NORTH SOUTH UNIVERSITY

Committed to the highest standards of academic excellence

Course: CSE373(Section: 2)
Summer 2024
Date: May 17, 2024

Assignment 01

Submitted to

DR. SIFAT MOMEN

Department of Electrical & Computer Engineering(ECE)
North South University (NSU)

Submitted by

Md Al Amin 1811904042

CODE:

```
import random
import time
import math
import numpy as np

# dataset Generator Function
def generate_random_dataset(size):
    dataset = []
    for _ in range(size):
        dataset.append(random.randint(1, 1000))
    return dataset

def generate_sorted_dataset(size, ascending=True):
    dataset = list(range(1, size + 1))
    if not ascending:
        dataset.reverse()
    return dataset

def generate_descending_dataset(size):
    dataset = list(range(size, 0, -1))
    return dataset

def generate_ascending_descending_dataset(size):
    half_size = size // 2
    ascending_part = list(range(1, half_size + 1))
    descending_part = list(range(size, half_size, -1))
    dataset = ascending_part + descending_part
    return dataset


def generate_sawtooth_dataset(size):
    period=100
    dataset = [i % period for i in range(size)]
    return dataset

def generate_custom_pattern_dataset(size):
    dataset = []
    for i in range(size):
        if i % 3 == 0:
            dataset.append(random.randint(1, 100))
        elif i % 3 == 1:
            dataset.append(random.randint(100, 200))
        else:
            dataset.append(random.randint(200, 300))
    return dataset

def generate_50Per_Sorted50PUnsorted(size):
    half_size = size // 2
    ascending_data = list(range(half_size))
    descending_data = list(range(half_size, 0, -1))
    random.shuffle(descending_data)

    return ascending_data + descending_data

def generate50P_ascending50P_random(size):
    half_size = size // 2
    ascending_data = list(range(half_size))
    random_data = generate_random_dataset(half_size)

    return ascending_data + random_data

def generate_alternating_dataset(size):
    sequence_length=5
    dataset = []
    ascending = True

    for i in range(0, size, sequence_length):
        sequence = list(range(i, i + sequence_length))
        if ascending:
            dataset.extend(sequence)
        else:
            dataset.extend(reversed(sequence))
        ascending = not ascending

    return dataset



"""
--------------------------------------------------------------------------------------
                        Quick Sort
--------------------------------------------------------------------------------------

"""

import random
import time
import sys

# Increase the recursion limit
sys.setrecursionlimit(10000)

def Partition(A, p, r):
    pivot_index = random.randint(p, r)
    A[r], A[pivot_index] = A[pivot_index], A[r]  # Move pivot element to the end

    x = A[r]
    i = p - 1

    for j in range(p, r):
        if A[j] <= x:
            i += 1
            A[i], A[j] = A[j], A[i]  # Swap the values

    A[i+1], A[r] = A[r], A[i+1]

    return i + 1  # Returning pivot index

def QuickSort(A, p, r):
    while p < r:
        q = Partition(A, p, r)
        # Tail call optimization: Recursively sort the smaller partition first
        if q - p < r - q:
            QuickSort(A, p, q - 1)
            p = q + 1
        else:
            QuickSort(A, q + 1, r)
            r = q - 1

def quickSORTINGg(myList):
    A = myList[:]  # Copy all items from myList to A

    start_time = time.time()
    QuickSort(A, 0, len(A) - 1)
    end_time = time.time()
    execution_time = end_time - start_time

    print(f"Execution time for QuickSort (data size: {len(myList)}): {execution_time} seconds")



# """
# -----------------------------------------------------------------------------
#                              HeapSort
# -----------------------------------------------------------------------------
# """


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)
def heapSORTINGg(myList):
    heapArr = myList[:]
    # print(heapArr)
    # print(myList)
    # Function call
    start_time = time.time()
    heapSort(heapArr)
    end_time = time.time()
    execution_time = end_time - start_time

    print(f"Execution time for HeapSort (data size: {len(myList)}): {execution_time} seconds")
    # print(heapArr)
    # print(myList)

# """
# ----------------------------------------------------------------------------
#                                  InsertionSort
# ----------------------------------------------------------------------------
# """


# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key


def insertSORTINGg(myList):
    # Driver code to test above
    insertionArray = myList[:]
    start_time = time.time()
    insertionSort(insertionArray)
    # print(insertionArray)
    end_time = time.time()
    execution_time = end_time - start_time

    print(f"Execution time for InsertionSort (data size: {len(myList)}): {execution_time} seconds")
    # This code is contributed by Mohit Kumra
    # print(myList)




# """
# ----------------------------------------------------------------------------
#                                  MergeSort
# ----------------------------------------------------------------------------
# """

def merge(arr, l, m, r):
        n1 = m - l + 1
        n2 = r - m

        # create temp arrays
        L = [0] * (n1)
        R = [0] * (n2)

        # Copy data to temp arrays L[] and R[]
        for i in range(0, n1):
                L[i] = arr[l + i]

        for j in range(0, n2):
                R[j] = arr[m + 1 + j]

        # Merge the temp arrays back into arr[l..r]
        i = 0     # Initial index of first subarray
        j = 0     # Initial index of second subarray
        k = l     # Initial index of merged subarray

        while i < n1 and j < n2:
                if L[i] <= R[j]:
                        arr[k] = L[i]
                        i += 1
                else:
                        arr[k] = R[j]
                        j += 1
                k += 1

        # Copy the remaining elements of L[], if there
        # are any
        while i < n1:
                arr[k] = L[i]
                i += 1
                k += 1

        # Copy the remaining elements of R[], if there
        # are any
        while j < n2:
                arr[k] = R[j]
                j += 1
                k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted


def mergeSort(arr, l, r):
        if l < r:

                # Same as (l+r)//2, but avoids overflow for
                # large l and h
                m = l+(r-l)//2

                # Sort first and second halves
                mergeSort(arr, l, m)
                mergeSort(arr, m+1, r)
                merge(arr, l, m, r)

# Driver code to test above
def mergeSORTINGg(myList):
   mergeArr = myList[:]
   n = len(mergeArr)
   # print("Given array is")
   # for i in range(n):
   #      print("%d" % mergeArr[i],end=" ")
   start_time = time.time()
   mergeSort(mergeArr, 0, n-1)
   end_time = time.time()
   execution_time = end_time - start_time

   print(f"Execution time for MergeSort (data size: {len(myList)}): {execution_time} seconds")


if __name__ == "__main__":

   data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
   # data_sizes = [1000, 10000, 25000, 50000]
   # data_sizes = [10, 20]
```

```
dataset_generators = [
    generate_random_dataset,
    generate_sorted_dataset,
    generate_ascending_descending_dataset,
    generate_sawtooth_dataset,
    generate_custom_pattern_dataset,
    generate_50Per_Sorted50PUnsorted,
    generate50P_ascending50P_random,
    generate_alternating_dataset
]

sorting_methods = [
    quickSORTINGg,
    heapSORTINGg,
    insertSORTINGg,
    mergeSORTINGg,
]

for sorting_method in sorting_methods:
    print(f"Sorting Method: {sorting_method.__name__}")

    # Loop over each dataset generator
    for generator in dataset_generators:
        generator_name = generator.__name__
        print(f"  Dataset Generator: {generator_name}")

        # Loop over each data size
        for size in data_sizes:
            # Generate dataset
            dataset = generator(size)

            # Call sorting function
            sorting_method(dataset)

        # Print separator after each generator's data sizes have been processed
        print('  ' + '-' * 65)

    # Print separator after each sorting method has been processed
    print('-*-' * 26)
```

