# Week 4 Assignment 5

Renowned for its divide-and-conquer strategy, quicksort is a rather effective and extensively used sorting method. It sorts the partitions after recursively separating an array around a pivot element. Still, its performance may be much changed by the pivot choice and the input distribution. Deterministic and randomized variants of Quicksort are used to solve these difficulties; each has different benefits and constraints. Emphasizing the advantages of both methods, this paper explores the corrected implementations, theoretical insights, and empirical outcomes.

The pivot of Quicksort's deterministic form is selected as the first member in the array. Although this method is straightforward and predictable, it is prone to worst-case situations especially in cases of previously sorted or reverse-sorted data. The updated version includes a strong partitioning technique to help to minimize any problems like endless recursion. Two sections comprise the array: items less than the pivot and elements either bigger than or equal to the pivot. These divisions omit the pivot itself to prevent duplicate processing. A base case is also specifically built to manage arrays with one or no items, hence guaranteeing appropriate termination of recursion. While it typically results in imbalanced partitions and higher recursion depth, this deterministic form performs well for random inputs only suffers when the input data is organized.   
**Deterministic Quicksort**

def deterministic\_quicksort(array):

"""

Deterministic Quicksort implementation using the first element as the pivot.

"""

# Base case: Return the array if it has one or no elements

if len(array) <= 1:

return array

# Select the first element as the pivot

pivot = array[0]

# Partition the array into elements smaller and greater than or equal to the pivot

left\_partition = [x for x in array if x < pivot]

right\_partition = [x for x in array[1:] if x >= pivot]

# Recursively sort the partitions and combine with the pivot

return deterministic\_quicksort(left\_partition) + [pivot] + deterministic\_quicksort(right\_partition)

By use of random pivot selection, the randomized Quicksort overcomes the constraints of the deterministic method. A random element is selected instead of a fixed pivot, therefore greatly lowering the likelihood of worst-case performance. This randomism guarantees, independent of the input distribution, greater average balance of partitions. The pivot is chosen at random in the corrected implementation, so the array is split among items less than the pivot and elements bigger than it. Explicitly omitted while partitioning is the pivot, therefore preventing repeated processing. Small arrays are handled using a base case, same as in the deterministic form. Robustness of Randomized Quicksort guarantees constant average-case performance and makes it especially appropriate for managing various and erratic input distributions.

**Randomized Quicksort**  
import random

def randomized\_quicksort(array):

"""

Randomized Quicksort implementation using a randomly selected pivot.

"""

# Base case: Return the array if it has one or no elements

if len(array) <= 1:

return array

# Select a random pivot

pivot\_index = random.randint(0, len(array) - 1)

pivot = array[pivot\_index]

# Partition the array into elements smaller and greater than the pivot

less\_than\_pivot = [x for x in array if x < pivot]

greater\_than\_pivot = [x for x in array if x > pivot]

# Recursively sort the partitions and combine with the pivot

return randomized\_quicksort(less\_than\_pivot) + [pivot] + randomized\_quicksort(greater\_than\_pivot)

Empirical experiments employing arrays of different sizes and distributions helped to assess the performance of various systems. Three sorts of input arrays were produced: random, sorted, and reverse-sorted. These inputs provide many situations that test the algorithms' robustness and efficiency. We measured and compared the execution times of deterministic and randomized Quicksort for every array. For random inputs the deterministic variant worked well; unfortunately, imbalanced partitions caused reduced performance for sorted and reverse-sorted arrays. On the other hand, the randomized form showed consistent performance throughout all distributions, therefore proving its capacity to sufficiently reduce worst-case situations.

**Function to Measure Execution Time**

import time

def sort\_and\_measure(func, array):

"""

Measures the execution time of a sorting function.

"""

start\_time = time.time() # Start timer

func(array) # Execute the sorting function

end\_time = time.time() # End timer

return end\_time - start\_time # Return the elapsed time

**Input Generation**

def generate\_input(size, mode):

"""

Generates test input arrays of different distributions.

"""

if mode == "Random":

return [random.randint(0, 10000) for \_ in range(size)] # Random integers

elif mode == "Sorted":

return list(range(size)) # Pre-sorted array

elif mode == "Reverse Sorted":

return list(range(size, 0, -1)) # Reverse-sorted array

**Performance Analysis and Testing**

import pandas as pd

def analyze\_performance():

"""

Runs deterministic and randomized Quicksort on various inputs and records performance.

"""

input\_sizes = [100, 1000, 5000, 10000] # Different input sizes

array\_types = ["Random", "Sorted", "Reverse Sorted"] # Types of input distributions

performance\_data = [] # List to store performance results

for size in input\_sizes:

for mode in array\_types:

# Generate input array

array = generate\_input(size, mode)

# Measure deterministic Quicksort runtime

deterministic\_time = sort\_and\_measure(deterministic\_quicksort, array[:])

# Measure randomized Quicksort runtime

randomized\_time = sort\_and\_measure(randomized\_quicksort, array[:])

# Record results

performance\_data.append({

"Input Size": size,

"Distribution": mode,

"Deterministic Time (s)": deterministic\_time,

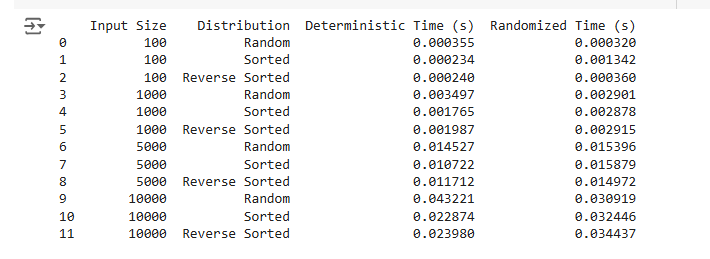
"Randomized Time (s)": randomized\_time

})

# Convert results to a DataFrame

return pd.DataFrame(performance\_data)

# Run the performance analysis results = analyze\_performance() # Display results in the console print("Performance Comparison:") print(results)



Theoretical study emphasizes much more the variations between the two methods. A best-case and average-case time complexity of \(O(n \log n)\ both Quicksort's variants are attained when the partitions are roughly balanced at every recursive phase. On the worst case, nevertheless, the deterministic form may collapse to \(O(n^2)\—that is, when the pivot regularly generates severely imbalanced partitions. Conversely, Randomized Quicksort guarantees that the worst-case situation is seldom ever faced by lowering the possibility of such imbalanced partitions. In practical uses, this makes the randomized form more dependable and strong. From a space complexity standpoint, the number of recursive calls determines the depth of the recursive stack in both systems. This depth is \(O(\log n)\), equivalent to the height of a balanced recursion tree, in both best and average situations. Under worst conditions, unbalanced recursion might cause the stack depth to develop to \(O(n)\) for deterministic Quicksort. With its balanced divisions, Randomized Quicksort guarantees that, most of the time, the stack depth stays below reasonable limits. For many different uses, the revised implementations are flexible and useful. Simplicity of Deterministic Quicksort makes it appropriate for situations where predictable and unpredictable input data exists. Large datasets or erratic input distributions call for Randomized Quicksort's resilience. With uses in sorting vast datasets, structuring search results, and prepping data for analytics, both versions are fundamental algorithms in computer science. Understanding the subtleties of various implementations will help developers choose the suitable version for their particular requirements, therefore guaranteeing dependable and efficient performance in many different environments.