Source code for each algorithm

1. **Quicksort**

To write the source code of quicksort where is used insertion sort for appropriate size small input;

```
def insertion_sort(arr, left=0, right=None):
    if right is None:
        right = len(arr) - 1
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Secondly, write the source code which needs to be determined experimentally, by finding the cutoff point between pure insertion sort and pure quicksort;

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
def quicksort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1
    if low < high:
        pi = partition(arr, low, high)
        quicksort(arr, low, pi - 1)
        quicksort(arr, pi + 1, high)

# Function to generate random arrays
def generate_random_array(n):
    return [random.randint(0, 20000) for _ in range(n)]
# Define small input sizes to test
input_sizes = list(range(5, 5000, 5))  # Small inputs from 5 to 5000
insertion_times = []
```

```
quicksort_times = []
for size in input_sizes:
    arr1 = generate_random_array(size)
    arr2 = arr1.copy()
    # Measure Insertion Sort Time
    start = time.time()
    insertion_sort(arr1)
    insertion_times.append(time.time() - start)
    # Measure Quicksort Time
    start = time.time()
    quicksort(arr2)
    quicksort_times.append(time.time() - start)
# Find the cutoff point where Insertion Sort becomes faster
cutoff_index = np.argmax(np.array(insertion_times) < np.array(quicksort_times))
best_cutoff = input_sizes[cutoff_index]

print(f"Best Cutoff Point: {best_cutoff}")
```

Next, to try randomized pivoting;

```
# Re-define necessary quicksort functions as execution state was reset
def randomized_partition(arr, low, high):
    """ Randomized Partition for Quicksort """
    rand_pivot = random.randint(low, high)
    arr[high], arr[rand_pivot] = arr[rand_pivot], arr[high]
    return partition(arr, low, high)
def quicksort_random(arr, low, high, cutoff):
    """ Quicksort with Randomized Pivoting """
    if high - low + 1 <= cutoff:
        insertion_sort(arr[low:high + 1])
        return
    if low < high:
        pivot = randomized_partition(arr, low, high)
        quicksort_random(arr, low, pivot - 1, cutoff)
        quicksort_random(arr, pivot + 1, high, cutoff)
```

Finally, to find the median-of-three pivoting (for possible improvements)

```
def median_of_three_partition(arr, low, high):
```

```
    """ Median-of-three Partition for Quicksort """
    mid = (low + high) // 2
    pivots = [(arr[low], low), (arr[mid], mid), (arr[high], high)]
    pivots.sort()
    median_index = pivots[1][1]
    arr[high], arr[median_index] = arr[median_index], arr[high]
    return partition(arr, low, high)
def quicksort_median(arr, low, high, cutoff):
    """ Quicksort with Median-of-Three Pivoting """
    if high - low + 1 <= cutoff:
        insertion_sort(arr[low:high + 1])
        return
    if low < high:
        pivot = median_of_three_partition(arr, low, high)
        quicksort_median(arr, low, pivot - 1, cutoff)
        quicksort_median(arr, pivot + 1, high, cutoff)
```

2. Radix sort

Here, to write the source code for radix sort which issues counting sort as the stable sort of subroutine;

```
  def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1

    for i in range(n):
        arr[i] = output[i]
```

Secondly, to find the parameterize the radix sort using the base as an input parameter and find the best base to use for fastest sort on your machine;

```
def radix_sort(arr, base=10):
    max_element = max(arr)
    exp = 1
    while max_element // exp > 0:
        counting_sort(arr, exp)
        exp *= base
# Function to generate random integers
def generate_random_array(n):
    return [random.randint(0, 20000) for _ in range(n)]

# Experimenting with different bases for Radix Sort

bases = [10, 50, 100, 256, 512, 1024]  # Different radix bases
radix_times = []

for base in bases:
    arr = generate_random_array(10000)  # Fixed input size
    start = time.time()
    radix_sort(arr, base)
    radix_times.append(time.time() - start)

# Identify the best base for Radix Sort
best_radix_base = bases[np.argmin(radix_times)]
best_radix_base
```

System Information

1. **A brief description about the computer system:**

- **Computer System:** Intel(R) Core (TM) i7-11800H, 2.30GHz
- **Operating System:** Windows 10
- **Programming Language:** Python 3.11.9
- **Random Number Generator:** Python's random.randint() function, generating integers in the range [0, 20000].

