**Banaras Hindu University**  
**Institute of Science**  
**Department of Computer Science**



**“Design and Analysis of Algorithms (CS-203)”**

**Submitted To:**  
**Dr. Marisha**  
Department of Computer Science

**Submitted By:**  
Sagar Timalsena

(24419CMP025)  
M.Sc. (Computer Science)

**Table of Contents**

[1. Quick Sort 3](#_Toc196817928)

[2. Merge Sort 5](#_Toc196817929)

[3. Bubble Sort 7](#_Toc196817930)

[4. Insertion Sort 9](#_Toc196817931)

[5. Radix sort 11](#_Toc196817932)

[6. Heap Sort 15](#_Toc196817933)

[7. Counting Sort 17](#_Toc196817934)

[8. Depth First Sort 19](#_Toc196817935)

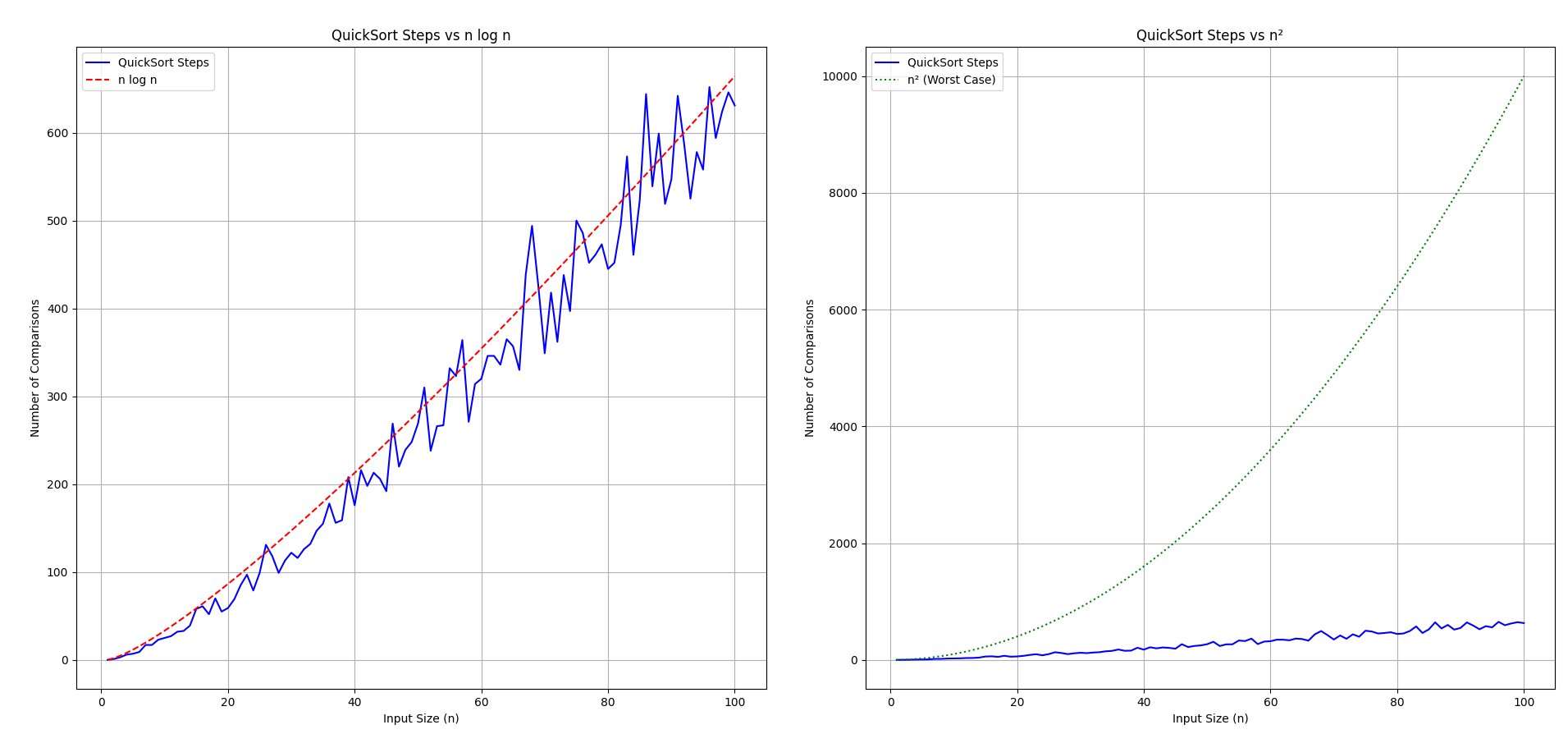
[9. Breath-First Sort 20](#_Toc196817936)

