Q1  
  
Quick Sort  
  
​​def partition(arr, low, high):

# Choose the pivot

pivot = arr[high]

i = low - 1

# Traverse arr[low..high] and move all smaller

# elements on the left side. Elements from low to

# i are smaller after every iteration

for j in range(low, high):

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

# Move pivot after smaller elements and

# return its position

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

# The QuickSort function implementation

def quick\_sort(arr, low, high):

if low < high:

# pi is the partition return index of pivot

pi = partition(arr, low, high)

# Recursion calls for smaller elements

# and greater or equals elements

quick\_sort(arr, low, pi - 1)

quick\_sort(arr, pi + 1, high)

# Function to print an array

def print\_array(arr):

for i in arr:

print(i, end=" ")

print()

# Driver code

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

arr = [10, 7, 8, 9, 1, 5]

print("Given array is")

print\_array(arr)

quick\_sort(arr, 0, len(arr) - 1)

print("\nSorted array is")

print\_array(arr)

Time Complexity:

Best Case : Ω (N log (N))

The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.

In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

Average Case: θ ( N log (N))

Quicksort’s average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

Worst Case: O(N ^ 2)

The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort ) to shuffle the element before sorting.



3-Way Merge Sort

# Python Program to perform 3 way Merge Sort

""" Merge the sorted ranges [low, mid1), [mid1,mid2)

and [mid2, high) mid1 is first midpoint

index in overall range to merge mid2 is second

midpoint index in overall range to merge"""

def merge(gArray, low, mid1, mid2, high, destArray):

i = low

j = mid1

k = mid2

l = low

# Choose smaller of the smallest in the three ranges

while ((i < mid1) and (j < mid2) and (k < high)):

if(gArray[i] < gArray[j]):

if(gArray[i] < gArray[k]):

destArray[l] = gArray[i]

l += 1

i += 1

else:

destArray[l] = gArray[k]

l += 1

k += 1

else:

if(gArray[j] < gArray[k]):

destArray[l] = gArray[j]

l += 1

j += 1

else:

destArray[l] = gArray[k]

l += 1

k += 1

# Case where first and second ranges

# have remaining values

while ((i < mid1) and (j < mid2)):

if(gArray[i] < gArray[j]):

destArray[l] = gArray[i]

l += 1

i += 1

else:

destArray[l] = gArray[j]

l += 1

j += 1

# case where second and third ranges

# have remaining values

while ((j < mid2) and (k < high)):

if(gArray[j] < gArray[k]):

destArray[l] = gArray[j]

l += 1

j += 1

else:

destArray[l] = gArray[k]

l += 1

k += 1

# Case where first and third ranges have

# remaining values

while ((i < mid1) and (k < high)):

if(gArray[i] < gArray[k]):

destArray[l] = gArray[i]

l += 1

i += 1

else:

destArray[l] = gArray[k]

l += 1

k += 1

# Copy remaining values from the first range

while (i < mid1):

destArray[l] = gArray[i]

l += 1

i += 1

# Copy remaining values from the second range

while (j < mid2):

destArray[l] = gArray[j]

l += 1

j += 1

# Copy remaining values from the third range

while (k < high):

destArray[l] = gArray[k]

l += 1

k += 1

""" Performing the merge sort algorithm on the

given array of values in the rangeof indices

[low, high). low is minimum index, high is

maximum index (exclusive) """

def mergeSort3WayRec(gArray, low, high, destArray):

# If array size is 1 then do nothing

if (high - low < 2):

return

# Splitting array into 3 parts

mid1 = low + ((high - low) // 3)

mid2 = low + 2 \* ((high - low) // 3) + 1

# Sorting 3 arrays recursively

mergeSort3WayRec(destArray, low, mid1, gArray)

mergeSort3WayRec(destArray, mid1, mid2, gArray)

mergeSort3WayRec(destArray, mid2, high, gArray)

# Merging the sorted arrays

merge(destArray, low, mid1, mid2, high, gArray)

def mergeSort3Way(gArray, n):

# if array size is zero return null

if (n == 0):

return

# creating duplicate of given array

fArray = []

# copying elements of given array into

# duplicate array

fArray = gArray.copy()

# sort function

mergeSort3WayRec(fArray, 0, n, gArray)

# copy back elements of duplicate array

# to given array

gArray = fArray.copy()

