iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 3 **Issue:** IV **Month of publication:** April 2015

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

Volume 3 Issue IV, April 2015
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

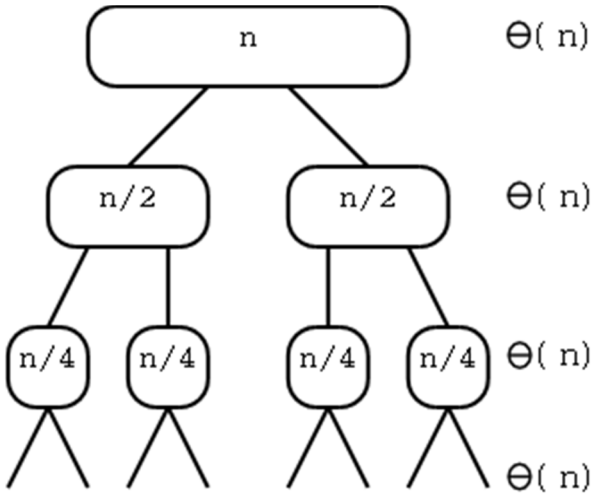
The Divide & Conquer Technique Used In the Analysis and Design of Algorithms

Shivangi Shandilya[1], Surekha Sangwan[2], Ritu Yadav[3]
[1,2,3]Dept. of Computer Science Engineering, Dronacharya College Of Engineering, Gurgaon

Abstract-Some sorting algorithms are simple and spontaneous, such as the bubble sort. Others, such as the quick sort are enormously complex, but produce superfast results. Some sorting algorithm work on less number of elements, some are suitable for floating point numbers, some are good for specific range, some sorting algorithms are used for huge number of data, and some are used if the list has repeated values. We sort data either in statistical order or lexicographical, sorting numerical value either in increasing order or decreasing order and alphabetical value. The complexity of algorithmic is generally written in a form known as Big – O (n) notation, where the O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against. The two groups of sorting algorithms are O(), which includes the bubble, insertion, selection, and shell sorts, and O(n log n) which includes the heap, merge, and quick sort. Since the advancement in computing, much of the research is done to solve the sorting problem, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. It is always very difficult to say that one sorting algorithm is better than another. Performance of various sorting algorithms depends upon the data being sorted.

I. INTRODUCTION

Sorting is used in many important applications and there have been a plenty of performance analyses.
Since the advancement in computing, much of the research is done to solve the sorting problem, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement.
Sorting is one of the most significant and well-studied subject area in computer science. Most of the first-class algorithms are known which offer various trade-offs in efficiency, simplicity, memory use, and other factors.



In the recent past, there has been a growing interest on enhancements to sorting algorithms that do not have an effect on their asymptotic complexity but rather tend to improve performance by enhancing data locality.
Sorting is an essential task that is performed by most computers. It is used commonly in a large variety of important applications. Database applications used by universities, banks, and other institutions all contain sorting code. Due to the importance of sorting in these applications, quite a large number of sorting algorithms have been developed over the decades with varying complexity.

Some of the time consuming sorting methods, for example bubble sort, Design and Analysis of Optimized Selection

©IJRASET 2015: All Rights are Reserved

Sort Algorithm insertion sort, and selection sort have a hypothetical complexity of O( ). Although these algorithms are very slow for sorting larger amount of data, yet these algorithms are simple, so they are not useless. If an application only needs to sort smaller amount of data, then it is suitable to use one of the simple slow sorting algorithms as opposed to a faster, but more complicated sorting algorithm.

A. Region Expressions

Each region expression is represented in the form [l; u], where l is the lower bound and u is the upper bound. Both l and u are symbolic expressions in terms of the reference set of the currently analyzed procedure. These expressions are of the form p + exp, where p is a pointer into the accessed allocation block and exp is an integer expression representing the pointer of set.