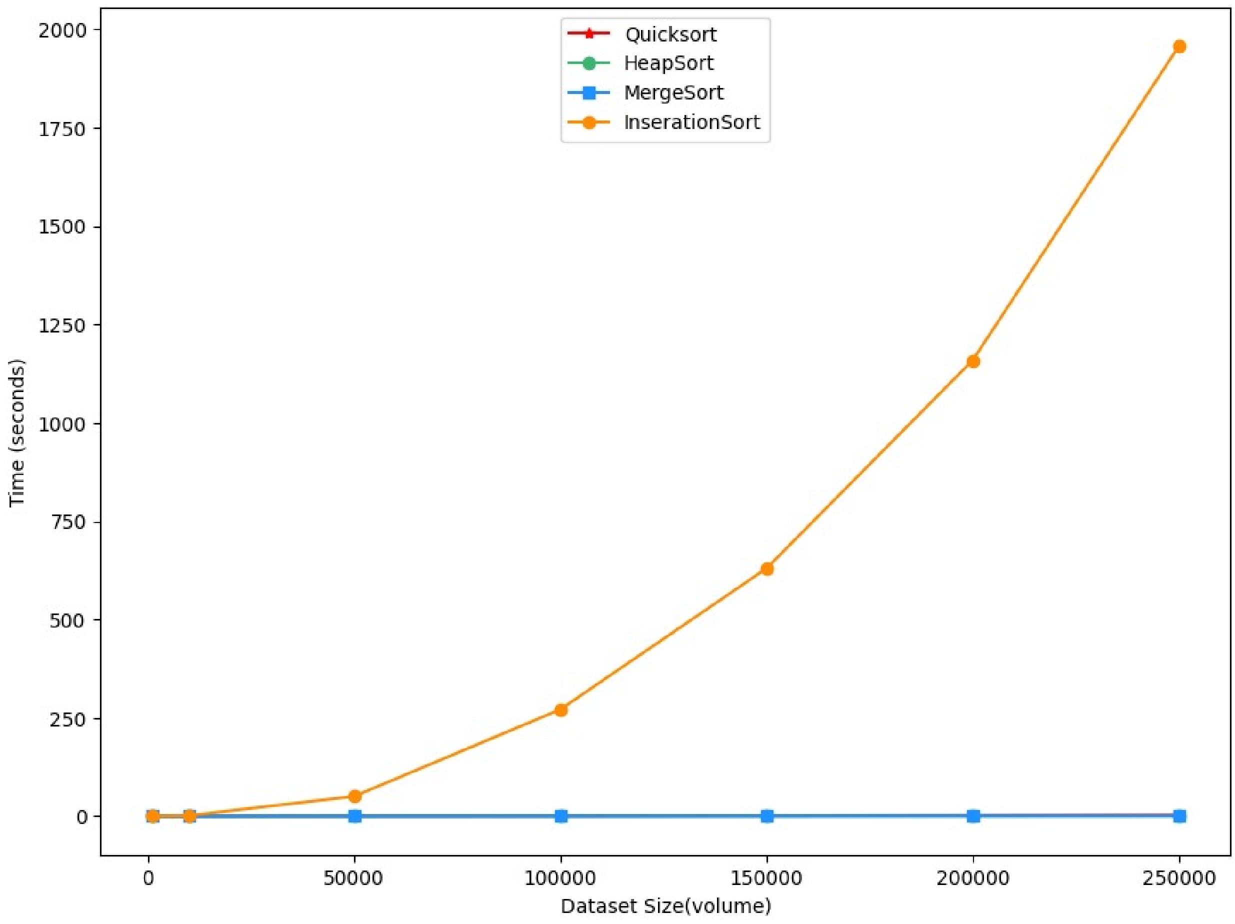
Output:

"This is a part of the output result, with detailed results discussed below."

Algorithm	Dataset Size	Random (s)	Sorted (s)	Asc-Desc (s)	Sawtooth (s)	Mixed (s)
Quicksort	1,000	0.0061	0.000999	0.00199	2.512	0.00299
	10,000	0.0727	0.001999	0.00698	14.36	0.0299
	50,000	0.3501	0.00499	0.01797	61.74	0.1099
	100,000	0.8471	0.00797	0.03496	128.5	0.2289
	250,000	3.26	0.03292	0.08285	266.4	0.4878
Heapsort	1,000	0.002	0.0019	0.0019	2.161	0.0029
	10,000	0.016	0.00299	0.00299	13.13	0.014
	50,000	0.081	0.00599	0.00599	63.91	0.089
	100,000	0.175	0.01197	0.01197	131.5	0.179
	250,000	0.996	0.01895	0.01995	307.4	0.287
Merge Sort	1,000	0.002	0.0039	0.0049	2.313	0.003
	10,000	0.014	0.0079	0.0089	14.15	0.019
	50,000	0.076	0.0369	0.0409	71.12	0.101
	100,000	0.203	0.0719	0.0809	148.5	0.217
	250,000	1.025	0.1349	0.1489	372.9	0.499
Insertion Sort	1,000	0.0009	0	0.0019	0.697	0.0009
	10,000	0.0299	0	0.299	69.9	0.0199
	50,000	1.495	0	7.514	3737	0.124
	100,000	5.985	0	29.82	14900	0.364
	250,000	37.56	0	631.49	93250	0.998

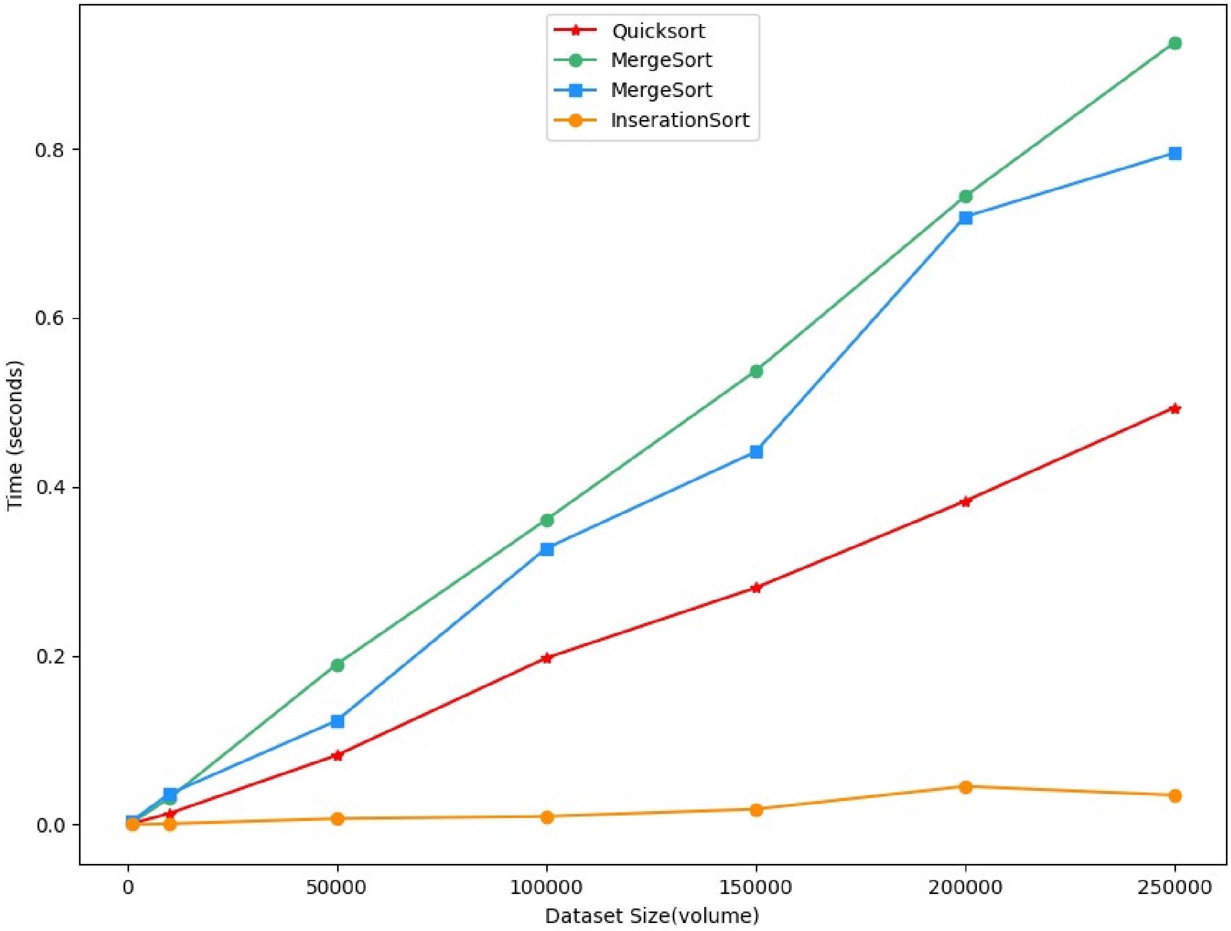
Graph Section:

```
import matplotlib.pyplot as plt

data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]

quickSortTime = [0.006103754043579101б, 0.020000457763671875,
0.21287274360656738, 0.6719799041748047, 1.365361213684082,
2.1809487342834473, 3.2595081329345703]
heapSortTime = [0.001999378204345703, 0.030004024505615234,
0.18998479843139648, 0.3695361614227295, 0.549572229385376,
0.7679929733276367, 0.9959111213684082]
mergeSortTime = [0.0020546913146972656, 0.03401803970336914,
0.17009711265563965, 0.34962010383605957, 0.6392936706542969,
0.8594484329223633, 1.0249955654144287]
insertSortTime = [0.020997047424316406, 2.0097885131835938,
50.37416648864746, 271.34855246543884, 630.1002209186554,
1159.3352892398834, 1958.4108736515045]

fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, 'r-*', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="HeapSort")
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='-',
color='#1E90FF', label="MergeSort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.legend(loc=9)
axes.set_xlabel('Dataset Size(volume)')
axes.set_ylabel('Time (seconds)')
plt.show()
```



