## **Source code**

**Quicksort:**

**Using 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*

**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}")*

**Implementing randomized pivoting:**

*def randomized\_partition(arr, low, high):*

*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):*

*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)*

**Implementing median-of-three pivoting:**

*def median\_of\_three\_partition(arr, low, high):*

*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):*

*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)*

**Radix sort**

**Using 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]*

**Parameterizing the radix sort using the base as an input parameter and finding the best base to use for fastest sort:**

*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)]*

## **Report**

## **Experimental Setup**

**Processor:** Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 Mhz, 2 Core(s)

**Language:** Python

**Random Number Generator:** random.randint() function using random library in Python.

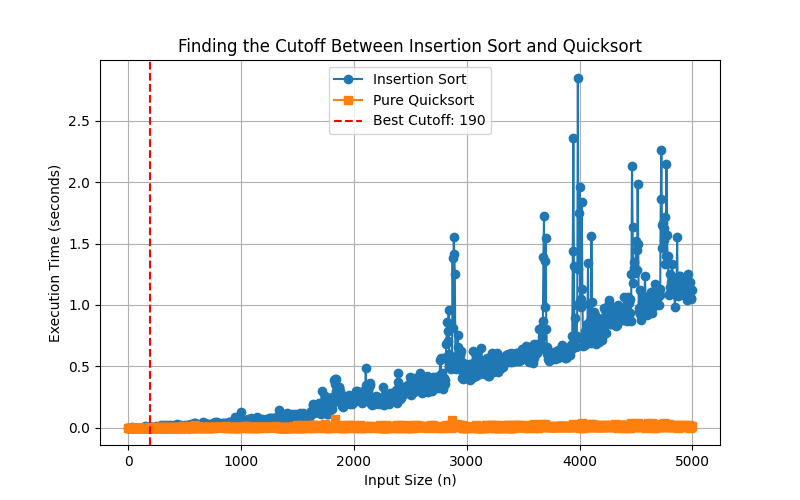
**Optimizations for quicksort and for radix sort:**

For Quicksort, I initially used Insertion sort using small arrays and then compared with pure quicksort to find the optimal cutoff point. After that, I used randomized pivoting and then median of three pivoting to further optimize the process.

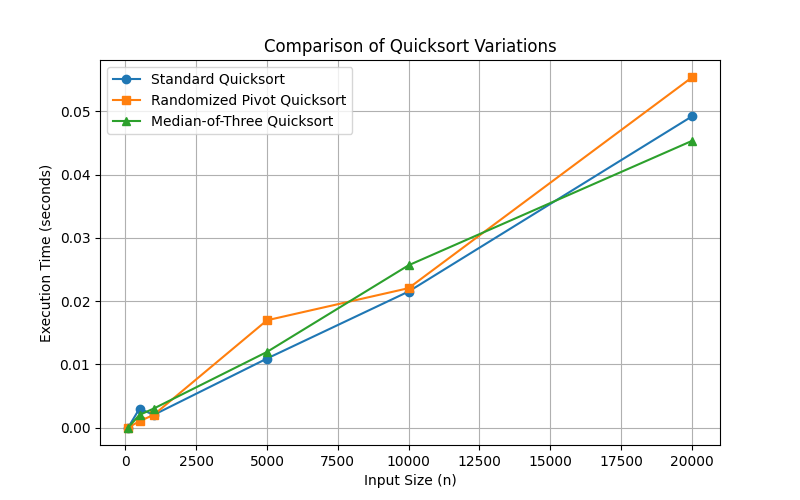
For radix sort, I started by using counting sort as a stable subroutine. Then, I iterated over all the possible options to find out the most effective base and used that best base to apply for sorting. This resulted in a more efficient response.