# return the sorted array

return gArray

data = [45, -2, -45, 78, 30, -42, 10, 19, 73, 93]

data = mergeSort3Way(data, 10)

print("After 3 way merge sort: ", end="")

for i in range(10):

print(f"{data[i]} ", end="")

# This code is contributed by Susobhan Akhuli

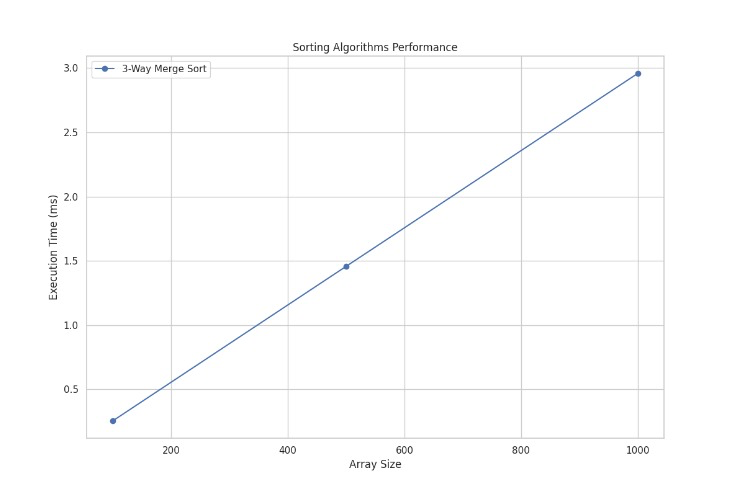
Time Complexity: In case of 2-way Merge sort we get the equation: T(n) = 2T(n/2) + O(n)

Similarly, in case of 3-way Merge sort we get the equation: T(n) = 3T(n/3) + O(n)

By solving it using Master method, we get its complexity as O(n log 3n).

Although time complexity looks less compared to 2 way merge sort, the time taken actually may become higher because number of comparisons in merge function go higher. Please refer Why is Binary Search preferred over Ternary Search? for details.

Auxiliary Space Complexity: The space complexity of 3-way merge sort is same as 2-way merge sort: O(n)



Heap Sort

# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.

# n is size of heap

def heapify(arr, N, i):

largest = i # Initialize largest as root

l = 2 \* i + 1 # left = 2\*i + 1

r = 2 \* i + 2 # right = 2\*i + 2

# See if left child of root exists and is

# greater than root

if l < N and arr[largest] < arr[l]:

largest = l

# See if right child of root exists and is

# greater than root

if r < N and arr[largest] < arr[r]:

largest = r

# Change root, if needed

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] # swap

# Heapify the root.

heapify(arr, N, largest)

# The main function to sort an array of given size

def heapSort(arr):

N = len(arr)

# Build a maxheap.

for i in range(N//2 - 1, -1, -1):

heapify(arr, N, i)

# One by one extract elements

for i in range(N-1, 0, -1):

arr[i], arr[0] = arr[0], arr[i] # swap

heapify(arr, i, 0)

# Driver's code

if \_\_name\_\_ == '\_\_main\_\_':

arr = [12, 11, 13, 5, 6, 7]

# Function call

heapSort(arr)

N = len(arr)

print("Sorted array is")

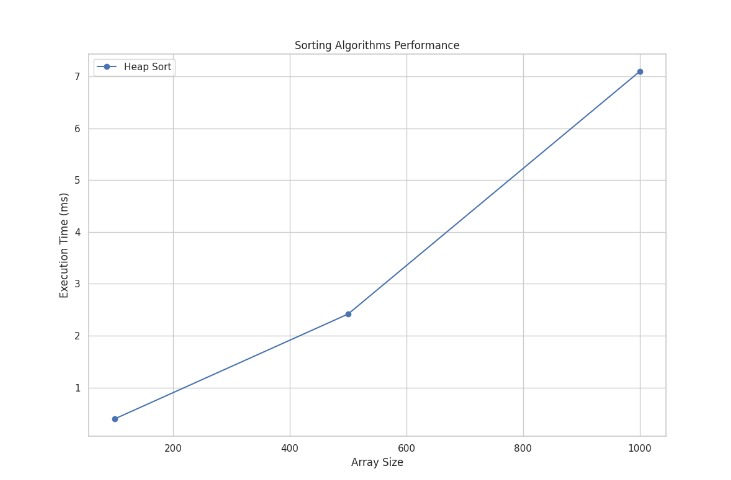
for i in range(N):

print("%d" % arr[i], end=" ")

# This code is contributed by Mohit Kumra

Time Complexity: O(n log n)

Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.



