Comparative Study of Two Divide and Conquer Sorting Algorithms: Modified Quick Sort and Merge Sort

Ibtehal Mishal
Computer System Engineering Department,
Faculty of Engineering, Al-Balqa Applied University,
Al Salt, Jordan

Rasha AL-Khatib
Computer System Engineering Department,
Faculty of Engineering, Al-Balqa Applied University,
Al Salt, Jordan

Razan Hiasat
Computer System Engineering Department,
Faculty of Engineering, Al-Balqa Applied University,
Al Salt, Jordan

ABSTRACT
Divide and conquer is a well-known technique for sorting algorithms. Such include Quick sort and Merge sort sorting algorithms. These two algorithms have been extensively used for sorting. However, discovering the most efficient sorting algorithm among the two has always been a contentious problem. Most of the existing research have compared quick sort and merge sort, this study intends to compare the intelligent Quick Sort algorithm based on a dynamic pivot selection technique "modified quicksort" and the merge sort. Using machine-dependent factors such as computational and employed machine-independent internal/external sorting factors, memory usage, stability, algorithm complexity: best, average, and worst cases. This study intends to contribute to this discussion using both machine-dependent and independent factors. Results obtained revealed that in terms of computational speed using an array of small sizes, the classical Quicksort algorithm is almost fast, meanwhile, the Merge sort algorithm is faster with an array of large sizes, However, modified quicksort is the fastest available option in all sizes. Also, the best case for both merge sort and classical quick sort complexity is O(nlogn), but the modified quicksort best case is O(n) which happened when the array is already sorted while the three sorts are of O(nlogn) average case, and the worst case for classical quicksort is O($\boldsymbol{n2}$) and that of merge sort and modified quick sort remains unchanged. In terms of stability, modified Quicksort is stable while Merge sort is not. Despite the excellent performance of the Merge sort algorithm, the need for an auxiliary memory for sorting makes it less preferable than the modified Quicksort algorithm for applications where a good cache locality is of paramount importance.

General Terms

Computer Science ,sorting algorithems.

Keywords
Computer Science, Software engineering, Sorting algorithms, Computational Mathematics

1. INTRODUCTION
Sorting requires arranging or organizing the elements of a list (1D Array) in a specified manner.[3] The sort order is a way of comparing two items for the purpose of sorting.
A common and simple example where sorting is always applied is a list of items. However, in computer science, there are many problems in which it is less obvious that sorting is required.[6] Some of the factors that are normally considered when it comes to the choice of sorting algorithm to be used include: format of input, amount, and nature of data, and machine-specific criteria.[3]
Of all the algorithm design techniques, the divide and conquer technique is the most extensively applied technique. [5]
It employs the following approach:
a) It divides a problem into several sub-problems of the same type.
b) These sub-problems are sorted recursively
c) Most times the resulting solutions are combined to get a final sorting solution to the original problem.
Two perfect examples of sorting algorithms that are a product of the divide and conquer algorithm design technique are Merge sort and Quicksort algorithms. And in the advanced level modified quicksort has been widely employed to solve various real-life sorting problems, however, the choice of the more preferred of the merge and quick sort algorithms have always resulted in heated arguments and controversies. As a result, different comparison that has been carried out by researchers, leading to enhancement on the quicksort reducing its time complexity Most of these comparisons have been implemented on virtual and real computers using different number of inputs. However, most works have not employed large range of data to examine the true behavior of these algorithms.
The machine-dependent factors were carefully selected using a range of different data sizes and different data types to better understand the algorithms' true behavior for both small and large data sizes.
The factors such as time complexity, stability memory space, and the actual time is taken when each of the algorithms is implemented are used as the basis of comparison [7].

