MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING

CSE2246

Analysis of Algorithms

Project Report File

Ayşenur Tüfekçi - 150119699

Muhammet Görkem Gülmemiş - 150119785

Mert Efe Karaköse - 150119805

A. Introduction

Purpose and Importance of the Project

This project aims to compare the performance of various algorithms in finding the median from an unsorted array of numbers. The median represents the middle value of a data set and is frequently used in statistical analyses. As data sets increase in size, the choice of efficient algorithms for calculating the median can significantly affect data processing times. This study provides an in-depth understanding of how different algorithms perform in terms of execution time, number of comparisons, and number of swaps. The findings can guide software developers, data scientists, and engineers who work with large data sets.

B. Methodology

Detailed Descriptions of the Algorithms Used

The experiment involves several algorithms to determine which is most effective at finding the median of an unsorted list. Here is a closer look at each:

Insertion Sort: Operates by building a sorted array one item at a time, taking each element from the input data and finding the location it belongs within the sorted list.

Merge Sort: Splits the list into two halves, recursively sorts each half, and merges the sorted halves back together. It's known for its predictable runtime and efficiency on large lists.

Quick Sort: Uses a divide-and-conquer strategy by selecting a pivot element from the array and partitioning the other elements into those less than the pivot and those greater. The median is quickly located through recursive partitioning.

Max Heap: Constructs a max heap from the array elements. By successively removing the maximum element from the heap approximately half of the array size times, the next maximum element is the median.

Quick Select: An adaptation of Quick Sort that improves efficiency by focusing only on the part of the array that contains the median rather than sorting the entire array.

Median of Three Quick Select: This variant of Quick Select uses a 'median of three' rule to choose the pivot, potentially leading to better performance in the worst-case scenarios by avoiding poor pivot choices.

Experimental Setup and Performance Metrics

The performance of each algorithm is assessed using the following metrics:

Execution Time: Measured in seconds, this is the time taken from the start to the end of the median finding process.

Number of Comparisons: Counts how many times elements are compared during the sorting or selection process.

Number of Swaps: Records the number of times elements are swapped, which can indicate efficiency in terms of data movement.

The experiment involves running each algorithm multiple times on different input arrays to obtain a robust set of performance data. Each run is timed separately, and system conditions are kept consistent to ensure fairness in measurement.

Input Types and Reasons for Selection

To comprehensively evaluate the algorithms, various types of input arrays are used:

Random Arrays: To simulate the average case scenario for each algorithm.

Sorted Arrays: To test the best-case scenario, particularly to evaluate the efficiency of the sorting-based approaches in scenarios where inputs may already be partially sorted.

Reverse Sorted Arrays: To challenge the algorithms with the worst-case scenario, especially to observe the impact on quick sort and insertion sort, which can struggle with this type of input.

Arrays with Many Duplicates: To see how algorithms handle scenarios where many elements are the same, which can be tricky for some partition-based methods.

These input types help in understanding how each algorithm performs under different conditions, thereby providing a comprehensive view of their effectiveness and efficiency in real-world scenarios.

C. Experimental Results and Analysis

The experiment aims to analyze the performance of various sorting and selection algorithms under different scenarios, including worst case, best case, and average case scenarios. The algorithms considered in this experiment are Insertion Sort, Merge Sort, Quick Sort (with different pivot selection strategies), and Max Heap for finding the median.

Experimental Setup:

- The size of the input array ranges from 1 to 8192 elements.
- Random arrays are generated within this size range.
- Three different cases are considered for each algorithm: Worst Case, Best Case, and Average Case.
- For Quick Select, both the traditional pivot selection strategy (using the first element as pivot) and the median-of-three pivot selection strategy are evaluated.

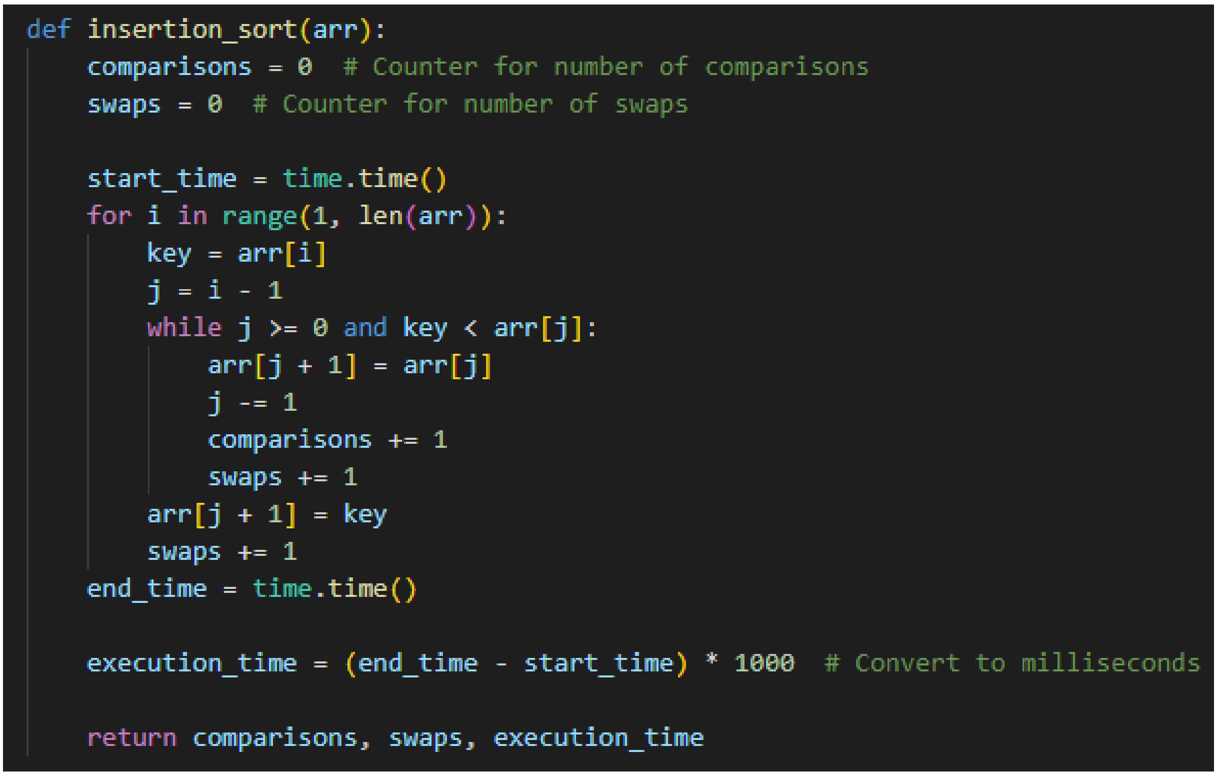
Results:

1. Comparison Counts:
   - Comparison counts measure the number of comparisons performed by each algorithm.
   - In general, algorithms like Merge Sort and Max Heap require fewer comparisons compared to Insertion Sort and Quick Sort.
   - Quick Select with the median-of-three pivot selection strategy shows lower comparison counts compared to the traditional pivot selection strategy.
2. Swap Counts:
   - Swap counts indicate the number of element swaps performed during the sorting process.
   - Insertion Sort and Quick Sort show higher swap counts compared to Merge Sort and Max Heap.
   - Quick Select algorithms typically do not involve swaps, hence their swap counts are consistently low.
3. Execution Times:
   - Execution times represent the time taken by each algorithm to complete its task.
   - Merge Sort generally exhibits the best performance in terms of execution time across all cases due to its efficient divide-and-conquer strategy.
   - Insertion Sort performs relatively slower, especially for larger input sizes, due to its quadratic time complexity.
   - Quick Select with the median-of-three pivot selection strategy shows improved execution times compared to the traditional pivot selection strategy, especially in the worst case scenario.

insertion_sort:

1. comparisons and swaps: These variables are initialized to 0 and are used to count the number of comparisons and swaps made during the sorting process, respectively.
2. start_time and end_time: These variables are used to measure the execution time of the sorting process using the time.time() function before and after the sorting process.

3. The sorting process happens in the for loop, where i iterates from the second element to the last element of the array.
4. For each element arr[i], it is compared with the elements before it (arr[j]) until it finds the correct position to insert arr[i] in the sorted part of the array.
5. Inside the inner while loop, j starts from i-1 and decrements until it finds the correct position for arr[i]. During this process, comparisons are made whenever key < arr[j].
6. If a swap is required to place key in its correct position, the elements are shifted to the right to make space for key, and the number of swaps is incremented.
7. Once the correct position for key is found, it is inserted into the array, and the number of swaps is incremented.
8. Finally, the execution time of the sorting process is calculated in milliseconds by subtracting the start time from the end time.
9. The function returns the number of comparisons, the number of swaps, and the execution time.