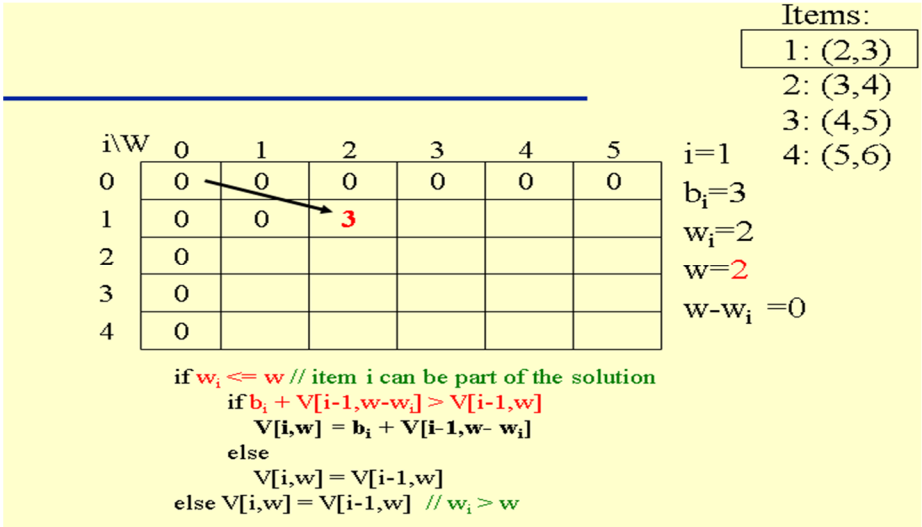
If the region expression summarizes how a statement or procedure reads data, it is marked as a read expression; if it summarizes how a statement or procedure writes data, it is marked as a write expression. It is important to realize that each region expression identies a region of memory within a specific set of al location blocks. The pointer analysis determines the set of allocation blocks for each region expression. So even if the symbolic analysis is unable to generate symbolic bounds for a region expression in terms of the reference set, the region expression does not denote all of memory. It merely denotes all of the memory in the allocation blocks that the pointer analysis extracted for that region expression.

B. Intraprocedural Region Analysis

The intraprocedural region analysis generates the local re- gion set of the procedure, or a minimal set of region ex- pressions that characterize how the procedure accesses data.

The local region set is expressed in terms of the procedure's reference set. The analysis starts by using the results of the bounds analysis to extract a region expression for each pointer dereference in the procedure. At each pointer dereference, it uses the order information (as translated by the initial value analysis into the reference set of the procedure) to obtain upper and lower bounds for the region that the dereference accesses. Together, these bounds make up the region expression for that dereference. The special bound 1 is used as the lower bound for dereferences with nolower bound from the bounds analysis; +1 is used as the upper bound for dereferences with no upper bound from the bounds analysis.

The local region set is initialized to the union of the region expressions from all of the pointer dereferences in the procedure. An iterative algorithm repeatedly ends in two region expressions in this set whose upper and lower bounds are either adjacent or overlap. It then merges the regions into a new region as follows. The lower bound of the new region is the minimum of the lower bounds of the original regions, and the upper bound is the maximum of the upper bounds of the original regions. The original regions are re- moved from the local region set, the new region is inserted into the set, and the algorithm iterates until there are no more adjacent or overlapping regions.



This algorithm assumes that the analysis can compare the upper and lower bounds of region expressions and generate the minimum and maximum of two bounds. These bounds are symbolic expressions in the reference set of enclosing procedure. During the

©IJRASET 2015: All Rights are Reserved

bounds analysis, each bounds expression is transformed into a sum of terms and each term is a product of a coefficient and a variable from the reference set.

The algorithm compares two such expressions by comparing corresponding terms: one expression is larger than another if all of its terms are larger. The algorithm compares terms by comparing their coefficients. If the variable from the reference set is positive or zero, the compiler chooses the term with the larger coefficient as the larger term. If the variable is negative, the term with the larger coefficient is the smaller term. This approach requires that the compiler know the sign of the integer variables in the reference set. The cur- rent implementation relies on the programmer to declare all such variables as C unsigned variables, which forces them to be non-negative. It would be straightforward to implement an inter-procedural abstract analysis to determine the sign of variables in the reference set.

The algorithm that computes the minimum and maxi- mum of two bounds expressions also operates at the granularity of terms. The minimum of two bounds expressions is the sum, over all pairs of corresponding terms, of the smaller term in the pair; the maximum is the sum of the larger terms in corresponding pairs.

For the example in Figure 1, the local region sets for main and sort are empty. The local region set for merge reads

[l10; h101] and [l20; h201] and writes [d0; d0 + (h10l10)+h20 l201].

The local region set for insertion sort reads and writes [l10; h101].

C. Interprocedural Region Analysis

For each procedure, the interprocedural region expression analysis uses the local region sets to derive a global region set, or a minimal set of region expressions that characterize how the entire execution of the procedure accesses data.