2. COMPARATIVE STUDY OF MERGE SORT AND MODIFIED QUICKSORT ALGORITHM

The comparative study of the two algorithms can be studied using factors that are machinedependent and machine-independent. Machinedependent factors are factors that can be measured and compared to specific machine configurations. For example, Computational complexity where the time taken by each algorithm to perform the sorting operation is measured, is considered as a machine-dependent factor. Machine independent factors are the factors that can be measured and compared from a general point of view using a mathematical entity or based on the behavior of each algorithm. [7] Factors such as internal/external sorting, system complexity which measures metrics such as worst case, average case and best case, memory usage, and stability are examples of machineindependent factors. The factors

considered in this paper are defined as follows: [7] a) Internal Sorting: this examines the mode of sorting carried out in the main memory. This could be direct or indirect. b) External Sorting: this examines the mode of sorting carried out in the auxiliary memory. c) System complexity: these can be classified using metrics such as: worst, average, and bestcase scenarios. d) Computational complexity: this could be measured using the number of swaps carried out by the algorithm during the sorting process. e) Memory Usage: each algorithm has different memory requirements. This can be used to differentiate them. f) Stability: this measures the state of the input and output records as shown by the order of its elements before and after sorting. g) Input size: different sizes of arrays are used to test the program where the two algorithms are implemented.

2.1 System Complexity of Merge Sort Algorithms

Suppose we had to sort an array A. A subproblem would be to sort a subsection of this array starting at index p and ending at index r, expressed as A[p...r].[4]

- . Divide If q is the half-way point between p and r, then we can split the subarray A[p..r]
  into two sub-arrays A[p...q] and A[q+1, r]
- . Conquer In the conquer step, we try to sort both the sub-arrays A[p…q] and A[q+1, r]., we again divide both these subarrays and try to sort them.
- . Combine When the conquer step reaches the base step and we get two sorted subarrays

A[p…q] and A[q+1, r] for array A[p..r], we combine

the results by producing a sorted array A[p...r] from two sorted sub-arrays A[p…q] and A[q+1,r].

The following Figure (1) shows the complete merge sort process for an example array

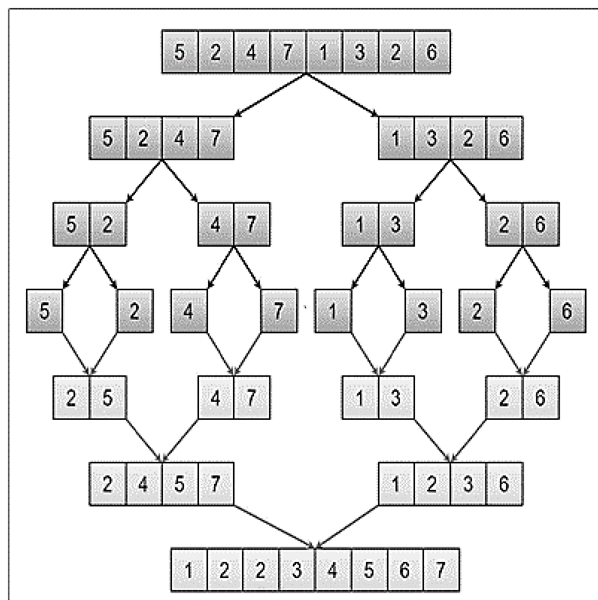

Fig1:merge sort process

For example, with a list containing elements: {5, 2, 4, 7, 1, 3, 2, 6 }

, Merge Sort algorithm will employ the master's theorem explained in Fig 1 to sort the elements. The analysis of the Merge Sort algorithm requires the use of the master's theorem[4].

The Master's theorem is given as follows:

Given function f (n) with constants a≥ 1 and b>1; then the time complexity of a recursive relation is given by

T(n) = aT(n/b) + f(n) where,

T(n) has the following asymptotic bounds:

If f(n) = O(nlogb a-ϵ), then T(n) = Θ(nlogb a).

If f(n) = $\mathbf{\Theta(n^{logb\ a})}$, then T(n) = $\mathbf{\Theta(n^{logba} * log\ n)}$.

If f(n) = $\mathbf{\Omega(n^{logb\ a+\epsilon})}$, then T(n) = $\mathbf{\Theta(f(n))}$.

$\epsilon > 0$ is a constant

the time for merge Sort function will become n(log n + 1), which gives us a time complexity of O(nlog n) for the Worst Case ,Best Case and Average Time Space Complexity of O(n) is used.

Time complexity of Merge Sort is O(nLog n) in the 3 cases as merge sort always divides the array in two halves and takes linear time to merge two halves[4].

