Algorithm Complexity Analysis of Merge Sort Variant

Shandy 13516097
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
shandy.gunawan@rocketmail.com

***Abstract*—Sorting algorithm is one of the most popular algorithm in computer science. There are a lot of algorithms to achieve a same result, that is a sorted array. This paper will take a look at merge sort algorithm as one of the most popular and fastest sorting algorithm and its variation based on the algorithm complexity.**

***Keywords*—sorting, algorithm complexity, merge sort, variation.**

I. Introduction

Sorting algorithm is one of the most important algorithm in human life and have been used since a long time ago. An example of the usage of a sorting algorithm is the librarian's step to sort books alphabetically.

The usage of sorting algorithm has been increasing since the end of the 20th century, along with the development of information technology. The introduction of internet, gadget, along with the increasing number of implementation of information technology among institutions and companies has played a large part of the increase in the amount of information and data available around the world. Here is a graph showing the explosion of digital information since 2005 [1].

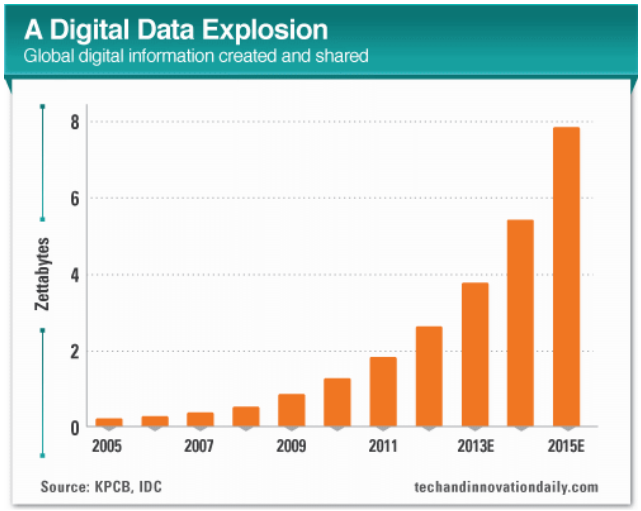

Figure 1.1 Bar graph of Digital Data Explosion

Let us remind ourselves that 1 Zettabytes (ZB) is equal to 10^{15} Megabytes (MB). If we take the average size of MP3 music file, which is approximately 3.5 MB [2]. The difference size of digital information between 2005 and 2015, which is a roughly 8 ZB, is enough to contain 2.28×10^{15} MP3 files. To put it in scale, Apple iTunes' music collection contains[3] 43 million songs or 4.3×10^{7} songs, which is just 0.0000000188% of 2.28×10^{15}.

The big amount of digital data stored and used by companies and institutions makes it difficult for programmers to handle the data manually or with a heuristic approach because the drawback in performance and time will be very significant.

This is one of many reasons that encourages programmers and algorithm developers to improve the existing algorithms and methodologies that handle big data management, sorting algorithm being one of them.

Today, there are a lot of sorting algorithms used by programmers to complete the task. Examples of some popular sorting algorithms are bubble sort, insertion sort, merge sort, radix sort, shell short, quick sort, and bogo sort. Each of these algorithms uses a very different technique to sort processed data. Bubble sort bubbles it's biggest element to the top or bottom of the array, quick sort picks a pivot in the middle of the array and place corresponding elements to the left or right of the pivot, etc. Each of these algorithms has its own advantages and disadvantages. One algorithm can be better than other algorithms in specific cases.

To compare these algorithms, we use a technique called algorithm complexity analysis. A complexity of an algorithm is determined by its time complexity and memory complexity. Time complexity, noted by T(n) with n as the amount of data processed, shows how fast the algorithm will be processed from the start to finish. A good algorithm has a low time complexity which shows the algorithm uses little time to finish the job. Memory complexity, noted by S(n), shows the amount of memory used to run the algorithm, this algorithm takes the data structure being used by the algorithm into account. A good algorithm has a low memory complexity which shows the algorithm uses little memory to be executed. Here is a graph showing how each algorithm performs based on time complexity [3].