[10. Travelling Salesman Problem (TSP) 21](#_Toc196817937)

## Quick Sort

|  |
| --- |
| import matplotlib.pyplot as plt  import random  import math  def quicksort(arr, counter):      """Recursive QuickSort implementation with comparison counter."""      if len(arr) <= 1:          return arr      pivot = arr[0]      left, right = [], []      for x in arr[1:]:          counter[0] += 1          if x < pivot:              left.append(x)          else:              right.append(x)      return quicksort(left, counter) + [pivot] + quicksort(right, counter)  def run\_quicksort\_with\_counter(n):      """Generates random data and returns number of comparisons for QuickSort."""      arr = random.sample(range(n \* 2), n)      counter = [0]      quicksort(arr, counter)      return counter[0]  # Generate data  sizes = list(range(1, 101))  qs\_steps = [run\_quicksort\_with\_counter(n) for n in sizes]  n\_log\_n = [n \* math.log2(n) for n in sizes]  n\_squared = [n \*\* 2 for n in sizes]  # Plot both graphs side by side  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))  # Plot 1: QuickSort vs n log n  ax1.plot(sizes, qs\_steps, label='QuickSort Steps', color='blue')  ax1.plot(sizes, n\_log\_n, label='n log n', color='red', linestyle='--')  ax1.set\_title('QuickSort Steps vs n log n')  ax1.set\_xlabel('Input Size (n)')  ax1.set\_ylabel('Number of Comparisons')  ax1.legend()  ax1.grid(True)  # Plot 2: QuickSort vs n^2  ax2.plot(sizes, qs\_steps, label='QuickSort Steps', color='blue')  ax2.plot(sizes, n\_squared, label='n² (Worst Case)', color='green', linestyle=':')  ax2.set\_title('QuickSort Steps vs n²')  ax2.set\_xlabel('Input Size (n)')  ax2.set\_ylabel('Number of Comparisons')  ax2.legend()  ax2.grid(True)  plt.tight\_layout()  plt.show() |