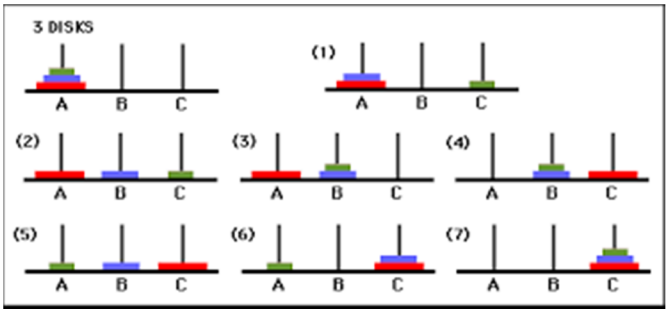
1) *Non-Recursive Procedures:* For non-recursive procedures, the analysis extracts the global region sets by propagating region sets up from the leaves of the call graph towards the root. For each procedure, the global region set is initialized to the procedure's local region set. At each propagation step, the analysis performs a symbolic un-mapping as follows. It first translates the region expressions from the reference set of the caller to expressions in the variables of the caller. It then translates the expressions from the variables of the caller to expressions in the reference set of the caller. The resulting expressions are added to the current global region set of the caller, with adjacent or overlapping regions. The global region set for merge contains the read region expression [l10 ; h101]. The analysisfirst un-maps this region expression into the variables of the sort procedure to obtain the region expression [d1; d21].It then uses the bounds analysis information to obtain the lower bound d0 for d1 and the upper bound d0 +n0=41 for d21. Note that both bounds expressions are in terms of the reference set for sort. The compiler combines these bounds to obtain the region expression [d0; d0 + n0=41], which is the translation of the original region expression from the global region set of merge into a region expression that characterizes,in part, how a specific call to o merge accesses data.
2) *Recursive Procedures:* The analysis uses a fixed point algorithm to handle recursive procedures. For each recursive procedure, the analysis initially sets the procedure's global region set to its local region set. It then applies the bottom-up symbolic un-mapping algorithm described above to propagate region expressions from callees to callers. It terminates the recursion by using the procedure's current global region set in the unmapping as an approximation of its actual global region set. When- ever possible, the unmapped region expressions are coalesced into the current global region set of the caller. The analysis uses the coalescing algorithm. The algorithm continues until it reaches a fixed point. In some cases, this analysis generates an unbounded number of region expressions that cannot be coalesced. This may happen, for instance, when the recursive function accesses a statically unbounded number of disjoint regions. In this case, the analysis as described above will not terminate. Even if the analysis is always able to coalesce the region expressions from recursive calls into the current global region set, the analysis as described above may not terminate if the bounds always increase or decrease. The compiler therefore imposes a finite bound on the number of analysis iterations. If the analysis fails to con- verge within this bound, we replace the extracted region expressions with corresponding region expressions that identify the entire allocation blocks as potentially accessed. The global region set for sort starts out empty. The interprocedural region analysis for non-recursive procedures propagates the region expressions from the calls to insertion sort and merge into the sort procedure, and coalesces the resulting region expressions to add the read region [d0; d0 + n01] and the write region [t0 ; t0 + n01] to the global region set for sort. The

©IJRASET 2015: All Rights are Reserved

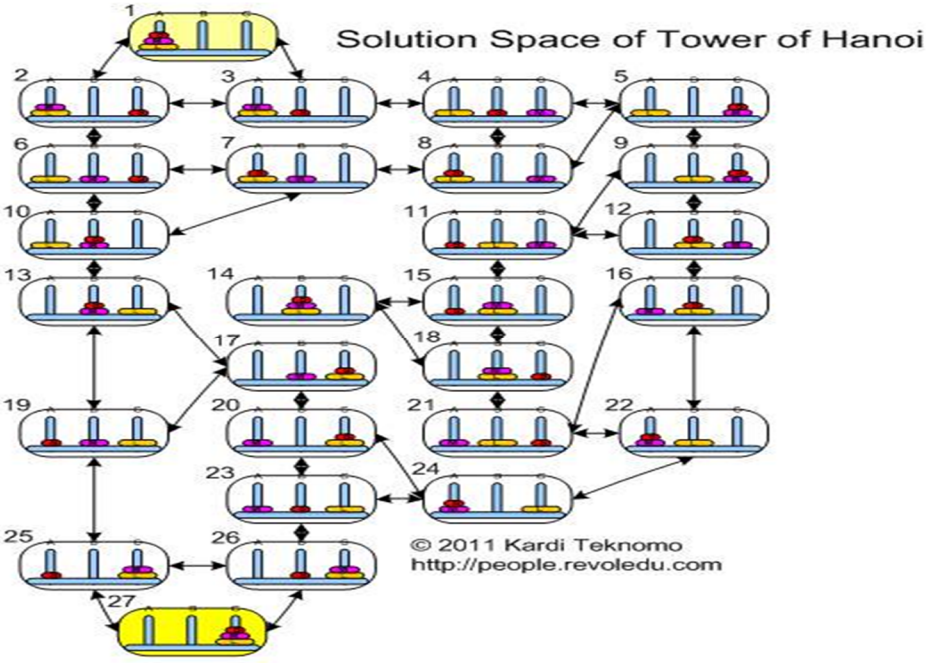
interprocedural region analysis for recursive procedures uses this global region set as an approximation to derive re- gion expressions that characterize how the recursive calls to sort access data. After coalescing these region expressions back into the current global region set for sort, the analysis reaches a fixed point.