## **Plots**

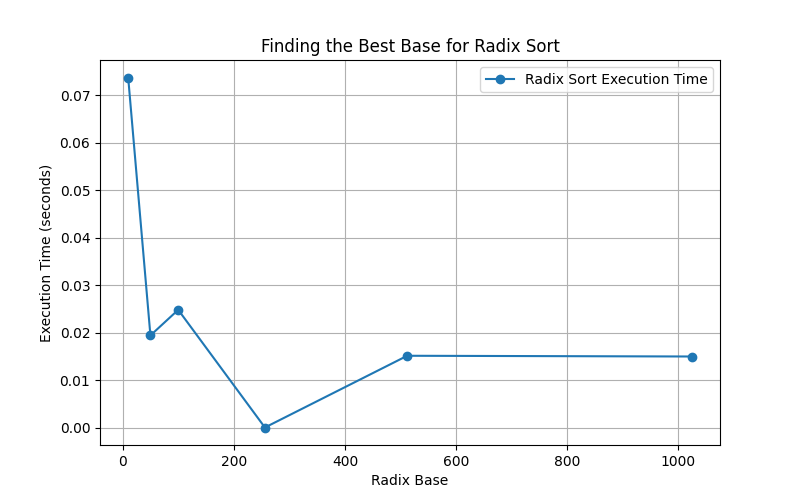
1. **Plot to find cut off point of insertion sort and quicksort**:



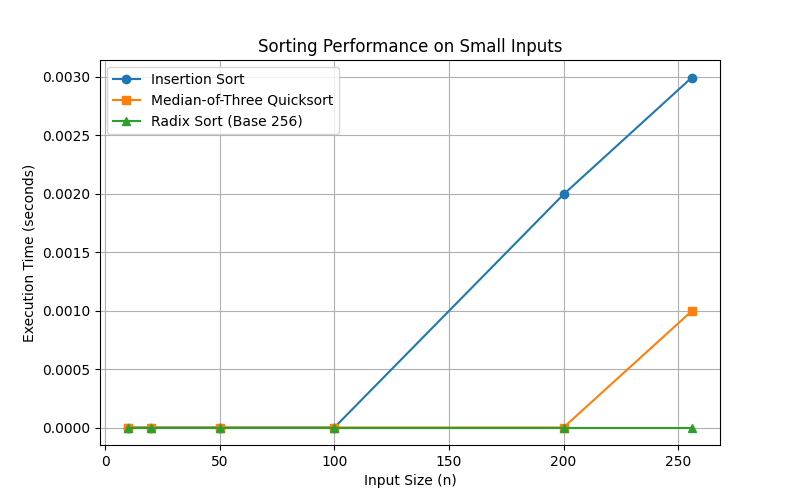
1. **Plot to find the best quicksort among its all variations:**



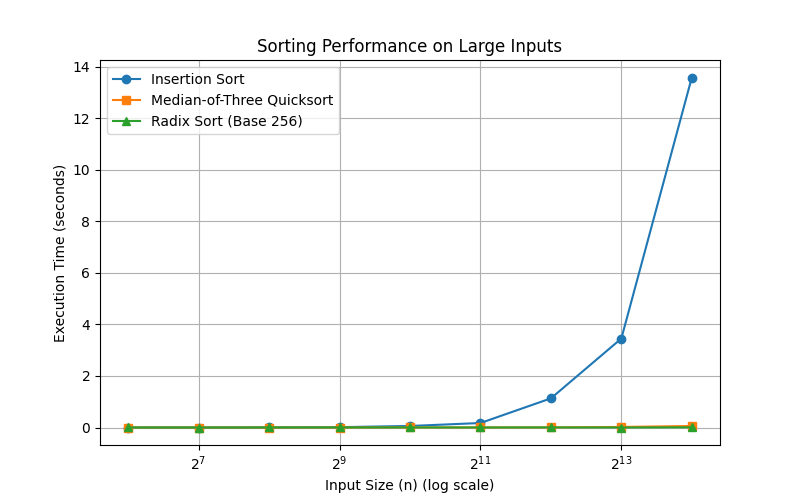
**3. Plot to find the best base to be employed for Radix sort:**



**4(a). Plot to compare best quicksort, insertion sort, and radix sort (input lengths up to 256):**



**4(b). Plot to compare best quicksort, insertion sort, and radix sort (with input lengths ranging from 64 up to 214):**



**The average execution times with the plot are given below:**

Average Quicksort Time: 0.007853s

Average Radix Sort Time: 0.005267s

