**Information Search and Analysis Skill**

**(ISAS)**

**Quick Sort Implementation with Python**



Arranged by:

GROUP 4:

1. Abdur Rashid Firdaus (2320010109)
2. Ahmad Maulana Ibrahim (2320010111)
3. Ayunda Pramita Kurnia Hapsari (2320010189)

Class: 2CS1 Faculty: Ivan Firdaus, S.T

**CEP CCIT**

**FAKULTAS TEKNIK UNIVERSITAS INDONESIA**

**2024**

# PREFACE

Praise be to Allah the Almighty for His blessings and grace, so we can complete this ISAS (Information Search and Analysis Skills) task both in the form of presentation and paper in a timely manner.

The motivation of this analysis is to provide information regarding topics that we learn throughout the first phase of the second semester. This research’s aim is not only to present findings but also to encourage more discussion among the class. We thank our faculty Mr. Ivan Firdaus, S.T. for the insights and suggestions to enhance this paper quality so that readers can easily digest and understand the topic that is covered in the paper.

Finally, the authors hope that this paper can be useful for all to gain a better understanding into the algorithm and data structure. We also realize that this paper may come with its imperfections, we accept all suggestions and criticisms from readers who are constructive in order for the perfection of this paper. We truly hope that this paper will provide only cover the best analysis.

Depok, March 2024

Author

# TABLE OF CONTENTS

[PREFACE I](#_Toc168575928)

[TABLE OF CONTENTS 0](#_Toc168575929)

[TABLE OF FIGURES 0](#_Toc168575930)

[CHAPTER I INTRODUCTION 1](#_Toc168575931)

[I.1. Background 1](#_Toc168575932)

[I.2. Writing Objective 1](#_Toc168575933)

[I.3. Problem Domain 1](#_Toc168575934)

[I.4. Writing Methodology 2](#_Toc168575935)

[I.5. Writing Framework 2](#_Toc168575936)

[I.5.1. Chapter I Introduction 2](#_Toc168575937)

[1.5.2. Chapter II Basic Theory 2](#_Toc168575938)

[1.5.3. Chapter III Problem Analysis 2](#_Toc168575939)

[1.5.4. Chapter IV Conclusion and Suggestions 2](#_Toc168575940)

[CHAPTER II BASIC THEORY 3](#_Toc168575941)

[II.1. Algorithm 3](#_Toc168575942)

[II.2. Data Structure 4](#_Toc168575943)

[II.3. Big O Notation 5](#_Toc168575944)

[II.4. Sorting 6](#_Toc168575945)

[II. 5. Classification of Sorting Algorithms 6](#_Toc168575946)

[II.6. Python 8](#_Toc168575947)

[CHAPTER III PROBLEM ANALYSIS 9](#_Toc168575948)

[III.1. Quick Sort 9](#_Toc168575949)

[III.2. Quick Sort algorithm 9](#_Toc168575950)

[III.3. Characteristics of Quick Sort 10](#_Toc168575951)

[III.4. Implementing Quick Sort Algorithm with Python 10](#_Toc168575952)

[CHAPTER IV CONCLUSION AND SUGGESTION 11](#_Toc168575953)

[IV.1. Conclusion 11](#_Toc168575954)

[IV.2. Suggestion 11](#_Toc168575955)

[BIBLIOGRAPHY 12](#_Toc168575956)

# TABLE OF FIGURES

# CHAPTER I INTRODUCTION

# I.1. Background

In one of the many things that we learn as a computer science student is the sorting algorithm. Sorting is a fundamental operation in computer science for optimizing various algorithms and applications. The need for efficient way of sorting has been constant with the ever-growing volumes of data that modern computing must handle. In this case, the Quick Sort algorithm emerges as a standout solution.

Developed and introduced by Tony Hoare in 1960, Quick Sort marked a shift in sorting methodologies. Using the divide and conquer strategy, Quick Sort showed an amazing efficiency in practice, earning its place as one of the most widely used sorting algorithms in the field.

As the computer science subject evolved thanks to the advancements of technology, the analysis of sorting algorithms remains to be looked upon for a significantly long time. This paper provides a comprehensive analysis of the Quick Sort algorithm, explaining its mechanics, efficiency, and its implementation in python programming language.

# I.2. Writing Objective

The purpose of this paper titled "implementing Quick Sort in Python" is to fulfilling the task of operating the algorithm and data structure task and to help authors and readers understand more about sorting algorithm.