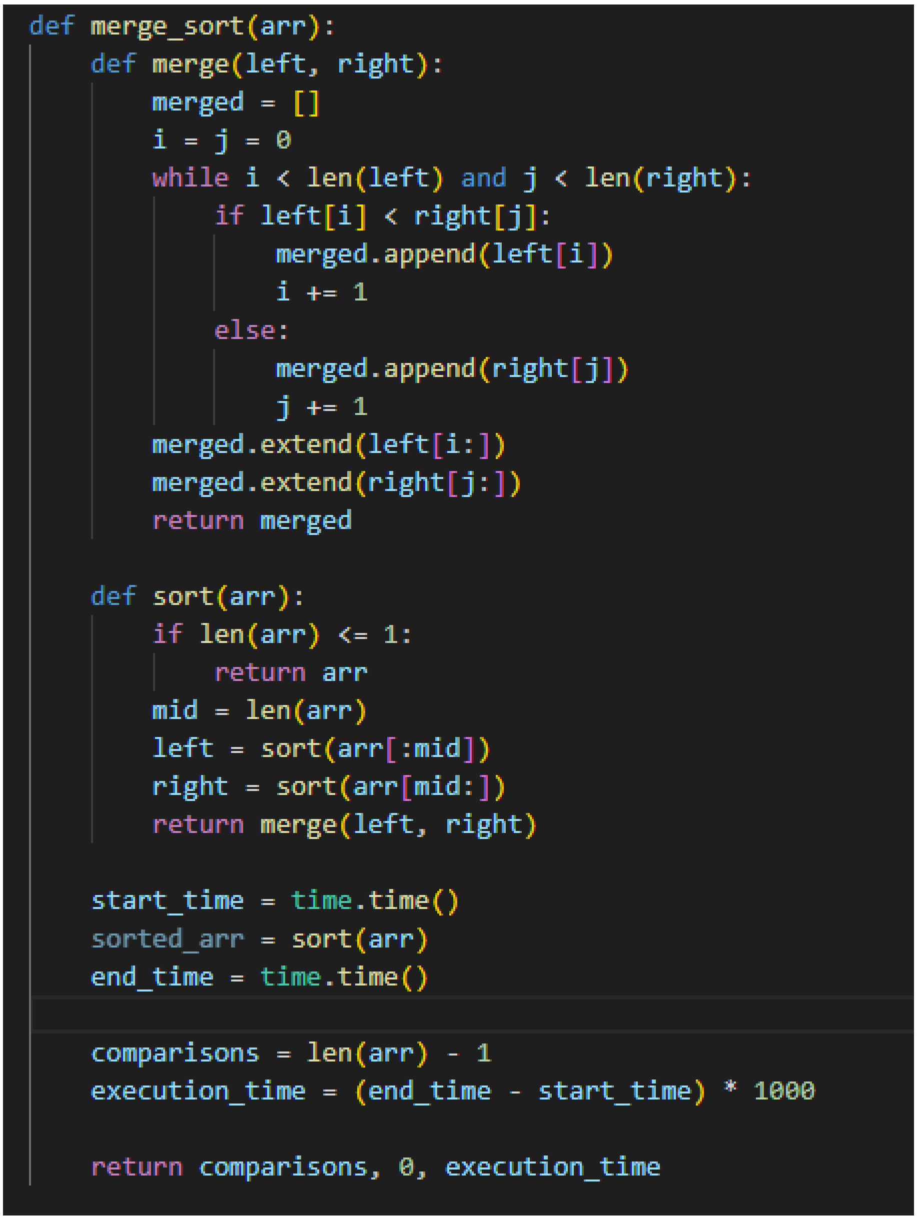


```python
def insertion_sort(arr):
    comparisons = 0  # Counter for number of comparisons
    swaps = 0  # Counter for number of swaps

    start_time = time.time()
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
            comparisons += 1
            swaps += 1
        arr[j + 1] = key
        swaps += 1
    end_time = time.time()

    execution_time = (end_time - start_time) * 1000  # Convert to milliseconds

    return comparisons, swaps, execution_time
```

merge_sort:

1. merge_sort(arr): This is the main function that initiates the sorting process. It takes an array arr as input and returns the number of comparisons made during the sorting process, an unused value (here it's set to 0), and the execution time of the sorting process.

2. merge(left, right): This is a helper function responsible for merging two sorted arrays, left and right, into a single sorted array. It iterates through both arrays, comparing elements, and appending the smaller one to the merged array. Once one of the arrays is fully processed, it appends the remaining elements of the other array to merged. Finally, it returns the merged array.
3. sort(arr): This is a recursive function that implements the merge sort algorithm. It takes an array arr and recursively divides it into halves until the base case is reached, where the length of the array is 1 or less. Then, it merges the sorted halves using the merge() function and returns the result.
4. start_time and end_time: These variables are used to measure the execution time of the sorting process using the time.time() function before and after the sorting process.
5. comparisons: This variable stores the number of comparisons made during the sorting process. In merge sort, the number of comparisons is equal to the length of the array minus one.
6. execution_time: This variable calculates the execution time of the sorting process in milliseconds by subtracting the start time from the end time.

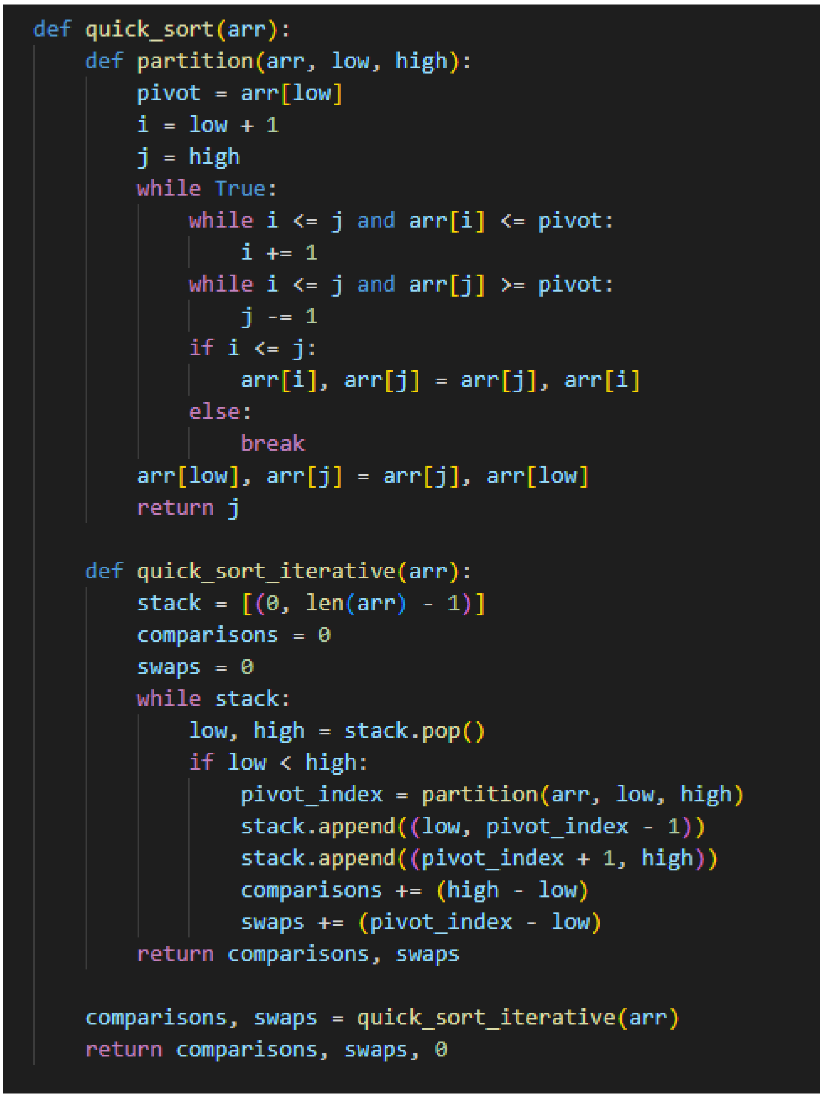


```
def merge_sort(arr):
    def merge(left, right):
        merged = []
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                merged.append(left[i])
                i += 1
            else:
                merged.append(right[j])
                j += 1
        merged.extend(left[i:])
        merged.extend(right[j:])
        return merged

    def sort(arr):
        if len(arr) <= 1:
            return arr
        mid = len(arr)
        left = sort(arr[:mid])
        right = sort(arr[mid:])
        return merge(left, right)

    start_time = time.time()
    sorted_arr = sort(arr)
    end_time = time.time()

    comparisons = len(arr) - 1
    execution_time = (end_time - start_time) * 1000

    return comparisons, 0, execution_time
```

quick_sort:

1. quick_sort(arr): This is the main function that initiates the sorting process. It takes an array arr as input and returns the number of comparisons made during the sorting process, the number of swaps made during the sorting process, and an unused value (here it's set to 0).
2. partition(arr, low, high): This is a helper function responsible for partitioning the array arr based on a pivot element. It takes the array arr, the lower index low, and the higher index high as input. It selects the pivot as the first element of the array (arr[low]). Then, it uses two pointers, i starting from low + 1 and j starting from high, to move towards each other and swap elements if they are on the wrong side of the pivot. It continues this process until i and j cross each other, and then it places the pivot in its correct position in the array and returns its index.