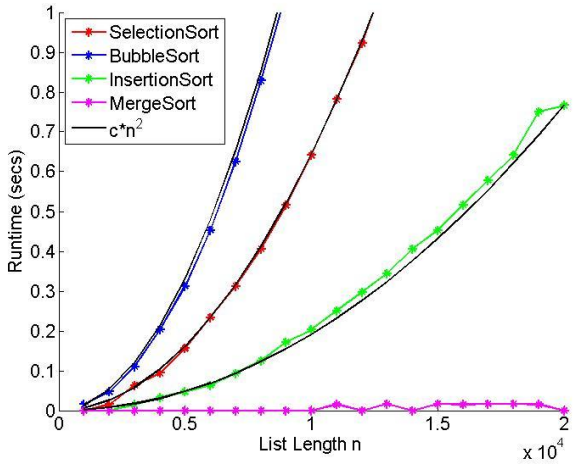

Figure 1.2 Comparison of Sorting Algorithms

The above graph shows that merge sort has a very consistent low runtime, which means it has a low time complexity compared to other sorting algorithms. This graph encouraged the writer to research more about merge sort and found that merge sort has two variant of sorting techniques. These techniques are called bottom up merge sort and top down merge sort. Then, the writer tried to compare these techniques of merge sort using the algorithm complexity analysis.

II. THEORY

A. Definition of Sorting

The sorting word itself derived from the basic root word of sort. According to Oxford online dictionaries[4], the word 'sort' in the computing aspect is defined as the arrangement of data in a prescribed sequence. In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. Numerical order and lexicographical order are usually used as the base for sorting. A sorting algorithm is important for other algorithms to work correctly. For example, binary search, which is more efficient than sequential search, can only work in a sorted database.

The sorting problem, despite its simple requirement and outcome, has become one of the most researched problem in computer science. Researchers and programmers have been trying to develop the sorting algorithm to become more efficient both in time and memory.

B. Algorithm Complexity

Algorithm, according to Rosen, is a finite sequence of precise instructions for performing a computation or for solving a problem. An algorithm can be analyzed mathematically by using the algorithm complexity, which is time complexity and memory complexity.

Time complexity ($T_{(n)}$) defines the steps required for an algorithm to accomplish a specific task with *n* number of data. Good algorithms have a lower time complexity compared to other algorithms to accomplish similar task, which means, the algorithm requires fewer steps to complete the task. Time complexity usually written as a polynomial.

Memory complexity ($S_{(n)}$) defines the memory required for an algorithm to work correctly with *n* number of data. The memory complexity of an algorithm is affected by the data structure being used in the algorithm. Good algorithms have a lower memory complexity compared to other algorithms to accomplish similar task, which means, the algorithm requires lesser memories to work correctly.

C. Definition of Big O Notation

Big-*O* notation has been used in mathematics for more than a century with Paul Bachmann, a German mathematician, introduced it in 1892. Big-*O* notation widely used in the analysis of algorithms in computer science.

A big O notation is defined as the following statement [6]:

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that f(x) is O(g(x)) if there are constants C and k such that.

$$|f(x)| \leq C|g(x)|$$

whenever x > k. [This is read as "f(x) is big-oh of g(x)."].

The constants *C* and *k* in the definition of big-*O* notation are called **witnesses** to the relationship *f(x)* is *O(g(x))*.

D. Application of Big-O Notation in Time Complexity

The definition of Big-O notation can be tweaked to fit the context of time complexity [7].

Let T(n) = O(f(n)) (This is read as "T(n) is O(f(n))" which means T(n) has f(n) as the biggest order) if there exist the constants C and n_o such that

$$T(n) \leq C(f(n))$$

for $n \geq n_o$.

f(n) is the upper bound of *T(n)* for a very big *n*.

Theorem in Big-*O* notation:

1. Exponential dominates any power ($y^n > n^p$, y>1).
2. Power dominates ln *n* ($n^p > \ln n$).
3. All logarithms grow in the same rate ($a \log(n) = b \log(n)$).
4. *n* log *n* grows faster than *n* but slower than n^2.