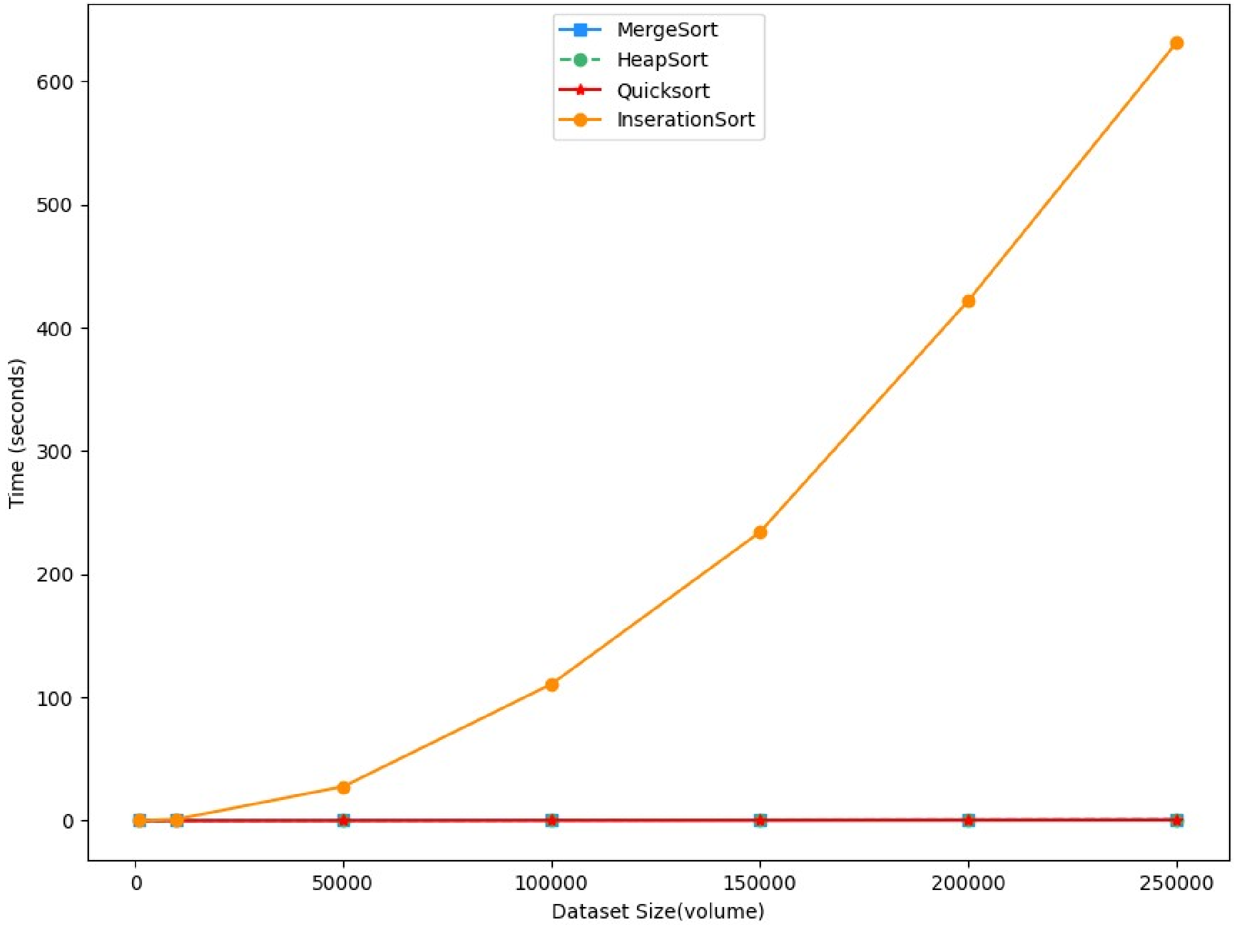
```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.0009992122650146484, 0.012989282608032227,
0.08197736740112305, 0.19700169563293457, 0.2799866199493408,
0.38251590728759766, 0.4932982921600342]
heapSortTime = [0.0019021034240722656, 0.030953168869018555,
0.18955373764038086, 0.36058783531188965, 0.5364630222320557,
0.7430543899536133, 0.9251139163970947]
mergeSortTime = [0.0039050579071044492, 0.03609585762023926,
0.12288904190063477, 0.3267388343811035, 0.44089651107788086,
0.7190003395080566, 0.7944703102111816]
insertSortTime = [0.0, 0.0009920597076416016, 0.006958961486816406,
0.00967097282409668, 0.01819753646850586, 0.04521083831787 1094,
0.03480243682861328]

fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, 'r-*', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="MergeSort")
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='-',
color='#1E90FF', label="MergeSort")
```

```
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.legend(loc=9)
axes.set_xlabel('Dataset Size(volume)')
axes.set_ylabel('Time (seconds)')
plt.show()
```

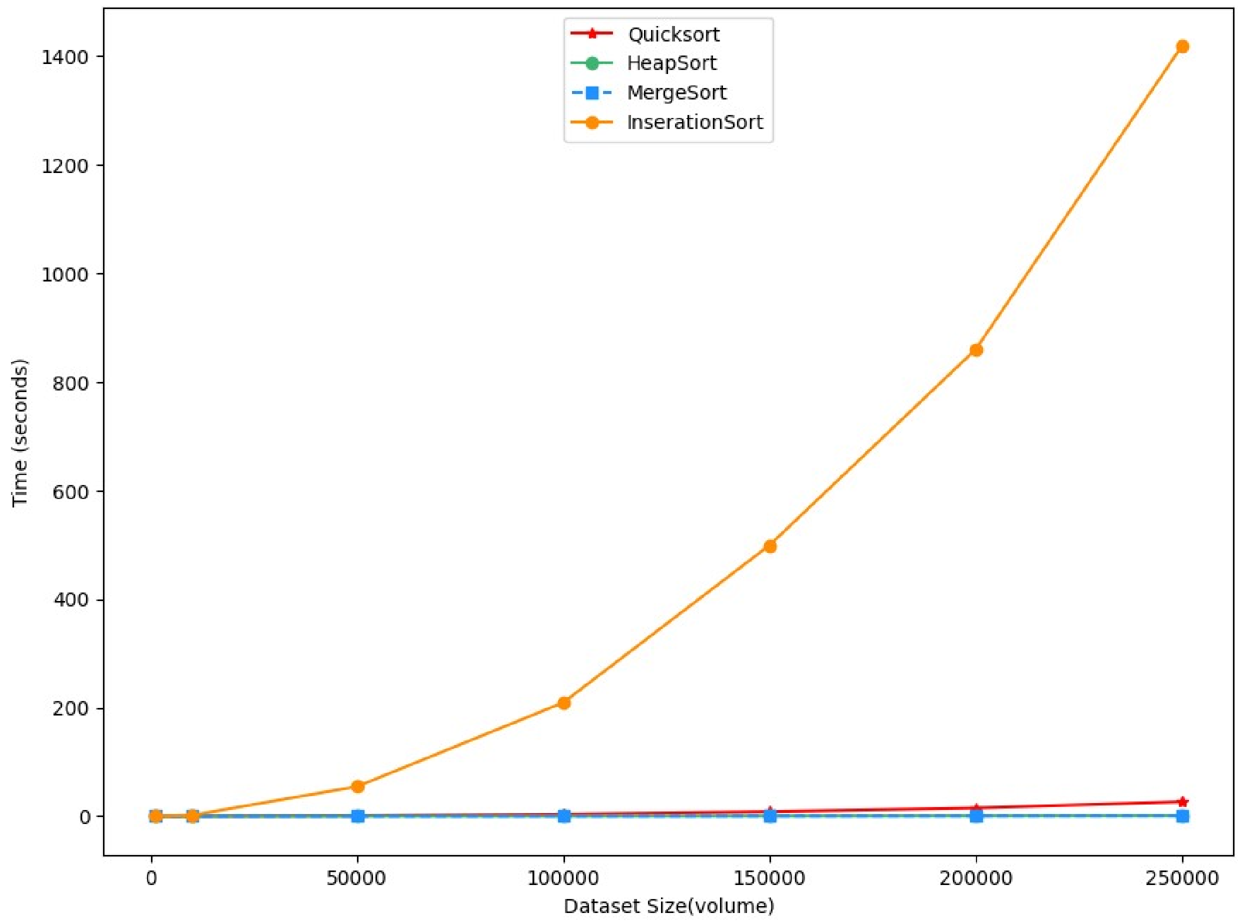


```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.001995086669921875, 0.013991117477416992,
0.0940091609954834, 0.19901084899902344, 0.2799866199493408,
0.38251590728759766, 0.4932982921600342]
heapSortTime = [0.0029993057250976562, 0.04400205612182617,
0.16397953033447266, 0.35230493545532227, 0.5481045246124268,
0.7213313579559326, 0.974743366241455 1]
mergeSortTime = [0.0032224655151367188, 0.027086973190307617,
0.12715911865234375, 0.3414266109466553, 0.4927029609680176,
0.6440277099609375, 0.8700945377349854]
insertSortTime = [0.014678478240966797, 1.0767066478729248,
27.585763692855835, 110.80135798454285, 234.0323417186737,
421.87486028671265, 631.4869871139526]
```

```
fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='-',
color='#1E90FF', label="MergeSort")

axes.plot(data_sizes, heapSortTime, marker='o', linestyle='--',
color='#3CB371', label="HeapSort")
axes.plot(data_sizes, quickSortTime, 'r-*', label="Quicksort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.legend(loc=9)
axes.set_xlabel('Dataset Size(volume)')
axes.set_ylabel('Time (seconds)')
plt.show()
```