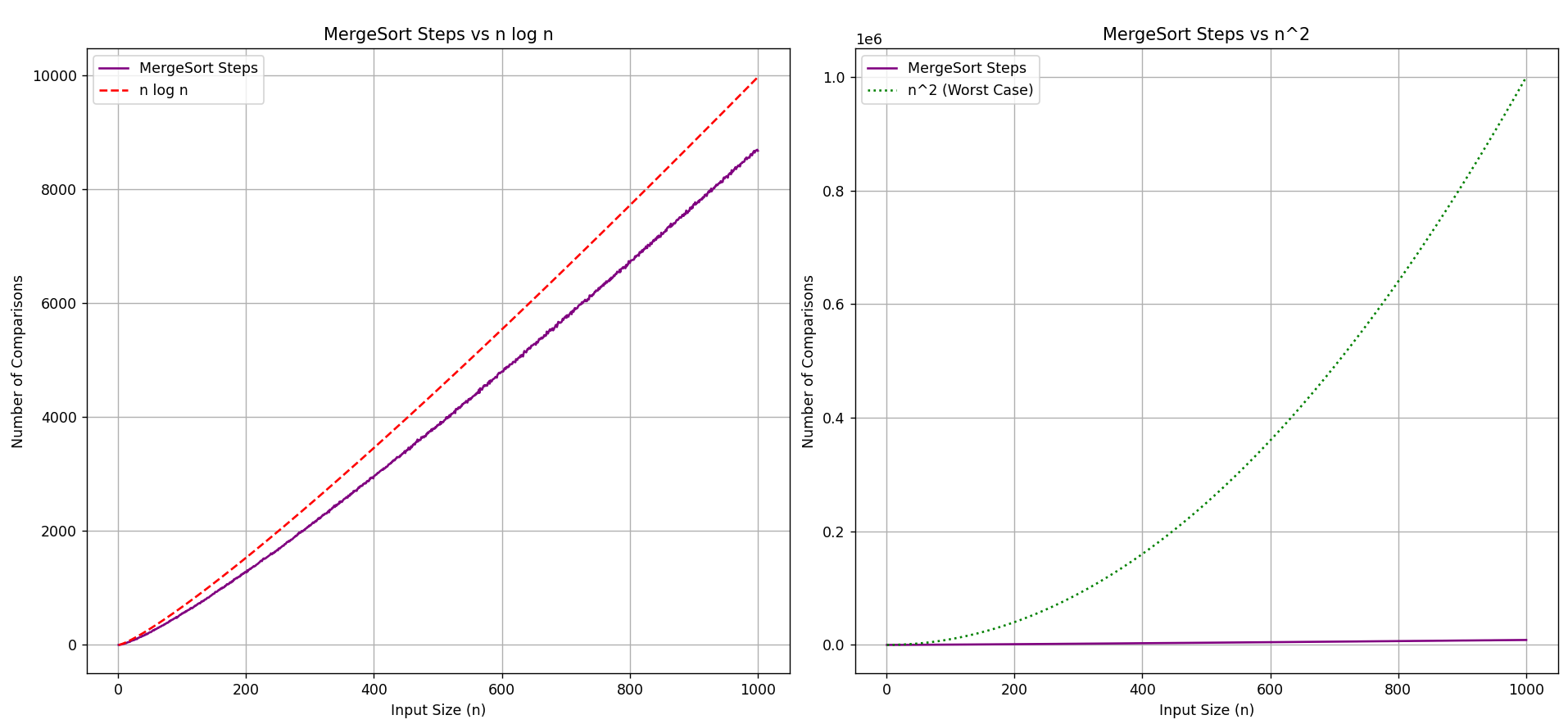
**Output:**



## Merge Sort

|  |
| --- |
| import matplotlib.pyplot as plt  import random  import math  def mergesort(arr, counter):      """Performs MergeSort and counts comparison steps."""      if len(arr) <= 1:          return arr      mid = len(arr) // 2      left = mergesort(arr[:mid], counter)      right = mergesort(arr[mid:], counter)      return merge(left, right, counter)  def merge(left, right, counter):      merged = []      i = j = 0      while i < len(left) and j < len(right):          counter[0] += 1          if left[i] < right[j]:              merged.append(left[i])              i += 1          else:              merged.append(right[j])              j += 1      merged.extend(left[i:])      merged.extend(right[j:])      return merged  def run\_mergesort\_with\_counter(n):      arr = random.sample(range(n \* 2), n)  # Unique elements      counter = [0]      mergesort(arr, counter)      return counter[0]  # Main execution  sizes = list(range(1, 1001))  ms\_steps = [run\_mergesort\_with\_counter(n) for n in sizes]  n\_log\_n = [n \* math.log2(n) for n in sizes]  n\_squared = [n \*\* 2 for n in sizes]  # Plot both comparisons side by side  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 5))  # Plot 1: MergeSort vs n log n  ax1.plot(sizes, ms\_steps, label='MergeSort Steps', color='purple')  ax1.plot(sizes, n\_log\_n, label='n log n', color='red', linestyle='--')  ax1.set\_title('MergeSort Steps vs n log n')  ax1.set\_xlabel('Input Size (n)')  ax1.set\_ylabel('Number of Comparisons')  ax1.legend()  ax1.grid(True)  # Plot 2: MergeSort vs n^2  ax2.plot(sizes, ms\_steps, label='MergeSort Steps', color='purple')  ax2.plot(sizes, n\_squared, label='n^2 (Worst Case)', color='green', linestyle=':')  ax2.set\_title('MergeSort Steps vs n^2')  ax2.set\_xlabel('Input Size (n)')  ax2.set\_ylabel('Number of Comparisons')  ax2.legend()  ax2.grid(True)  plt.tight\_layout()  plt.show() |