2. **Quicksort (Optimized):**

- Uses **Insertion Sort** for small subarrays (cutoff point determined experimentally).

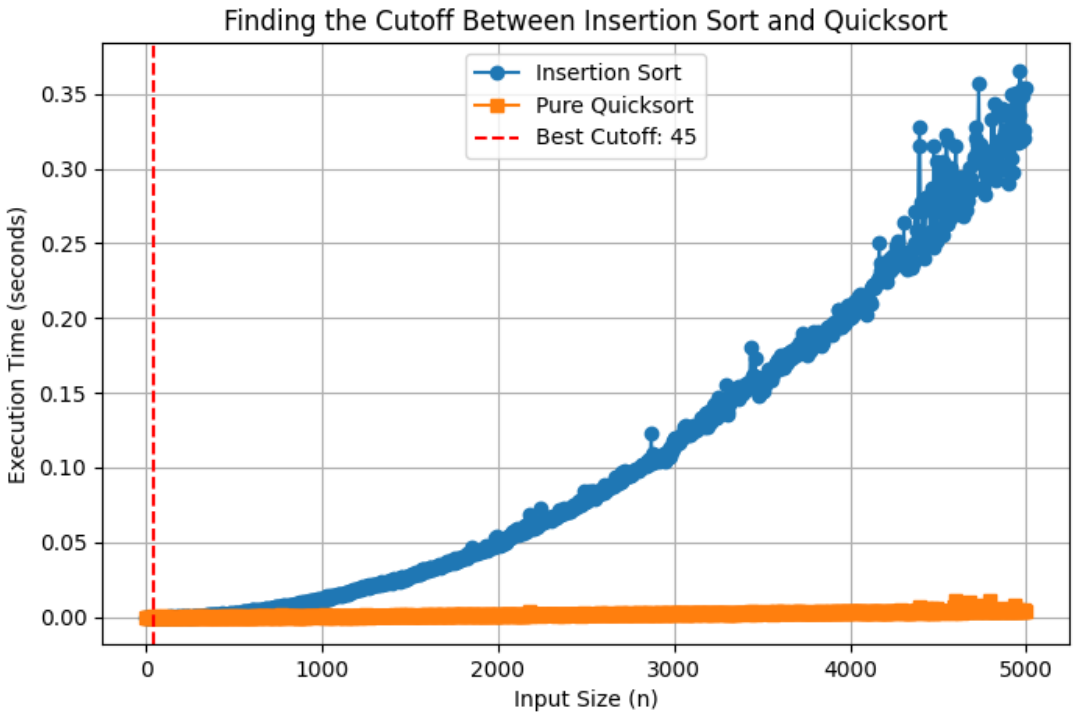
- Implements **Randomized Pivoting** and **Median-of-Three Pivoting** to improve performance.

Radix Sort (Optimized):

