RESEARCH REPORT

TOPIC: The Quicksort Algorithm: History and the Idea

CONTENTS

SUMMARY OF RESEARCH REPORT:

- Quicksort, a method which involves use of divide and conquer procedure to sort a given list.
- It has three different cases depending on running time which include:
    1) Best Case
    2) Worst Case
    3) Average Case
- History of Quicksort from discovery to modern day ramifications.
- Quicksort has evolved over time and Dual Pivot Selection Technique is efficient form of Quicksort, this method cuts the number of steps taken to sort the list which in return reduces it's running time making the algorithm more efficient.
- Conclusion is drawn from the content and research done about Quicksort Algorithm and related topics.

The content for this research report was collected by reading various books, journals, and articles containing knowledge about Quicksort algorithm and few videos from online sources were also very helpful for clear understanding of concept of this algorithm.

The author of this report also deployed his own understanding by uploading hand written version of algorithm in this report.

INTRODUCTION :

This paper gives an overview about quicksort algorithm, it's history, algorithm and about the future of algorithm. Sorting is one of the most used algorithm in modern day computing to deal with huge chunks of data and quicksort is one of the method of sorting which is widely used of them all.

In this paper, general idea of sorting is given along with detailed research on Quicksort algorithm which includes codes, screenshots of algorithms, analysis of algorithm and few other aspects of Quicksort. So, let's begin and understand everything about the topic from basic to advanced level.

SORTING

The word sort is defined as 'to put a number of things in an order or to separate them into groups' (Cambridge Dictionary).

For example:

<u>INPUT:</u>

J = [x1, x2, x3, x4, ………………., xN]

Here we have the input in form of an array which has N number of elements in integer format.

<u>OUTPUT:</u>

In ascending order:

J = [a1, a2, a3, a4, ……………….., aN-1, aN], where [a1<a2, a2<a3, a3<a4,…………………,aN-1<aN]

In descending order:

J = [b1, b2, b3, b4, ………………., bN-1, bN] where [b1>b2, b2>b3, b3>b4, …………………., bN-1>bN] (Cormen et al., 2009).

QUICKSORT

Quicksort is a method of sorting which works on Divide and Conquer principle and has different running time for worst and best case.

In worst case it is has running time of (O n^2), which is similar to the running time of Insertion sort and in best case quicksort has running time same as than of Merge sort that is (O *n* lg(*n*)) (Cormen et al., 2009).

History of Quicksort:

The method of Quicksort was developed by British computer scientist Tony Hoare (Turing award recipient) in year 1959, while on his visit as a visiting student to Moscow State University in Russia. He worked on a Machine translation project and while working on this project he had to sort some Russian words and instead of resorting to a known method of Insertion sort (a slow algorithm to sort a list), he developed a different and more efficient method known as Quicksort. Although he was only able to write code for partitioning (first step for Quicksort) while in Russia but was able to complete the code when he returned back to England and learned about recursion. Finally, his code was published in premier computer science journal known as Communications of the Association for Computer Machinery.

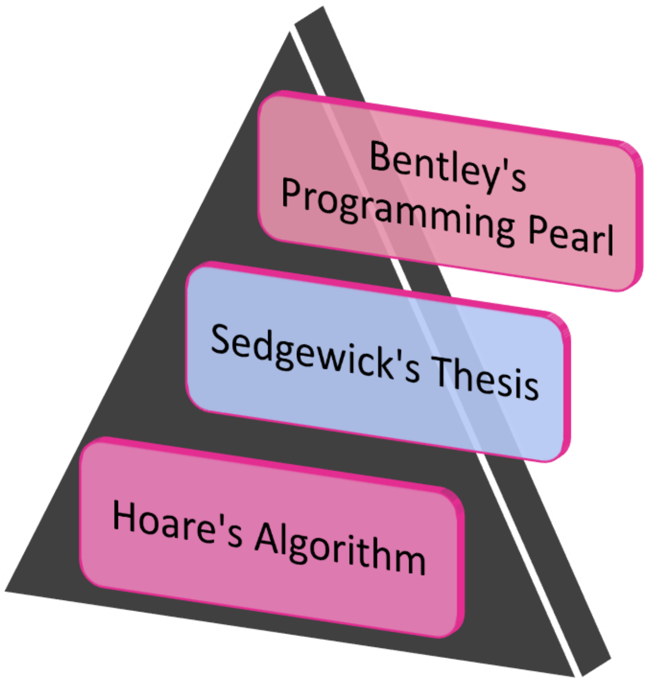
In year 1975, Robert Sedgewick in his Ph.D. thesis resolved many problems regarding Pivot selection while partitioning step and also worked on derivation on predicting number of comparisons and swaps while sorting the list.

Afterwards, Jon Bentley in his book Programming Pearls familiarized the readers with Lomuto’s partition scheme which according to him is much simpler than Hoare’s partitioning scheme.

Lately in year 2009, a dual pivot Quicksort algorithm was proposed by Vladimir Yaroslavskiy.

This algorithm after substantial actual performance tests has been chosen as new default sorting in Oracle’s Java 7 runtime library.

SUMMARIZING HISTORY OF QUICKSORT



All in all, Quicksort has evolved since the time it was developed by Tony Hoare in year 1959 with latest improvement done in year 2009. Since it is one of the most preferred way to sort complex data structures, one can say that this algorithm still has a lot of scope for improvement in upcoming years (Cormen et al., 2009).