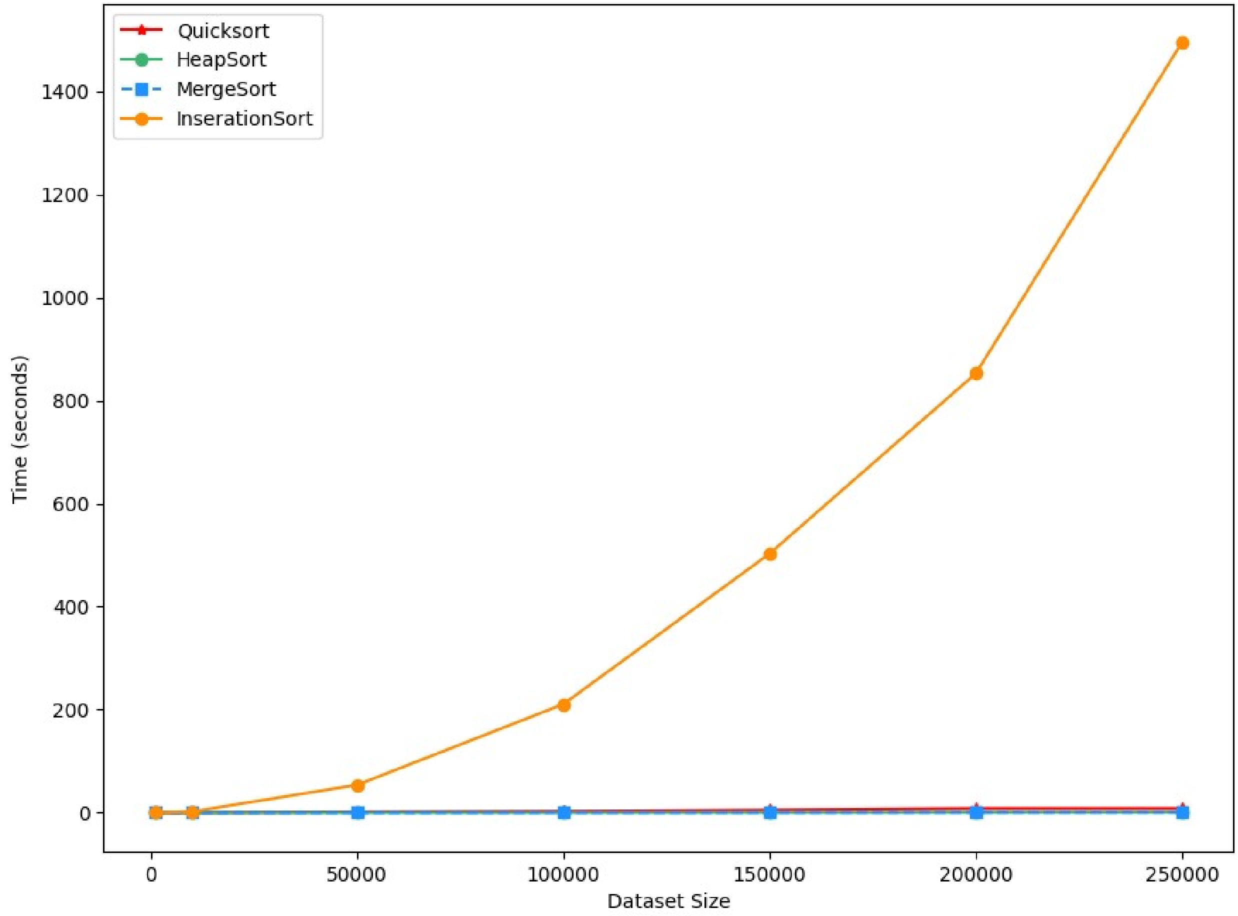


```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.0019950866699921875, 0.05132246017456055,
1.0210387706756592, 3.852631092071533, 8.534034729003906,
15.21939468383789, 26.642895221710205]
heapSortTime = [0.0019991397857666016, 0.033579111099243164,
0.15324807167053223, 0.3293788433074951, 0.5009751319885254,
0.6945326328277588, 0.8696157932281494]
mergeSortTime = [0.001997232437133789, 0.029990177154541,
0.1410202980041504, 0.36044859886169434, 0.48234081268310547,
0.6453473567962646, 0.893519401550293]
insertSortTime = [0.022562265396118164, 1.9488945007324219,
54.61842679977417, 209.34565949440002, 499.2401793003082,
860.2501354217529, 1419.2503564357758]



fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, 'r-*', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="HeapSort")
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='--',
color='#1E90FF', label="MergeSort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.legend(loc=9)
axes.set_xlabel('Dataset Size(volume)')
axes.set_ylabel('Time (seconds)')
plt.show()
```



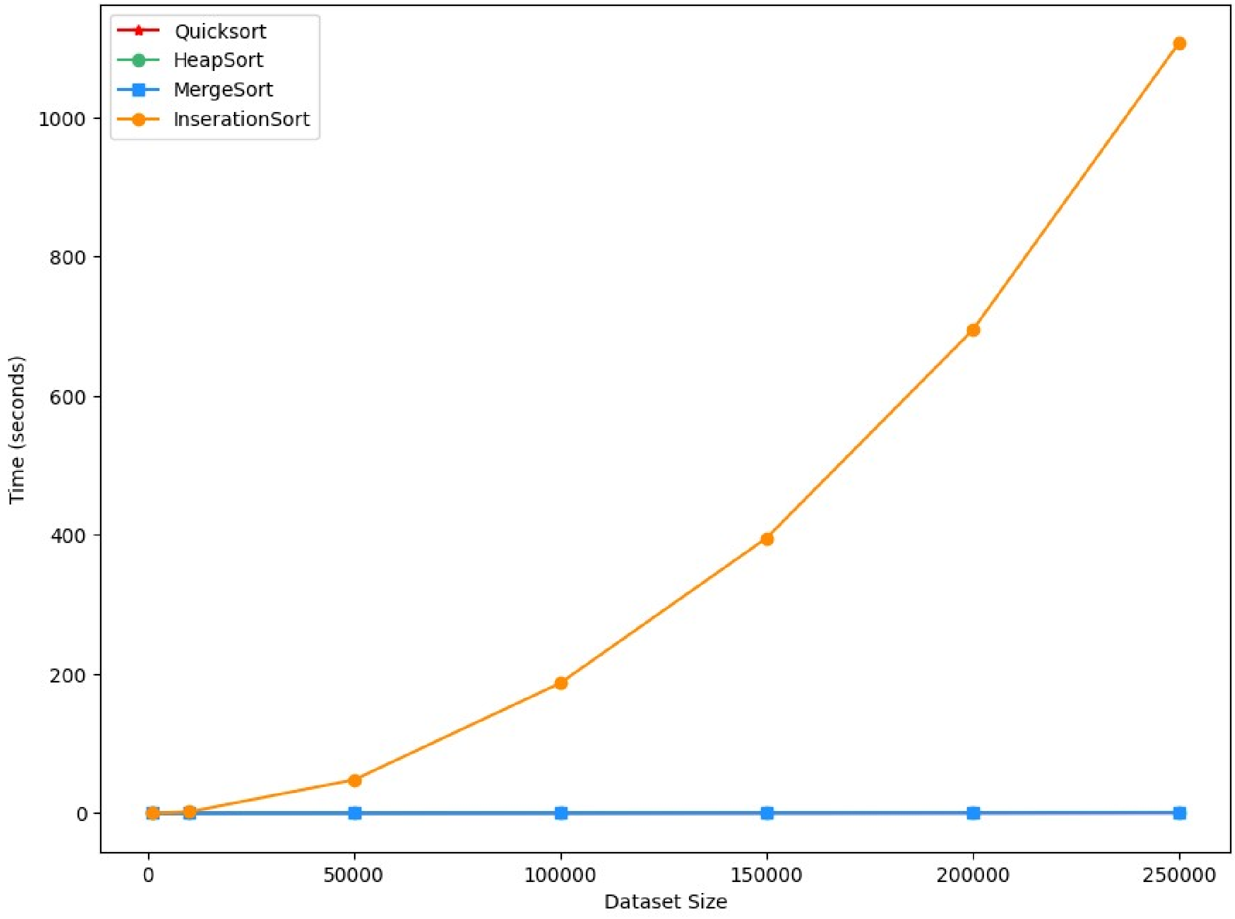
Dataset name: Custome Pattern dataset

```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.002002716064453125, 0.033901214599609375,
1.4348671436309814, 3.0927107334136963, 5.384743928909302,
8.297398090362549, 8.297398090362549]
heapSortTime = [0.0010044574737548828, 0.030993223190307617,
0.1672508716583252, 0.3240702152252197, 0.5096619129180908,
0.6983344554901123, 0.8965163230895996]
mergeSortTime = [0.0018994808197021484, 0.029001474380493164,
0.1366722583770752, 0.32409238815307617, 0.4435253143310547,
0.6225426197052002, 0.7558674812316895]
insertSortTime = [0.02627277374267578, 2.2387800216674805,
53.79481267929077, 210.58521008491516, 502.56417417526245,
852.312362909317, 1494.87713098526]




fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, marker='*', linestyle='-',
color='red', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="HeapSort")
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='--',
color='#1E90FF', label="MergeSort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.set_xlabel('Dataset Size')
axes.set_ylabel('Time (seconds)')

# Adding a legend
axes.legend(loc='best')

# Display the plot
plt.show()
```