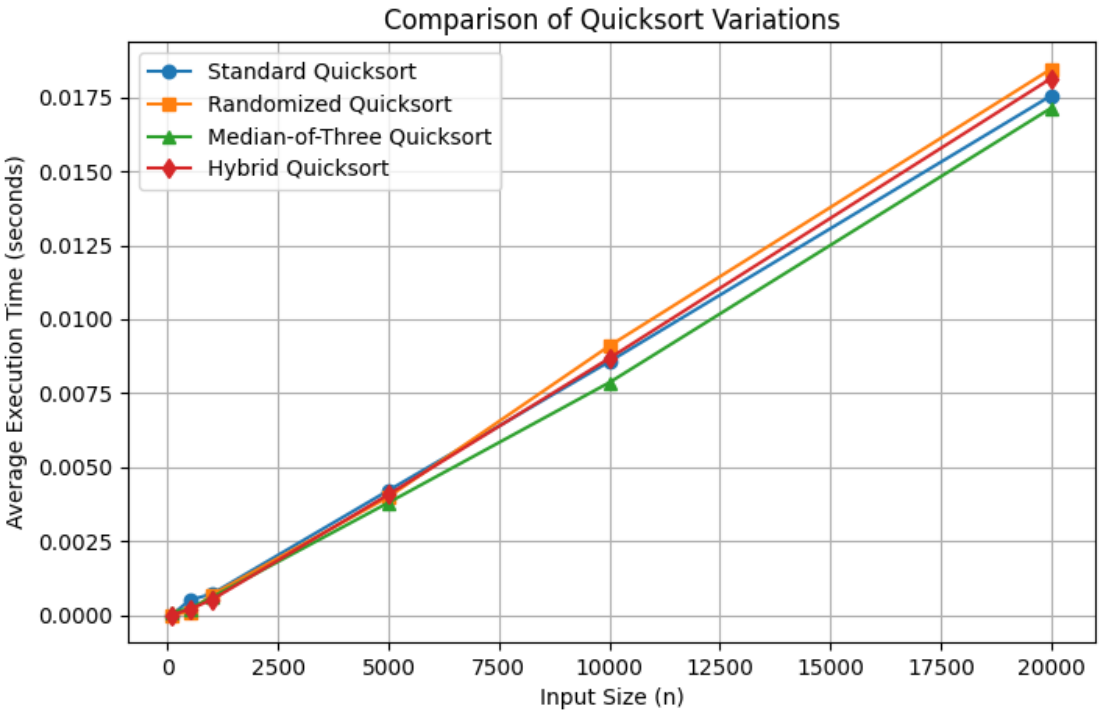
- Uses **Counting Sort** as a stable subroutine.
- Optimizes base selection by experimenting with different values to find the best base.

Plots to submit:

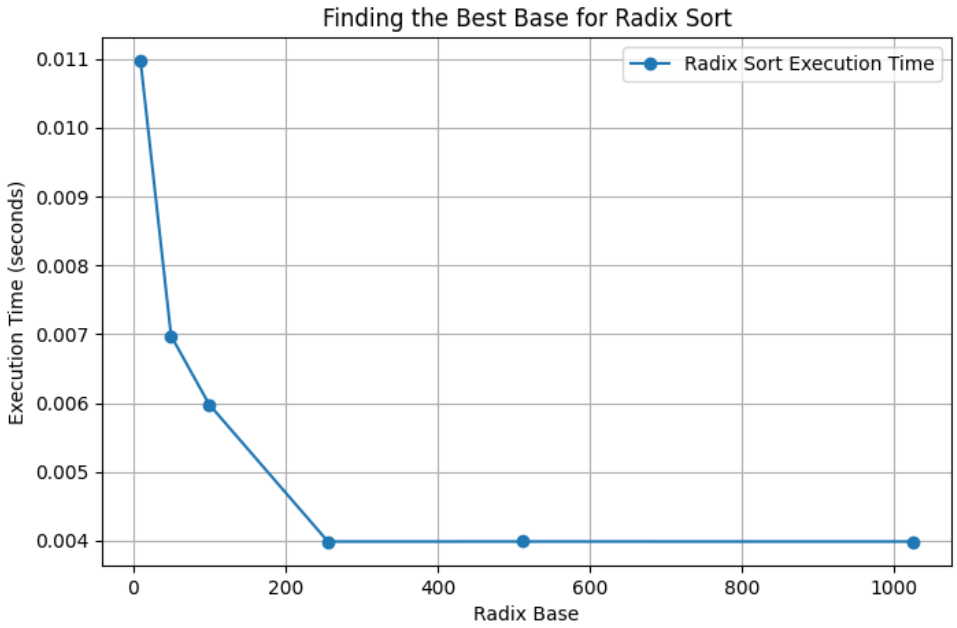
1. **Plot to find cut off point of insertion sort and quicksort**