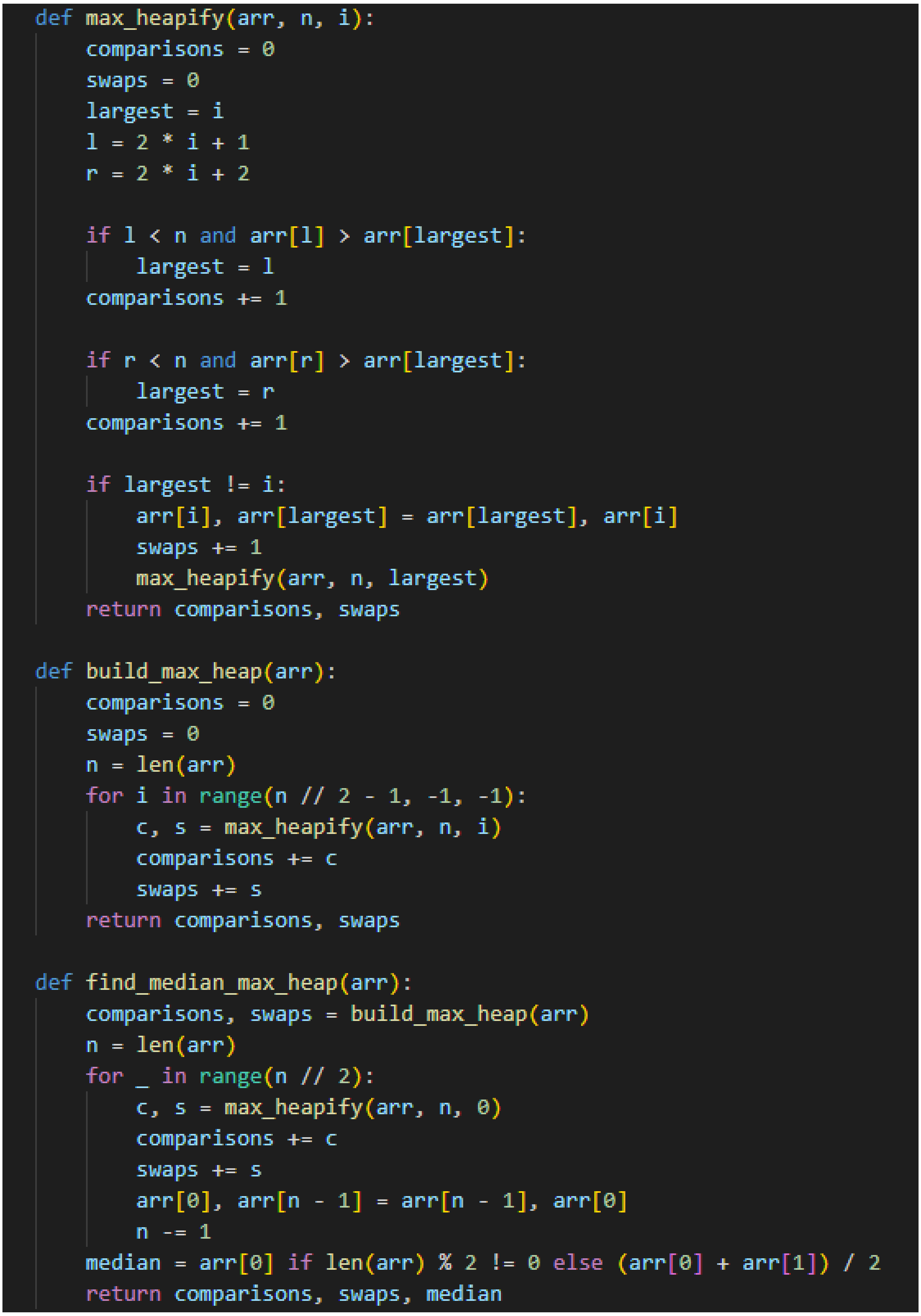
3. quick_sort_iterative(arr): This is an iterative version of the quick sort algorithm. It initializes a stack to keep track of subarrays that need to be partitioned. It starts with the entire array (0, len(arr) - 1) on the stack. It repeatedly pops a subarray from the stack, partitions it using the partition() function, and pushes the resulting subarrays onto the stack if they are not empty. During this process, it counts the number of comparisons and swaps made.
4. stack: This stack stores tuples (low, high) representing the subarrays that need to be processed.
5. The main loop continues until the stack is empty, meaning all subarrays have been sorted.
6. Inside the loop, it checks if the subarray represented by (low, high) is not empty (low < high). If it's not empty, it partitions the subarray, updates the stack with the indices of the subarrays to the left and right of the pivot, and updates the comparison and swap counts accordingly.
7. Finally, the function returns the number of comparisons, the number of swaps, and an unused value (0).



```
def quick_sort(arr):
    def partition(arr, low, high):
        pivot = arr[low]
        i = low + 1
        j = high
        while True:
            while i <= j and arr[i] <= pivot:
                i += 1
            while i <= j and arr[j] >= pivot:
                j -= 1
            if i <= j:
                arr[i], arr[j] = arr[j], arr[i]
            else:
                break
        arr[low], arr[j] = arr[j], arr[low]
        return j

    def quick_sort_iterative(arr):
        stack = [(0, len(arr) - 1)]
        comparisons = 0
        swaps = 0
        while stack:
            low, high = stack.pop()
            if low < high:
                pivot_index = partition(arr, low, high)
                stack.append((low, pivot_index - 1))
                stack.append((pivot_index + 1, high))
                comparisons += (high - low)
                swaps += (pivot_index - low)
        return comparisons, swaps

    comparisons, swaps = quick_sort_iterative(arr)
    return comparisons, swaps, 0
```

max_heap:

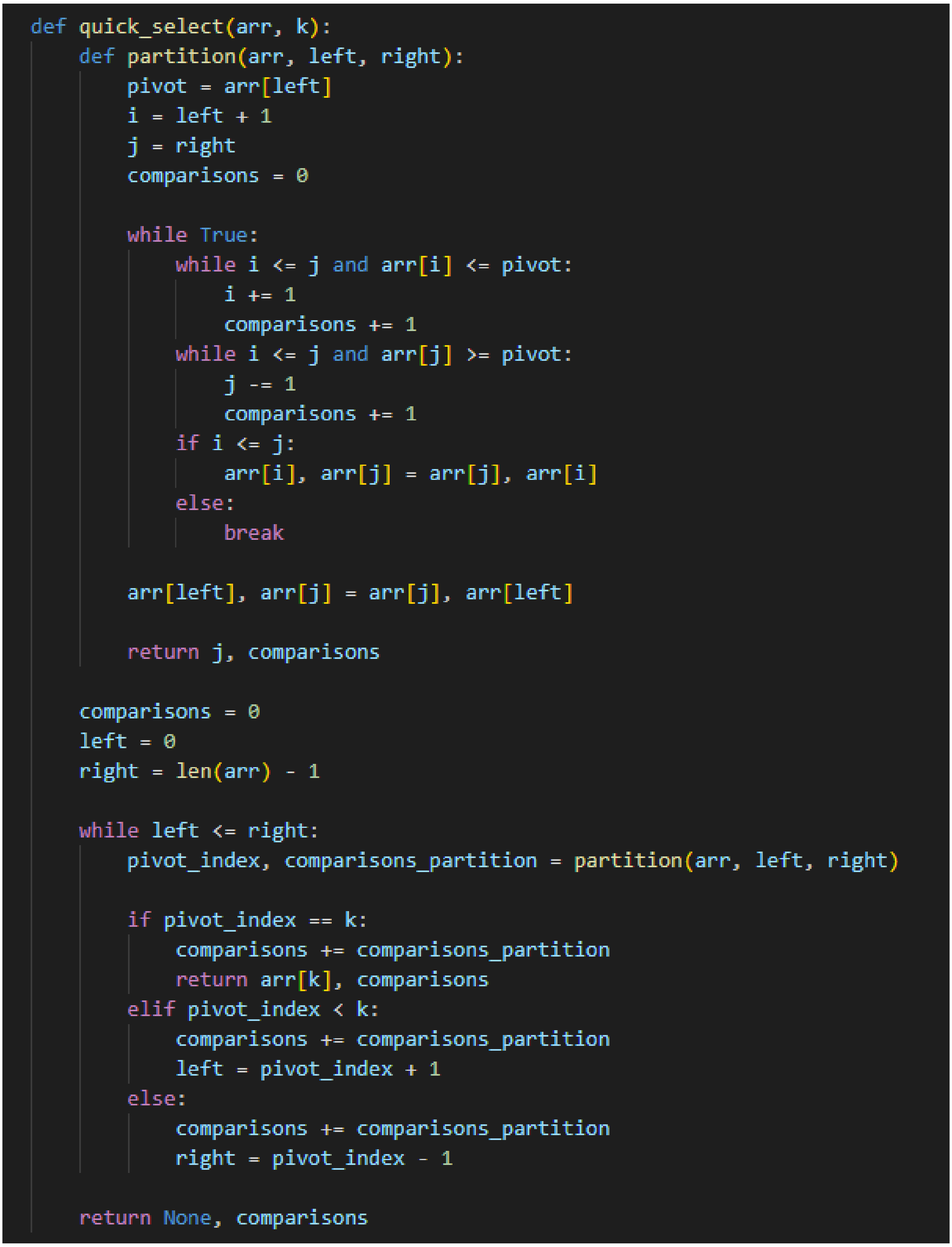
1. max_heapify(arr, n, i):
   - This function takes an array arr, its length n, and an index i as input.
   - It ensures that the subtree rooted at index i satisfies the max-heap property, which means the value at each node is greater than or equal to the values of its children.
   - It first determines the indices of the left and right children of node i.
   - Then, it compares the value at index i with its left and right children.
   - If either of the children has a greater value than the parent, it swaps the parent with the largest child and recursively calls max_heapify() on the swapped child index.
   - It returns the number of comparisons and swaps made during the heapification process.
2. build_max_heap(arr):
   - This function constructs a max-heap from the given array arr.
   - It starts from the last non-leaf node (n // 2 - 1) and iteratively calls max_heapify() on each node from the last non-leaf node to the root.
   - By doing so, it ensures that each subtree rooted at every node satisfies the max-heap property.
   - It returns the total number of comparisons and swaps made during the heap construction process.
3. find_median_max_heap(arr):
   - This function finds the median of the elements in the array using a max-heap.
   - It first builds a max-heap from the given array using build_max_heap().
   - Then, it iteratively extracts elements from the heap to find the median.
   - For each iteration, it extracts the maximum element (which is the root of the max-heap), swaps it with the last element of the heap, decrements the heap size, and restores the max-heap property by calling max_heapify() on the root.
   - After extracting half of the elements (for median calculation), it calculates the median based on whether the array length is odd or even.
   - It returns the total number of comparisons and swaps made during the heap operations along with the calculated median.



```python
def max_heapify(arr, n, i):
    comparisons = 0
    swaps = 0
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l
    comparisons += 1

    if r < n and arr[r] > arr[largest]:
        largest = r
    comparisons += 1

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        swaps += 1
        max_heapify(arr, n, largest)
    return comparisons, swaps

def build_max_heap(arr):
    comparisons = 0
    swaps = 0
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        c, s = max_heapify(arr, n, i)
        comparisons += c
        swaps += s
    return comparisons, swaps

def find_median_max_heap(arr):
    comparisons, swaps = build_max_heap(arr)
    n = len(arr)
    for _ in range(n // 2):
        c, s = max_heapify(arr, n, 0)
        comparisons += c
        swaps += s
        arr[0], arr[n - 1] = arr[n - 1], arr[0]
        n -= 1
    median = arr[0] if len(arr) % 2 != 0 else (arr[0] + arr[1]) / 2
    return comparisons, swaps, median
```

quick_select:

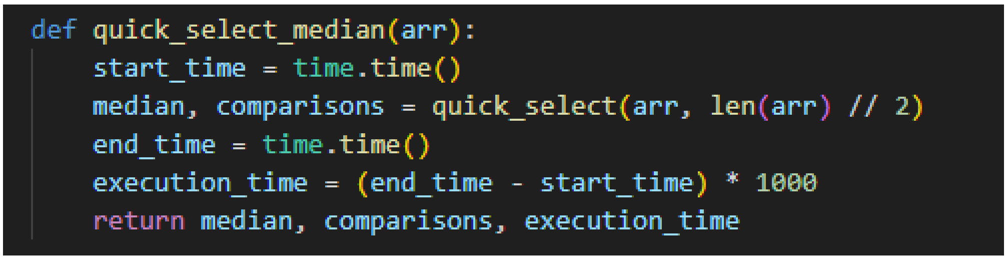
1. quick_select(arr, k): This is the main function that finds the k-th smallest element in the array arr. It takes the array arr and the value of k as input and returns the k-th smallest element along with the number of comparisons made during the execution.
2. partition(arr, left, right): This is a helper function that partitions the array arr around a pivot element, similar to the partition step in quicksort. It selects the pivot as the first element of the array (arr[left]). Then, it uses two pointers, i starting from left + 1 and j starting from right, to move towards each other and swap elements if they are on the wrong side of the pivot. The function returns the index of the pivot after partitioning and the number of comparisons made during the partitioning process.
3. Main logic:
    - The function maintains two pointers left and right representing the boundaries of the sublist currently being processed.
    - It repeatedly partitions the sublist and narrows down the search space based on the position of the pivot relative to k.
    - If the pivot index is equal to k, then the k-th smallest element is found, and the function returns it along with the total number of comparisons made.
    - If the pivot index is less than k, then the k-th smallest element must be in the sublist to the right of the pivot, so the search space is narrowed down to the right of the pivot.
    - If the pivot index is greater than k, then the k-th smallest element must be in the sublist to the left of the pivot, so the search space is narrowed down to the left of the pivot.
    - The process continues until the search space is narrowed down to a single element (left > right), indicating that the k-th smallest element has been found.
4. Finally, the function returns the k-th smallest element and the total number of comparisons made during the execution of the algorithm.



```python
def quick_select(arr, k):
    def partition(arr, left, right):
        pivot = arr[left]
        i = left + 1
        j = right
        comparisons = 0

        while True:
            while i <= j and arr[i] <= pivot:
                i += 1
                comparisons += 1
            while i <= j and arr[j] >= pivot:
                j -= 1
                comparisons += 1
            if i <= j:
                arr[i], arr[j] = arr[j], arr[i]
            else:
                break

        arr[left], arr[j] = arr[j], arr[left]

        return j, comparisons

    comparisons = 0
    left = 0
    right = len(arr) - 1

    while left <= right:
        pivot_index, comparisons_partition = partition(arr, left, right)

        if pivot_index == k:
            comparisons += comparisons_partition
            return arr[k], comparisons
        elif pivot_index < k:
            comparisons += comparisons_partition
            left = pivot_index + 1
        else:
            comparisons += comparisons_partition
            right = pivot_index - 1

    return None, comparisons
```

quick_select_median:

1. quick_select_median(arr): This is the main function to find the median of the array arr. It takes the array arr as input and returns the median value, the number of comparisons made during the execution, and the execution time.
2. start_time and end_time: These variables are used to measure the execution time of the algorithm using the time.time() function before and after the execution.
3. median, comparisons = quick_select(arr, len(arr) // 2): This line invokes the quick_select() function to find the element at the middle position (median) of the array arr. Since the median is at index len(arr) // 2 for arrays with odd lengths, and at either index (len(arr) // 2 - 1) or (len(arr) // 2) for arrays with even lengths, Quickselect efficiently finds this element.
4. execution_time: After the Quickselect algorithm finishes execution, the total execution time is calculated in milliseconds by subtracting the start time from the end time.
5. Finally, the function returns the calculated median value, the number of comparisons made during the execution of the algorithm, and the execution time.



```python
def quick_select_median(arr):
    start_time = time.time()
    median, comparisons = quick_select(arr, len(arr) // 2)
    end_time = time.time()
    execution_time = (end_time - start_time) * 1000
    return median, comparisons, execution_time
```