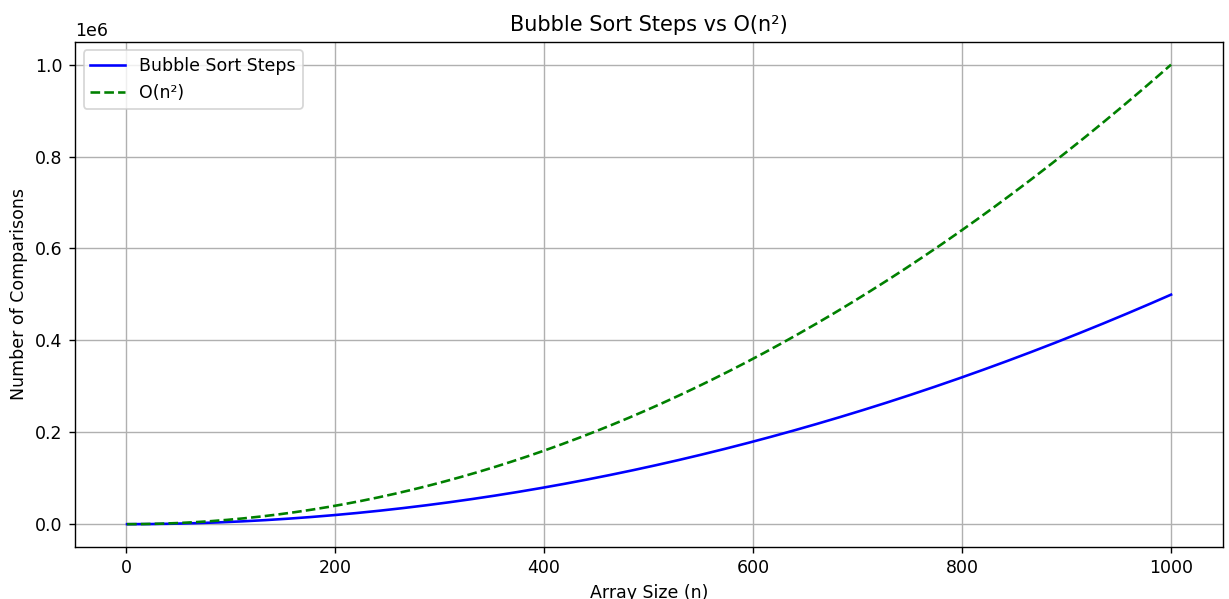
**Output:**



## Bubble Sort

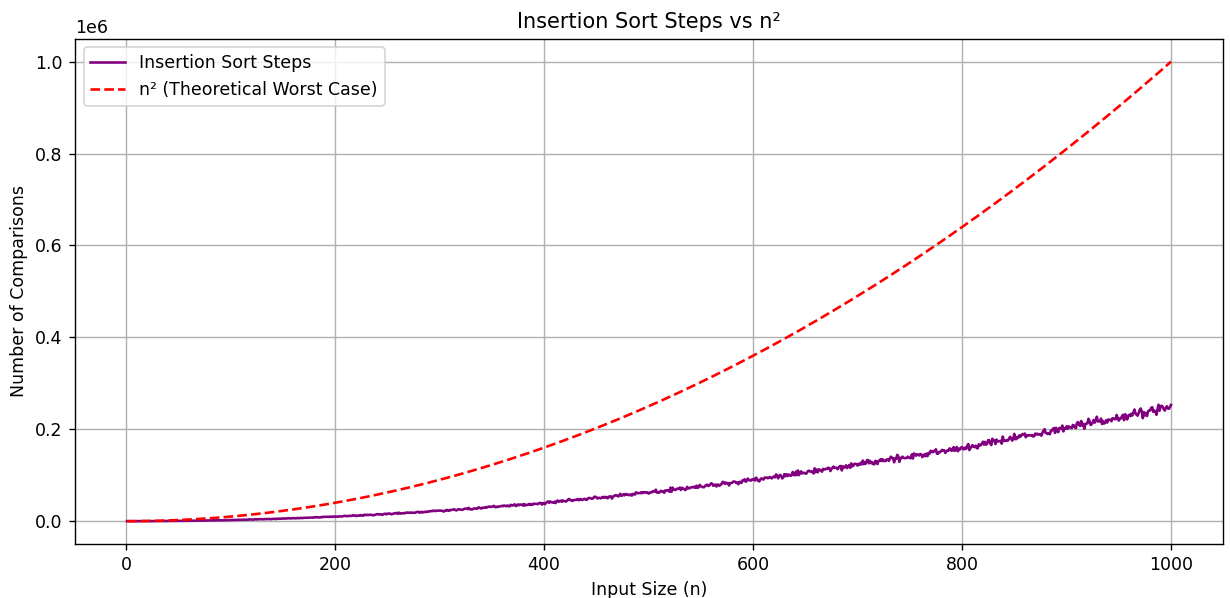
|  |
| --- |
| import matplotlib.pyplot as plt  import random  def bubble\_sort(arr):      """Bubble Sort with comparison counter."""      count = 0      n = len(arr)      for i in range(n - 1):          for j in range(n - i - 1):              count += 1              if arr[j] > arr[j + 1]:                  arr[j], arr[j + 1] = arr[j + 1], arr[j]      return count  def run\_bubble\_sort\_tests(max\_size):      """Run Bubble Sort for array sizes from 1 to max\_size and collect comparison counts."""      sizes = list(range(1, max\_size + 1))      comparisons = []      for n in sizes:          arr = [random.randint(1, 100) for \_ in range(n)]          comparisons.append(bubble\_sort(arr))      return sizes, comparisons  # Run and plot  max\_size = 1000  x\_values, count\_values = run\_bubble\_sort\_tests(max\_size)  n\_squared = [n \*\* 2 for n in x\_values]  plt.figure(figsize=(10, 5))  plt.plot(x\_values, count\_values, label='Bubble Sort Steps', color='blue')  plt.plot(x\_values, n\_squared, label='O(n²)', color='green', linestyle='--')  plt.xlabel('Array Size (n)')  plt.ylabel('Number of Comparisons')  plt.title('Bubble Sort Steps vs O(n²)')  plt.legend()  plt.grid(True)  plt.tight\_layout()  plt.show() |

