int \*arr, int l, int mid, int r)

{

int arr1[r + 1];

int i = l, j = mid + 1, k = l;

while (i <= mid && j <= r)

{

if (arr[i] < arr[j])

{

arr[j] = arr[i];

i++;

}

else

{

arr1[k] = arr[j];

j++;

}

k++;

}

if (i > mid)

while (j <= r)

{

arr1[k] = arr[j];

k++;

j++;

}

else if (j > r)

while (i <= mid)

{

arr1[k] = arr[i];

k++;

i++;

}

for (k = l; k <= r; k++)

arr[k] = arr1[k];

}

void sort(int \*arr, int l, int r)

{

if (l < r)

{

int mid = (l + r) / 2;

sort(arr, l, mid);

sort(arr, mid + 1, r);

merge(arr, l, mid, r);

}

}

int main()

{

int n;

cin >> n;

int arr[n];

for (int i = 0; i < n; i++)

cin >> arr[i];

sort(arr, 0, n - 1);

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

cout << endl;

return 0;

}

**Derivation of best case and worst-case time complexity (Merge Sort)**

T(n) = 2T(n/2) + n

a = b = 2

logb a = 1 , c=1

Hence, θ( nclog n) = θ(n log n)

Best Case = Worst Case = θ(n log n)

**CONCLUSION:**

Understood and implemented Merge sort and quick sort their time and space complexities