

|  |
| --- |
| **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:**

|  |  |
| --- | --- |
| Sr. No | Objective |
| CO 1 | Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations. |
| CO 2 | Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies. |
| CO 3 | Analyze and solve problems for   different string matching algorithms. |



**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 s a** 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;

}

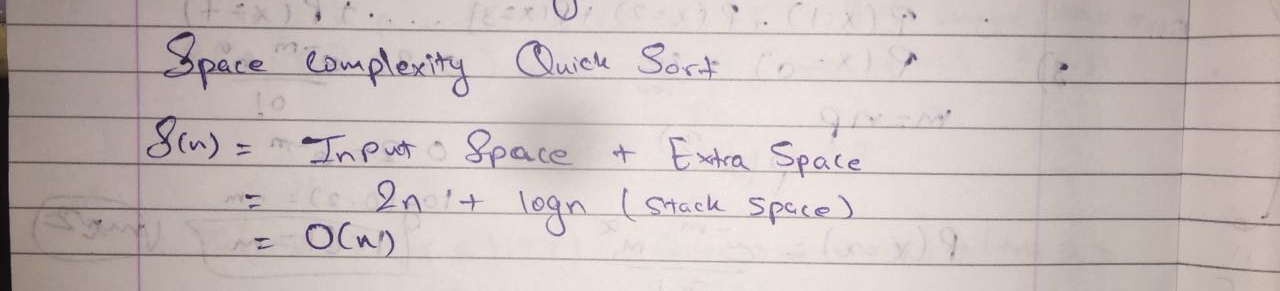
**The space complexity of QuickSort:**

The in-place version of quicksort has a space complexity of O(log n), even in the worst case, when it is carefully implemented using the following strategies:

in-place partitioning is used. This unstable partition requires O(1) space.

After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most O(log n) space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgewick, and keeps the stack depth bounded by O(log n).

Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most O(log n) nested recursive calls, it uses O(log n) space. However, without Sedgewick's trick to limit the recursive calls, in the worst case quicksort could make O(n) nested recursive calls and need O(n) auxiliary space.

****

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

In quick sort, we pick an element called the pivot in each step and re-arrange the array in such a

way that all elements less than the pivot now appear to the left of the pivot, and all elements larger than the pivot appear on the right side of the pivot. In all subsequent iterations of the sorting algorithm, the position of this pivot will remain unchanged, because it has been put in its correct place. The total time taken to re-arrange the array as just described, is always O(n) , or αn where α is some constant . Let us suppose that the pivot we just chose has divided the array into two parts - one of size k and the other of size n − k. Notice that both these parts still need to be sorted. This gives us the following relation:

T(n) = T(k) + T(n − k) + αn

where T(n) refers to the time taken by the algorithm to sort n elements.

**WORST CASE ANALYSIS:**

Now consider the case, when the pivot happened to be the least element of the array, so that

we had k = 1 and n − k = n − 1. In such a case, we have:

T(n) = T(1) + T(n − 1) + αn

Now let us analyse the time complexity of quick sort in such a case in detail by solving the recurrence

as follows:

T(n) = T(n − 1) + T(1) + αn

= [T(n − 2) + T(1) + α(n − 1)] + T(1) + αn

(Note: I have written T(n − 1) = T(1) + T(n − 2) + α(n − 1) by just substituting n − 1 instead

of n. Note the implicit assumption that the pivot that was chosen divided the original subarray of

size n − 1 into two parts: one of size n − 2 and the other of size 1.)

= T(n − 2) + 2T(1) + α(n − 1 + n) (by simplifying and grouping terms together)

= [T(n − 3) + T(1) + α(n − 2)] + 2T(1) + α(n − 1 + n)

= T(n − 3) + 3T(1) + α(n − 2 + n − 1 + n)

= [T(n − 4) + T(1) + α(n − 3)] + 3T(1) + α(n − 2 + n − 1 + n)

**=** T(n − 4) + 4T(1) + α(n − 3 + n − 2 + n − 1 + n)

= T(n − i) + iT(1) + α(n − i + 1 + ..... + n − 2 + n − 1 + n) (Continuing likewise till the i

th

step.)