**Output:**



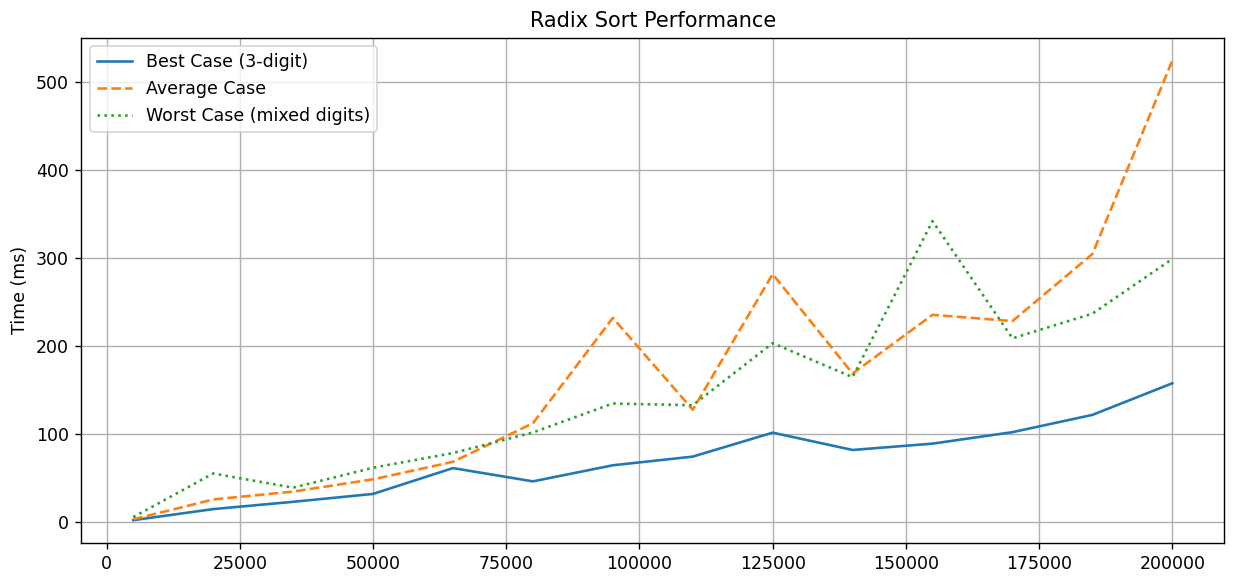
## Insertion Sort

|  |
| --- |
| import matplotlib.pyplot as plt  import random  def insertion\_sort(arr, counter):      """Insertion Sort with comparison counter."""      for i in range(1, len(arr)):          key = arr[i]          j = i - 1          while j >= 0 and arr[j] > key:              counter[0] += 1              arr[j + 1] = arr[j]              j -= 1          arr[j + 1] = key      return arr  def run\_insertion\_sort(n):      """Run Insertion Sort and return number of comparisons."""      arr = random.sample(range(2 \* n), n)      counter = [0]      insertion\_sort(arr, counter)      return counter[0]  # Generate input sizes and step counts  sizes = list(range(1, 1001))  is\_steps = [run\_insertion\_sort(n) for n in sizes]  n\_squared = [n \*\* 2 for n in sizes]  # Plotting  plt.figure(figsize=(10, 5))  plt.plot(sizes, is\_steps, label='Insertion Sort Steps', color='purple')  plt.plot(sizes, n\_squared, label='n² (Theoretical Worst Case)', color='red', linestyle='--')  plt.xlabel('Input Size (n)')  plt.ylabel('Number of Comparisons')  plt.title('Insertion Sort Steps vs n²')  plt.legend()  plt.grid(True)  plt.tight\_layout()  plt.show() |