2. **Plot to find the best quicksort among its all variations**

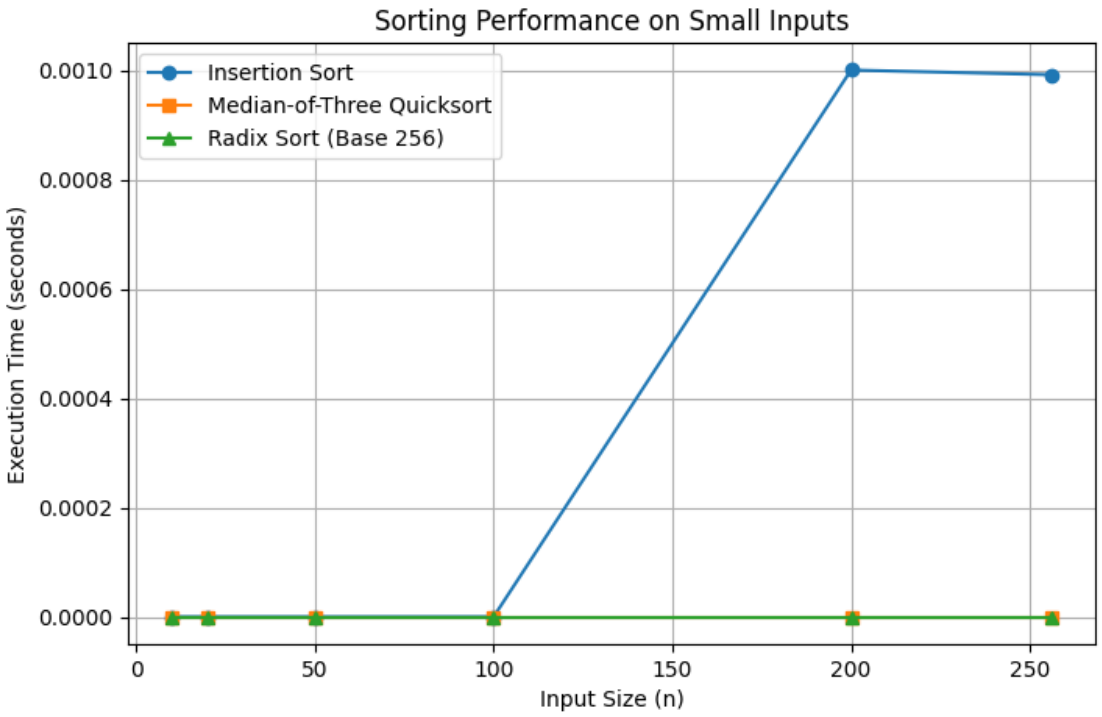


3. Plot to find the best base to be employed for Radix sort (typical bases are high such as 100 or more)

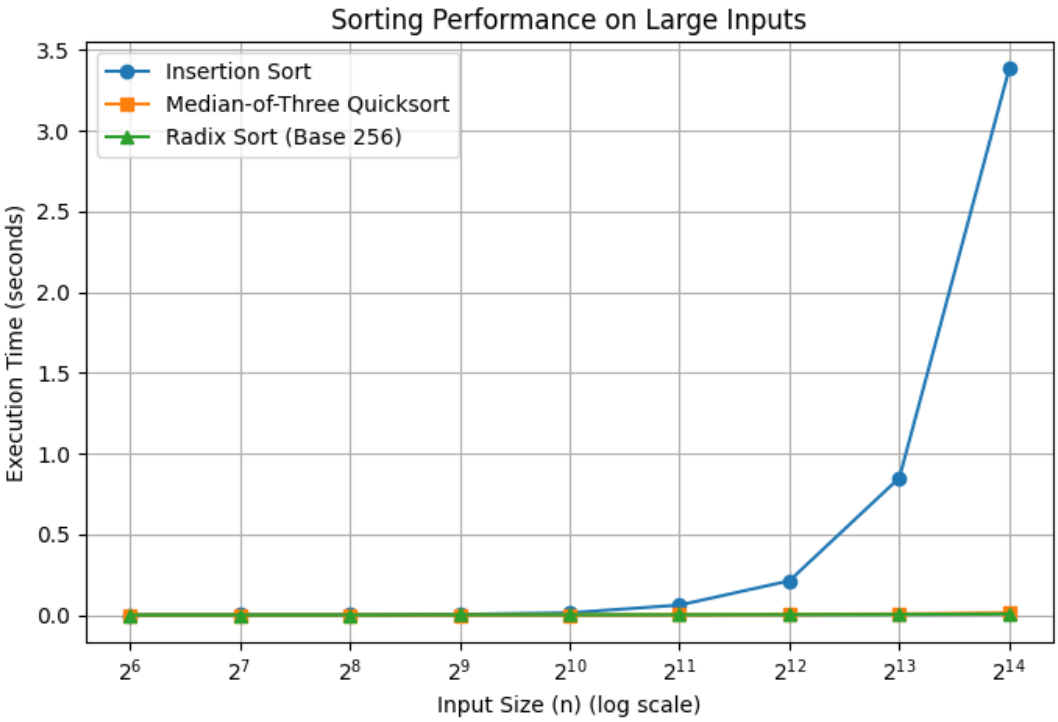


4. Plots to compare best quicksort, insertion sort, and radix sort

a) The first plot has range of input lengths up to 256 to compare the algorithms on small inputs



b) The second plot has range of input from 64 through 214 (or more, in powers of 2's) using log2 scale to compare the algorithms over a large range of input lengths