Example of *T(n)* conversion to big-*O* notation:

- $T(n) = 2^n + 2n^2 = O(2^n)$.
- $T(n) = 2n \log(n) + 3n = O(n \log(n))$.
- $T(n) = \log(n^3) = 3 \log(n) = O(\log(n))$.
- $T(n) = 2n \log(n) + 3n^2 = O(n^2)$.

E. Classification of Algorithm Based on Big-O Notation

Big-*O* notation can be used to classify algorithms based on their complexity. Below is the table showing the classification of algorithm [7].

Classification	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	*n log n*
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Table 2.1 Big-*O* Classification of Algorithm

The order sequence of the classification from the fastest to the slowest is.
$O(1)<O(\log n)<O(n)<O(n \log n)<O(n^2)<O(n^3)<O(2^n)<O(n!)$.

F. Classification of Sorting Algorithm

Sorting algorithms can be classified by [6]:

- Computational Complexity.
  Sorting algorithms can be classified based on its performances in best, average, and worst scenario. Best case scenario for comparison-based sorting algorithms is O(nlogn) and the worst case is $O(n^2)$.
- Memory usage
  Some sorting algorithms are called "In-Place" algorithm. In In-Place algorithm, no additional data structure required for sorting.
- Recursion
  Some sorting algorithms use recursive method, some use non-recursive method, and some use both.
- Stability
  The stability of sorting algorithm is determined by the technique used by the algorithm. A stable algorithm sorts identical elements in the same order that they appear in the input.
- Adaptability
  An adaptive algorithm changes its complexity based on pre-sortedness of the input. A non-adaptive algorithm does not change its complexity regardless of the input.

G. Types of Sorting Algorithm

The curiosity of the researchers to optimize sorting algorithm resulted in many types of sorting algorithms. There are bubble sort, selection sort, insertion sort, counting sort, quick sort, and merge sort.

- Bubble Sort
  Bubble sort starts at the beginning of the data set. Then, it compares the first two elements, and if the first is greater than the second, it swaps them. This step will continue for each pair until the end of data set. Then, the algorithm starts again from the first two elements, repeating until there is no swap could be done so it is not efficient to sort a large data set. The average and worst case for bubble sort is $O(n^2)$.
- Selection Sort
  Selection sort finds the minimum value of the data set, then swaps it with the value in the first position. This step will be repeated for the remainder of the list. Selection sort has $O(n^2)$ complexity, like bubble sort, is not efficient to sort a large data set.
- Insertion Sort
  Insertion sort works by taking elements from the list one by one and inserting them in correct position into a new sorted list. Insertion operation is expensive so this algorithm is not preferred for a large data set.
- Counting Sort
  Counting sort only works if each input is known to belong to a particular set *S*. It works by creating an integer array of size $|S|$ and using the *i*th bin to count the occurrences of the *i*th member of *S* in the input. Counting sort have $O(|S|+n)$ time complexity and $O(|S|)$ space complexity where *n* is the length of the input.
- Quick Sort
  Quick sort works by choosing a pivot in the middle of the data set. Then, move lesser elements of data set in the left of the pivot and the greater elements in the right. Then, the lesser and greater sublists are recursively sorted. Quick sort works in $O(n \log n)$.
- Merge Sort
  Merge sort will be discussed further in later chapters.

H. Introduction to Divide and Conquer Algorithm.

Divide and conquer algorithm divides a problem into smaller sub-problems and then each problem is solved independently. The sub-problems will keep divided into small sub-problems until the stage where no more possible division is reached. Those "atomic" smallest possible sub-problem are solved and merged with other solution of atomic sub-problems in order to obtain the solution of an original problem [8].