= T(n − i) + iT(1) + α(

Pi−1

j=0(n − j)) (Look carefully at how the summation is being written.)

Now clearly such a recurrence can only go on until i = n − 1 (Why? because otherwise n − i

would be less than 1). So, substitute i = n − 1 in the above equation, which gives us:

T(n) = T(1) + (n − 1)T(1) + α

Pn−2

j=0 (n − j)

= nT(1) + α(n(n − 2) − (n − 2)(n − 1)/2) (Notice that Pn−2

j=0 j =

Pn−2

j=1 j = (n − 2)(n − 1)/2

by a formula we earlier derived in class)

which is O(n

2

).

This is the worst case of quick-sort, which happens when the pivot we picked turns out to be

the least element of the array to be sorted, in every step (i.e. in every recursive call). A similar

situation will also occur if the pivot happens to be the largest element of the array to be sorted.

**BEST CASE ANALYSIS:**

The best case of quicksort occurs when the pivot we pick happens to divide the array into two

exactly equal parts, in every step. Thus we have k = n/2 and n − k = n/2 for the original array of

size n.

Consider, therefore, the recurrence:

T(n) = 2T(n/2) + αn

= 2(2T(n/4) + αn/2) + αn

(Note: I have written T(n/2) = 2T(n/4) + αn/2 by just substituting n/2 for n in the equation

T(n) = 2T(n/2) + αn.)

= 22T(n/4) + 2αn (By simplifying and grouping terms together).

= 22

(2T(n/8) + αn/4) + 2αn

= 23T(n/8) + 3αn

= 2kT(n/2

k

) + kαn (Continuing likewise till the k

th step)

Notice that this recurrence will continue only until n = 2k

(otherwise we have n/2

k < 1), i.e.

until k = log n. Thus, by putting k = log n, we have the following equation:

2

T(n) = nT(1) + αn log n, which is O(n log n).

This is the best case for quicksort.

It also turns out that in the average case (over all possible pivot configurations), quicksort has

a time complexity of O(n log n), the proof of which is beyond the scope of our class.

**Algorithm MergeSort**

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 subarray *A*[*p* .. *q*] is sorted and subarray //*A*[*q* + 1 .. *r*] is sorted. By restrictions on *p*, *q*, *r*, neither subarray is empty.

**//OUTPUT**: The two subarrays are merged into a single sorted subarray 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* )