Given below is an implementation of merge sort using C++ in the figure (2):

```
template<class t>
void merge(t* a, int l, int m, int r)
{

    int n1 = m - l + 1;
    int n2 = r - m;
    t* L = new t[n1];
    t* R = new t[n2];
    for (int i = 0; i < n1; i++)
        L[i] = a[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = a[m + 1 + j];

    int i = 0;
    int j = 0;
    int k = l;
    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            a[k] = L[i];
            i++;
        }

        else {

            a[k] = R[j];
            j++;

        }
        k++;
    }

    while (i < n1) {

        a[k] = L[i];
        i++;
        k++;

    }

    while (j < n2) {

        a[k] = R[j];
        j++;
        k++;

    }
}
template<class t>
void mergeSort(t* ar, int l, int r) {

    if (l >= r) {

        return;//returns recursively

    }

    int m = l + (r - l)/2;
    mergeSort(ar, l, m);
    mergeSort(ar, m + 1, r);
    merge(ar, l, m, r);
}
```

Fig 2: theimplementation of merge sort

2.2 System Complexity Of Modified Quick Sort Algorithms

The QuickSort algorithm is the fastest sorting algorithm based on different studies.

Quicksort follows the technique of divide and conquer by recursively splitting each array into two sub-arrays, which makes it easier to solve smaller problems than a single larger one

In Quicksort, a pivot is selected from the unsorted array and used to split the array into two sub arrays for which the same algorithm is called recursively until the sub arrays have size one or zero.

For an input size n, the worst-case scenario of $\Theta(\boldsymbol{n2})$ when sorting an already sorted list while choosing the largest element as a pivot.

However, the QuickSort algorithm has an average runtime complexity of Θ(nlog n)

The runtime of the Quicksort algorithm mainly depends on the splitting of the array and the following sub-arrays.

If splitting constantly results in a small reduction in the size of the array or sub-array, the runtime will be:

T(n) = n + T(n-c), where c is a constant.

Thereby,

$T(n) = \Theta(n)$ (1)

However, if splitting constantly results almost equal size subarrays, the runtime complexity of Quick sort will be:

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

Thereby,

$T(n) = \Theta(n \log n)$ (2)

Splitting the array into almost equal halves ensures the best performance for the Quicksort algorithm and reduces the number of recursive calls which will eventually reduce execution time.

The enhancement is concerned with the pivot selection technique which has proven to be the most determining factor in dividing the array into sub-arrays.

Various techniques have been proposed to avoid the worst-case scenario previously explained in (1). [2]

Classical Quicksort algorithm uses the left-most or the right-most element as a pivot which can easily cause the worst-case behavior when sorting a sorted or a partially sorted list.

Therefore, it drives the worst case behavior to be O(n log n).

The proposed technique also verifies an already sorted array or sub-array which is done while comparing the elements of the array to the pivot. If the array is already sorted, it will not be sorted any further which modifies the$O(\boldsymbol{n2})$ complexity into the best-case behavior of the algorithm, i.e. O(n).

The proposed technique operates as follows, at first, the pivot value is chosen to be the value of the rightmost element of the array. Each element value will be compared with the pivot value and two counters are used to count the number of elements with values smaller than the pivot versus the number of elements with values larger than the pivot.

The sum of the values of the elements smaller than the pivot and the sum of those larger than the pivot is calculated.

The integer average of the values smaller than the pivot is passed as the pivot value of the recursive call for the left sub-array. Likewise, the integer average of the values larger than the pivot is passed as the pivot value of the recursive call for the right sub array.

This pivot selection technique helps in successively splitting the array into nearly equal halves which in turn decrease the complexity of the Quick Sort algorithm.

A Boolean variable is used by the algorithm to recognize an already sorted array or sub array which reduces the number of recursive calls. Along with the reduction in recursive calls, the proposed technique converts the worst-case scenario for the classical Quick Sort algorithm into a best-case scenario with Θ(n) runtime in the modified quick sort[2].

The dynamic pivot selection technique is independent on the position of the values stored in the array. Figure(3) displays the splitting of the arrays in Figure (3 /a) and (3/b) where the same splitting tree is generated for both arrays which indicate that the Modified Quick Sort algorithm is not sensitive to the order of elements in the array[2].