**Output**:

## Radix sort

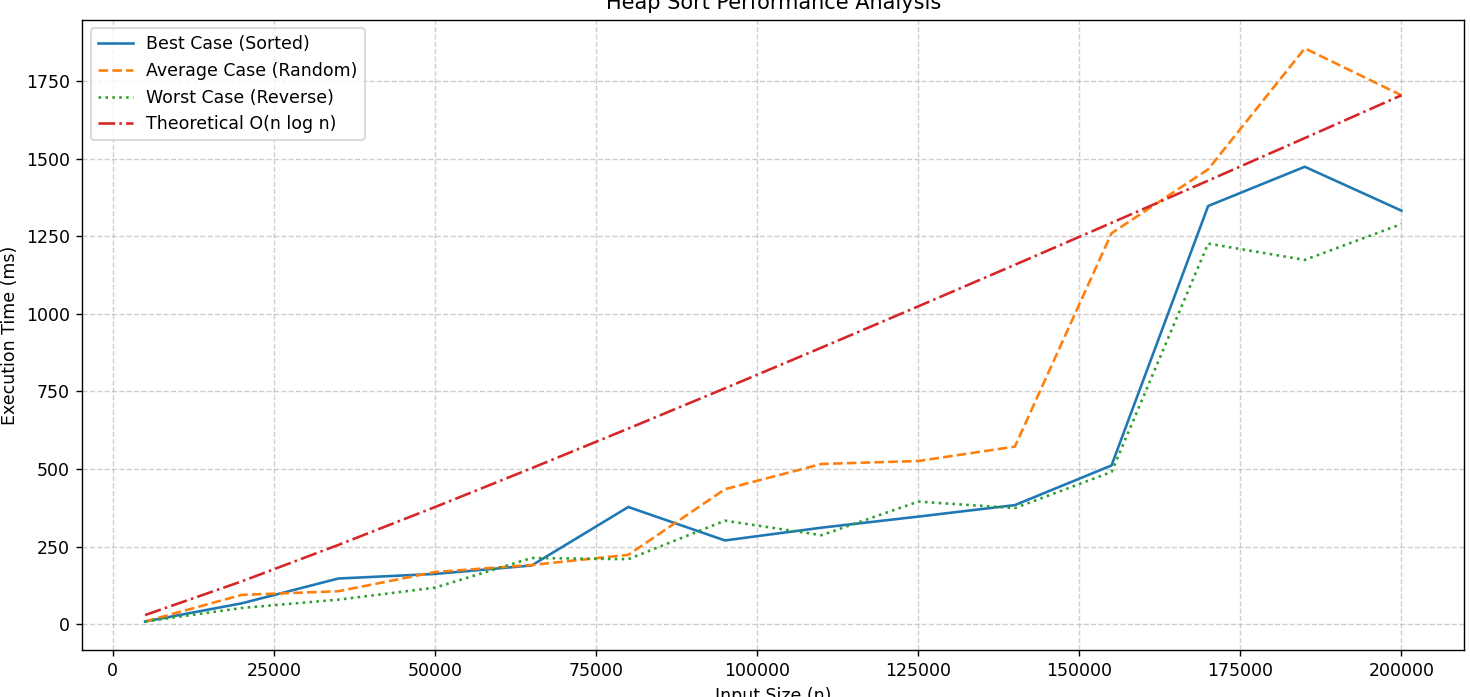
|  |
| --- |
| import random  import time  import matplotlib.pyplot as plt  def counting\_sort(arr, exp):      output = [0] \* len(arr) # hold the sorted values based on the current digit      count = [0] \* 10      for num in arr:          index = (num // exp) % 10          count[index] += 1      for i in range(1, 10):          count[i] += count[i - 1]      for num in reversed(arr):          index = (num // exp) % 10          output[count[index] - 1] = num          count[index] -= 1      for i in range(len(arr)):          arr[i] = output[i]  def radix\_sort(arr):      max\_val = max(arr)      exp = 1      while max\_val // exp > 0:          counting\_sort(arr, exp)          exp \*= 10  # Test parameters  sizes = list(range(5000, 200001, 15000))  results = {'Best': [], 'Average': [], 'Worst': []}  # Test case generators  def best\_case(size):      return [random.randint(100, 999) for \_ in range(size)]  def avg\_case(size):      return [random.randint(0, size) for \_ in range(size)]  def worst\_case(size):      return [random.choice([1, 10\*\*6]) for \_ in range(size)]  # Run tests  for size in sizes:      # Best Case      arr = best\_case(size)      start = time.time()      radix\_sort(arr)      results['Best'].append((time.time() - start) \* 1000)      # Average Case      arr = avg\_case(size)      start = time.time()      radix\_sort(arr)      results['Average'].append((time.time() - start) \* 1000)      # Worst Case      arr = worst\_case(size)      start = time.time()      radix\_sort(arr)      results['Worst'].append((time.time() - start) \* 1000)      print(f"Size: {size:6d} | Best: {results['Best'][-1]:6.2f} ms | "            f"Avg: {results['Average'][-1]:6.2f} ms | Worst: {results['Worst'][-1]:6.2f} ms")  # Plot results  plt.figure(figsize=(10, 5))  plt.plot(sizes, results['Best'], '-', label='Best Case (3-digit)')  plt.plot(sizes, results['Average'], '--', label='Average Case')  plt.plot(sizes, results['Worst'], ':', label='Worst Case (mixed digits)')  plt.title('Radix Sort Performance')  plt.xlabel('Input Size')  plt.ylabel('Time (ms)')  plt.legend()  plt.grid(True)  plt.tight\_layout()  plt.show() |