ALGORITHM:

1) Dividing /Partitioning the Array:

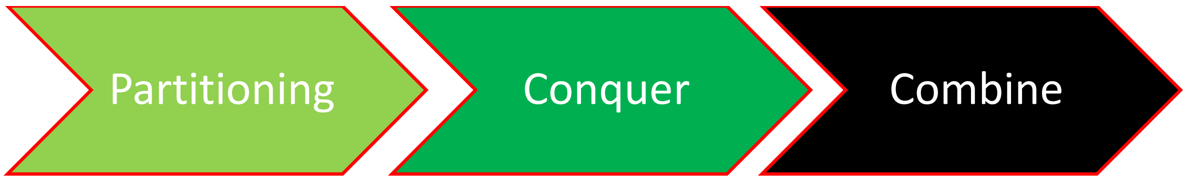
The first step is to divide array into two sub-arrays and for this step we use select an item from array itself which is called pivot. After selecting pivot array is divided into three sub-blocks such that elements less than pivot are put in left sub block, whereas elements greater than pivot are put in right sub block and the pivot is placed in middle sub block (Kaushal, 2017).

Note: There are these various ways to pick a pivot:

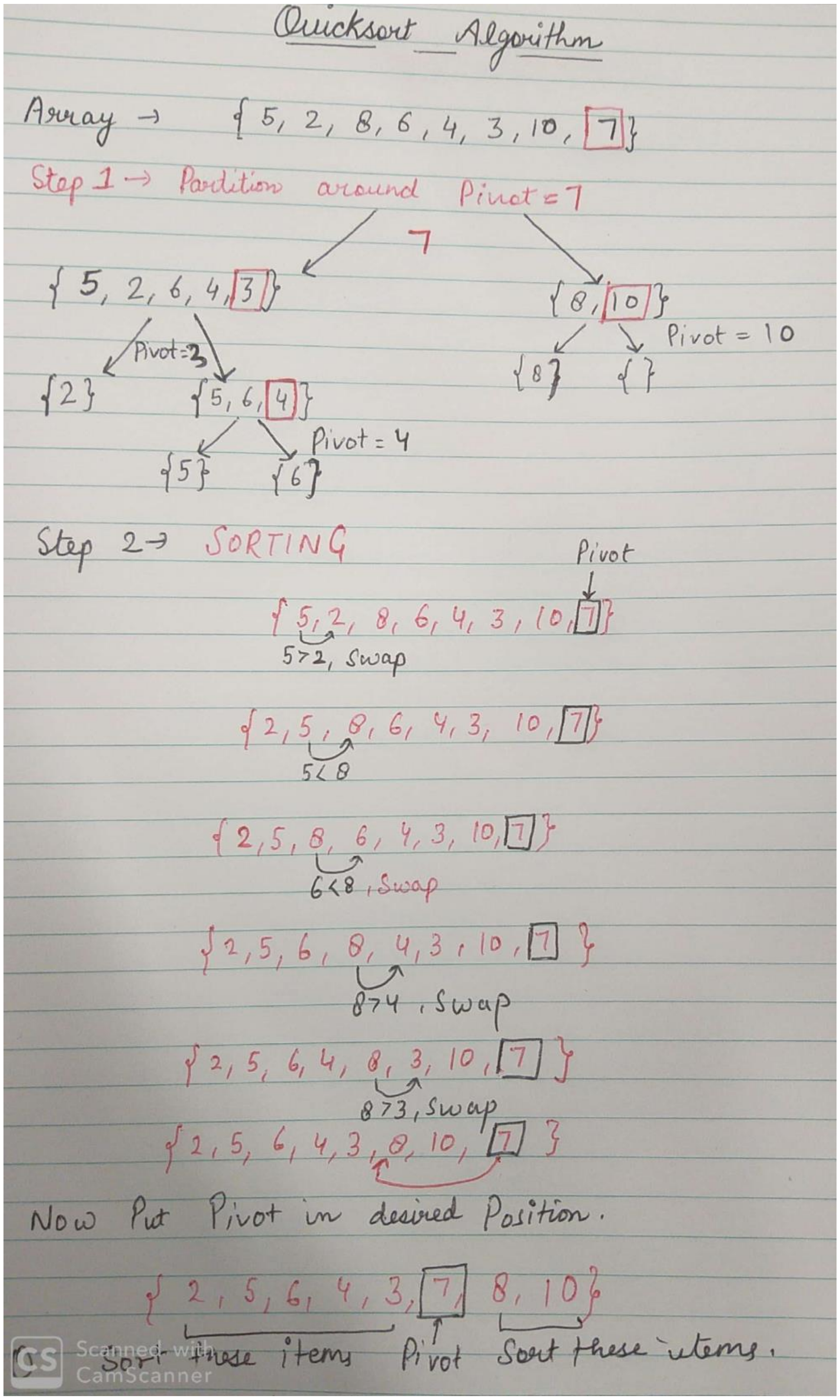
- Pick first or last element of array as pivot.
- Pick random element from arrays which is not out of index.
- Pick median of random elements in array.

2) Conquer: In first step the array was divided into two sub arrays, now in second step theses sub arrays are sorted recursively (Kaushal, 2017)

3) Combine: Since the subarrays gets sorted there is no need to combine these sub arrays (Kaushal, 2017) .



EXAMPLE FOR QUICKSORT ALGORITHM: (Ang, 2014)



Quicksort Algorithm

Array → {5, 2, 8, 6, 4, 3, 10, $\boxed{7}$}

Step 1 → Partition around Pivot = 7

7

{5, 2, 6, 4, $\boxed{3}$}

{8, $\boxed{10}$}

Pivot = 3

{2}

{5, 6, $\boxed{4}$}

Pivot = 4

{5}

{6}

Pivot = 10

{8}

{ }

Step 2 → SORTING

Pivot

{5, 2, 8, 6, 4, 3, 10, $\boxed{7}$}

5>2, Swap

{2, 5, 8, 6, 4, 3, 10, $\boxed{7}$}

5<8

{2, 5, 8, 6, 4, 3, 10, $\boxed{7}$}

6<8, Swap

{2, 5, 6, 8, 4, 3, 10, $\boxed{7}$}

8>4, Swap

{2, 5, 6, 4, 8, 3, 10, $\boxed{7}$}

8>3, Swap

{2, 5, 6, 4, 3, 8, 10, $\boxed{7}$}

Now Put Pivot in desired Position.