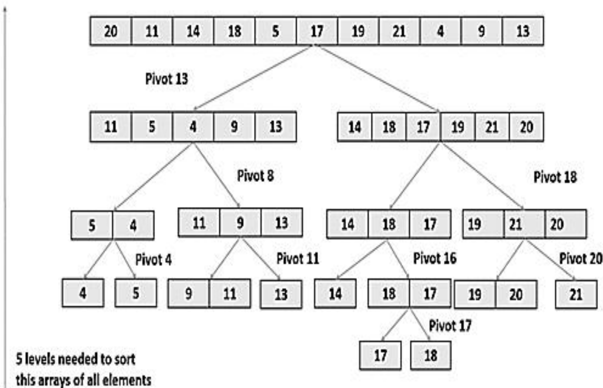

Fig 3/a :MQuick sort processor on the first order

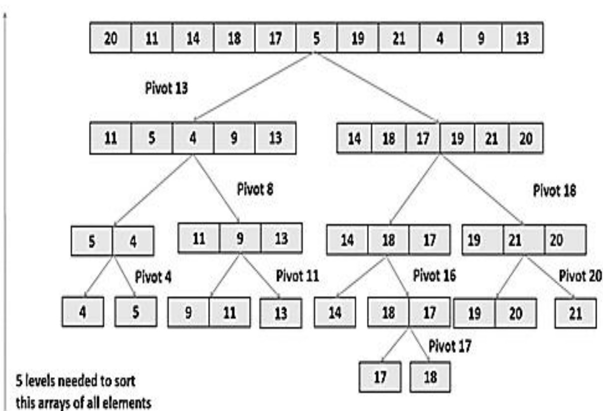

Fig 3/b :MQuick sort processor on the second order

Given below is an implementation of modified quick sort in the figure(4)

```
template <class t>
void exchange(t* f, int i, int j) {// swap fuction
    t swap;
    swap = f[i];
    f[i] = f[j];
    f[j] = swap;
}

template<class t>
void MQuickSort(t* F, int L, int R, t Pivot) { // L : first index , R : last index , Pivot :A [ R ]the first call .
    if (L < R)
    {
        int i, z, j, v;
        int  Countless, CountLarger;
        t Pivot1, Pivot2, K, Sunless, SumLarger; // K is used to check if array is sorted and update the boolean N
        bool N = true;
        i = z = L;
        j = v = R;
        Pivot1 = Pivot2 = Sumless = SumLarger = 0;
        Countless = CountLarger = 0;
        K = F[R];
        while (i <= j) {
            if (F[i] <= Pivot)
            {
                Countless++; // count the elenents less or equal to the pivot Sunless
                SumLarger += F[i]; // sum of elenents less or equal to the pivot
                if (N == true && K >= (Pivot - F[i])) // the differenace in this step is less than the difference in the pervious step
                    K = Pivot - F[i];//which is mean the element in the pervious step is less than the elment now which is mean it is sorted
                else
                    N = false;// any element is not sorted and then the array is not sorted array  and it will not  care about this condition again
                i++;
            }
            else {
                CountLarger++;
                SunLarger += F[i];
                exchange(F, i, j);// swap the elenents i and j in array F
                j--;
            }
        }

        if (Countless != 0) {
            Pivot1 = Sunless / Countless;
            floor(Pivot1);
            if (N != true) // if subarray is not sorted
                MQuickSort(F, z, i - 1, Pivot1);
        }

        if (CountLarger != 0) {
            Pivot2 = SunLarger / CountLarger;
            floor(Pivot2);
            MQuickSort(F, i, v, Pivot2);

        }
    }
}
```

Fig 4:the implementation of MQuick sort

3. RESULTS AND DISCUSSION

3.1Results Of System Complexity for Quicksort and Merge Sort and Modified Quick Sort Algorithms

An overview of the system complexity, stability, and internal versus external characteristics of modified quick sort, merge sort and classical quicksort algorithms are provided in Table (1). Regarding the system complexity, the best case for both merge sort and classical quick sort complexity is O(nlogn), but the modified quicksort best case is O(n) which occurred when the array is already sorted while the three sorts are of O(nlogn) average case, and the worst case for classical quicksort is O($\boldsymbol{n2}$) and that of merge sort and modified quick sort remains unchanged

Quicksort algorithm does not keep elements with equal values in the same relative order in the output as they were in the input (unstable) while merge and modified quick sorts do (stable).

Quick Sort and modified quicksort do not need auxiliary memory therefore it is an in-place (internal) algorithm while

merge sort needs auxiliary memory (external).