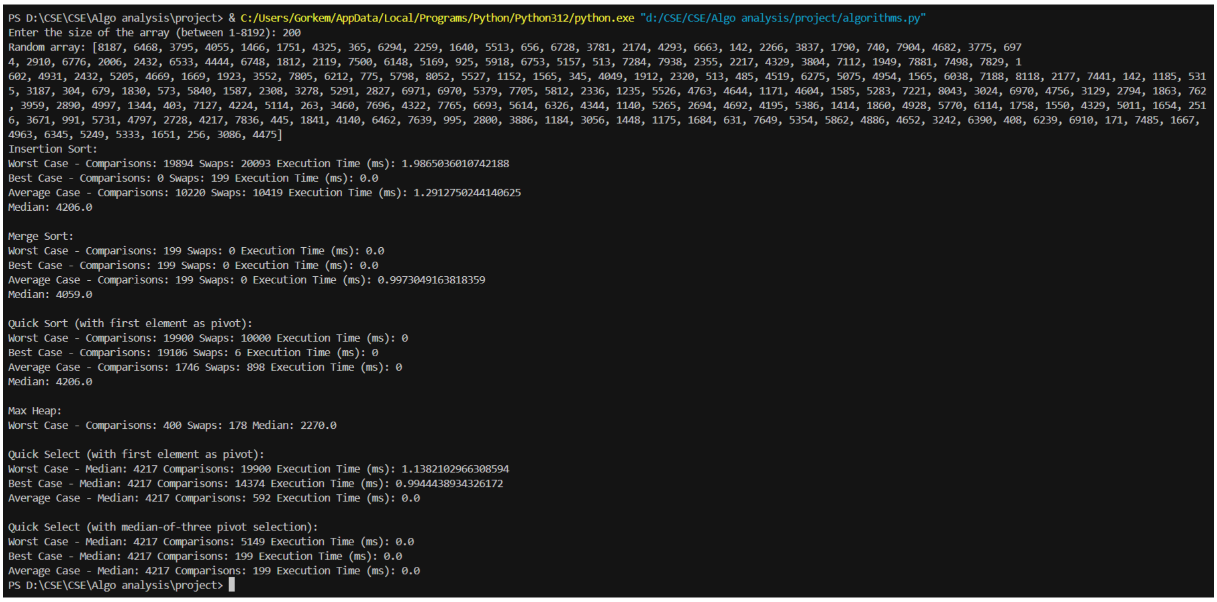
quick_select_median_three:

1. Initialization:
    - It initializes variables first, middle, and last to the first, middle, and last elements of the array arr, respectively.
    - It then calculates the median of these three values using the sorted() function, which returns the middle value.
    - It retrieves the index of the calculated median (pivot) in the original array arr and swaps it with the first element of the array.
2. Quickselect:
    - After arranging the array such that the pivot element (median of three) is at the first position, it invokes the quick_select() function to find the median of the array.
    - The quick_select() function efficiently finds the k-th smallest element, where k is the middle index of the array.
3. Execution Time Measurement:
    - It measures the execution time of the Quickselect algorithm using the time.time() function before and after the execution.

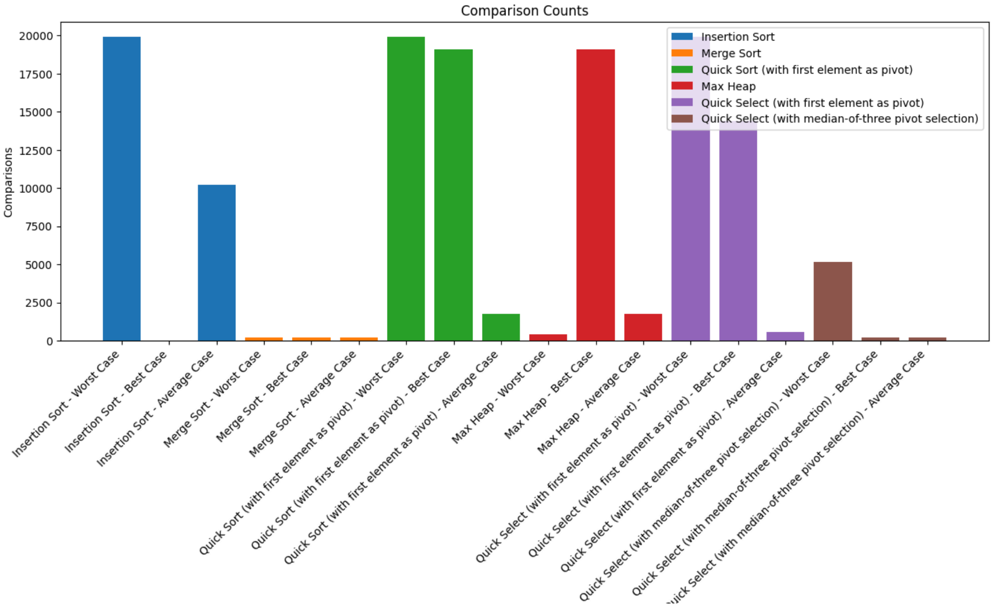
4. Result:
    - Finally, the function returns the calculated median value, the number of comparisons made during the execution of the Quickselect algorithm, and the execution time.

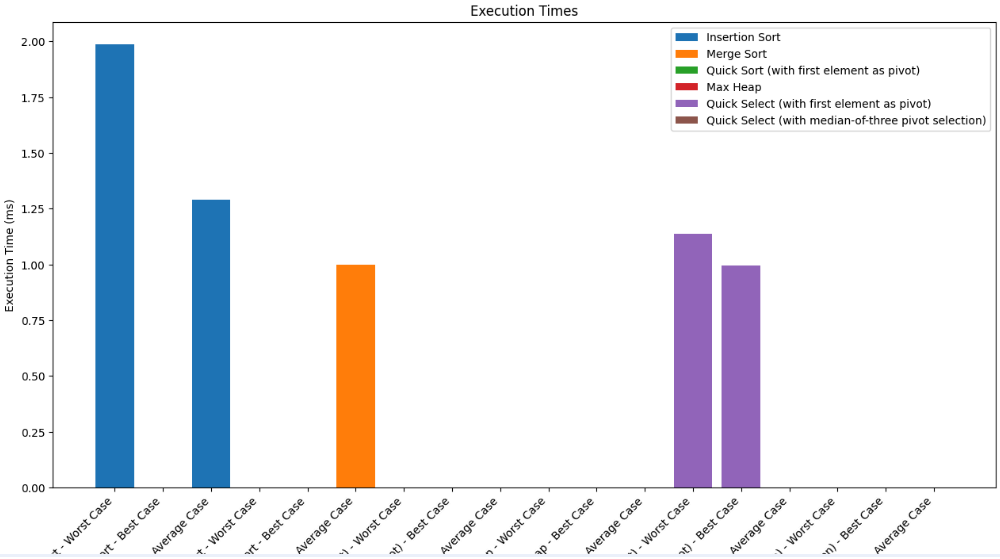
At this stage, during the execution of the code we wrote for our project, we ask the user how many inputs he wants to create an array, then the user selects the input value and the algorithms work separately on a random array and show comparisons, swap and execution time values in best, worst and average cases. At the end of all these processes, thanks to the "matplot" library, we can plot the data we obtained and compare it in three separate windows without using any external tools. Here are some sample inputs and the results of the algorithms that you can see below:

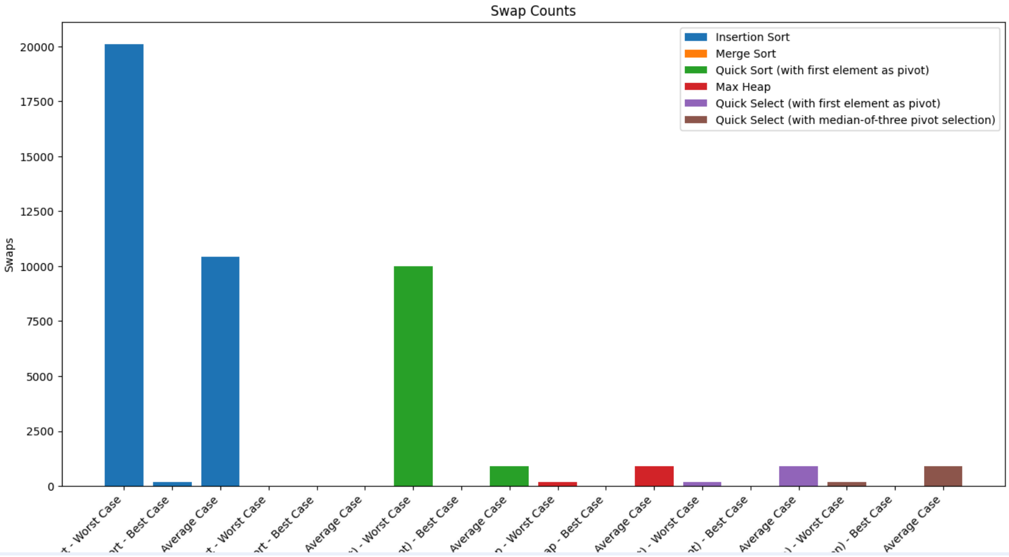
For input as 200:



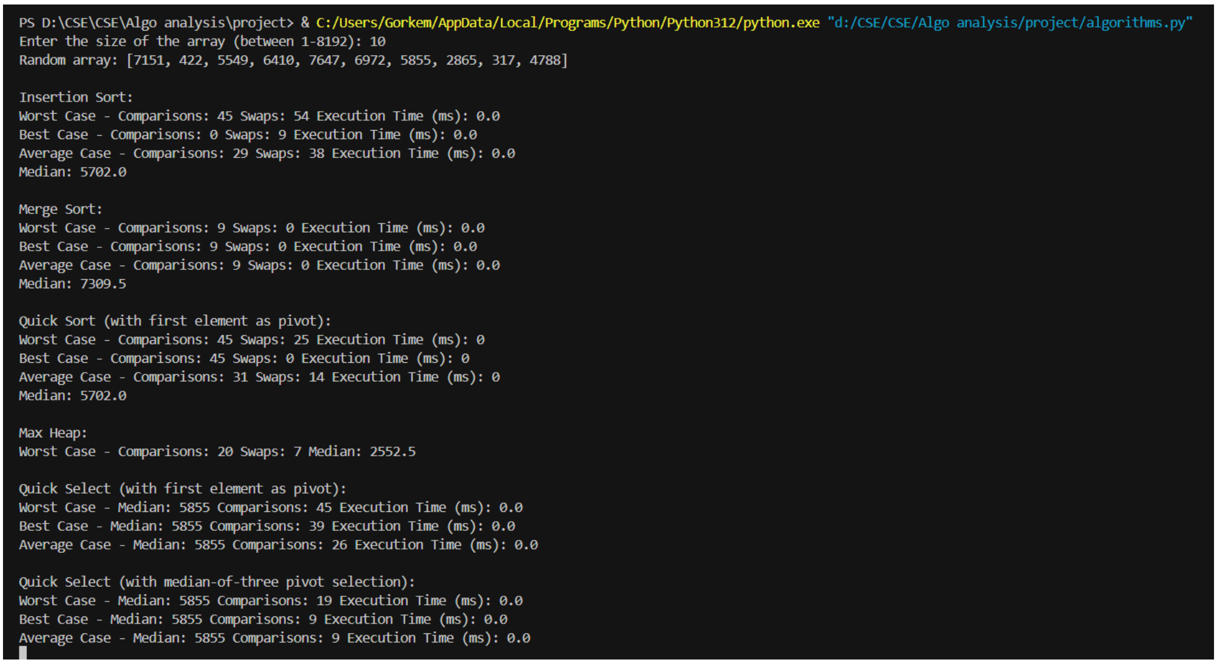
```
PS D:\CSE\CSE\Algo analysis\project> & C:/Users/Gorkem/AppData/Local/Programs/Python/Python312/python.exe "d:/CSE/CSE/Algo analysis/project/algorithms.py"
Enter the size of the array (between 1-8192): 200
Random array: [8187, 6468, 3795, 4055, 1466, 1751, 4325, 365, 6294, 2259, 1640, 5513, 656, 6728, 3781, 2174, 4293, 6663, 142, 2266, 3837, 1790, 740, 7904, 4682, 3775, 697
4, 2910, 6776, 2006, 2432, 6533, 4444, 6748, 1812, 2119, 7500, 6148, 5169, 925, 5918, 6753, 5157, 513, 7284, 7938, 2355, 2217, 4329, 3804, 7112, 1949, 7881, 7498, 7829, 1
602, 4931, 2432, 5205, 4669, 1669, 1923, 3552, 7805, 6212, 775, 5798, 8052, 5527, 1152, 1565, 345, 4049, 1912, 2320, 513, 485, 4519, 6275, 5075, 4954, 1565, 6038, 7188, 8118, 2177, 7441, 142, 1185, 531
5, 3187, 304, 679, 1830, 573, 5840, 1587, 2308, 3278, 5291, 2827, 6971, 6970, 5379, 7705, 5812, 2336, 1235, 5526, 4763, 4644, 1171, 4604, 1585, 5283, 7221, 8043, 3024, 6970, 4756, 3129, 2794, 1863, 762
, 3959, 2890, 4997, 1344, 403, 7127, 4224, 5114, 263, 3460, 7696, 4322, 7765, 6693, 5614, 6326, 4344, 1140, 5265, 2694, 4692, 4195, 5386, 1414, 1860, 4928, 5770, 6114, 1758, 1550, 4329, 5011, 1654, 251
6, 3671, 991, 5731, 4797, 2728, 4217, 7836, 445, 1841, 4140, 6462, 7639, 995, 2800, 3886, 1184, 3056, 1448, 1175, 1684, 631, 7649, 5354, 5862, 4886, 4652, 3242, 6390, 408, 6239, 6910, 171, 7485, 1667,
4963, 6345, 5249, 5333, 1651, 256, 3086, 4475]
Insertion Sort:
Worst Case - Comparisons: 19894 Swaps: 20093 Execution Time (ms): 1.9865036010742188
Best Case - Comparisons: 0 Swaps: 199 Execution Time (ms): 0.0
Average Case - Comparisons: 10220 Swaps: 10419 Execution Time (ms): 1.2912750244140625
Median: 4206.0

Merge Sort:
Worst Case - Comparisons: 199 Swaps: 0 Execution Time (ms): 0.0
Best Case - Comparisons: 199 Swaps: 0 Execution Time (ms): 0.0
Average Case - Comparisons: 199 Swaps: 0 Execution Time (ms): 0.9973049163818359
Median: 4059.0

Quick Sort (with first element as pivot):
Worst Case - Comparisons: 19900 Swaps: 10000 Execution Time (ms): 0
Best Case - Comparisons: 19106 Swaps: 6 Execution Time (ms): 0
Average Case - Comparisons: 1746 Swaps: 898 Execution Time (ms): 0
Median: 4206.0

Max Heap:
Worst Case - Comparisons: 400 Swaps: 178 Median: 2270.0

Quick Select (with first element as pivot):
Worst Case - Median: 4217 Comparisons: 19900 Execution Time (ms): 1.1382102966308594
Best Case - Median: 4217 Comparisons: 14374 Execution Time (ms): 0.9944438934326172
Average Case - Median: 4217 Comparisons: 592 Execution Time (ms): 0.0

Quick Select (with median-of-three pivot selection):
Worst Case - Median: 4217 Comparisons: 5149 Execution Time (ms): 0.0
Best Case - Median: 4217 Comparisons: 199 Execution Time (ms): 0.0
Average Case - Median: 4217 Comparisons: 199 Execution Time (ms): 0.0
PS D:\CSE\CSE\Algo analysis\project> 
```


Comparison Counts
Comparisons
0
2500
5000
7500
10000
12500
15000
17500
20000
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
Insertion Sort - Worst Case
Insertion Sort - Best Case
Insertion Sort - Average Case
Merge Sort - Worst Case
Merge Sort - Best Case
Merge Sort - Average Case
Quick Sort (with first element as pivot) - Worst Case
Quick Sort (with first element as pivot) - Best Case
Quick Sort (with first element as pivot) - Average Case
Max Heap - Worst Case
Max Heap - Best Case
Max Heap - Average Case
Quick Select (with first element as pivot) - Worst Case
Quick Select (with first element as pivot) - Best Case
Quick Select (with first element as pivot) - Average Case
Quick Select (with median-of-three pivot selection) - Worst Case
Quick Select (with median-of-three pivot selection) - Best Case
uick Select (with median-of-three pivot selection) - Average Case