****

## Heap Sort

|  |
| --- |
| import random  import time  import math  import matplotlib.pyplot as plt  # Heapify function to maintain max-heap property  def heapify(arr, n, i):  largest = i  left = 2 \* i + 1  right = 2 \* i + 2  if left < n and arr[left] > arr[largest]:  largest = left  if right < n and arr[right] > arr[largest]:  largest = right  if largest != i:  arr[i], arr[largest] = arr[largest], arr[i]  heapify(arr, n, largest)  # Heap sort algorithm  def heap\_sort(arr):  n = len(arr)  # Build max-heap  for i in range(n // 2 - 1, -1, -1):  heapify(arr, n, i)  # Extract elements one by one  for i in range(n - 1, 0, -1):  arr[i], arr[0] = arr[0], arr[i]  heapify(arr, i, 0)  # Test case generators  def best\_case(size):  return list(range(size)) # Already sorted  def worst\_case(size):  return list(range(size, 0, -1)) # Reverse sorted  def average\_case(size):  arr = list(range(size))  random.shuffle(arr)  return arr  # Input sizes to test  input\_sizes = list(range(5000, 200001, 15000))  # Lists to hold execution times  best\_times = []  worst\_times = []  avg\_times = []  # Benchmarking loop  for size in input\_sizes:  # Best Case  arr = best\_case(size)  start = time.time()  heap\_sort(arr)  best\_times.append((time.time() - start) \* 1000)  # Worst Case  arr = worst\_case(size)  start = time.time()  heap\_sort(arr)  worst\_times.append((time.time() - start) \* 1000)  # Average Case  arr = average\_case(size)  start = time.time()  heap\_sort(arr)  avg\_times.append((time.time() - start) \* 1000)  print(f"Size: {size:6}, Best: {best\_times[-1]:7.2f} ms, "  f"Worst: {worst\_times[-1]:7.2f} ms, Avg: {avg\_times[-1]:7.2f} ms")  # Theoretical n log n curve for comparison  n\_log\_n = [size \* math.log2(size) for size in input\_sizes]  scale = avg\_times[-1] / n\_log\_n[-1]  n\_log\_n\_scaled = [x \* scale for x in n\_log\_n]  # Plotting results  plt.figure(figsize=(12, 6))  plt.plot(input\_sizes, best\_times, label='Best Case (Sorted)', linestyle='-')  plt.plot(input\_sizes, avg\_times, label='Average Case (Random)', linestyle='--')  plt.plot(input\_sizes, worst\_times, label='Worst Case (Reverse)', linestyle=':')  plt.plot(input\_sizes, n\_log\_n\_scaled, label='Theoretical O(n log n)', linestyle='-.')  plt.title('Heap Sort Performance Analysis')  plt.xlabel('Input Size (n)')  plt.ylabel('Execution Time (ms)')  plt.legend()  plt.grid(True, linestyle='--', alpha=0.6)  plt.tight\_layout()  plt.show() |