II. ADVANTAGES OF DIVIDE & CONQUER

The first, and probably most recognizable benefit of the divide and conquer paradigm is the fact that it allows us to solve difficult and often impossible looking problems such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it. However, with the divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.



Another advantage to this paradigm is that it often plays a part in finding other efficient algorithms, and in fact it was the central role in finding the quick sort and merge sort algorithms. It also uses memory caches effectively.



The reason for this is the fact that when the sub problems become simple enough, they can be solved within a cache, without having to access the slower main memory, which saves time and makes the algorithm more efficient. And in some cases, it can even produce more precise outcomes in computations with rounded arithmetic than iterative methods would.

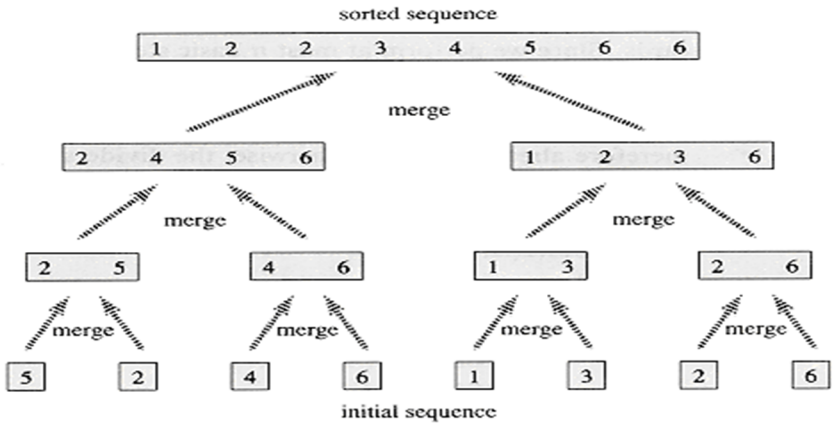
III. LIMITATIONS OF DIVIDE & CONQUER

©IJRASET 2015: All Rights are Reserved

One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process. Another concern with it is the fact that sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together. Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization. And the last recognizable implementation issue is that these algorithms can be carried out by a non-recursive program that will store the different sub problems in things called explicit stacks, which gives more freedom in deciding just which order the sub problems should be solved. These implementation issues do not make this process a bad decision when it comes to solving difficult problems, but rather this paradigm is the basis of many frequently used algorithms.

IV. CONCLUSION

The above technique enables us to solve difficult problems, helps to find other efficient algorithms, and is sometimes the better choice than an iterative approach would be. One of the only downfalls to this paradigm is the fact that recursion is slow, and this fact alone can almost negate all the other advantages of this process. Merge Sort and Quick Sort are both sorting algorithms that are based on this process of the divide and conquer paradigm. Other examples that are based on this process are matrix multiplication, pair wise summation, and the fast Fourier transform Once a closer look is taken at the running times of these different divide and conquer algorithms, and the paradigm in general, you will see that this paradigm's running time will beat out many of the other algorithm running times proving to be the most time efficient and the most practical choice.

