Source code for each algorithm

1. Quicksort

Write the source code for the Quicksort algorithm, incorporating Insertion Sort for handling small input sizes efficiently.

```python
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 to experimentally determine the optimal cutoff point between using pure Insertion Sort and pure Quicksort.

```python
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)
def generate_random_array(n):
    return [random.randint(0, 20000) for _ in range(n)]
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)
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;

```python
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, implement the median-of-three pivot selection method to enhance performance.

```python
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

Write the source code for the Radix Sort algorithm, utilizing Counting Sort as a stable sorting 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, implement Radix Sort with a configurable base as an input parameter and determine the optimal base for the fastest sorting performance 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)]
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)
best_radix_base = bases[np.argmin(radix_times)]
best_radix_base
```

System Information

A brief overview of the computer system:

- **Processor:** 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 within the range [0, 20000].

Quicksort (Optimized)

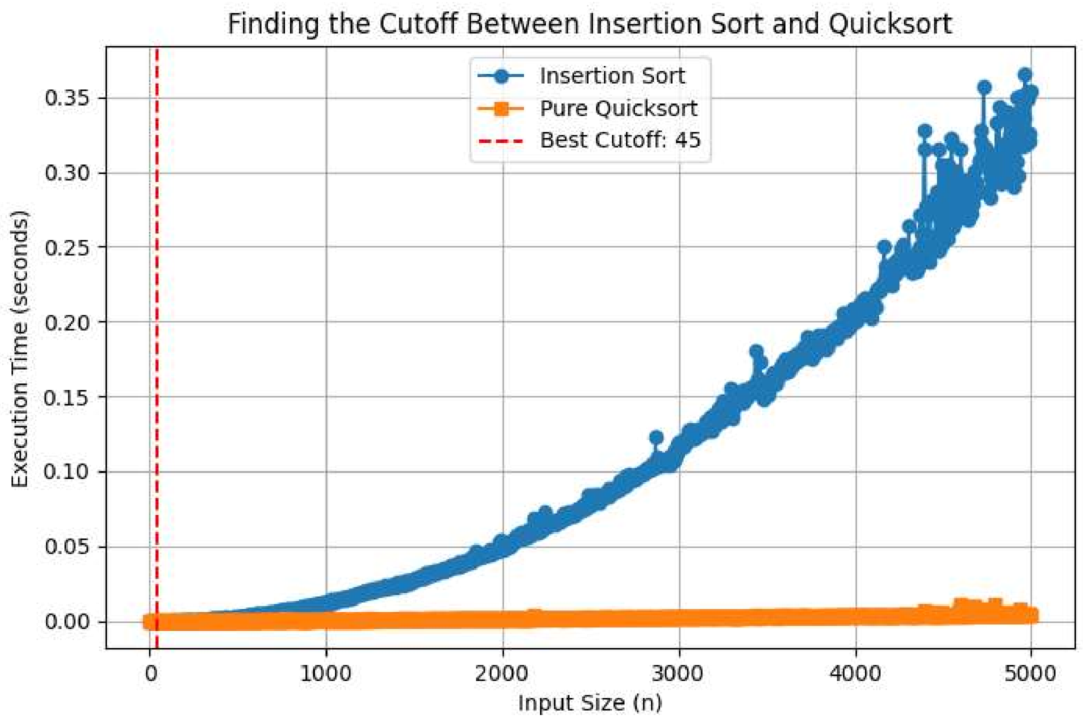
- Incorporates **Insertion Sort** for small subarrays, with the cutoff point determined experimentally.
- Enhances performance through **Randomized Pivoting** and **Median-of-Three Pivoting** techniques.