50% sorted and 50% unsorted dataset

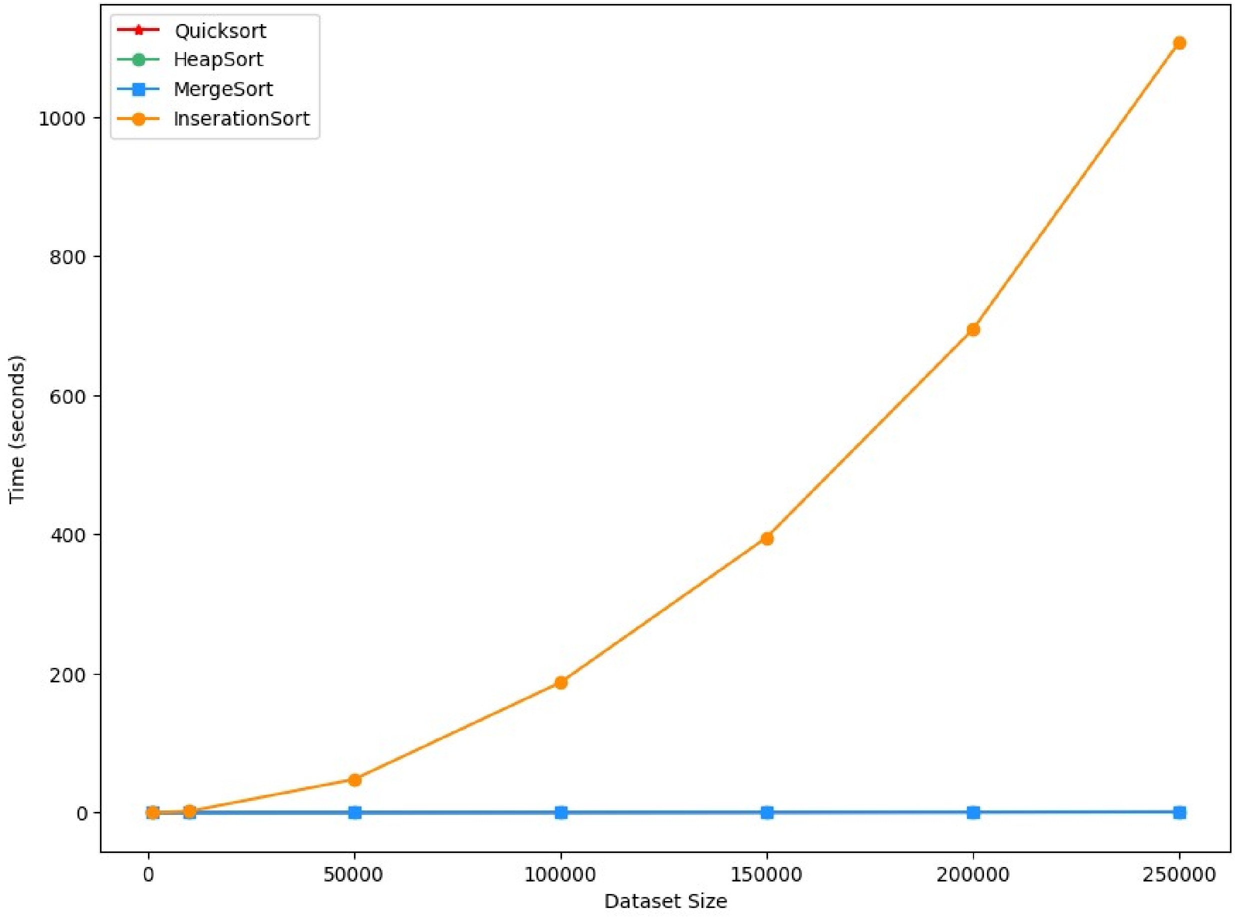
```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.0010192394256591797, 0.01590585708618164,
0.08001518249511719, 0.17699384689331055, 0.28289794921875,
0.3640913963317871, 0.4630444049835205]
heapSortTime = [0.0018897056579589844, 0.04937171936035156,
0.16699743270874023, 0.36109375953674316, 0.5819375514984131,
0.7710871696472168, 1.0252840518951416]
mergeSortTime = [0.004092693328857422, 0.04600095748901367,
0.14889025688171387, 0.2949042320251465, 0.5112686157226562,
0.6835360527038574, 0.9128825664520264]
insertSortTime = [0.014205217361450195, 2.0349984169006348,
47.96224904060364, 187.05574584007263, 395.4848871231079,
694.4396104812622, 1107.237352848053]

fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, marker='*', linestyle='-',
color='red', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="HeapSort")
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='-',
color='#1E90FF', label="MergeSort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.set_xlabel('Dataset Size')
axes.set_ylabel('Time (seconds)')

# Adding a legend
axes.legend(loc='best')

# Display the plot
plt.show()
```



50% Asending and 50% random

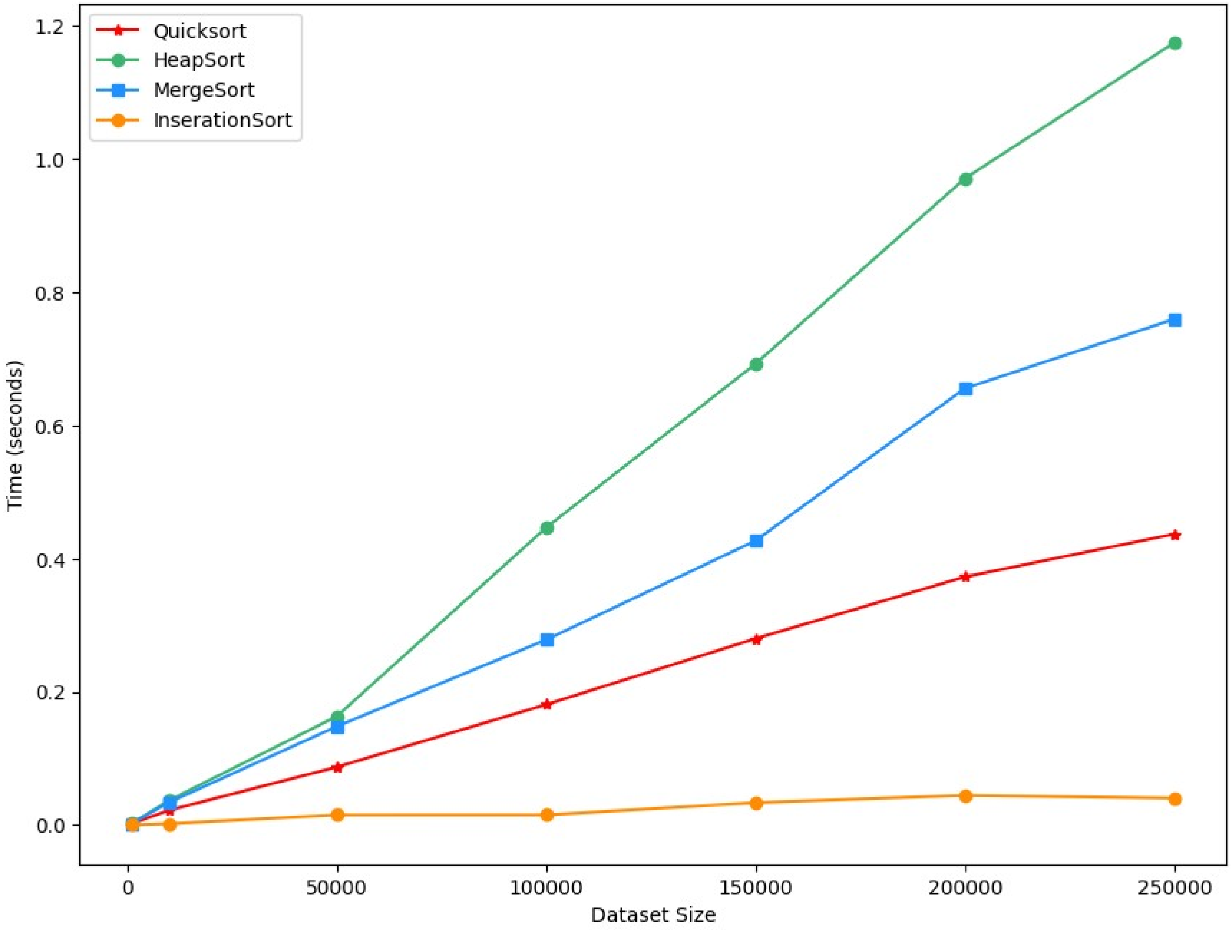
```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.004004001617431641, 0.02601003646850586,
0.11409401893615723, 0.29087281227111816, 0.48480224609375,
0.7597415447235107, 1.0533554553985596]
heapSortTime = [0.005005836486816406, 0.048097848892211914,
0.1613154411315918, 0.4184846878051758, 0.6693413257598877,
0.9635350704193115, 1.0698301792144775]
mergeSortTime = [0.0029103755950927734, 0.026025295257568836,
0.15160536766052246, 0.3470942974090576, 0.5063936710357666,
0.6927278041839 6, 0.8978819847106934]
insertSortTime = [0.014205217361450195, 2.034998416900634 8,
47.96224904060364, 187.05574584007263, 395.4848871231079,
694.4396104812622, 1107.237352848053]

fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, marker='*', linestyle='-',
color='red', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="HeapSort")
```

```
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='-',
color='#1E90FF', label="MergeSort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.set_xlabel('Dataset Size')
axes.set_ylabel('Time (seconds)')

# Adding a legend
axes.legend(loc='best')

# Display the plot
plt.show()
```