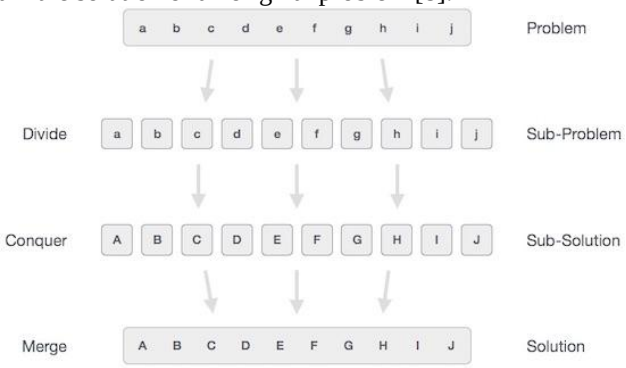

Figure 2.1 Illustration of Divide and Conquer Algorithm

The divide and conquer algorithm can be divided into a three-step process.

1. Divide/Break
   This step involves breaking the problem into smaller sub-problems. This step usually takes a recursive approach to divide the problem until atomic sub-problem is reached.
2. Conquer/Solve
   This step receives a lot of smaller sub-problems to be solved and solve them with intended operation.
3. Merge/Combine
   When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem.

I. Introduction to Merge Sort

Merge sort was invented by John Von Neumann in 1945 and has become one of the most popular sorting algorithm. Merge sort based on divide and conquer algorithm and is a comparison-based sorting algorithm. Merge sort use a recursive method and its implementation produces a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. The best, worst, and average scenario for

merge sort is O(n log(n)), which means that merge sort is not an adaptive algorithm. Memory wise, worst case for merge sort is O(n). In real life implementation, there are at least two major variations for merge sort, which is bottom-up merge sort and top-down merge sort.

J. Algorithm of Merge Sort

Conceptually, merge sort divides a list into two halves or two sub-lists until it can no more be divided (consists only one element). Then, the algorithm will merge the sub-list into a sorted list by traversing both of the sub-lists and insert the lower valued element into the list. The algorithm of merge sort can be expressed by these steps:

1. If it is only one element in the list, it is already sorted, return.
2. Divide the list recursively into two halves until it can no more be divided.
3. Merge the smaller lists into new list in sorted order

III. ALGORITHM COMPLEXITY ANALYSIS OF MERGE SORT VARIANT

A. Algorithm of Merge Sort Variant

In real life implementation, there are two major variations of merge sort. They are top-down merge sort and bottom-up merge sort.

The most common example of merge sort is the top-down merge sort, as this method also used as the example in the previous chapter. The top-down merge sort uses recursion. It starts by splitting the list into two, make the recursive calls, and merge the results. To elaborate, the writer will divide the top-down merge sort into two parts. The first part is the divide part, the algorithm will keep splitting the list into two halves until each of the sub-lists consist only one element from the initial list. The second part is the conquer part, the algorithm will merge the sub-lists, starting from the sub-lists with one element, and continue to merge the merged lists to form a new sorted list until all the elements of the initial list sorted. The top-down merge sort can be illustrated as below.

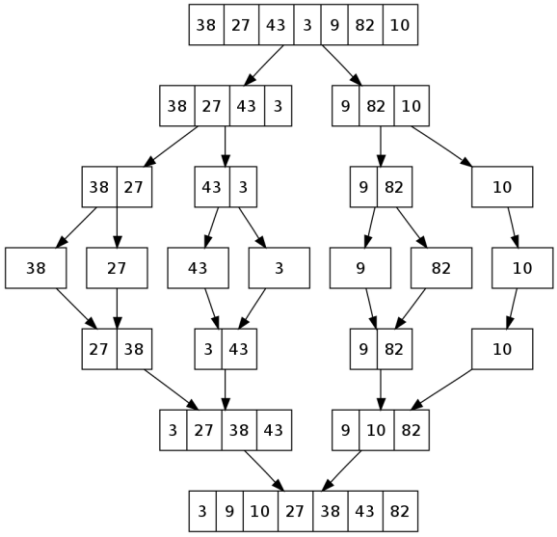

Figure 3.1 Visualization of Top-Down Merge Sort

Bottom-up merge sort also uses recursion. It directly starts at the sub-lists consists of one element and proceed by iterating over the pieces and merging them. To elaborate, the algorithm starts by comparing the elements 1-by-1 and merges them, then move to the next pair, this step will keep repeating until the end of the list, then the algorithm makes the recursive call and start comparing the pairs that have been merged from the previous algorithm call. These steps will continue until all the elements of the list sorted. In other words, bottom-up merge sort is almost identical with top-down merge sort but without the splitting part. The bottom-up merge sort can be visualized below.