Radix Sort (Optimized)

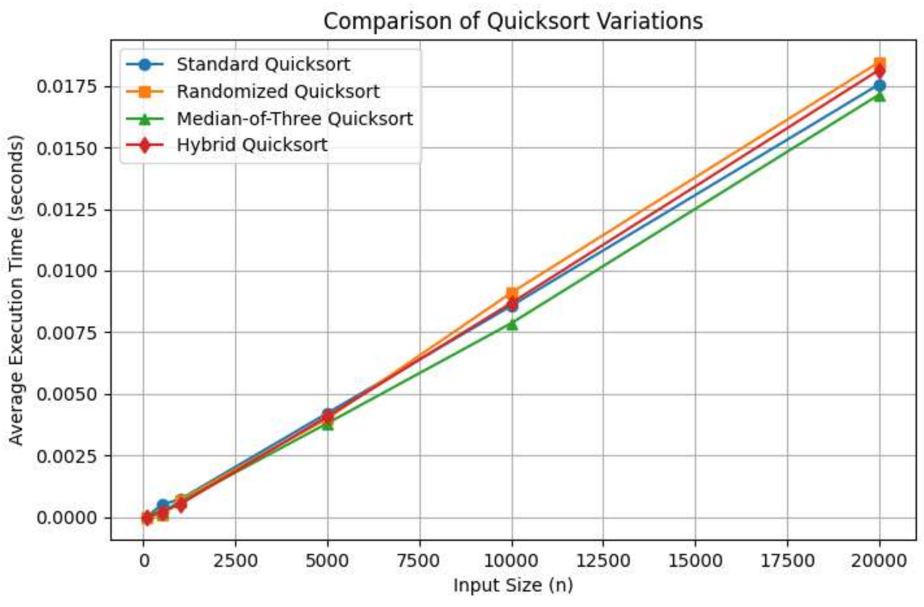
- Utilizes **Counting Sort** as a stable subroutine.
- Optimizes base selection by experimenting with different values to determine the most efficient base.

Plots to submit:

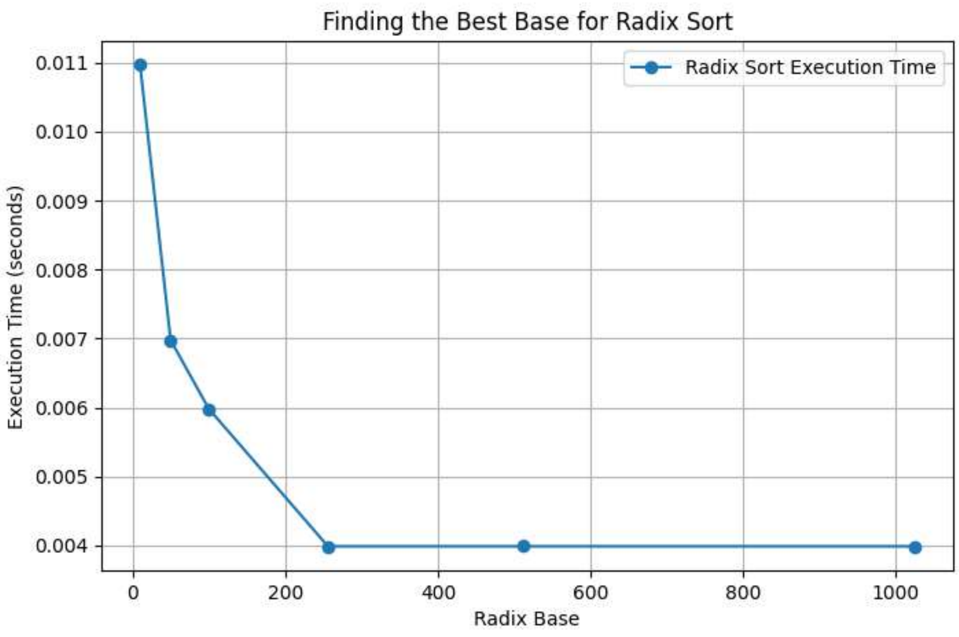
1. Plot to determine the optimal cutoff point between Insertion Sort and Quicksort



2. **Generate a plot to determine the most efficient variation of the Quicksort algorithm.**

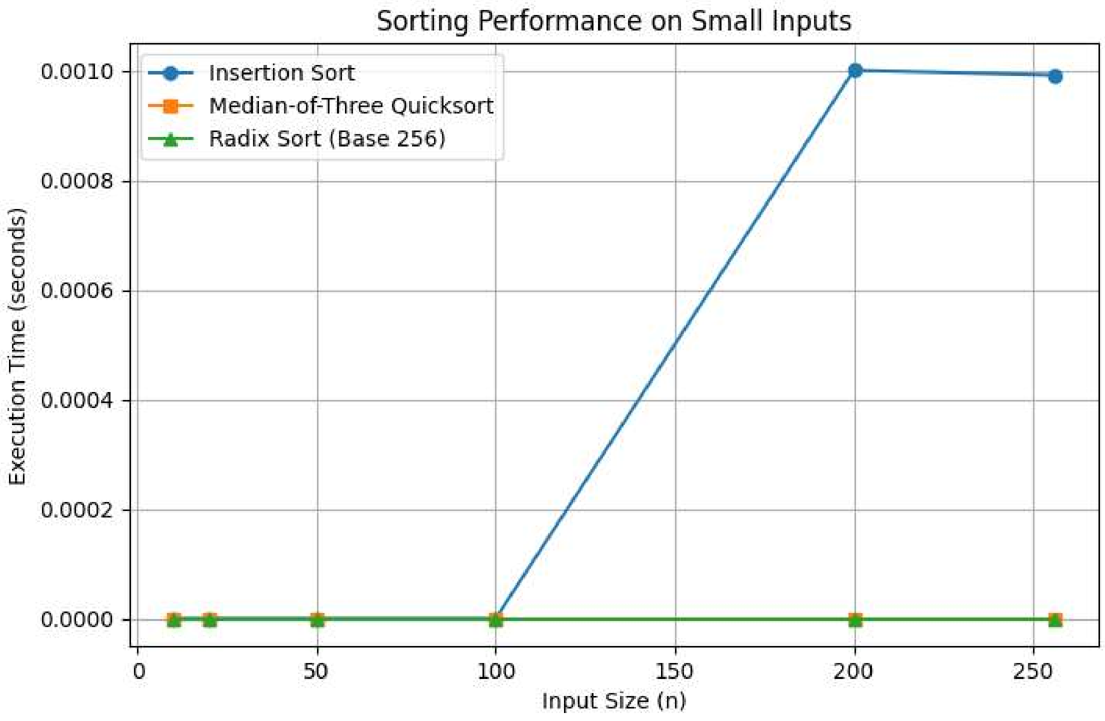