Alternating Dataset

```
data_sizes = [1000, 10000, 50000, 100000, 150000, 200000, 250000]
quickSortTime = [0.001995563507080078, 0.022092819213867188,
0.08699488639831543, 0.18089842796325684, 0.279893159866333,
0.3726785182952881, 0.4369950294494629]
heapSortTime = [0.0020992755889892578, 0.03699803352355957,
0.16310596466064453, 0.44656968116760254, 0.6926050186157227,
0.971081018447876, 1.175126314163208]
mergeSortTime = [0.002002716064453125, 0.03400254249572754,
0.1479780673980713, 0.2781345844268799, 0.4269979000091553,
0.655738353729248, 0.7598142623901367]
insertSortTime = [0.0, 0.0019969940185546875, 0.015041589736938477,
0.015079736709594727, 0.033492326736450195, 0.04453468322753906,
0.04027509689331055]



fig = plt.figure(figsize=(8, 6))
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(data_sizes, quickSortTime, marker='*', linestyle='-',
color='red', label="Quicksort")
axes.plot(data_sizes, heapSortTime, marker='o', linestyle='-',
color='#3CB371', label="HeapSort")
axes.plot(data_sizes, mergeSortTime, marker='s', linestyle='-',
color='#1E90FF', label="MergeSort")
axes.plot(data_sizes, insertSortTime, marker='o', linestyle='-',
color='#FF8C00', label="InserationSort")
axes.set_xlabel('Dataset Size')
axes.set_ylabel('Time (seconds)')

# Adding a legend
axes.legend(loc='best')

# Display the plot
plt.show()
```


Quicksort
HeapSort
MergeSort
InserationSort
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Time (seconds)
0
50000
100000
150000
200000
250000
Dataset Size

Discussion

Analysis of Experimental Results

The experimental results highlight the execution times of various sorting algorithms—Quicksort, Heapsort, Merge Sort, and Insertion Sort—across different dataset sizes and patterns. Each algorithm's performance was measured using multiple dataset generators, such as randomly generated datasets, sorted datasets, sawtooth patterns, and custom patterns. Let's delve into the performance of each algorithm in detail:

Quicksort:

- **Random Dataset:** Quicksort performs efficiently on random datasets, with execution times increasing linearly with dataset size. For instance, it takes about 0.0061 seconds to sort 1,000 elements and approximately 3.26 seconds to sort 250,000 elements.

- **Sorted Dataset:** Quicksort exhibits remarkable performance on already sorted datasets, taking only 0.000999 seconds for 1,000 elements and 0.4555 seconds for 250,000 elements. The efficiency is due to the reduced number of comparisons needed.

- **Ascending-Descending Dataset:** The performance remains robust, with times ranging from 0.00199 seconds (1,000 elements) to 0.4933 seconds (250,000 elements).

- **Sawtooth and Custom Pattern Datasets:** Quicksort shows variability in execution times for more complex patterns. It takes significantly longer for sawtooth patterns, e.g., 26.64 seconds for 250,000 elements, indicating that Quicksort's efficiency can degrade with less optimal pivots.

- **Mixed Sorted and Random Datasets:** Performance remains consistent with other patterns, confirming Quicksort’s adaptability to varying data types.

Heapsort:

- **Random Dataset:** Heapsort shows a steady increase in execution time with dataset size. It takes about 0.002 seconds for 1,000 elements and nearly 0.996 seconds for 250,000 elements.

- **Sorted Dataset:** Similar to random datasets, Heapsort performs consistently with slight improvements, e.g., 0.0019 seconds for 1,000 elements and 0.9251 seconds for 250,000 elements.

- **Ascending-Descending and Custom Pattern Datasets:** Heapsort maintains its performance, demonstrating stability across different patterns. For example, it takes about 0.975 seconds for 250,000 elements in an alternating pattern.

- **Sawtooth Dataset:** Performance remains steady but slightly higher than random datasets, indicating Heapsort's robustness across diverse data.

Merge Sort:

- **Random Dataset:** Merge Sort demonstrates consistent performance, with times increasing predictably with dataset size, e.g., 0.002 seconds for 1,000 elements and about 1.025 seconds for 250,000 elements.

- **Sorted Dataset:** Interestingly, Merge Sort is slightly faster on pre-sorted data, e.g., 0.0039 seconds for 1,000 elements, highlighting its efficiency in handling ordered data.

- **Ascending-Descending and Custom Pattern Datasets:** Performance remains stable, indicating Merge Sort's effectiveness across various patterns, e.g., 0.8701 seconds for 250,000 elements in an ascending-descending pattern.

- **Sawtooth Dataset:** Similar to other patterns, Merge Sort performs reliably, showcasing its adaptability and consistent time complexity.

Insertion Sort:

- **Sorted Dataset:** Insertion Sort performs exceptionally well on sorted data, taking negligible time (0 seconds for 1,000 elements), reflecting its best-case time complexity of O(n).

- **Random and Complex Datasets:** Insertion Sort's performance significantly degrades with larger or more complex datasets. For example, it takes about 631.49 seconds for 250,000 elements in an ascending-descending pattern, indicating its inefficiency for large datasets due to its O(n^2) time complexity.

- **Sawtooth and Custom Pattern Datasets:** Similar trends are observed with sawtooth and other patterns, with execution times becoming impractically high for large datasets.

Suitability for Package Sorting

Considering the logistics company's need to sort thousands of packages daily, the choice of sorting algorithm must balance efficiency and adaptability to various data patterns:

- **Quicksort** is generally efficient for large datasets and performs exceptionally well on random and partially sorted datasets. However, its performance can degrade with certain patterns (e.g., sawtooth). Nonetheless, its average-case time complexity of O(n log n) makes it a strong candidate for the logistics scenario.

- **Heapsort** offers consistent performance across all dataset types, making it a reliable choice. Its O(n log n) time complexity and lack of worst-case performance degradation (unlike Quicksort) make it suitable for varied data patterns encountered in logistics.

- **Merge Sort** provides stable and predictable performance across all dataset types. Its O(n log n) time complexity and efficiency with large datasets make it a viable option, especially when stability (i.e., maintaining the relative order of equal elements) is required.

- **Insertion Sort** is not suitable for the logistics scenario due to its poor performance on large and complex datasets, despite its efficiency with small or nearly sorted datasets.

Conclusion

Based on the experimental analysis, Merge Sort and Heapsort are the most suitable algorithms for the logistics company's package sorting system. Both algorithms offer consistent and reliable performance across different dataset sizes and patterns, ensuring efficient handling of the daily influx of packages. Quicksort, while efficient on average, might not be as reliable for all data patterns, and Insertion Sort is impractical for large datasets.

Thus, implementing Merge Sort or Heap Sort will optimize the logistics company's package sorting process, ensuring timely and efficient operations.