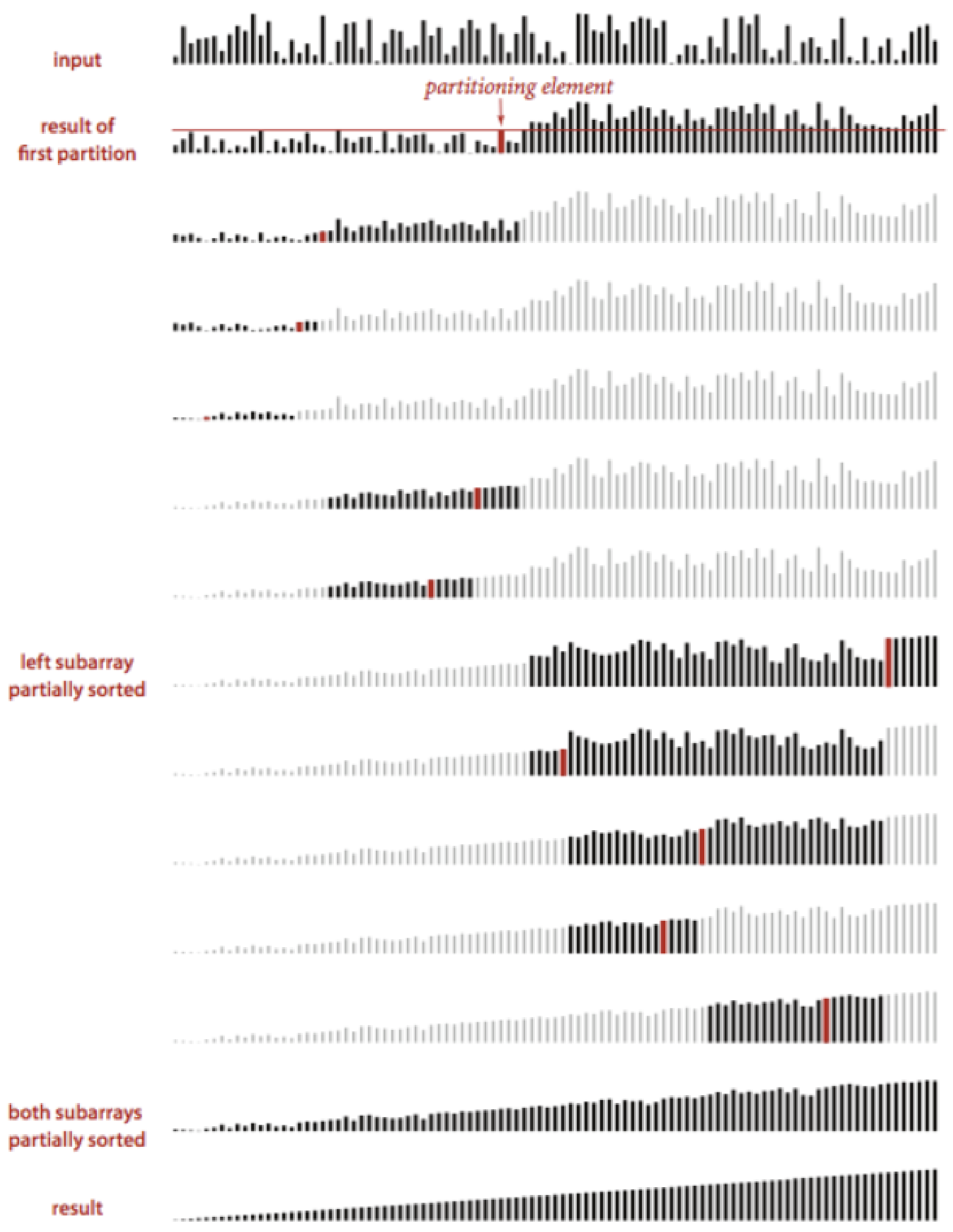
{2, 5, 6, 4, 3, $\boxed{7}$, 8, 10}

Sort these items

Pivot

Sort these items.

