**Batch: B3 Roll No.: 121**

**Experiment No.\_\_\_3\_\_\_\_**

**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. Rajasekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algorithms”,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**
12. **https://www.enjoyalgorithms.com/blog/quick-sort-algorithm**
13. **https://www.enjoyalgorithms.com/blog/merge-sort-algorithm**
14. **https://www.javatpoint.com/daa-merge-sort**
15. **https://www.scaler.com/topics/merge-sort-time-complexity/**
16. **https://www.geeksforgeeks.org/quicksort-better-mergesort/**

**Pre Lab/ Prior Concepts:**

Data structures, various sorting techniques

**Historical Profile:**

**Quicksort and merge sort are** divide**-**and-conquer sorting algorithms 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*rearranges *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:**

The space used by Quicksort depends on the version used.

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

* + 1. In-place partitioning is used. This unstable partition required O(1) space.
    2. After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most O(logn) space. Then the other partition is sorted using tail recursion or iteration, which does not add to the call stack. This idea keeps the stack depth bounded by O(logn).

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(logn) nested recursive calls, it uses O(logn) space. However, without the above-mentioned tricks to limit the recursive calls, in the worst case quicksort could make O(n) nested recursive calls and need O(n) auxiliary space.

From a bit complexity viewpoint, variables such as lo and hi do not use constant space; it takes O(logn) bits to index into a list of n items. Because there are such variables in every stack frame, quicksort using the above-mentioned trick requires O((logn)2) bits of space. This space requirement is acceptable as if the list contained distinct elements, it would need at least O(nlogn) bits of space.

Another, less common, not in-place version of Quick sort uses O(n) space for working storage and can implement a stable sort. The working storage allows the input array to be easily partitioned in a stable manner and then copied back to the input array for successive recursive calls.

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

Let it be assumed that T(n) is the worst-case time complexity of quicksort for n integers. Let it be analysed by breaking down time complexities of each process:

Divide part: This is equal to the time complexity of partition algorithm, which is O(n).

Conquer part: Two subproblems of different sizes are being recursively solved. The size of a subproblem depends on the choice of pivot in the partition process. Suppose after the partition, I elements are in left subarray, and n - i – 1 elements are in the right subarray.

Time complexity of conquer part = Time complexity of recursively sorting left subarray + Time complexity of recursively sorting right subarray = T(i) +T(n - i - 1)

Combine part: This is a trivial part of quick sort algorithm. The time complexity is O(1).

For calculating overall time complexity T(n), the time complexities of divide, conquer and combine parts need to be added:

T(n) = O(n) + T(i) + T(n - i - 1) + O(1)

= T(i) + T(n - i - 1) + O(n)

= T(i) + T(n - i - 1) + cn

Recurrence relation of quick sort

T(n) = c, if n = 1

T(n) = T(i) + T(n - i - 1) + cn, if n > 1

**Best case time complexity of Quick Sort:**

The best-case behaviour for quicksort occurs when there is good luck and partition process always picks the middle element as the pivot. In other words, this is a case of balanced partition, where both sub-problems are n/2 size each.

Let us assume that balanced partitioning arises in each recursive call. For calculating the time complexity in best case, i = n/2 is put in the above formula of T(n).

T(n) = T(n/2) + T(n - 1 - n/2) + cn

= T(n/2) + T(n/2 - 1) + cn

≈ 2T(n/2) + cn

There are following three cases of analysis using master theorem:

* + - * 1. If f(n) = O(nk) where k < logb(a) then T(n) = O(nlogb(a))
        2. If f(n) = O(nk) where k = logb(a) then T(n) = O(nklogn)
        3. If f(n) = O(nk) where k > logb(a) then T(n) = O(nk)

Comparing T(n) = 2T(n/2) + cn with T(n) = aT(n/b) + O(nk),

a = 2, b = 2

O(nk) = cn = O(n1) => k = 1

Similarly, logb(a) = log2(2) = 1 = k

It means that the second case is satisfied. Therefore, Quick sort time complexity T(n) = O(nklogn) = O(nlogn)

**Worst case time complexity of Quick Sort:**

Worst-case scenario of quicksort occurs when the partition process always picks the largest or the smallest element as the pivot. In this scenario, partition process would be highly unbalanced i.e., one subproblem would have n-1 elements and the other would have 0 elements. This situation occurs when the array is sorted in ascending or descending order.