[Kazim. (n.d.). A. A Comparative Study of Well Known Sorting Algorithms.] explains this as an advantage that quicksort has over merge sort, the fact that quicksort does not need additional storage space makes it presents good cache locality.[3]

Table1.systems complexity for three algorithms

S/N	Parameters		Quick sort	Merge sort	Modified Quicksort
1	System Complexity	Best Case	O(nlogn)	O(nlogn)	O(n)
		Average Case	O(nlogn)	O(nlogn)	O(nlogn)
		Worst Case	$O(n^2)$	O(nlogn)	O(nlogn)
2	Stability		Unstable	Stable	Stable
3	Internal vs External memory		Internal	External	Internal

3.2 Results of Sorting Time between Quicksort and Merge Sort and Modified Quick Sort Algorithms

The time taken by three algorithms to sort data of different data type sizes and ether for sorted and unsorted arrays were documented. (Already sorted array of integer, unsorted (integer, character and double) data type which data sizes between 100 and 100000 were evaluated. The division is done to observe the true behavior of the three algorithms in the divided sections. Table (2) show the recorded time for both sorted and unsorted integer array:

Table2.recoded time for sorted and unsorted integer array

Array type	Number of inputs (n)	Quicksort (ms)	Merge sort (ms)	Modified Quick Sort (ms)
Already sorted integer array	100	0.0200	0.0770	0.0080
	200	2.0700	0.1540	0.0100
	500	7.4850	0.3050	0.0100
	1000	31.9520	0.6630	0.0130
	10000	----	4.6860	0.0400
	50000	-----	23.297	0.1580
	100000	-----	46.6900	0.3190
unsorted integer array	100	0.0290	0.0730	0.0250
	200	0.0790	0.1810	0.0540
	500	0.3080	0.3060	0.1910
	1000	0.6350	0.5330	0.2400
	10000	35.4000	4.9600	2.200
	50000	841.007	28.903	11.008
	100000	3278.697	50.7030	23.0880

Table (3) show the recorded time for both unsorted (character and double) array:

Array type	Number of inputs (n)	Quicksort (ms)	Merge sort (ms)	Modified Quick Sort (ms)
Unsorted character array	100	0.0320	0.0660	0.0200
	200	0.1010	0.1400	0.0430
	500	0.4760	0.3060	0.0870
	1000	2.1940	0.8090	0.1640
	10000	132.5910	4.963	1.953
	50000	3163.7420	23.328	7.711
	100000	12669.5020	48.238	17.487
Unsorted double array	100	0.0150	0.0770	0.0460
	200	0.0330	0.1200	0.0720
	500	0.0880	0.2970	0.1820
	1000	0.1800	0.5600	0.4290
	10000	2.6960	5.3500	5.0750
	50000	27.952	29.302	29.446
	100000	84.270	55.309	66.280

For already sorted integer array :Quicksort is Much slower than the other types, which is compatible with the complexity in the case of the already sorted array which is the worst case for quicksort which is equal to $O(n2)$.

While, the modified quicksort is much faster in this case, which is compatible with the complexity in this case which is the best case for modified quick sort which is equal to O(n).However, the merge sort is the intermediate state

between them because the complexity of it is the intermediate Compared to other sorts which is equal to O(nlogn)As shown in figure(5):

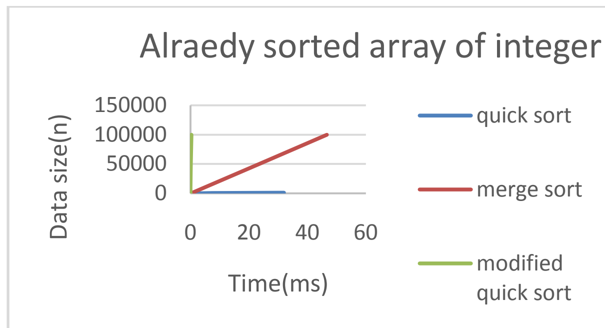

Fig 5:already sorted array of integers

For unsorted integer, character, and double array:Quicksort algorithm sorting time as shown in Figure (6), (7), and (8), revealed that quicksort is faster than merge sort for small data sizes elements while for large data size merge sort seems faster than quicksort. however, modified quicksort is the fastest for all data sizes.