3. **Create a plot to determine the optimal base for Radix Sort, typically using higher base values (100 or more).**

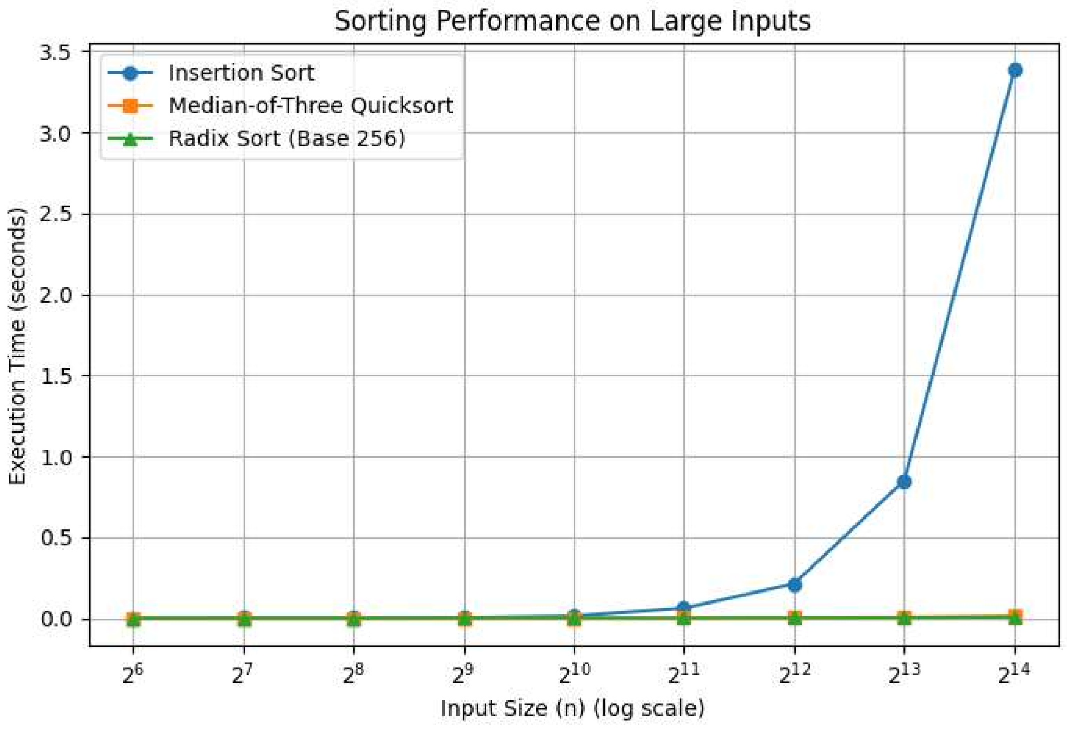


4. Graphs comparing the performance of the best Quicksort, Insertion Sort, and Radix Sort

a) The first graph displays input sizes up to 256, allowing a performance comparison of the algorithms on smaller datasets.



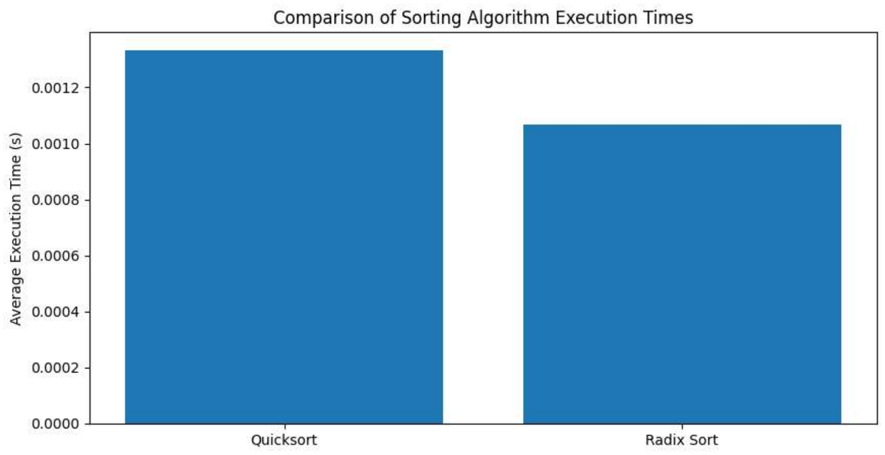
(b) The second plot displays input sizes ranging from 64 to 214 (or higher, in powers of 2) using a $\log_2$ scale to compare the algorithms across a wide range of input lengths.



The integers to be sorted should fall within the range of 0 to 20,000 (or higher). For each value of *n*, both Quicksort and Radix Sort are executed on 10 to 20 different randomly generated inputs, and their average execution time is recorded.

- Average Quicksort Execution Time: 0.001331 seconds
- Average Radix Sort Execution Time: 0.001068 seconds

Now, plot the graph of the average execution time for both sorting algorithms.


Comparison of Sorting Algorithm Execution Times
Average Execution Time (s)
Quicksort
Radix Sort