Let it be assumed that unbalanced partitioning arises at each recursive call. For calculating the time complexity in the worst case, i = n - 1 is put in the above formula for T(n).

T(n) = T(n - 1) + T(0) + cn

T(n) = T(n - 1) + cn

Now, the recurrence relation is simply expanded by substituting all intermediate values of i (from i = n - 1 to 1).

T(n) = T(n - 1) + cn

= T(n - 2) + c(n - 1) + cn

= T(n - 3) + c(n - 2) + c(n - 1) + cn

… and so on

= T(1) + 2c + 3c + … + c(n - 3) + c(n - 2) + c(n - 1) +cn

= c(1 + 2 + 3 + … + n-3 + n-2 + n-1 + n)

This is a case of Arithmetic Progression

T(n) = c(n(n + 1)/2) = O(n2)

**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 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:**

Merge sort space complexity depends on the extra space used by the merging process and the size of recursion call stack used by the recursion.

* Space complexity of merging process = O(n)
* Space complexity for recursion call stack = Height of merge sort recursion tree = O(logn)
* Space complexity of merge sort algorithm = O(n) + O(logn) = O(n)

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

Let T(n) be the time taken by the Merge sort algorithm.

* Sorting two halves will take at most n/2 time.
* When the sorted lists are merged, a total of n-1 comparisons is required because the last element which is left will need to be copied down to the combined list, and there will be no comparison.

Thus, the recurrence relation will be

-1 is ignored because the element will take some time to be copied in the merged list.

So equation 1 is

The stopping condition is T(1) = 0 because at last, there will be only one element left which can be copied, and hence, no comparison would be required.

Putting n = n/2 in equation 1,

(Equation 2)

Putting T(n/2) in equation 1,

(Equation 3)

Putting n = n/22 in equation 1,

(Equation 4)

Putting T(n/22) in Equation 3,

(Equation 5)

From equations 1, 3 and 5, it can be inferred that

(Equation 6)

From stopping condition,

From Equation 6,

Best case complexity:

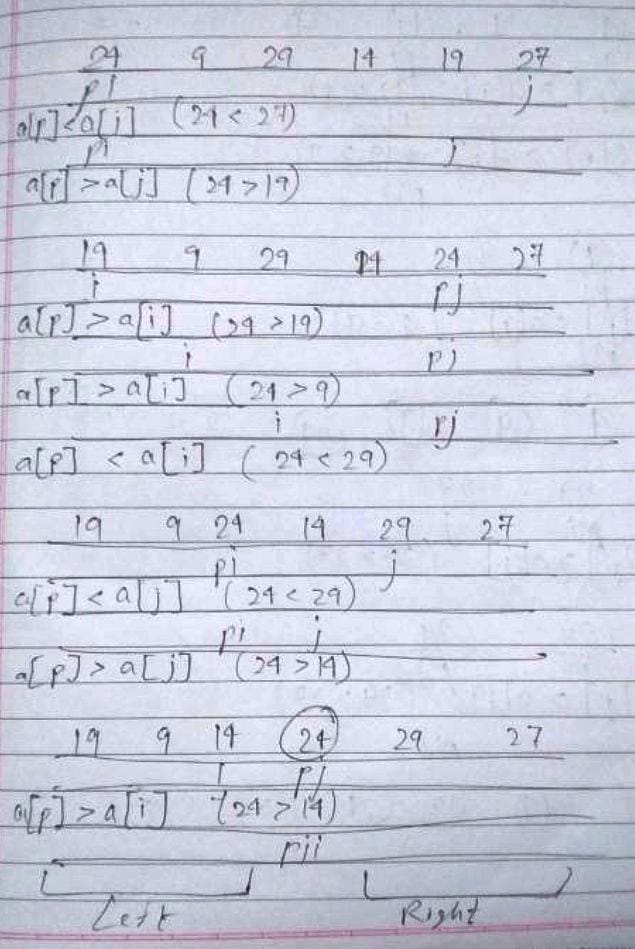
The best case scenario occurs when the elements are already sorted in ascending order.