# I.3. Problem Domain

This paper will only discuss Quick Sort. The main issues Includes:

* + 1. Quick Sort algorithm
    2. Characteristic of Quick Sort
    3. Implementation with python

# I.4. Writing Methodology

This paper will use a descriptive methodology to analyze the Quick Sort algorithm. It involves conducting in-depth research, discussions, and ensure all the available facts and data are gathered from trusted and reputable sources.

# I.5. Writing Framework

The writing systematics is written as follows:

# I.5.1. Chapter I Introduction

This chapter will discuss background, writing objectives, problem domains, writing methodology, and writing framework.

# Chapter II Basic Theory

This chapter will provide basic explanation of the things that are related to algorithm and data structures, and the topic for this paper.

# Chapter III Problem Analysis

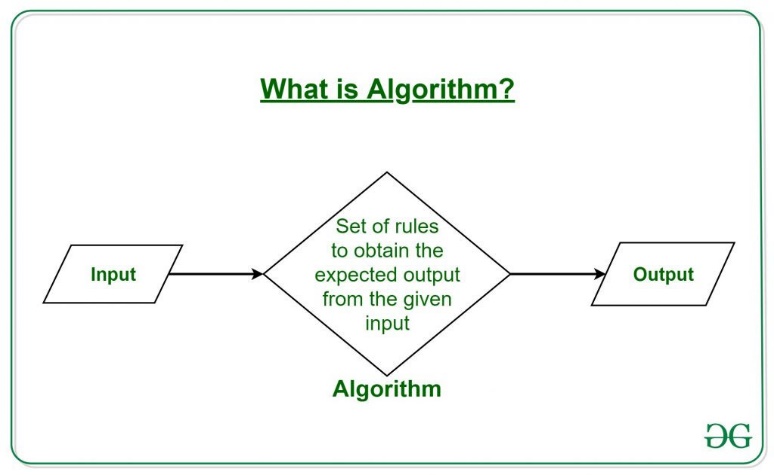
This chapter will provide the implementation of Quick Sort.

# Chapter IV Conclusion and Suggestions

The last chapter will provide a summary and recommendations that may prove useful to the reader

# CHAPTER II BASIC THEORY

## **II.1. Algorithm**

In writing programming code to a computer, a programmer must have a basic knowledge of decision making. A good programmer knows how to decide which solutions is the best for a specific problem. In that process, the writing of such solutions is called an algorithm. An algorithm is a process for carrying out a calculation or problem-solving. In hardware or software-based routines, algorithms function as a precise set of instructions that carry out predetermined operations one after the other.

**Figure II.1 Illustration of an Algorithm (https://www.geeksforgeeks.org/introduction-to-algorithms/)**

Algorithm are widely used throughout all areas of IT. An algorithm is typically used in mathematics, computer science, and computer programming to describe a brief process that resolves a recurring issue. Algorithms are also used as specifications for performing data processing and play a major role in automated systems.

## **II.2. Data Structure**

Data structure is a specialized format for organizing, processing, retrieving and storing data in a computer. It serves as a way of arranging data on a computer so that it can be accessed and updated efficiently. Data structures are divided into two categories, linear and non-linear. In linear data structure, data elements are arranged sequentially or linearly. Each element is connected to its previous and next adjacent elements. To understand the examples of data structures, Readers can imagine it as things that is seen in real life.

Arrays can be pictured as a line of boxes, where each box has a number. A child can put toys and trinkets in these boxes and easily find them by remembering the box number. Stacks can be pictured as a stack of plates. When stacking things on top of each other, one can only add or remove plates from the top.

Queues can be pictured as a line of fans waiting in line for an artist concert. The first fan will get the chance to sit or stand in front of the artist, and the next one follows. Trees can be imagined as a family tree. One might have a big tree with branches, and each branch has more branches. It helps organize and identify who is related to whom. Lastly, Graphs can be pictured as a map with cities connected by roads. One can travel between cities using these roads like a fun trip.

Understanding each data structures helped programmers make decisions. Selecting the right data structures for a specific problem is crucial in writing solutions as it helps programmers write memory and time efficient code.