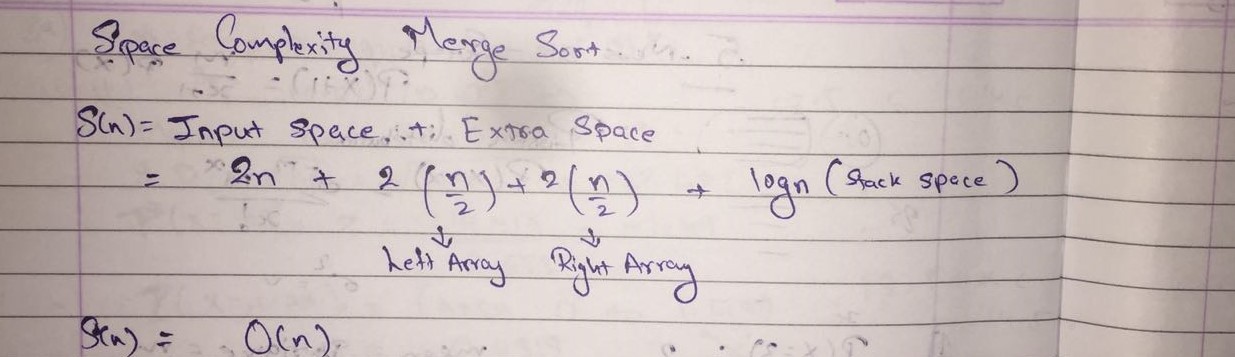
{

*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:**

Mergesort, if implemented to create arrays in the recursive calls, will create many of them, but they won't coexist at the same time. In every recursive call you create an array (or 2 depending on an implementation) for merging and they take no more than O(n) space, and then when the merging is done, these arrays are deleted and some new ones will be created after a moment in some other recursive call. If you counted how much space all the arrays that ever have been created took, it'd be O(n log n), but you don't need to care about this information - you don't need more than O(n) space, because when you need to create an array, all the other ones don't longer exist and don't occupy any memory. Note that you can simply declare 2 - or 3 - arrays in the beginning, each the length of n, and then store the sequence in one of them, while using the other for merging, it will improve the performance as well as show you beyond doubt there's no need for more than O(n) of memory.



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

mergesort( int [] a, int left, int right)

{

if (right > left)

{

middle = left + (right - left)/2;

mergesort(a, left, middle);

mergesort(a, middle+1, right);

merge(a, left, middle, right);

}

}

Assumption: N is a power of two.

For N = 1: time is a constant (denoted by 1)

Otherwise: time to mergesort N elements = time to mergesort N/2 elements plus

time to merge two arrays each N/2 elements.

Time to merge two arrays each N/2 elements is linear, i.e. N

Thus we have:

(1) T(1) = 1

(2) T(N) = 2T(N/2) + N

Next we will solve this recurrence relation. First we divide (2) by N:

(3) T(N) / N = T(N/2) / (N/2) + 1

N is a power of two, so we can write

(4) T(N/2) / (N/2) = T(N/4) / (N/4) +1

(5) T(N/4) / (N/4) = T(N/8) / (N/8) +1

(6) T(N/8) / (N/8) = T(N/16) / (N/16) +1

(7) ……

(8) T(2) / 2 = T(1) / 1 + 1

Now we add equations (3) through (8) : the sum of their left-hand sides

will be equal to the sum of their right-hand sides:

T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + … + T(2)/2 =

T(N/2) / (N/2) + T(N/4) / (N/4) + ….+ T(2) / 2 + T(1) / 1 + LogN

(LogN is the sum of 1s in the right-hand sides)