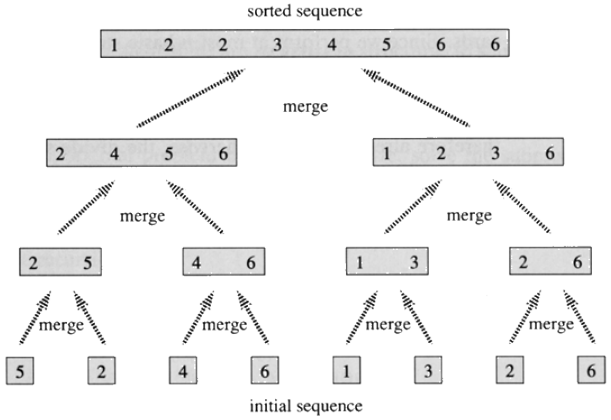

Figure 3.2 Visualization of Bottom-Up Merge Sort

B. Merge Sort Variant in C-like Language

The two variations of merge sort can be implemented in C language as shown below.

The C-like example of Top-Down merge sort:

```
// Array A has the items to sort; array B is a work array.
TopDownMergeSort(A[], B[], n)
{
  CopyArray(A, 0, n, B);      // duplicate array A into B
  TopDownSplitMerge(B, 0, n, A); // sort data from B into A
}


// Sort the given run of array A using array B as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
   if(iEnd - iBegin < 2)   // if run size == 1
      return;              //   consider it sorted

   // split the run longer than 1 item into halves
   iMiddle = (iEnd + iBegin) / 2;     // iMiddle = mid point

   // recursively sort both runs from array A into B
   TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left run
   TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run

   // merge the resulting runs from array B[] into A[]
   TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}
```

```
// Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is            B[ iBegin:iEnd-1   ].
TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
   i = iBegin, j = iMiddle;
   // While there are elements in the left or right runs...
   for (k = iBegin; k < iEnd; k++) {
      // If left run head exists and is <= existing right run head.
      if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
         B[k] = A[i];
         i = i + 1;
      } else {
         B[k] = A[j];
         j = j + 1;
      }
   }
}

CopyArray(A[], iBegin, iEnd, B[])
{
   for(k = iBegin; k < iEnd; k++)
      B[k] = A[k];
}
```

The C-like example of bottom-up merge sort:

```
// array A has the items to sort; array B is a work array
void BottomUpMergeSort(A[], B[], n)
{
   // Each 1-element run in A is already "sorted".
   // Make successively longer sorted runs of length 2, 4, 8, 16...
until whole array is sorted.
   for (width = 1; width < n; width = 2 * width)
   {
      // Array A is full of runs of length width.
      for (i = 0; i < n; i = i + 2 * width)
      {
          //Merge two runs:
          //A[i:i+width-1] and A[i+width:i+2*width-1] to B[]
         // or copy A[i:n-1] to B[] ( if(i+width >= n) )
         BottomUpMerge(A, i, min(i+width, n), min(i+2*width,
n), B);
      }
      // Now work array B is full of runs of length 2*width.
      // Copy array B to array A for next iteration.
      // A more efficient implementation would swap the roles
      // of A and B.
      CopyArray(B, A, n);
      // Now array A is full of runs of length 2*width.
   }
}

// Left run is A[iLeft :iRight-1].
// Right run is A[iRight:iEnd-1  ].
void BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
   i = iLeft, j = iRight;
   // While there are elements in the left or right runs...
   for (k = iLeft; k < iEnd; k++) {
      // If left run head exists and is <= existing right run head.
      if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
         B[k] = A[i];
         i = i + 1;
      } else {
         B[k] = A[j];
         j = j + 1;
      }
   }
}

void CopyArray(B[], A[], n)
{
   for(i = 0; i < n; i++)
      A[i] = B[i];
}
```