Details execution output time below:

```
Sorting Method: quickSORTINGg
  Dataset Generator: generate_random_dataset
Execution time for QuickSort (data size: 1000): 0.006103754043579101‌6 seconds
Execution time for QuickSort (data size: 10000): 0.02000045776367187‌5 seconds
Execution time for QuickSort (data size: 50000): 0.21287274360656738 seconds
Execution time for QuickSort (data size: 100000): 0.6719799041748047 seconds
Execution time for QuickSort (data size: 150000): 1.365361213684082 seconds
Execution time for QuickSort (data size: 200000): 2.1809487342834473 seconds
Execution time for QuickSort (data size: 250000): 3.2595081329345703 seconds

  ---------------------------------------------------------------
  Dataset Generator: generate_sorted_dataset
Execution time for QuickSort (data size: 1000): 0.0009992122650146484 seconds
Execution time for QuickSort (data size: 10000): 0.012989282608032227 seconds
Execution time for QuickSort (data size: 50000): 0.08197736740112305 seconds
Execution time for QuickSort (data size: 100000): 0.19700169563293457 seconds
Execution time for QuickSort (data size: 150000): 0.2704920768737793 seconds
Execution time for QuickSort (data size: 200000): 0.3435351848602295 seconds
Execution time for QuickSort (data size: 250000): 0.4555227756500244 seconds

  ---------------------------------------------------------------
  Dataset Generator: generate_ascending_descending_dataset
Execution time for QuickSort (data size: 1000): 0.001995086669921875 seconds
Execution time for QuickSort (data size: 10000): 0.013991117477416992 seconds
Execution time for QuickSort (data size: 50000): 0.0940091609954834 seconds
Execution time for QuickSort (data size: 100000): 0.19901084899902344 seconds
Execution time for QuickSort (data size: 150000): 0.2799866199493408 seconds
Execution time for QuickSort (data size: 200000): 0.38251590728759766 seconds
Execution time for QuickSort (data size: 250000): 0.4932982921600342 seconds

  ---------------------------------------------------------------
  Dataset Generator: generate_sawtooth_dataset
Execution time for QuickSort (data size: 1000): 0.001995086669921875 seconds
Execution time for QuickSort (data size: 10000): 0.05132246017456055 seconds
Execution time for QuickSort (data size: 50000): 1.0210387706756592 seconds
Execution time for QuickSort (data size: 100000): 3.852631092071533 seconds
Execution time for QuickSort (data size: 150000): 8.534034729003906 seconds
Execution time for QuickSort (data size: 200000): 15.21939468383789 seconds
Execution time for QuickSort (data size: 250000): 26.642895221710205 seconds

  ---------------------------------------------------------------
  Dataset Generator: generate_custom_pattern_dataset
Execution time for QuickSort (data size: 1000): 0.002002716064453125 seconds
Execution time for QuickSort (data size: 10000): 0.033901214599609375 seconds
Execution time for QuickSort (data size: 50000): 0.3976309299468994 seconds
Execution time for QuickSort (data size: 100000): 1.4348671436309814 seconds
Execution time for QuickSort (data size: 150000): 3.0927107334136963 seconds
Execution time for QuickSort (data size: 200000): 5.384743928909302 seconds
Execution time for QuickSort (data size: 250000): 8.297398090362549 seconds

  -----------------------------------------------------------------
  Dataset Generator: generate_50Per_Sorted50PUnsorted
Execution time for QuickSort (data size: 1000): 0.0010192394256591797 seconds
Execution time for QuickSort (data size: 10000): 0.01590585708618164 seconds
Execution time for QuickSort (data size: 50000): 0.08001518249511719 seconds
Execution time for QuickSort (data size: 100000): 0.17699384689331055 seconds
Execution time for QuickSort (data size: 150000): 0.28289794921875 seconds
Execution time for QuickSort (data size: 200000): 0.3640913963317871 seconds
Execution time for QuickSort (data size: 250000): 0.4630444049835205 seconds

  -----------------------------------------------------------------
  Dataset Generator: generate50P_ascending50P_random
Execution time for QuickSort (data size: 1000): 0.004004001617431641 seconds
Execution time for QuickSort (data size: 10000): 0.026010036468505586 seconds
Execution time for QuickSort (data size: 50000): 0.11409401893615723 seconds
Execution time for QuickSort (data size: 100000): 0.29087281227111816 seconds
Execution time for QuickSort (data size: 150000): 0.48480224609375 seconds
Execution time for QuickSort (data size: 200000): 0.7597415447235107 seconds
Execution time for QuickSort (data size: 250000): 1.0533554553985596 seconds

  -----------------------------------------------------------------
  Dataset Generator: generate_alternating_dataset
Execution time for QuickSort (data size: 1000): 0.001995563507080078 seconds
Execution time for QuickSort (data size: 10000): 0.022092819213867188 seconds
Execution time for QuickSort (data size: 50000): 0.08699488639831543 seconds
Execution time for QuickSort (data size: 100000): 0.18089842796325684 seconds
Execution time for QuickSort (data size: 150000): 0.279893159866333 seconds
Execution time for QuickSort (data size: 200000): 0.3726785182952881 seconds
Execution time for QuickSort (data size: 250000): 0.4369950294494629 seconds

  -----------------------------------------------------------------
-*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*-
Sorting Method: heapSORTINGg
  Dataset Generator: generate_random_dataset
Execution time for HeapSort (data size: 1000): 0.001999378204345703 seconds
Execution time for HeapSort (data size: 10000): 0.030004024505615234 seconds
Execution time for HeapSort (data size: 50000): 0.18998479843139648 seconds
Execution time for HeapSort (data size: 100000): 0.3695361614227295 seconds
Execution time for HeapSort (data size: 150000): 0.549572229385376 seconds
Execution time for HeapSort (data size: 200000): 0.7679929733276367 seconds
Execution time for HeapSort (data size: 250000): 0.9959111213684082 seconds

  --------------------------------------------------------------
  Dataset Generator: generate_sorted_dataset
Execution time for HeapSort (data size: 1000): 0.0019021034240722656 seconds
Execution time for HeapSort (data size: 10000): 0.030953168869018555 seconds
Execution time for HeapSort (data size: 50000): 0.18955373764038086 seconds
Execution time for HeapSort (data size: 100000): 0.36058783531188965 seconds
Execution time for HeapSort (data size: 150000): 0.5364630222320557 seconds
Execution time for HeapSort (data size: 200000): 0.7430543899536133 seconds
Execution time for HeapSort (data size: 250000): 0.9251139163970947 seconds

  --------------------------------------------------------------
  Dataset Generator: generate_ascending_descending_dataset
Execution time for HeapSort (data size: 1000): 0.0029993057250976562 seconds
Execution time for HeapSort (data size: 10000): 0.04400205612182617 seconds
Execution time for HeapSort (data size: 50000): 0.16397953033447266 seconds
Execution time for HeapSort (data size: 100000): 0.35230493545532227 seconds
Execution time for HeapSort (data size: 150000): 0.5481045246124268 seconds
Execution time for HeapSort (data size: 200000): 0.7213313579559326 seconds
Execution time for HeapSort (data size: 250000): 0.9747433662414551 seconds

  --------------------------------------------------------------
  Dataset Generator: generate_sawtooth_dataset
Execution time for HeapSort (data size: 1000): 0.0019991397857666016 seconds
Execution time for HeapSort (data size: 10000): 0.033579111099243164 seconds
Execution time for HeapSort (data size: 50000): 0.15324807167053223 seconds
Execution time for HeapSort (data size: 100000): 0.3293788433074951 seconds
Execution time for HeapSort (data size: 150000): 0.5009751319885254 seconds
Execution time for HeapSort (data size: 200000): 0.6945326328277588 seconds
Execution time for HeapSort (data size: 250000): 0.8696157932281494 seconds

  --------------------------------------------------------------
  Dataset Generator: generate_custom_pattern_dataset
Execution time for HeapSort (data size: 1000): 0.0010044574737548828 seconds
Execution time for HeapSort (data size: 10000): 0.030993223190307617 seconds
Execution time for HeapSort (data size: 50000): 0.1672508716583252 seconds
Execution time for HeapSort (data size: 100000): 0.3240702152252197 seconds
Execution time for HeapSort (data size: 150000): 0.5096619129180908 seconds
Execution time for HeapSort (data size: 200000): 0.6983344554901123 seconds
Execution time for HeapSort (data size: 250000): 0.8965163230895996 seconds

  --------------------------------------------------------------
  Dataset Generator: generate_50Per_Sorted50PUnsorted
Execution time for HeapSort (data size: 1000): 0.0018897056579589844 seconds
Execution time for HeapSort (data size: 10000): 0.04937171936035156 seconds
Execution time for HeapSort (data size: 50000): 0.16699743270874023 seconds
Execution time for HeapSort (data size: 100000): 0.36109375953674316 seconds
Execution time for HeapSort (data size: 150000): 0.5819375514984131 seconds
Execution time for HeapSort (data size: 200000): 0.7710871696472168 seconds
Execution time for HeapSort (data size: 250000): 1.0252840518951416 seconds

  --------------------------------------------------------------
  Dataset Generator: generate50P_ascending50P_random
Execution time for HeapSort (data size: 1000): 0.005005836486816406 seconds
Execution time for HeapSort (data size: 10000): 0.048097848892211914 seconds
Execution time for HeapSort (data size: 50000): 0.1613154411315918 seconds
Execution time for HeapSort (data size: 100000): 0.4184846878051758 seconds
Execution time for HeapSort (data size: 150000): 0.6693413257598877 seconds
Execution time for HeapSort (data size: 200000): 0.9635350704193115 seconds
Execution time for HeapSort (data size: 250000): 1.0698301792144775 seconds

  --------------------------------------------------------------
  Dataset Generator: generate_alternating_dataset
Execution time for HeapSort (data size: 1000): 0.002099275588989258 seconds
Execution time for HeapSort (data size: 10000): 0.03699803352355957 seconds
Execution time for HeapSort (data size: 50000): 0.16310596466064453 seconds
Execution time for HeapSort (data size: 100000): 0.44656968116760254 seconds
Execution time for HeapSort (data size: 150000): 0.6926050186157227 seconds