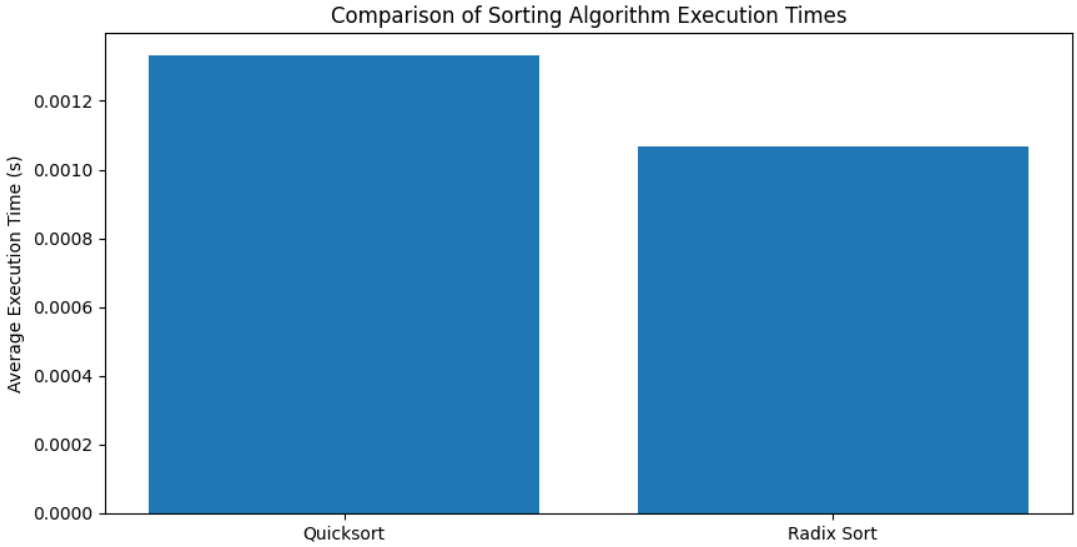


The integers to be sorted should be in the range 0 to 20,000 (or larger). For each n, to run algorithms (Quicksort and radix sort) on 10 to 20 different randomly generated inputs and the average execution time:

Average Quicksort Time: 0.001331 s

Average Radix Sort Time: 0.001068 s

Now, to draw the plot of the average execution time


Comparison of Sorting Algorithm Execution Times
Average Execution Time (s)
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
Quicksort
Radix Sort