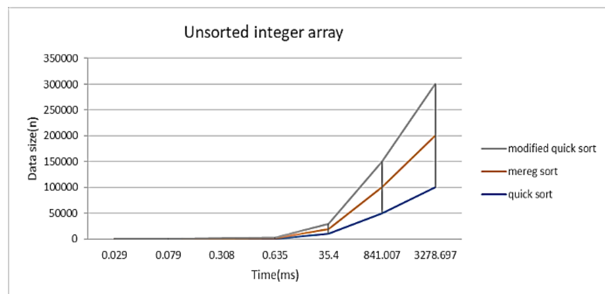

Fig 6:unsorted array of integers

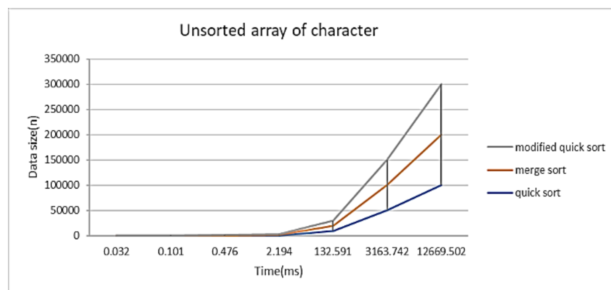

Fig 7:unsorted array of characters

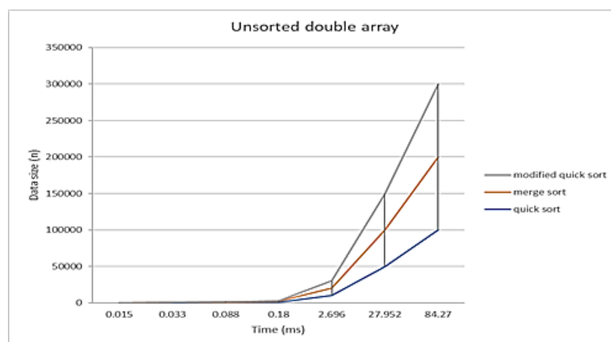

Fig 8:unsorted array of character

4. CONCLUSION

A comparative study of the three sorting algorithms that employ the divide and conquer technique is done in this study.

The analysis of the three algorithms was carried out based on System complexity (where the best, average, and worst cases were considered independent of the machine from the mathematical point of view), stability, internal versus external memory requirement, and computational complexity. Quicksort is faster than merge sort when the data size is small while merge sort is faster when the data size is large. However, Modified quick sort is the fastest for all data sizes. The Merge sort needs an additional memory space of O(n) for storing the extra array while modified quick sort and Quicksort need space of O(logn). If there is therefore the need to choose between Quicksort and Merge sort and modified quicksort algorithms for faster computation, Quicksort is preferred to Merge sort when the size of the data is small while Merge sort is recommended for data of large sizes, where there is the need to make use of cache locality, Modified Quicksort is preferred for all data sizes. It is believed that the information given in this paper will be of great value to programmers for choice modified quick sort in all time.

5. ACKNOWLEDGMENTS

The authors would like to thank Rula Al-Nusirat. EynasAjarmeh for their contributions during data collection phase.

6. REFERENCES

[1] Ashima. (n.d.). G. Implementation and Application of Bubble sort in D Array" International Journal for Scientific Research.

[2] Dalhoum1, A. l. (n.d.). Enhancing QuickSort Algorithm using a Dynamic Pivot Selection Technique.

[3] Kazim. (n.d.). A. A Comparative Study of Well Known Sorting Algorithms. International Journal of Advanced Research in Computer.

[4] Kumar, A. (n.d.). Merge Sort Algorithm.

[5] Mandeep. (n.d.). Why Quicksort is better than Mergesort? Retrieved May 14, 2019, from Geeksforgeeks: http:// www.geekforgeeks.org.

[6] Sorting and Efficient Searching. Lecture Note. Unpublished. 2008

[7] third international conference on computing and network communications (coconet'19) comparative study of two divide and conquer sorting algorithms: quicksort and mergesort.

[8] A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays Ahmed M. Aliyu, Dr. P. B. Zirra

[9] Optimizing Complexity of Quick Sort Md. Sabir Hossain Snaholata Mondal Rahma Simin Ali Mohammad Hasan.