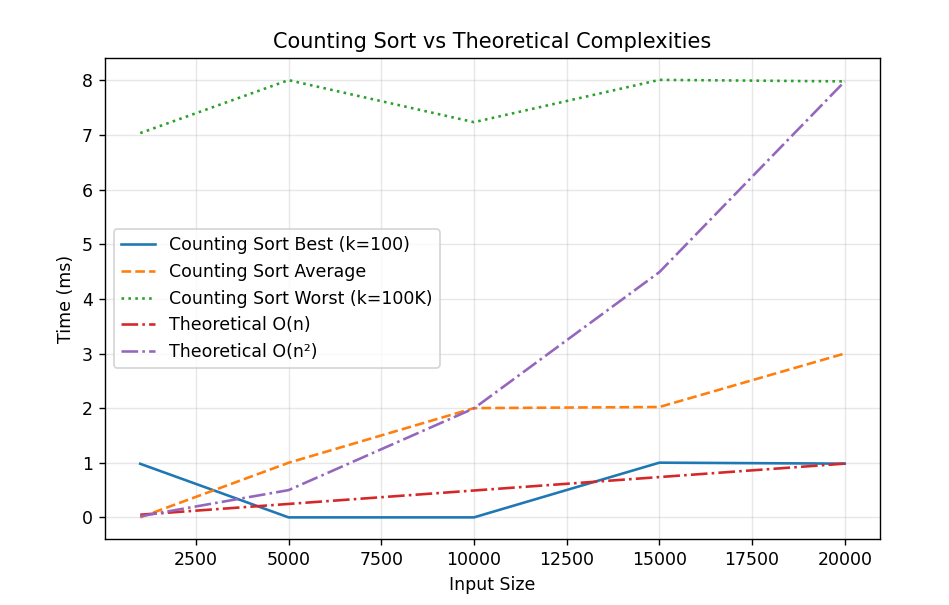
**Output:**

****

## Counting Sort

|  |
| --- |
| import random  import time  import matplotlib.pyplot as plt  def counting\_sort(numbers):  if not numbers:  return numbers.copy()  small = min(numbers)  large = max(numbers)  counts = [0] \* (large - small + 1)  for num in numbers:  counts[num - small] += 1  result = []  for i in range(len(counts)):  result.extend([i + small] \* counts[i])  return result  # Test sizes  sizes = [1000, 5000, 10000, 15000, 20000]  results = {'Best': [], 'Average': [], 'Worst': []}  # Generate test data  def make\_best(size):  return [random.randint(0, 100) for x in range(size)]  def make\_avg(size):  return [random.randint(0, size) for x in range(size)]  def make\_worst(size):  return [random.randint(0, 100000) for x in range(size)]  # Run tests  for size in sizes:  # Best case (small range)  data = make\_best(size)  start = time.time()  counting\_sort(data)  results['Best'].append((time.time() - start) \* 1000)  # Average case  data = make\_avg(size)  start = time.time()  counting\_sort(data)  results['Average'].append((time.time() - start) \* 1000)  # Worst case (large range)  data = make\_worst(size)  start = time.time()  counting\_sort(data)  results['Worst'].append((time.time() - start) \* 1000)  # Calculate theoretical curves (scaled to match)  scale\_n = results['Best'][-1] / sizes[-1]  scale\_n2 = results['Worst'][-1] / (sizes[-1] \*\* 2)  theory\_n = [size \* scale\_n for size in sizes]  theory\_n2 = [(size \*\* 2) \* scale\_n2 for size in sizes]  # Plot results  plt.figure(figsize=(8, 5))  plt.plot(sizes, results['Best'], '-', label='Counting Sort Best (k=100)')  plt.plot(sizes, results['Average'], '--', label='Counting Sort Average')  plt.plot(sizes, results['Worst'], ':', label='Counting Sort Worst (k=100K)')  plt.plot(sizes, theory\_n, '-.', label='Theoretical O(n)')  plt.plot(sizes, theory\_n2, '-.', label='Theoretical O(n²)')  plt.title('Counting Sort vs Theoretical Complexities')  plt.xlabel('Input Size')  plt.ylabel('Time (ms)')  plt.legend()  plt.grid(True, alpha=0.3)  plt.show() |

**Output:**

****

## Depth First Sort

## Breath-First Sort

## Travelling Salesman Problem (TSP)