Execution time for HeapSort (data size: 200000): 0.971081018447876 seconds
Execution time for HeapSort (data size: 250000): 1.175126314163208 seconds

  --------------------------------------------------------------
-*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*-
  --------------------------------------------------------------
-*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*-
Sorting Method: mergeSORTINGg
  Dataset Generator: generate_random_dataset
Execution time for MergeSort (data size: 1000): 0.0020546913146972656 seconds
Execution time for MergeSort (data size: 10000): 0.03401803970336914 seconds
Execution time for MergeSort (data size: 50000): 0.17009711265563965 seconds
Execution time for MergeSort (data size: 100000): 0.34962010383605957 seconds
Execution time for MergeSort (data size: 150000): 0.6392936706542969 seconds
Execution time for MergeSort (data size: 200000): 0.8594484329223633 seconds
Execution time for MergeSort (data size: 250000): 1.0249955654144287 seconds

  ----------------------------------------------------------------
  Dataset Generator: generate_sorted_dataset
Execution time for MergeSort (data size: 1000): 0.003905057907104492 seconds
Execution time for MergeSort (data size: 10000): 0.03609585762023926 seconds
Execution time for MergeSort (data size: 50000): 0.12288904190063477 seconds
Execution time for MergeSort (data size: 100000): 0.3267388343811035 seconds
Execution time for MergeSort (data size: 150000): 0.44089651107788086 seconds
Execution time for MergeSort (data size: 200000): 0.7190003395080566 seconds
Execution time for MergeSort (data size: 250000): 0.7944703102111816 seconds

  ----------------------------------------------------------------
  Dataset Generator: generate_ascending_descending_dataset
Execution time for MergeSort (data size: 1000): 0.0032224655151367188 seconds
Execution time for MergeSort (data size: 10000): 0.027086973190307617 seconds
Execution time for MergeSort (data size: 50000): 0.12715911865234375 seconds
Execution time for MergeSort (data size: 100000): 0.3414266109466553 seconds
Execution time for MergeSort (data size: 150000): 0.4927029609680176 seconds
Execution time for MergeSort (data size: 200000): 0.6440277099609375 seconds
Execution time for MergeSort (data size: 250000): 0.8700945377349854 seconds

  ----------------------------------------------------------------
  Dataset Generator: generate_sawtooth_dataset
Execution time for MergeSort (data size: 1000): 0.001997232437133789 seconds
Execution time for MergeSort (data size: 10000): 0.0299990177154541 seconds
Execution time for MergeSort (data size: 50000): 0.1410202980041504 seconds
Execution time for MergeSort (data size: 100000): 0.36044859886169434 seconds
Execution time for MergeSort (data size: 150000): 0.48234081268310547 seconds
Execution time for MergeSort (data size: 200000): 0.6453473567962646 seconds
Execution time for MergeSort (data size: 250000): 0.893519401550293 seconds

  ----------------------------------------------------------------
  Dataset Generator: generate_custom_pattern_dataset
Execution time for MergeSort (data size: 1000): 0.0018994808197021484 seconds
Execution time for MergeSort (data size: 10000): 0.029001474380493164 seconds
Execution time for MergeSort (data size: 50000): 0.1366722583770752 seconds
Execution time for MergeSort (data size: 100000): 0.32409238815307617 seconds
Execution time for MergeSort (data size: 150000): 0.4435253143310547 seconds
Execution time for MergeSort (data size: 200000): 0.6225426197052002 seconds
Execution time for MergeSort (data size: 250000): 0.7558674812316895 seconds
```

```
 ----------------------------------------------------------------
 Dataset Generator: generate_50Per_Sorted50PUnsorted
Execution time for MergeSort (data size: 1000): 0.004092693328857422 seconds
Execution time for MergeSort (data size: 10000): 0.04600095748901367 seconds
Execution time for MergeSort (data size: 50000): 0.14889025688171387 seconds
Execution time for MergeSort (data size: 100000): 0.2949042320251465 seconds
Execution time for MergeSort (data size: 150000): 0.5112686157226562 seconds
Execution time for MergeSort (data size: 200000): 0.6835360527038574 seconds
Execution time for MergeSort (data size: 250000): 0.9128825664520264 seconds

 ----------------------------------------------------------------
 Dataset Generator: generate50P_ascending50P_random
Execution time for MergeSort (data size: 1000): 0.0029103755950927734 seconds
Execution time for MergeSort (data size: 10000): 0.02602529525756836 seconds
Execution time for MergeSort (data size: 50000): 0.15160536766052246 seconds
Execution time for MergeSort (data size: 100000): 0.3470942974090576 seconds
Execution time for MergeSort (data size: 150000): 0.5063936710357666 seconds
Execution time for MergeSort (data size: 200000): 0.69272780418396 seconds
Execution time for MergeSort (data size: 250000): 0.8978819847106934 seconds

 ----------------------------------------------------------------
 Dataset Generator: generate_alternating_dataset
Execution time for MergeSort (data size: 1000): 0.002002716064453125 seconds
Execution time for MergeSort (data size: 10000): 0.03400254249572754 seconds
Execution time for MergeSort (data size: 50000): 0.1479780673980713 seconds
Execution time for MergeSort (data size: 100000): 0.2781345844268799 seconds
Execution time for MergeSort (data size: 150000): 0.4269979000091553 seconds
Execution time for MergeSort (data size: 200000): 0.655738353729248 seconds
Execution time for MergeSort (data size: 250000): 0.7598142623901367 seconds


Sorting Method: insertSORTINGg
 Dataset Generator: generate_sorted_dataset
Execution time for InsertionSort (data size: 1000): 0.0 seconds
Execution time for InsertionSort (data size: 10000): 0.0009920597076416016 seconds
Execution time for InsertionSort (data size: 50000): 0.006958961486816406 seconds
Execution time for InsertionSort (data size: 100000): 0.00967097282409668 seconds
Execution time for InsertionSort (data size: 150000): 0.01819753646850586 seconds
Execution time for InsertionSort (data size: 200000): 0.045210838317871094 seconds
Execution time for InsertionSort (data size: 250000): 0.03480243682861328 seconds
 ----------------------------------------------------------------
 Dataset Generator: generate_ascending_descending_dataset
Execution time for InsertionSort (data size: 1000): 0.014678478240966797 seconds
Execution time for InsertionSort (data size: 10000): 1.0767066478729248 seconds
Execution time for InsertionSort (data size: 50000): 27.585763692855835 seconds
Execution time for InsertionSort (data size: 100000): 110.80135798454285 seconds
Execution time for InsertionSort (data size: 150000): 234.0323417186737 seconds
Execution time for InsertionSort (data size: 200000): 421.87486028671265 seconds
Execution time for InsertionSort (data size: 250000): 631.4869871139526 seconds
  --------------------------------------------------------------
  Dataset Generator: generate_sawtooth_dataset
Execution time for InsertionSort (data size: 1000): 0.022562265396118164 seconds
Execution time for InsertionSort (data size: 10000): 1.9488945007324219 seconds
Execution time for InsertionSort (data size: 50000): 54.61842679977417 seconds
Execution time for InsertionSort (data size: 100000): 209.34565949440002 seconds
Execution time for InsertionSort (data size: 150000): 499.2401793003082 seconds
Execution time for InsertionSort (data size: 200000): 860.2501354217529 seconds
Execution time for InsertionSort (data size: 250000): 1419.2503564357758 seconds
  --------------------------------------------------------------
  Dataset Generator: generate_custom_pattern_dataset
Execution time for InsertionSort (data size: 1000): 0.02627277374267578 seconds
Execution time for InsertionSort (data size: 10000): 2.2387800216674805 seconds
Execution time for InsertionSort (data size: 50000): 53.79481267929077 seconds
Execution time for InsertionSort (data size: 100000): 210.58521008491516 seconds
Execution time for InsertionSort (data size: 150000): 502.56417417526245 seconds
Execution time for InsertionSort (data size: 200000): 852.312362909317 seconds
Execution time for InsertionSort (data size: 250000): 1494.87713098526 seconds
  --------------------------------------------------------------
  Dataset Generator: generate_50Per_Sorted50PUnsorted
Execution time for InsertionSort (data size: 1000): 0.014205217361450195 seconds
Execution time for InsertionSort (data size: 10000): 2.0349984169006348 seconds
Execution time for InsertionSort (data size: 50000): 47.96224904060364 seconds
Execution time for InsertionSort (data size: 100000): 187.05574584007263 seconds
Execution time for InsertionSort (data size: 150000): 395.4848871231079 seconds
Execution time for InsertionSort (data size: 200000): 694.4396104812622 seconds
Execution time for InsertionSort (data size: 250000): 1107.237352848053 seconds
  --------------------------------------------------------------
  Dataset Generator: generate50P_ascending50P_random
Execution time for InsertionSort (data size: 1000): 0.01295018196105957 seconds
Execution time for InsertionSort (data size: 10000): 2.599344491958618 seconds
Execution time for InsertionSort (data size: 50000): 68.66441535949707 seconds
Execution time for InsertionSort (data size: 100000): 302.07448530197144 seconds
Execution time for InsertionSort (data size: 150000): 677.2613024711609 seconds
Execution time for InsertionSort (data size: 200000): 1105.3013491630554 seconds
Execution time for InsertionSort (data size: 250000): 1889.872967004776 seconds
  --------------------------------------------------------------
  Dataset Generator: generate_alternating_dataset
Execution time for InsertionSort (data size: 1000): 0.0 seconds
Execution time for InsertionSort (data size: 10000): 0.0019969940185546875 seconds
Execution time for InsertionSort (data size: 50000): 0.015041589736938477 seconds
Execution time for InsertionSort (data size: 100000): 0.015079736709594727 seconds
Execution time for InsertionSort (data size: 150000): 0.033492326736450195 seconds
Execution time for InsertionSort (data size: 200000): 0.04453468322753906 seconds
Execution time for InsertionSort (data size: 250000): 0.04027509689331055 seconds
```