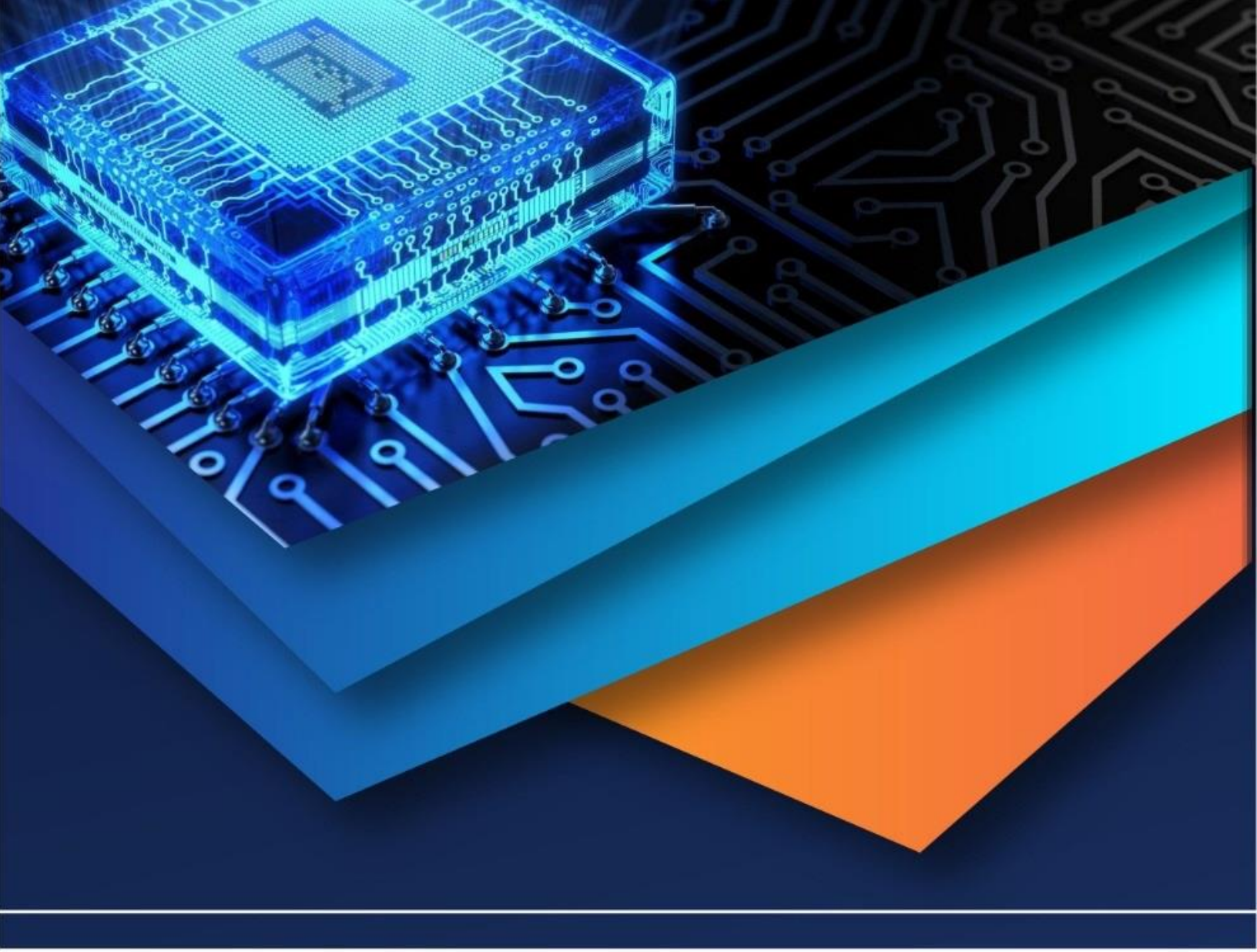


In reality it just makes everything in life simpler if you try and apply the process of divide and conquer. So in other words, wouldn't it only seem logical if you were to use the divide and conquer algorithm in order to make all of the complex problems in mathematics that much simpler and easier to solve? Overall, the point is simple. The divide and conquer algorithm design paradigm is a definite advantage to the world of mathematics and in this way, is a very good concept to know about and understand.

REFERENCES

[1]Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (MIT Press, 2000).
[2]Brassard, G. and Bratley, P. Fundamental of Algorithmics, Prentice-Hall, 1996.
[3]Radu Rugina and Martin Rinard, "Recursion unrolling for divide and conquer programs," in *Languages and Compilers for Parallel Computing*, chapter 3, pp. 34–48. *Lecture Notes in Computer Science* vol. 2017 (Berlin: Springer, 2001)
[4]Amato, Nancy. Sorting. slide 25 and 40 (2008) http://parasol.cs.tamu.edu/~amato/Courses/221/lectures/Ch10.Sorting.pdf
[5] www.philadelphia.edu.jo/academics/.../part-two(Divide-Conquer **)**

©IJRASET 2015: All Rights are Reserved



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)