C. Time Complexity Analysis on Merge Sort Variant

The usual typical operation for most comparison-based sorting algorithm is the comparison and swap operation. Merge sort is no difference. In both of the merge sort variations, the typical operation is the comparison of elements in sub-lists that are going to be inserted into a new sorted list. The most typical operation in both variations can be seen from the clipped illustration below.

```
for (k= iLeft; k < iEnd; k++) {
   if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
      B[k] = A[i];
      i = i + 1;
   } else {
      B[k] = A[j];
      j = j + 1;
   }
}
```

To elaborate the above illustration, A is the working array and B is the result array. Variable *k* will be the index of B, variable i will be the iterative index for the left half of A and variable j for the right half. The for loop will iterate all the elements in array A. Then the if condition will check if variable i is less than the starting index of the right half (iRight) and another check if variable j is bigger than the last index of the array A (iEnd) or the left half's element is less or equal to the right half's element. If those conditions are true then the algorithm will insert the left half's element into B and if the conditions fail, the algorithm will insert the right half's element into B.

This operation takes *n* time with *n* is the length of the array. Each time the algorithm divides the array into halves, the operation will take half the previous time but doubling in number of operation performed. The doubling and halving cancel each other so the operation time for each sub-array remains constant *n*. The time complexity requires cn(log(n)+1) with *c* is a constant coefficient. The big-O notation for cn(log(n)+1) is *O*(nlog(n)).

In the pseudocode for each of the merge sort variation, both of them use the same operation to merge sub-arrays into an

array, which means, that both of the algorithms have the time complexity of *O*(nlog(n)).

D. Space Complexity Analysis on Merge Sort Variant

Both of the variations have the same time complexity which is *O*(n log(n)), but the same statement can also be said for their space complexity. Both algorithms have *O*(n) for their space complexity since merge sort is not an In-Place sorting algorithm. It makes a copy of the entire array being sorted. From the example, we can see that there are two arrays, A and B that is used in the algorithm with A is the array with the initial unsorted items and B is the working array. Researchers have been trying to optimize merge sort into an In-Place sorting algorithm but the results still have not satisfactory and sometimes too complicated to use.

IV. CONCLUSION

The merge sort is one of the most popular sorting algorithm in computer science. It has two major variations in its implementation. They are called top-down merge sort and bottom-up merge sort. Both algorithms have the same time complexity (O(nlog(n)) and space complexity (O(n)) despite having a different technique to sort array elements.

VI. APPENDIX

1. Algorithm : a finite sequence of precise instructions for performing a computation or for solving a problem
2. Array : a data structure that contains a group of elements in sequence with the same data type.

VII. ACKNOWLEDGMENT

All my deepest gratitude to Ir. Rinaldi Munir, M.T. as my lecturer in IF2130 – Matematika Diskrit course and giving such this assignment to develop my understanding of discrete mathematics.

Also my special thanks for all of my family members and friends who have supported me in the process of creating this paper.

REFERENCES

[1] http://www.blackcloudanalytics.com/news/who-needs-big-data/ (accessed November 29th, 2017 20:20).
[2] https://www.apple.com/lae/itunes/music/ (accessed November 29th, 2017 20:30).
[3] http://www.cs.toronto.edu/~jepson/csc148/2007F/notes/sorting.html (accessed November 29th, 2017 21:13).
[4] https://en.oxforddictionaries.com/definition/sort (accessed November 29th, 2017 21:18).
[5] Rosen, Kenneth H, *Discrete Mathematics and Its Applications 7th Edition.* Monmouth University. New York:McGraw-Hill, 2011, pp. 205.
[6] condor.depaul.edu/ichu/csc383/notes/notes2/sorting.pdf (accessed November 30th, 2017, 18:30).
[7] Munir, Rinaldi, *MATEMATIKA DISKRIT Revisi Kelima*. Bandung: Penerbit Informatika. 2012.
[8] https://www.tutorialspoint.com/data_structures_algorithms/divide_and_conquer.htm (accessed December 2nd, 2017 23:33)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017

Shandy - 13516097