Bucket Sort

def insertion\_sort(bucket):

for i in range(1, len(bucket)):

key = bucket[i]

j = i - 1

while j >= 0 and bucket[j] > key:

bucket[j + 1] = bucket[j]

j -= 1

bucket[j + 1] = key

def bucket\_sort(arr):

n = len(arr)

buckets = [[] for \_ in range(n)]

# Put array elements in different buckets

for num in arr:

bi = int(n \* num)

buckets[bi].append(num)

# Sort individual buckets using insertion sort

for bucket in buckets:

insertion\_sort(bucket)

# Concatenate all buckets into arr[]

index = 0

for bucket in buckets:

for num in bucket:

arr[index] = num

index += 1

arr = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]

bucket\_sort(arr)

print("Sorted array is:")

print(" ".join(map(str, arr)))

Time Complexity: O(n2),

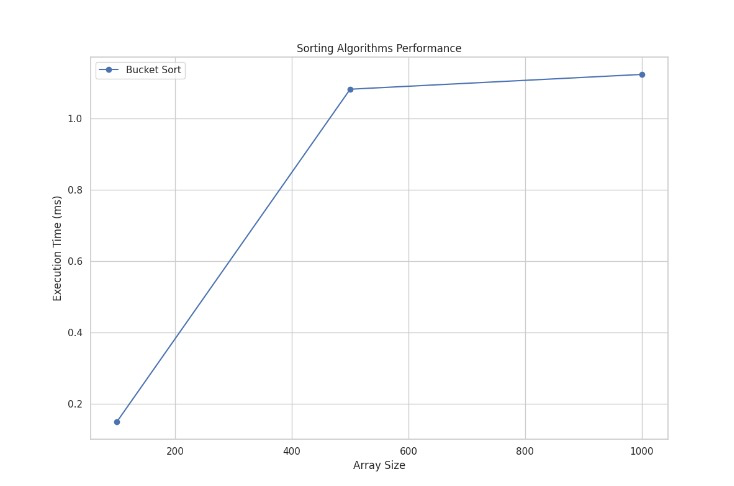
If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time.

The O(1) is easily possible if we use a linked list to represent a bucket.

Step 4 also takes O(n) time as there will be n items in all buckets.

The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed.

Auxiliary Space: O(n+k)



Radix Sort

# Python program for implementation of Radix Sort

# A function to do counting sort of arr[] according to

# the digit represented by exp.

def countingSort(arr, exp1):

n = len(arr)

# The output array elements that will have sorted arr

output = [0] \* (n)

# initialize count array as 0

count = [0] \* (10)

# Store count of occurrences in count[]

for i in range(0, n):

index = arr[i] // exp1

count[index % 10] += 1

# Change count[i] so that count[i] now contains actual

# position of this digit in output array

for i in range(1, 10):

count[i] += count[i - 1]

# Build the output array

i = n - 1

while i >= 0:

index = arr[i] // exp1

output[count[index % 10] - 1] = arr[i]

count[index % 10] -= 1

i -= 1

# Copying the output array to arr[],

# so that arr now contains sorted numbers

i = 0

for i in range(0, len(arr)):

arr[i] = output[i]

# Method to do Radix Sort

def radixSort(arr):

# Find the maximum number to know number of digits

max1 = max(arr)

# Do counting sort for every digit. Note that instead

# of passing digit number, exp is passed. exp is 10^i

# where i is current digit number

exp = 1

while max1 / exp >= 1:

countingSort(arr, exp)

exp \*= 10

# Driver code

arr = [170, 45, 75, 90, 802, 24, 2, 66]

# Function Call

radixSort(arr)

for i in range(len(arr)):

print(arr[i], end=" ")

# This code is contributed by Mohit Kumra

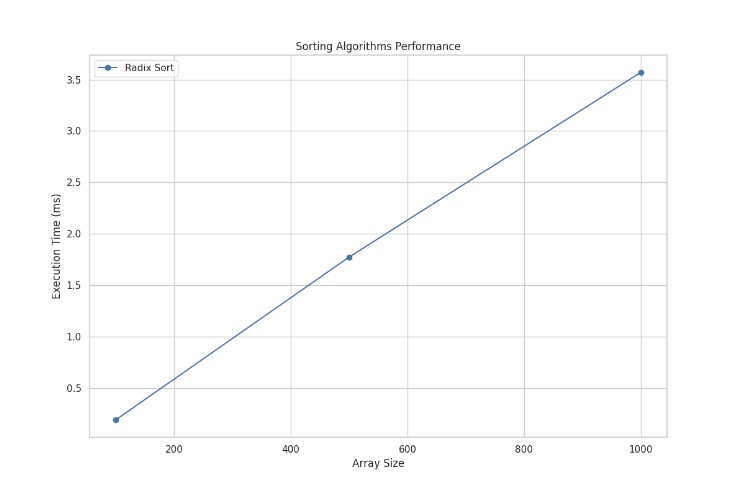
# Edited by Patrick Gallagher

Time Complexity:

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of O(d \* (n + b)), where d is the number of digits, n is the number of elements, and b is the base of the number system being used.

In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

Auxiliary Space:

Radix sort also has a space complexity of O(n + b), where n is the number of elements and b is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.  
  


Testbench for analyzing the sorting algorithms

import time

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# Import sorting algorithm classes

from q1 import QuickSort, ThreeWayMergeSort, HeapSort, BucketSort, RadixSortLinkedList

# Helper functions for measuring performance

def measure\_performance(sort\_class, arr):

"""Measures execution time, swaps, and comparisons of the sorting class."""

sorter = sort\_class() # Instantiate the sorting class

start\_time = time.time()

sorted\_arr, swaps, comparisons = sorter.sort(arr.copy())

end\_time = time.time()

execution\_time = (end\_time - start\_time) \* 1000 # Convert to milliseconds

return execution\_time, swaps, comparisons

def generate\_array(size):

"""Generates a random array of the given size."""

return np.random.randint(0, 10000, size).tolist()

def test\_sorting\_algorithm(sort\_class, sizes):

"""Tests the sorting algorithm and measures performance."""

times = []

for size in sizes:

arr = generate\_array(size)

# Measure performance

try:

exec\_time, swaps, comparisons = measure\_performance(sort\_class, arr)

times.append(exec\_time)

print(f"Size: {size}, Time: {exec\_time:.4f} ms, Swaps: {swaps}, Comparisons: {comparisons}")

except Exception as e:

print(f"Error sorting array of size {size}: {e}")

times.append(None) # Append None to indicate error

return times

def plot\_performance(sizes, times\_dict, filename="sorting\_performance.png"):

"""Plots performance results using Seaborn and saves the plot to a file."""

plt.figure(figsize=(12, 8))

sns.set\_theme(style="whitegrid")

for label, times in times\_dict.items():

plt.plot(sizes, times, marker='o', label=label)

plt.xlabel('Array Size')

plt.ylabel('Execution Time (ms)')

plt.title('Sorting Algorithms Performance')

plt.legend()

plt.grid(True)

plt.savefig(filename)

print(f"Plot saved as {filename}")

# Sizes to test

sizes = [100, 500, 1000]

# Test Quick Sort

print("Testing Quick Sort")

quick\_sort\_times = test\_sorting\_algorithm(QuickSort, sizes)

# Test 3-Way Merge Sort

print("Testing 3-Way Merge Sort")

merge\_sort\_times = test\_sorting\_algorithm(ThreeWayMergeSort, sizes)

# Test Heap Sort

print("Testing Heap Sort")

heap\_sort\_times = test\_sorting\_algorithm(HeapSort, sizes)

# Test Bucket Sort

print("Testing Bucket Sort")

bucket\_sort\_times = test\_sorting\_algorithm(BucketSort, sizes)

# Test Radix Sort

print("Testing Radix Sort")

def radix\_sort\_test(arr):

"""Wraps RadixSortLinkedList class for testing."""

if len(arr) > 25:

return arr # Avoid testing with large sizes

linked\_list = RadixSortLinkedList.LinkedList()

for num in arr:

linked\_list.append(num)

sorted\_linked\_list = RadixSortLinkedList().sort(linked\_list)

return sorted\_linked\_list.to\_list()

radix\_sort\_times = test\_sorting\_algorithm(lambda arr, sizes=sizes: radix\_sort\_test(arr), sizes=sizes)

# Plot the performance results

times\_dict = {

'Quick Sort': quick\_sort\_times,

'3-Way Merge Sort': merge\_sort\_times,

'Heap Sort': heap\_sort\_times,

'Bucket Sort': bucket\_sort\_times,

'Radix Sort': radix\_sort\_times

}

plot\_performance(sizes, times\_dict)