## **II.3. Big O Notation**

Big O notation is a unique way to determine how fast a computer program grows when given an extra work to do. One can imagine building a building a Lego tower with blocks, and each block represents a task the computer needs to do. Here’s how it works:

1. **Efficiency of Building Blocks**:
   * Each toy block takes some time to stack on top of another.
   * Some blocks are easy to stack, while others are trickier.
   * We want to know how long it will take to build our tower (or run the program).
2. **Big O Notation**:
   * Big O helps us understand how fast the tower grows as one add more blocks (or tasks).
   * It’s like counting how many blocks needed to build a tall Lego tower.
3. **Examples**:
   * Let’s say:
     + **N** is the number of blocks (or tasks) available.
     + **T** is the time it takes to build the tower (or run the program).
   * If adding each block takes the same time, we say it’s **O(N)**:
     + If one has10 blocks, it takes 10 units of time.
     + If one has 100 blocks, it takes 100 units of time.
   * But sometimes, adding blocks takes longer as the Lego tower gets taller:
     + For example, if one needs to check every block before adding a new one, it’s **O(N^2)**.
     + If one has 10 blocks, it takes 100 units of time (10 \* 10).
     + If one has 100 blocks, it takes 10,000 units of time (100 \* 100).

Big O notation provides comparisons how fast different programs or algorithms work and help one choose the most efficient way to build the Lego tower or to solve a problem.

## **II.4. Sorting**

Sorting is a process of putting elements of a list into an order. One can imagine organizing scattered books into a book shelf. When sorting the books, one can arrange things in a specific way from putting the books in line from biggest to smallest or arranging the books from newest to oldest. There are many ways of sorting including by size, color, shape, and any other rule one might choose. Sorting algorithm is a way computer organize data in order efficiently.

By exploring sorting algorithms, a programmer can easily sort things with computers. Nowadays there is a variety sorting algorithm to choose each with their own method of sorting. Some sort used by selecting the smallest or largest element and place it in a correct position like selection sort, some repeatedly compares adjacent elements and swaps them in the right order such as bubble sort, and some divide the array and sort it out individually.

Each sorting algorithm has its own strengths and limitations. The choice of algorithm highly depends on the specific problem and the specialty of the data that is been sort.

## **II. 5. Classification of Sorting Algorithms**

1. Stable or unstable sorting algorithm

If a sorting algorithm, after sorting the data items, does not change the sequence of similar data items in which the data appear in the input array then it is called stable sorting. If a sorting algorithm, after sorting the data items, changes the sequence of similar data that appeared, it is called unstable sorting. Examples of stable sorts are Merge Sort, Tim sort, Counting Sort, Insertion Sort, and Bubble Sort. Examples of non-stable sorts are Quicksort, Heap sort and Selection Sort

1. Recursive or non-Recursive (iterative)

Recursion is a phenomenon in which function calls itself again and again. Recursive sorting algorithms work by splitting the input array into two or more smaller subarrays and then sorting those sub arrays, it then combines the sorted subarrays into sorted output array. A non-recursive algorithm does the sorting all at once by using iterative statements/repetitive statements/loops only, without calling itself again and again. Examples of Recursive sorts are merge sort, quick sort Examples of not recursive bubble sort, selection sort, insertion sort.

1. Adaptive or Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them. A non-adaptive algorithm is one which does not take into account the elements which are already sorted. It tries to force every single element to be re-ordered to confirm their sorting. Examples of adaptive sorts are stranded sort, quick sort Examples of non-adaptive sorts are Selection Sort, Merge Sort, and Heap Sort.

1. Computational Complexity

The algorithmic complexity of different sorting algorithms varies depending on Best, worst and average case behaviour in terms of the size of the data list. For typical serial sorting algorithms, good behaviour is 𝑂(𝑛 𝑙𝑜𝑔 𝑛), with parallel sort is 𝑂(log2 𝑛), and bad behaviour is 𝑂(𝑛 2 ). Ideal behaviour for a serial sort is 𝑂(𝑛), but this is not possible in the average case. Optimal parallel sorting is 𝑂(𝑙𝑜𝑔 𝑛).

1. Online or offline

An online algorithm is one that can process its input one by one in a serial fashion, in the order that the input is fed to the algorithm, without having the entire input available from the start of the algorithm, for example, insertion sort. In contrast, an offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand.

1. Comparison sort or non-comparison sort

A comparison sort examines the data only by comparing two elements with a comparison operator. best complexity of comparison sort algorithm is 𝑂(𝑛 𝑙𝑜𝑔(𝑛)) A non-comparison algorithm sorts the data without using comparisons it uses the internal character/nature of the values for producing sorted output. For example, counting sort, and radix sort.