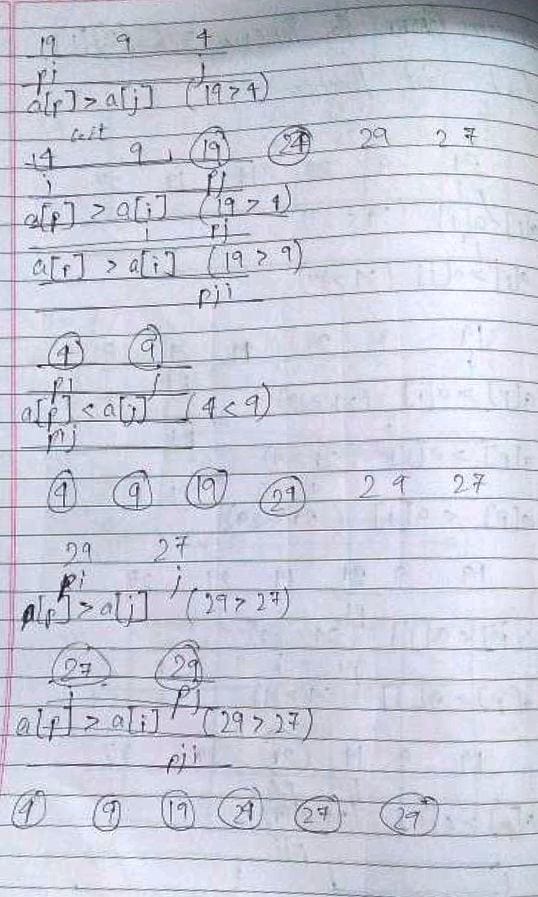
If two sorted arrays of size n need to be merged, the minimum number of comparisons will be n. This happens when all elements of the first array are less than the elements of the second array. The best case time complexity of merge sort is O(nlogn).

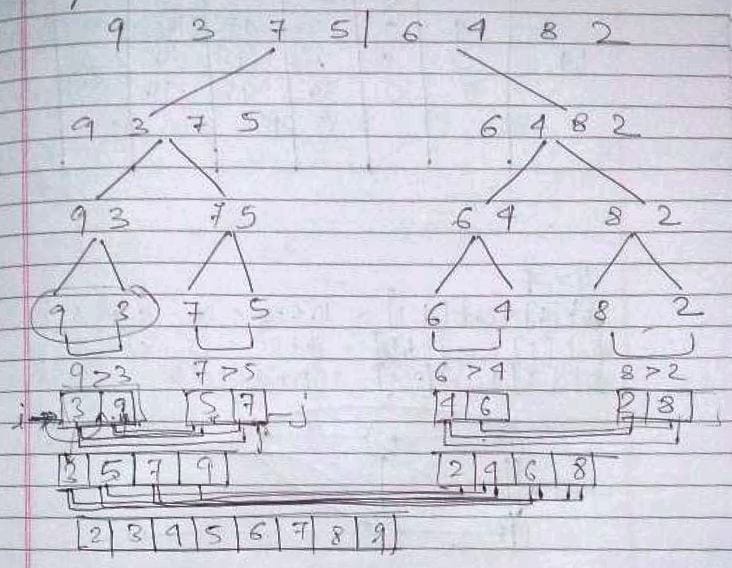
Worst case complexity:

The worst case scenario occurs when the elements are sorted in descending order, leading to maximum number of comparisons. In this case, for two sorted arrays of size n, the minimum number of comparisons will be 2n. The worst case time complexity of merge sort is O(nlogn).

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

****

****

****

**CONCLUSION:**

Thus, in this experiment, the sorting methods of Quick Sort and Merge Sort have been learnt and implemented in java. Although merge sort has better worst case performance than quick sort, yet quick sort is better overall. This is because merge sort uses extra space, while quick sort uses much less space; merge sort is not in-place, while quick sort is in-place. Also, the worst case of quick sort O(n2) can be avoided using randomized quick sort. However, merge sort is better for large data sets as it is a stable sort and can be adapted to operate on linked lists and very large lists.