After crossing the equal term, we get

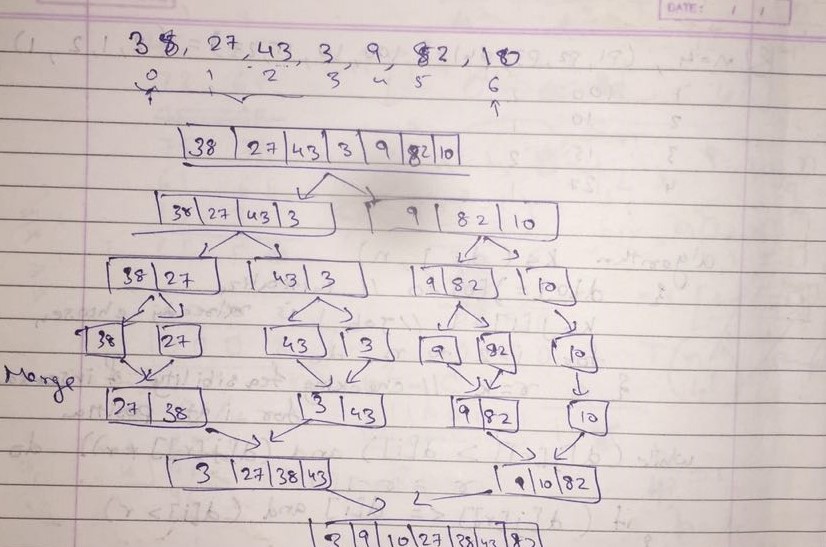
(9) T(N)/N = T(1)/1 + LogN

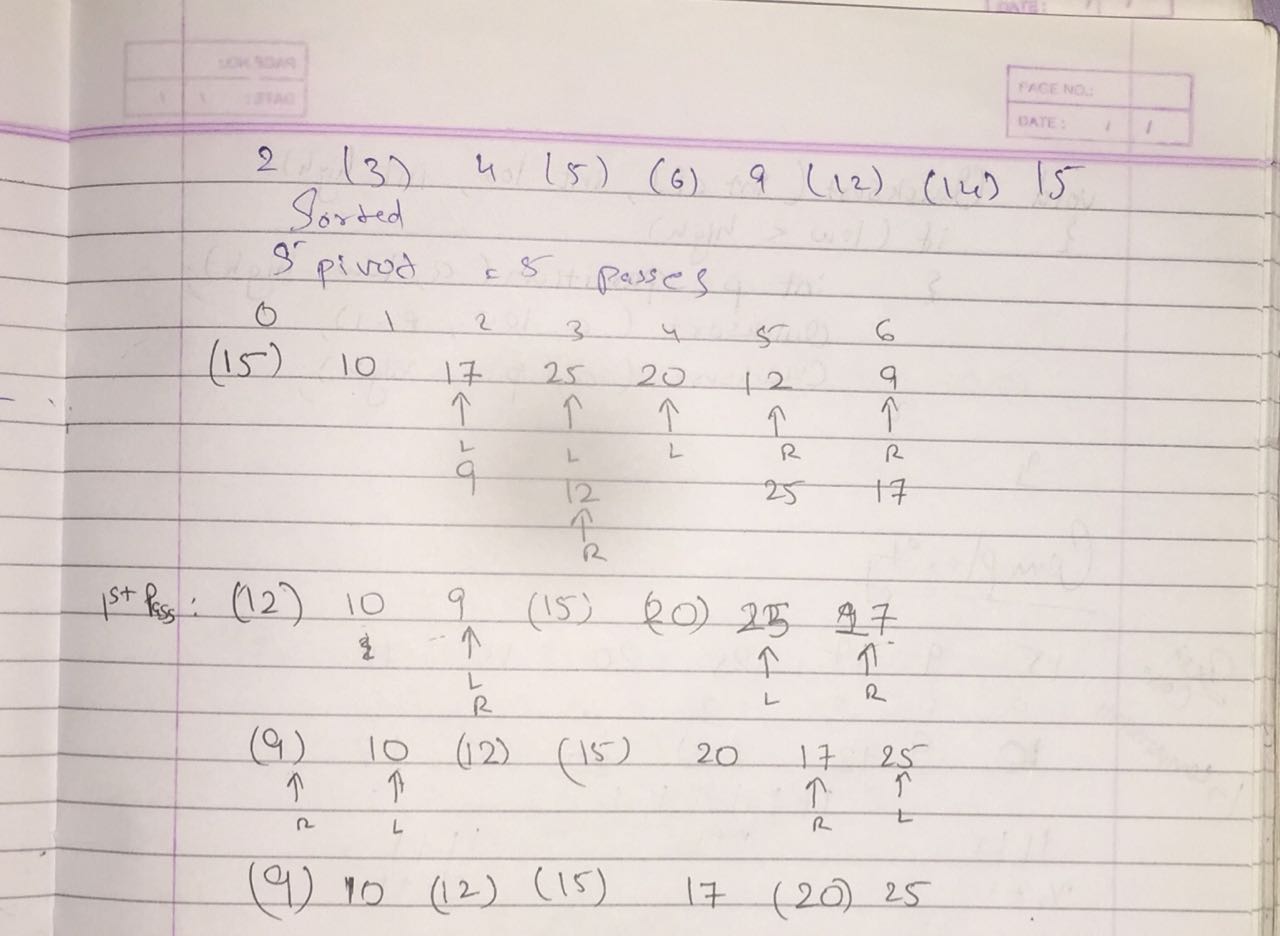
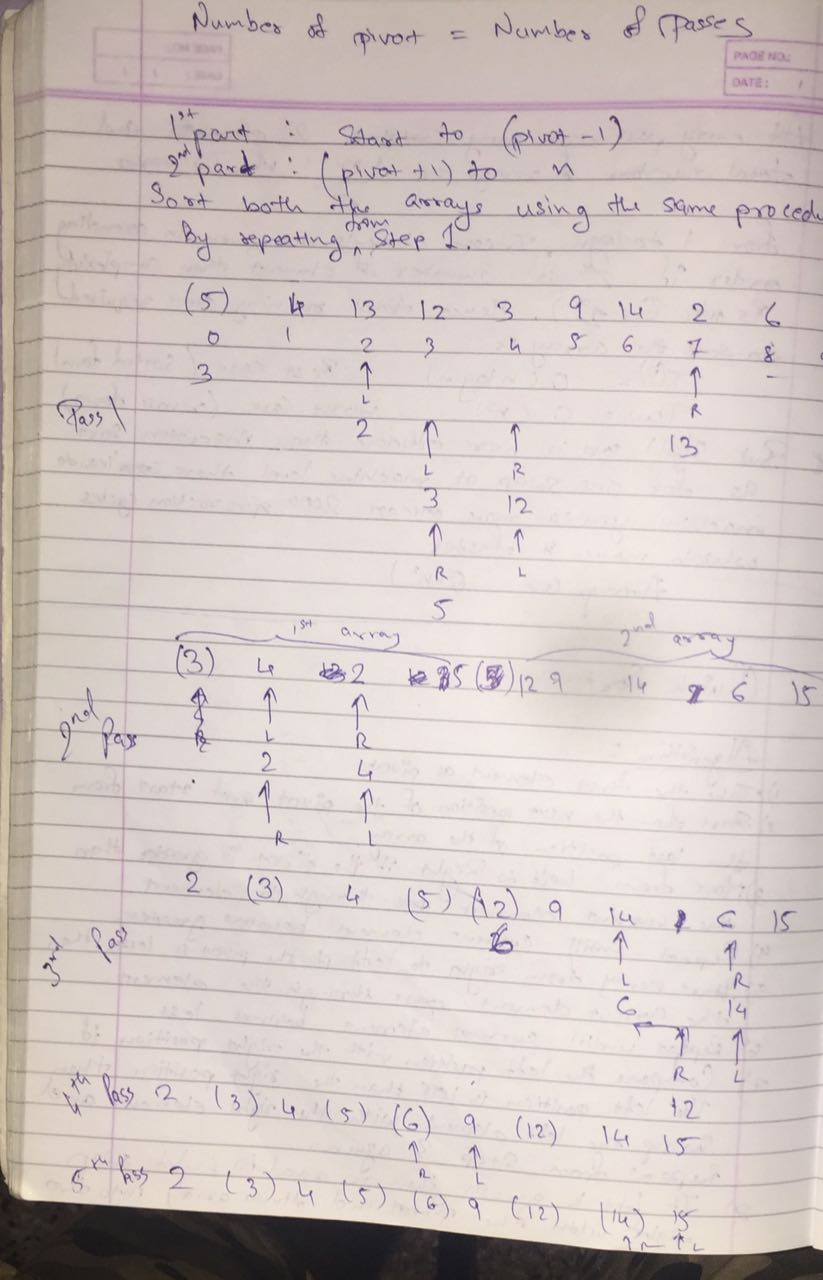
T(1) is 1, hence we obtain

(10) T(N) = N + NlogN = O(NlogN)

Hence the complexity of the MergeSort algorithm is O(NlogN).

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

****

****

**CONCLUSION: Thus Implementation of Quick sort/Merge sort algorithm was done successfully and space and time complexity was also found out successfully.**