1. Algorithm is serial or parallel

Parallel sorting algorithms are simply algorithms that allows parallel processing, that is, perform multiple operations in a given time.While serial sorting algorithms are simply algorithms that allows serial processing and don’t support parallel processing.

1. External and Internal sorting

In internal sorting all the data to be sorted is stored in memory (main memory or RAM) at all times while sorting algorithm is being executed by a processor.In external sorting data is stored outside memory or secondary memory (like on disk) and only loaded into memory in small blocks whenever needed. External sorting is usually applied in cases when data can't fit into main memory, entirely memory available is less than data to be sorted.

## **II.6. Python**

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python is simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Often, programmers fall in love with Python because of the increased productivity it provides. Since there is no compilation step, the edit-test-debug cycle is incredibly fast. Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on.

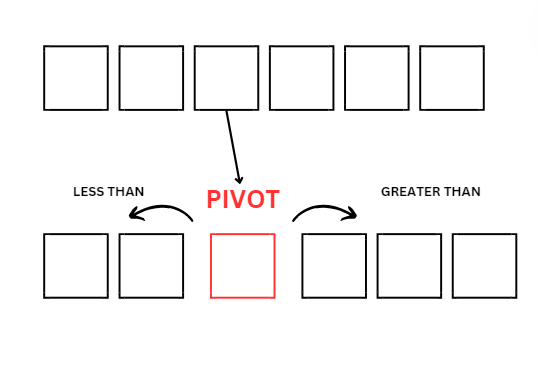
# CHAPTER III PROBLEM ANALYSIS

# Quick Sort

Quick sort is a sorting algorithm based on the divide and conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The way it works is to place the pivot at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot. Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

# Quick Sort algorithm

Every sorting begins with Quick Sort selects an element called pivot. The pivot is fixed in place by moving all the elements less than that to its left and all the elements greater than that to its right. The partitions of the element sequence into left, pivot, and right are referred as a sorting by partitioning.

**Figure III. 2 Quick Sort Algorithm**

**(https://www.canva.com/design/DAF5j9p4zP8/hIB9J\_Qif\_F89GFNHvykRA/edit)**

# Characteristics of Quick Sort

Quick Sort is a highly effective algorithm used for sorting arrays. It uses a divide and conquer approach. Quick Sort partitions the array into smaller segments, sorting them recursively, and combining the result at the last. This include selecting an element as a pivot, a pivot can be the first, middle or any element of the array. Pivot used as a way to partition the array by placing the element greater than to its right and the element less than to its left until the array is sorted.

Despite its efficiency, Quick Sort is not a stable sort, it means that the relative order of equal elements is not guaranteed to be preserved during the sorting process. Quick sort prioritizes speed and simplicity over stability. This could be a major consideration depending on the specific requirements of a given task.

# Implementing Quick Sort Algorithm with Python

Quick sort is more efficient when implemented on smaller data sets. Therefore, the examples written below only use small data sets. The data sets implemented here are sequences of integers.

def main():

    data = []

    while True:

        data\_length = int(input("How long is the array? "))

        if data\_length >= 1:

            break

    for i in range(data\_length):

        if (i + 1 == 1) or ((i + 1) % 10 == 1):

            data.append(int(input(f"{i + 1}st value: ")))

        elif (i + 1 == 2) or ((i + 1) % 10 == 2):

            data.append(int(input(f"{i + 1}nd value: ")))

        elif (i + 1 == 3) or ((i + 1) % 10 == 3):

            data.append(int(input(f"{i + 1}rd value: ")))

        else:

            data.append(int(input(f"{i + 1}th value: ")))

    print("")

    print(f"The array that you provided is: {data}")

    print(f"The quick sort process:")

    sorted\_data = quick\_sort(data)

    print(f"The sorted array: {sorted\_data}")

def quick\_sort(array):

    if len(array) <= 1:

        return array

    pivot = array[0]

    lesser\_than\_pivot = []

    greater\_than\_pivot = []

    for i in array[1:]:

        if i <= pivot:

            lesser\_than\_pivot.append(i)

        else:

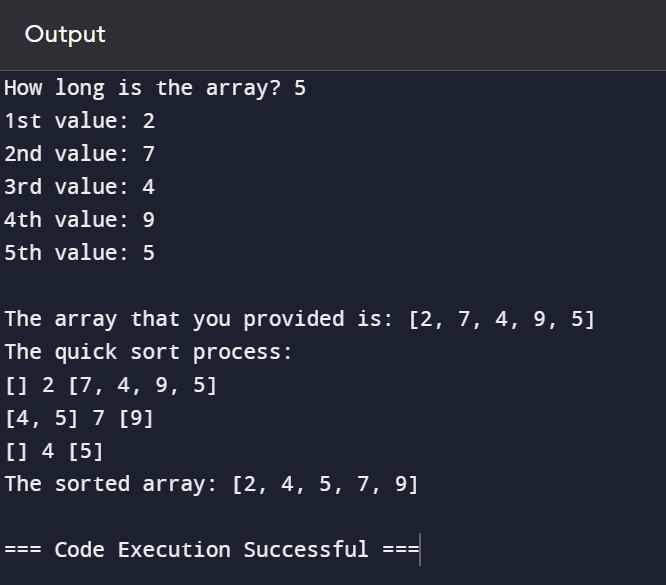
            greater\_than\_pivot.append(i)

    print(lesser\_than\_pivot, pivot, greater\_than\_pivot)

    return quick\_sort(lesser\_than\_pivot) + [pivot] + quick\_sort(greater\_than\_pivot)

if \_\_name\_\_ == "\_\_main\_\_":

    main()



# CHAPTER IV CONCLUSION AND SUGGESTION

# Conclusion

Quick sort is a sorting algorithm used by picking an element as a pivot. It determines the correct position for the pivot by checking each element in the array, and places it where all the smaller elements are on the left and larger ones are on the right. This sorting algorithm is preferred for arrays and is more efficient on smaller data sets.

In this paper, quick sort is implemented on two different data sets with even and odd number of elements. It can be concluded that the implementation is the same no matter the length. The main advantage of quick sort is that it is fast and efficient, and the main drawback is instability. Compared to two other sorting algorithms, quick sort’s uniqueness lies in its preference for arrays and its distinct way for partitioning arrays into any ratio.

# Suggestion

For readers of this paper, it is suggested that:

For researchers, it is hoped that this paper will be a valuable piece of reference in books or papers. For students, it is hoped that this paper will be a valuable piece of knowledge on introducing a type of sorting algorithms. Things to note when utilizing quick sort is to pick pivots wisely as it can be crucial to the time of sorting. Another thing is to consider another sorting when quick sort is not sorting effectively.

To enhance knowledge in the field of algorithms and data structures, it is highly recommended to seek the resources provided in the bibliography or any other articles, books, and courses regarding algorithms and data structures. Authors recognize the imperfections in this paper. Therefore, criticisms are highly needed for the improvements of the subsequent papers.

# BIBLIOGRAPHY

1. **Al-Kharabsheh et al**., 2013. *Review on Sorting Algorithm*. Aqoba: Academia.edu.
2. **Rouf Ali,** 2021. *Sorting and Classification Sorting Algorithms.* Kashmir:

Turkish Journal of Computer and Mathematics Education (TURKOMAT)

1. <https://youtu.be/4n6_x62-LZQ?si=8qdiHYIMSrJ2EweW>. Adil Ansari [3/8/2024]
2. <https://www.techquintal.com/advantages-and-disadvantages-of-quick-sort/>. Sahana [6/6/2024]
3. <https://onophp.blogspot.com/2018/11/quick-sort-pengertian-agoritma-dan.html>. Admin 1 [6/6/2024]
4. <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/>. Hackerearth [6/6/2024]
5. <https://www.techtarget.com/whatis/definition/algorithm#:~:text=An%20algorithm%20is%20a%20procedure,throughout%20all%20areas%20of%20IT>. Alexander. S. Gills [6/6/2024]
6. <https://www.geeksforgeeks.org/introduction-to-algorithms/>. RishabhPrabhu [6/6/2024]
7. <https://stackoverflow.com/questions/56717163/quicksort-algorithm-failed-with-stackoverflow-error>. Community [6/6/2024]
8. <https://stackoverflow.com/questions/164163/quicksort-choosing-the-pivot>. Community [6/6/2024]
9. <https://www.geeksforgeeks.org/python-program-for-quicksort/> [6/6/2024]