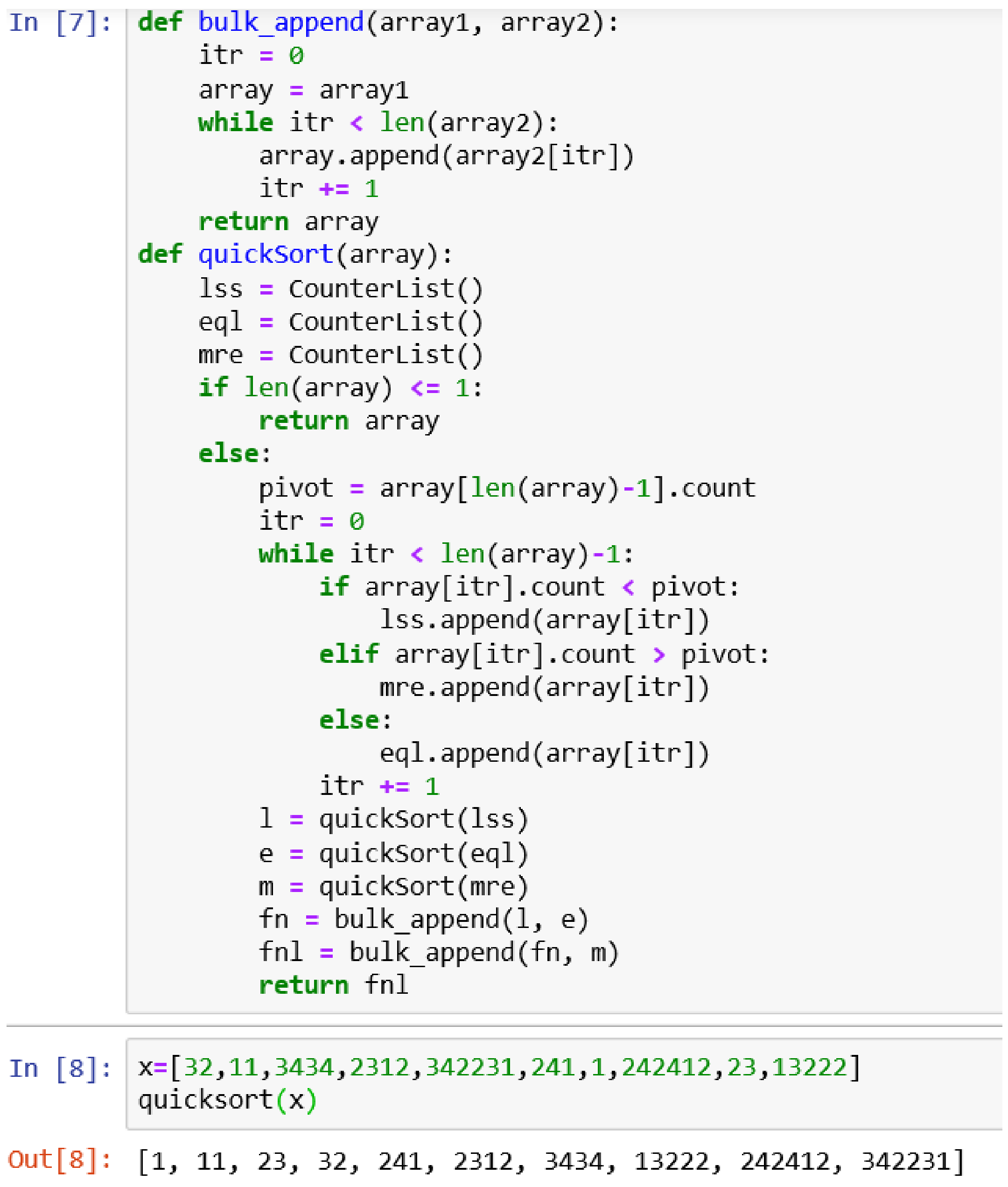
Visualization of Quicksort: (Sedgewick and Wayne, 2011)



Quicksort with median-of-3 partitioning and cutoff for small subarrays

CODE FOR QUICKSORT ALGORITHM:

1) Python code for quicksort with output



```
In [7]: def bulk_append(array1, array2):
            itr = 0
            array = array1
            while itr < len(array2):
                array.append(array2[itr])
                itr += 1
            return array
        def quickSort(array):
            lss = CounterList()
            eql = CounterList()
            mre = CounterList()
            if len(array) <= 1:
                return array
            else:
                pivot = array[len(array)-1].count
                itr = 0
                while itr < len(array)-1:
                    if array[itr].count < pivot:
                        lss.append(array[itr])
                    elif array[itr].count > pivot:
                        mre.append(array[itr])
                    else:
                        eql.append(array[itr])
                    itr += 1
                l = quickSort(lss)
                e = quickSort(eql)
                m = quickSort(mre)
                fn = bulk_append(l, e)
                fnl = bulk_append(fn, m)
                return fnl
```

```
In [8]: x=[32,11,3434,2312,342231,241,1,242412,23,13222]
        quicksort(x)

Out[8]: [1, 11, 23, 32, 241, 2312, 3434, 13222, 242412, 342231]
```

ANALYSIS OF QUICKSORT ALGORITHM:

Quicksort is a method which has large difference in running time for worst-case scenario and best-case scenario. Average-case of quicksort has almost same running time as that in best-case scenario. Here we analyze different scenarios of implementing quicksort algorithm :

BEST-CASE ANALYSIS:

The best case in quicksort arises when the item selected as pivot is exactly in middle of all elements after partitioning or in other words partitioning is very balanced.

The size of two sub-arrays to left of pivot and right of pivot is either equal or is within one of each other. This depends on number on elements in the array as given below:

- *Odd Number of Elements*:
  When the number of elements are odd in number then in best-case scenario the array is partitioned such that the pivot lies at:
  Let the number of elements in array be ‘n’ then,

  Pivot’s position = $\frac{(n-1)}{2}$

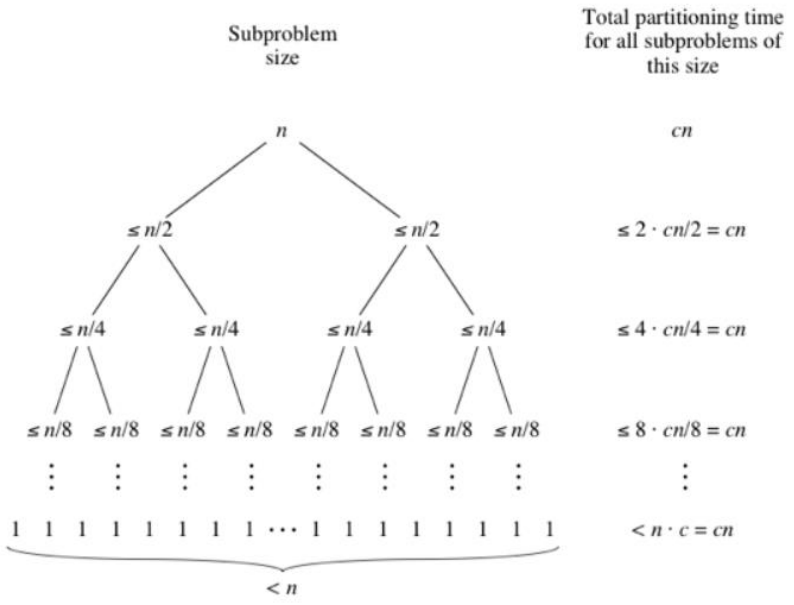
- *Even Number of Elements:*

  When the number of elements are even in number then in best-case scenario the array is partitioned such that the pivot lies at:

  Let the number of elements in array be ‘n’ then,

  Pivot’s position = $\left(\frac{n}{2}\right)$or $\left(\frac{n}{2} - 1\right)$

The figure below presents the analysis of best-case scenario where the pivot is present exactly at midpoint of arrays. The expected running time is also given corresponding to each partitioning of arrays. Big Oh notation is used for deriving the total expected running time and is displayed in form of Big Oh.



Using big-Θ notation running time in best-case scenario evaluates to (O *n* lg(*n*)), which is equal to the running time of Merge sort (Cormen and Balkcom, n.d.).

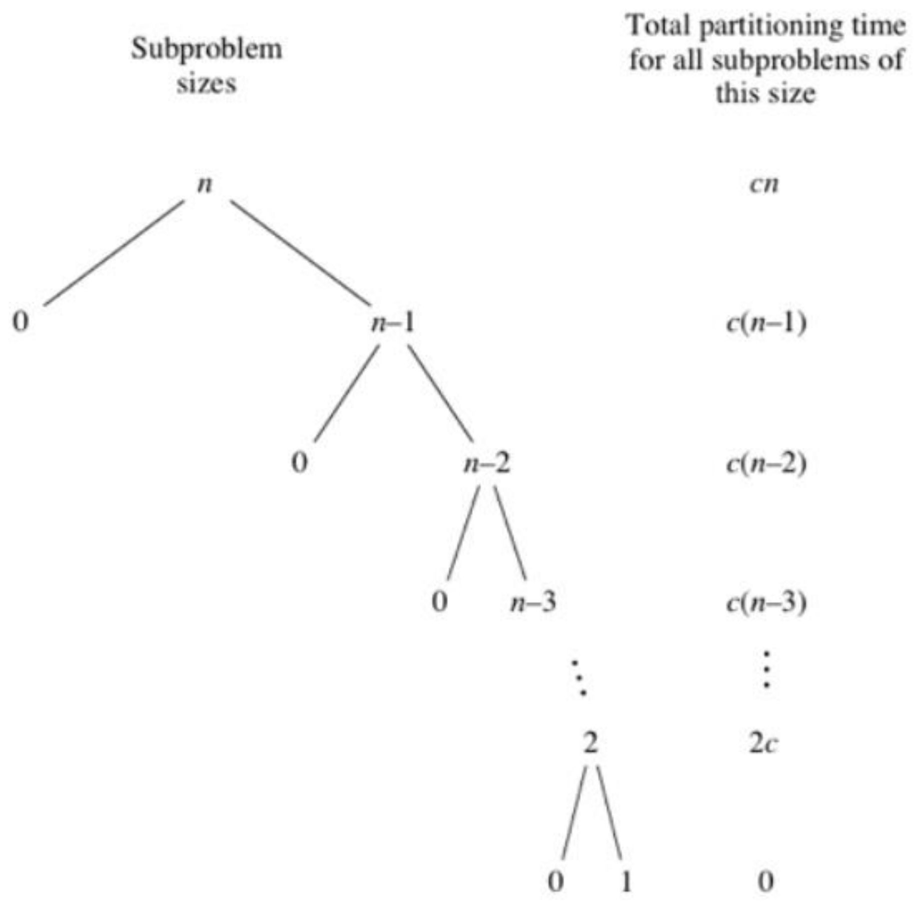
WORST-CASE ANALYSIS

The worst case in quicksort arises when the item selected as pivot is exactly in the corners of all elements after partitioning or in other words partitioning is not balanced and there is no item right to the pivot.

The size of two sub-arrays to left of pivot and right of the pivot has a huge difference between them, this arises a case where the recursion function calls itself equal to number of times as it would have in case of Insertion Sort.

The worst-case scenario arises most of the times when array is already sorted or is almost sorted.

The figure given below analysis how worst-case scenario works and it also demonstrates the time taken at each step:

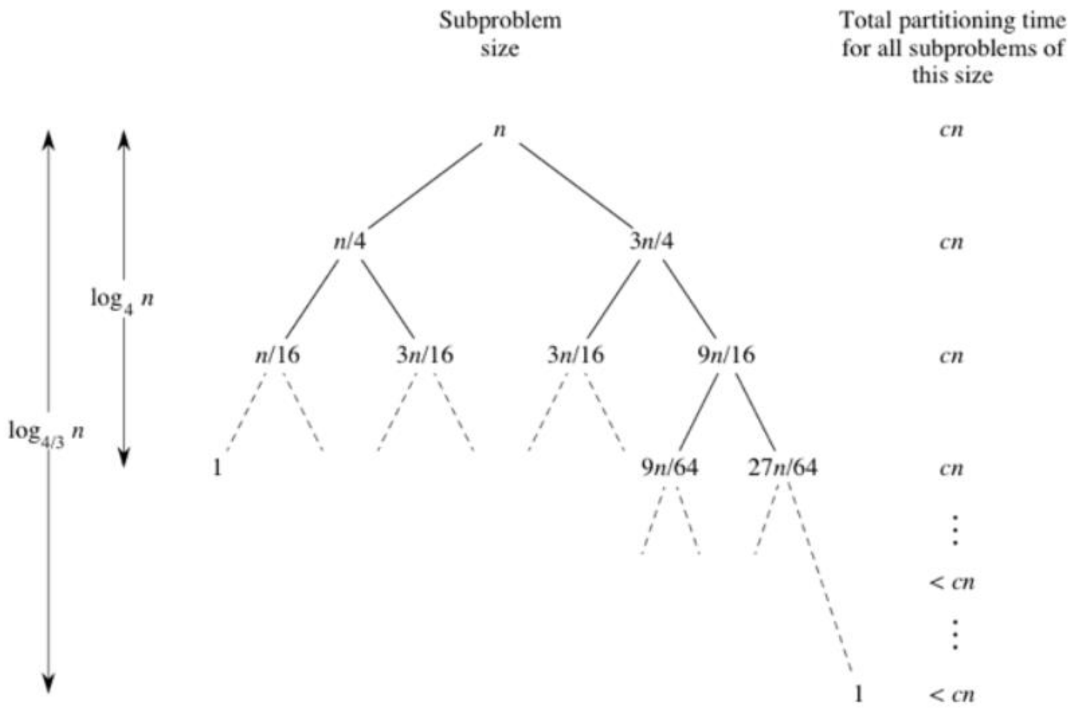


Adding the total partitioning time for all subproblems of each step and using Big Oh derivation, big oh notation for this case is (O n^2) (Cormen and Balkcom, n.d.).

AVERAGE CASE ANALYSIS

The average-case scenario arises when the partitioning in first step is neither very balanced as in best-case scenario nor very unbalanced as in worst-case scenario but is somewhere in between and in worst case is such that it has 3-to-1 split. Let's say array had n elements, so the pivot would be at such a position that one half has $\frac{3n}{4}$ elements while other half has $\frac{n}{4}$ elements.

The figure below explains partitioning in such scenario along with running time in each case:



Adding total partitioning time for all subproblems for each step and using Big Oh derivations, big O notation for this case is equivalent to that in best case scenario and that is (O *n* lg(*n*)) (Cormen and Balkcom, n.d.).

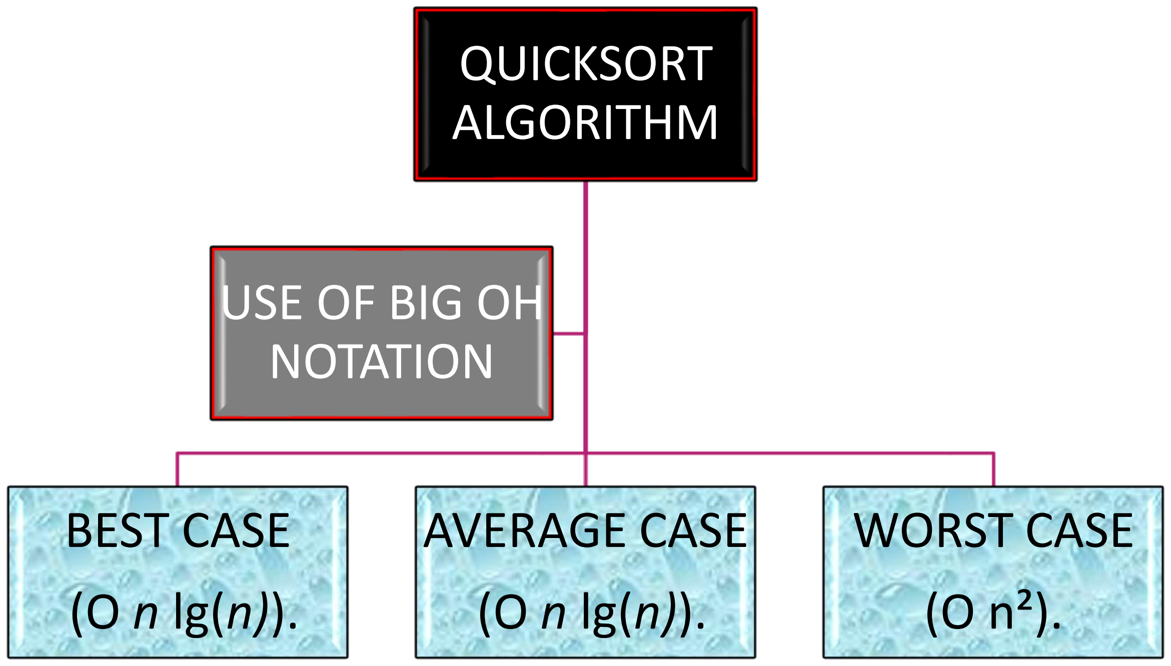
CONCLUSION:

The quicksort algorithm is most efficient when the pivot after partitioning is in middle of the array whereas, on the other hand this algorithm almost fails to deal with the scenario in which the rightmost pivot has no element to it’s right and the partitioning is very unbalanced. The average case as we read is also very efficient so it is not a big issue if the partitioning is neither very balanced nor very unbalanced.

This contrast in efficiency of quicksort algorithm is the point that must be considered before deploying this method to solve any task.

All in all, quicksort algorithm has everything to offer from a very quick solution to a sorting problem to a very slow solution to a sorting problem which totally depends upon the characteristic of the elements in the array and upon the position of pivot after partitioning (Cormen and Balkcom, n.d.).

SUMMARY OF CONCLUSION:



EVOLVING QUICKSORT!

Researchers are trying to eliminate the worst-case behaviour of Quicksort Algorithm using different techniques. These techniques are going to define the future of Quicksort Algorithm.One of the current advanced technique is Dynamic Pivot Selection Technique.

DUAL PIVOT SELECTION TECHNIQUE

This technique involves selection of pivot in such a way that the partitioning of array is balanced almost all the times and thus it propels to remove worst case behaviour and also enhances efficiency of average case behaviour.

HOW IT WORKS?

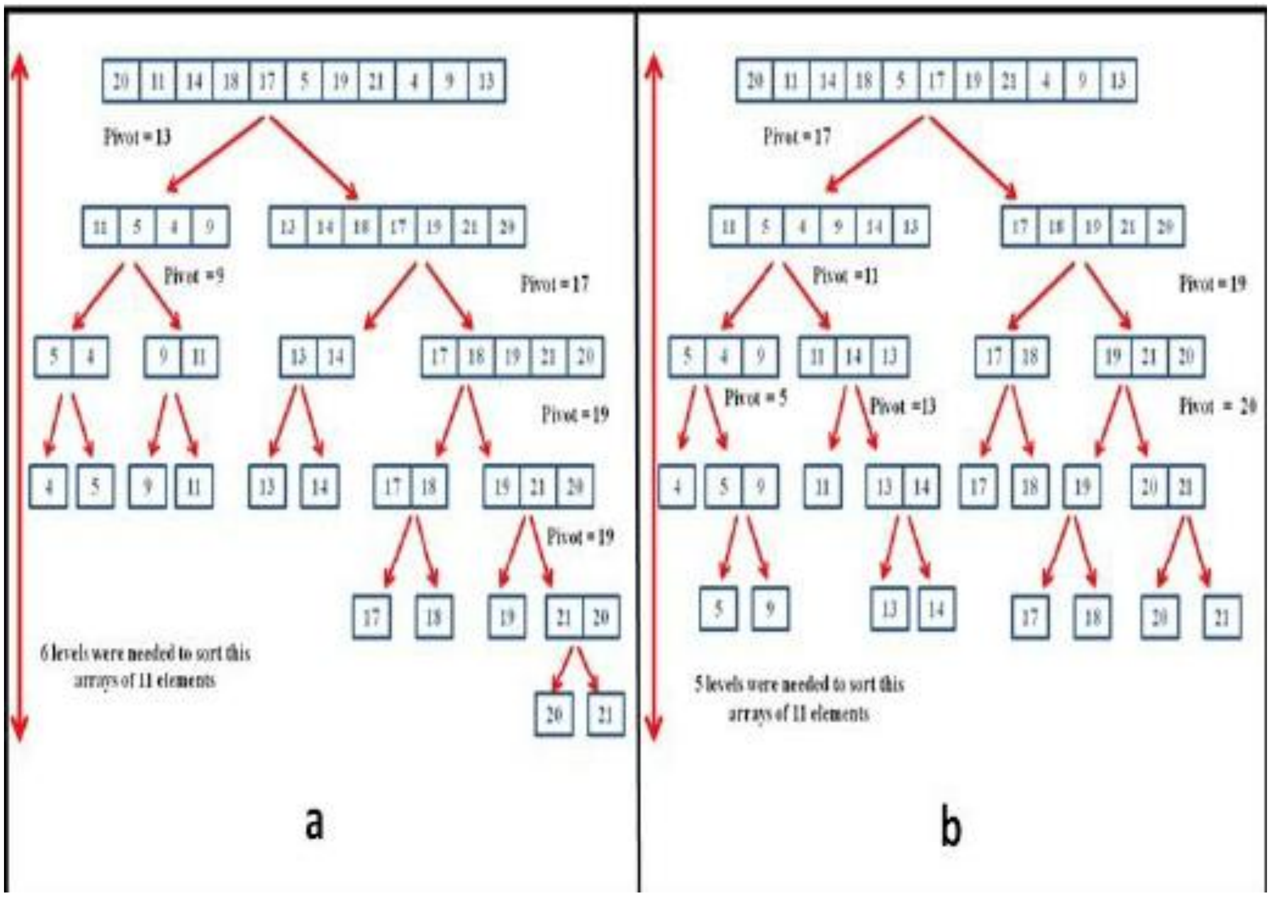
1) The rightmost element is picked as the pivot.
2) Other elements of array are compared with pivot and counters are used to count number of elements greater and smaller than the pivot, say CountLarge and CountLess respectively and their sums are also stored in SumLarge and SumLess respectively.
3) The integer averages of the SumLarge and SumLess are used as pivots for recursive calls in right and left sub-arrays respectively.

This is how we employ dual pivot selection technique for quicksorting a given array. This technique ensures equal distribution of array while partitioning such that algorithm always acquires best case running time that is represented as (O *n* lg(*n*)) in form of big oh notation (Dalhoum et al., 2012).

This algorithm is also know as MQuickSort Algorithm.

The image given analyses the working of MQuicksort Algorithm:

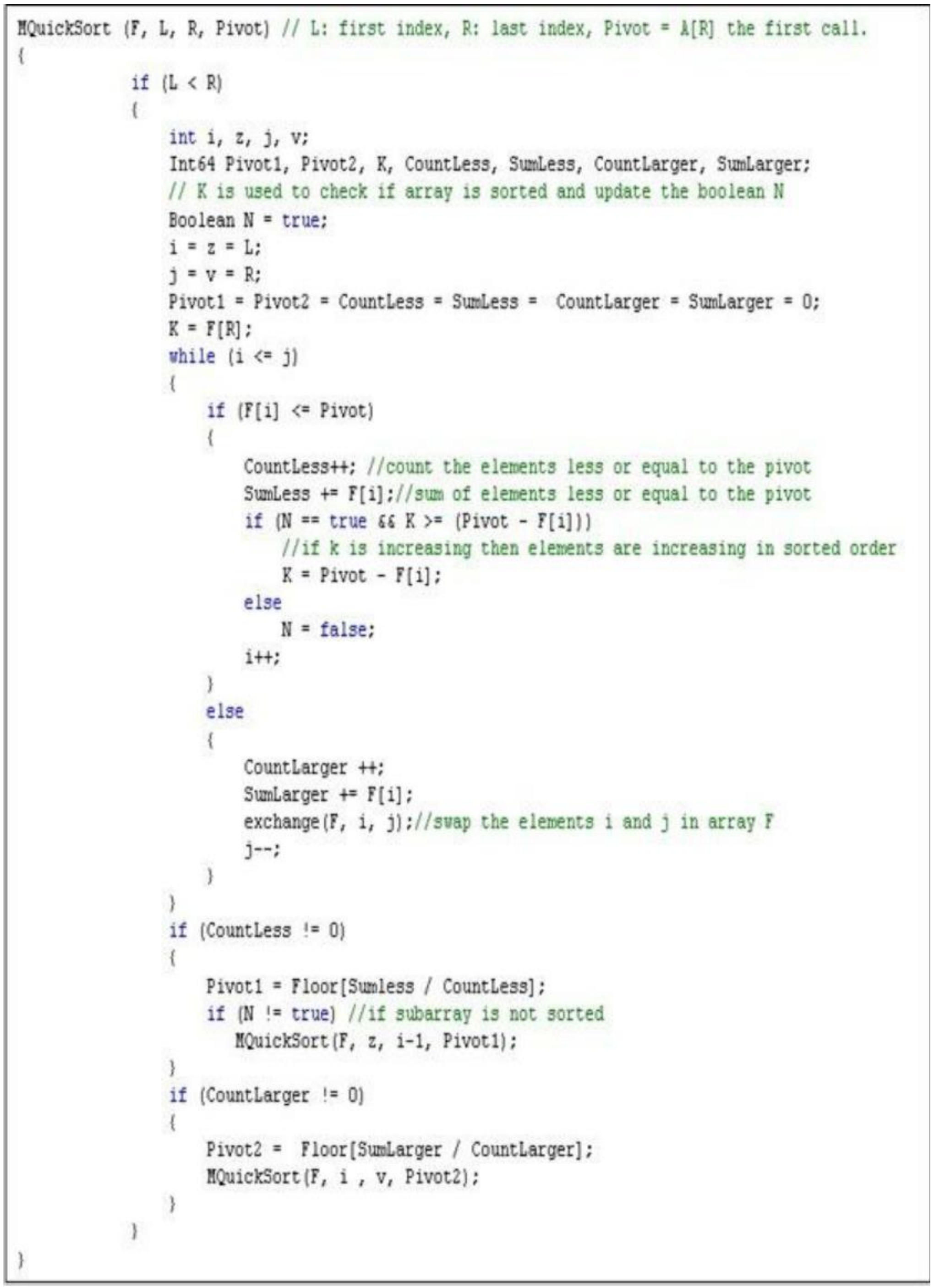
1


Pivot = 13
Pivot = 9
Pivot = 17
Pivot = 19
Pivot = 19
6 levels were needed to sort this arrays of 11 elements
a
Pivot = 17
Pivot = 11
Pivot = 19
Pivot = 5
Pivot = 13
Pivot = 20
5 levels were needed to sort this arrays of 11 elements
b

2

As evident from the image, this technique has reduced the number of levels or steps taken to sort the given array (Dalhoum et al., 2012).

Below is the code which executes the discussed algorithm in programming language Java:



```
MQuickSort (F, L, R, Pivot) // L: first index, R: last index, Pivot = A[R] the first call.
{
            if (L < R)
            {
                int i, z, j, v;
                Int64 Pivot1, Pivot2, K, CountLess, SumLess, CountLarger, SumLarger;
                // K is used to check if array is sorted and update the boolean N
                Boolean N = true;
                i = z = L;
                j = v = R;
                Pivot1 = Pivot2 = CountLess = SumLess =  CountLarger = SumLarger = 0;
                K = F[R];
                while (i <= j)
                {
                    if (F[i] <= Pivot)
                    {
                        CountLess++; //count the elements less or equal to the pivot
                        SumLess += F[i];//sum of elements less or equal to the pivot
                        if (N == true && K >= (Pivot - F[i]))
                            //if k is increasing then elements are increasing in sorted order
                            K = Pivot - F[i];
                        else
                            N = false;
                        i++;
                    }
                    else
                    {
                        CountLarger ++;
                        SumLarger += F[i];
                        exchange(F, i, j);//swap the elements i and j in array F
                        j--;
                    }
                }
                if (CountLess != 0)
                {
                    Pivot1 = Floor[Sumless / CountLess];
                    if (N != true) //if subarray is not sorted
                       MQuickSort(F, z, i-1, Pivot1);
                }
                if (CountLarger != 0)
                {
                    Pivot2 =  Floor[SumLarger / CountLarger];
                    MQuickSort(F, i , v, Pivot2);
                }
            }
}
```

CONCLUSION:

This report so has given us the understanding of how Quicksort algorithm works and how it’s executed with different time frames in different scenarios.

All in all, Quicksort is one of the most efficient means there is to sort an array or a list but still there is a lot of scope of improvement and the discussion on dynamic pivot selection technique helped us understand the part of Quicksort algorithm on which one can work to improve overall efficiency.

So, at the end one can conclude that sorting is one of the most helpful techniques developed to reduce the work put in to put huge chunks of data into a meaningful order which makes it easier to access the desired dataset and what better way other than Quicksort?

REFERENCE LIST

Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2009). *Introduction to algorithms*. 3rd ed. Cambridge, Massachusetts: MIT Press, pp.170-180.

Cormen, T. and Balkcom, D. (n.d.). *Overview of quicksort*. [online] Khan Academy. Available at: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort [Accessed 23 Apr. 2019]

Cormen, T. and Balkcom, D. (n.d.). *Analysis of quicksort*. [online] Khan Academy. Available at: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort [Accessed 23 Apr. 2019]

Ang, K. (2014). *Quicksort: Partitioning an array*. [video] Available at: https://www.youtube.com/watch?v=MZaf_9IZCrc [Accessed 20 Apr. 2019]

Kaushal, R. (2017). Why Sorting is So Important in Data Structures. - *International Journal for Scientific Research & Development*, [online] 5(7), pp.114-116. Available at: http://www.ijsrd.com/articles/IJSRDV5I70074.pdf [Accessed 26 Apr. 2019]

Sedgewick, R. and Wayne, K. (2011). *Quicksort*. [online] Algs4.cs.princeton.edu. Available at: https://algs4.cs.princeton.edu/23quicksort/ [Accessed 25 Apr. 2019]

Abu Dalhoum, Abdel & Kobbaey, Thaeer & Sleit, Azzam & Alfonseca, Manuel & De la Puente, Alfonso. (2012). Enhancing QuickSort Algorithm using a Dynamic Pivot Selection Technique. Wulfenia. 19. 543-552

WORD COUNT (only of content): 1823

Including summary: 1984

Total words: 2233