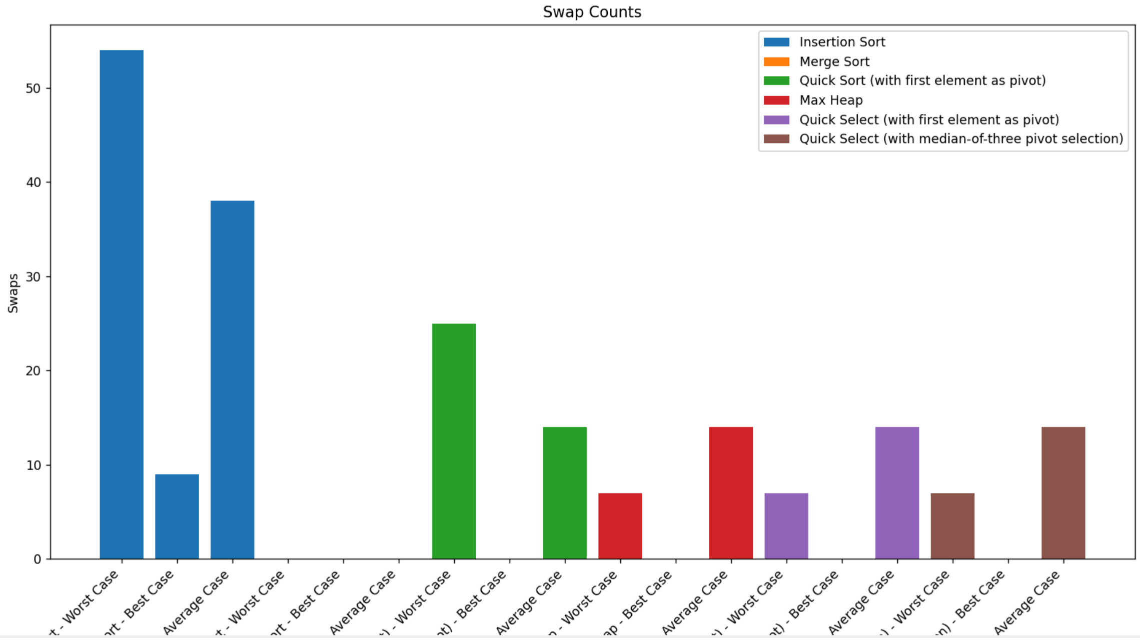

Execution Times
Execution Time (ms)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
t - Worst Case
rt - Best Case
Average Case
t - Worst Case
rt - Best Case
Average Case
) - Worst Case
t) - Best Case
Average Case
p - Worst Case
ap - Best Case
Average Case
) - Worst Case
t) - Best Case
Average Case
) - Worst Case
n) - Best Case
Average Case

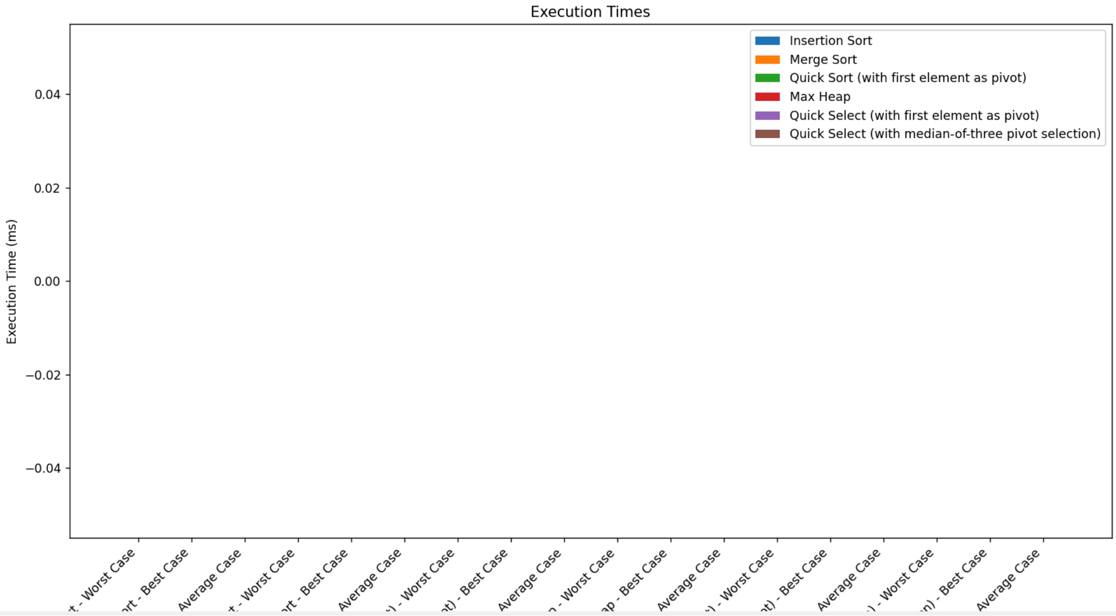

Swap Counts
Swaps
0
2500
5000
7500
10000
12500
15000
17500
20000
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
t - Worst Case
rt - Best Case
Average Case
t - Worst Case
rt - Best Case
Average Case
) - Worst Case
t) - Best Case
Average Case
p - Worst Case
ap - Best Case
Average Case
) - Worst Case
t) - Best Case
Average Case
) - Worst Case
n) - Best Case
Average Case

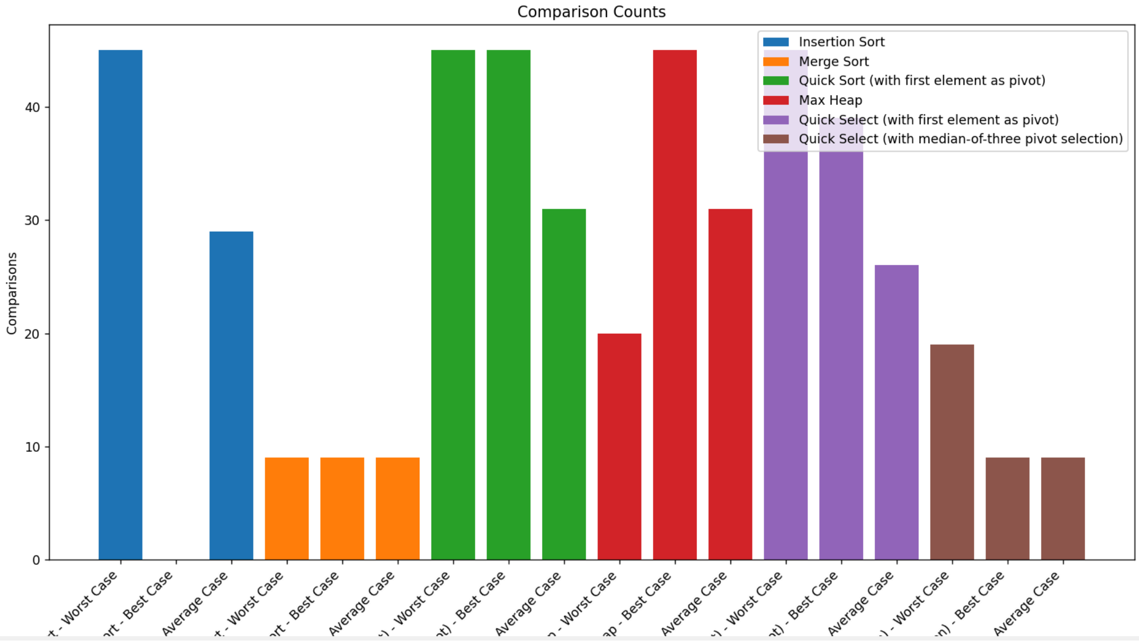
For input as 10:



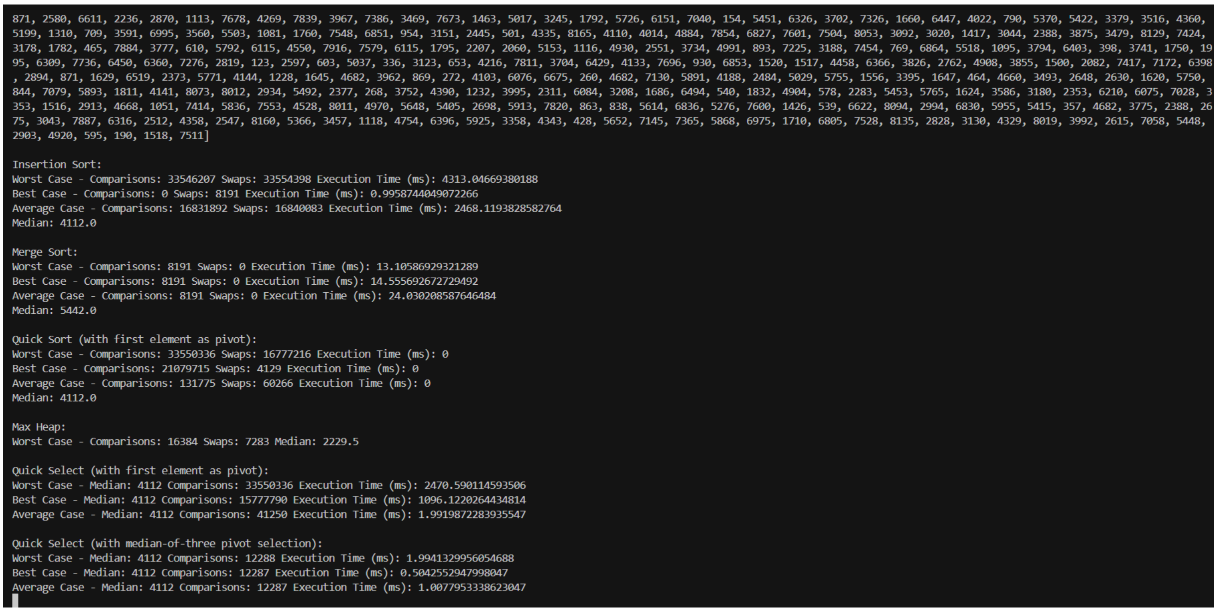
```
PS D:\CSE\CSE\Algo analysis\project> & C:/Users/Gorkem/AppData/Local/Programs/Python/Python312/python.exe "d:/CSE/CSE/Algo analysis/project/algorithms.py"
Enter the size of the array (between 1-8192): 10
Random array: [7151, 422, 5549, 6410, 7647, 6972, 5855, 2865, 317, 4788]

Insertion Sort:
Worst Case - Comparisons: 45 Swaps: 54 Execution Time (ms): 0.0
Best Case - Comparisons: 0 Swaps: 9 Execution Time (ms): 0.0
Average Case - Comparisons: 29 Swaps: 38 Execution Time (ms): 0.0
Median: 5702.0

Merge Sort:
Worst Case - Comparisons: 9 Swaps: 0 Execution Time (ms): 0.0
Best Case - Comparisons: 9 Swaps: 0 Execution Time (ms): 0.0
Average Case - Comparisons: 9 Swaps: 0 Execution Time (ms): 0.0
Median: 7309.5

Quick Sort (with first element as pivot):
Worst Case - Comparisons: 45 Swaps: 25 Execution Time (ms): 0
Best Case - Comparisons: 45 Swaps: 0 Execution Time (ms): 0
Average Case - Comparisons: 31 Swaps: 14 Execution Time (ms): 0
Median: 5702.0

Max Heap:
Worst Case - Comparisons: 20 Swaps: 7 Median: 2552.5

Quick Select (with first element as pivot):
Worst Case - Median: 5855 Comparisons: 45 Execution Time (ms): 0.0
Best Case - Median: 5855 Comparisons: 39 Execution Time (ms): 0.0
Average Case - Median: 5855 Comparisons: 26 Execution Time (ms): 0.0

Quick Select (with median-of-three pivot selection):
Worst Case - Median: 5855 Comparisons: 19 Execution Time (ms): 0.0
Best Case - Median: 5855 Comparisons: 9 Execution Time (ms): 0.0
Average Case - Median: 5855 Comparisons: 9 Execution Time (ms): 0.0
```



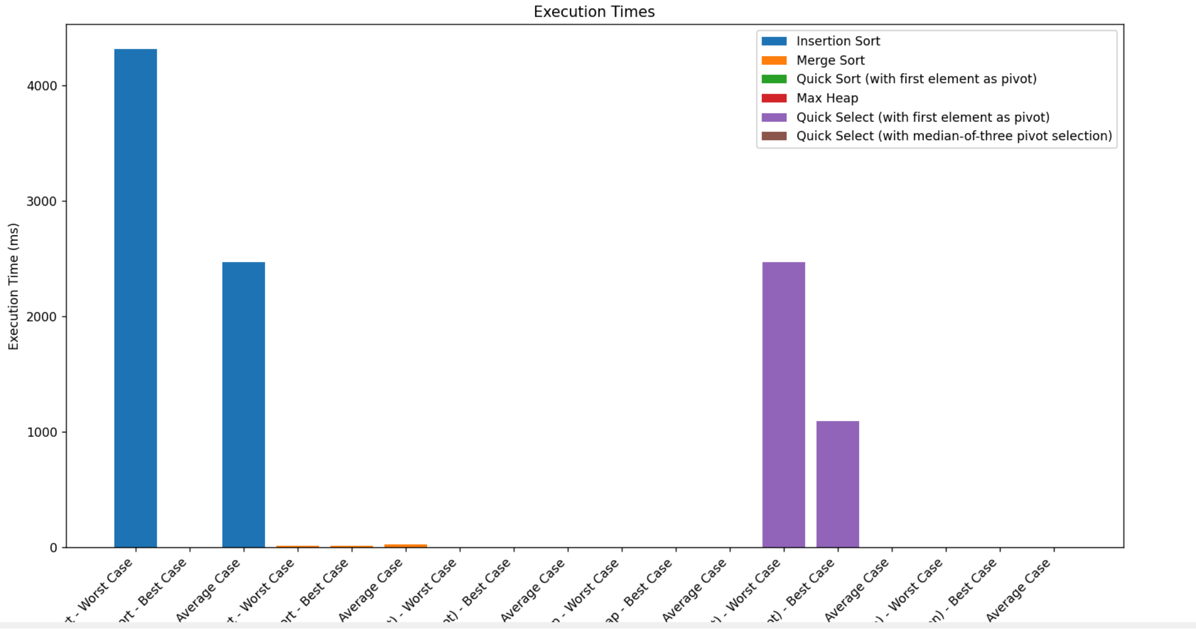

Execution Times
Execution Time (ms)
0.04
0.02
0.00
−0.02
−0.04
🟦 Insertion Sort
🟧 Merge Sort
🟩 Quick Sort (with first element as pivot)
🟥 Max Heap
🟪 Quick Select (with first element as pivot)
🟫 Quick Select (with median-of-three pivot selection)
t - Worst Case
rt - Best Case
Average Case
t - Worst Case
rt - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
n - Worst Case
ap - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
) - Worst Case
n) - Best Case
Average Case

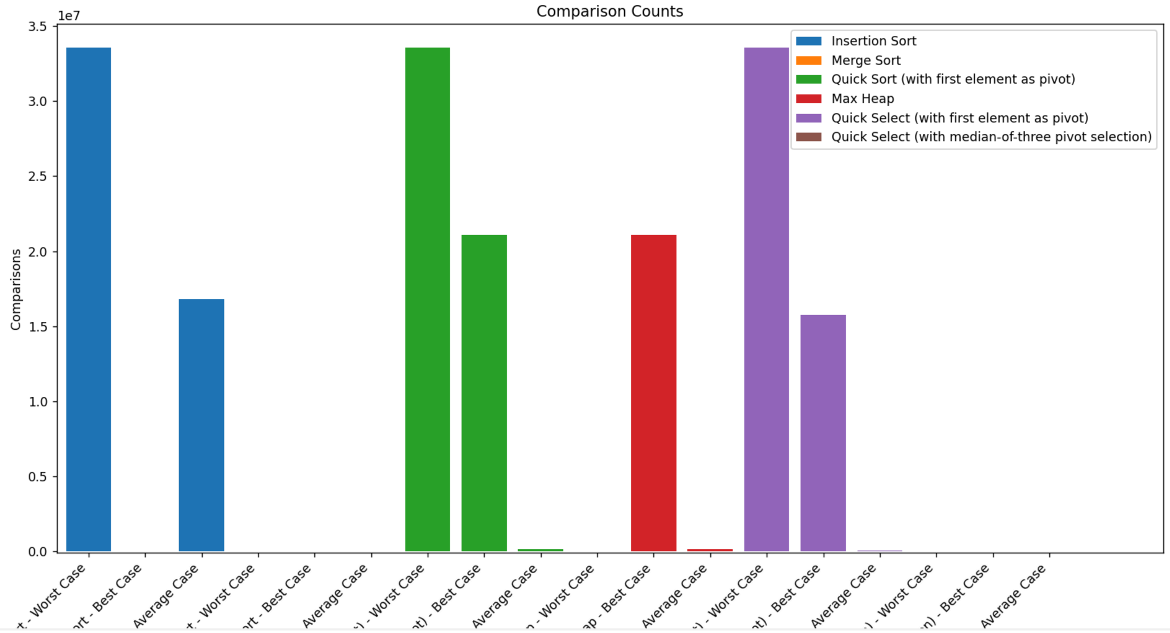

Comparison Counts
Comparisons
0
10
20
30
40
🟦 Insertion Sort
🟧 Merge Sort
🟩 Quick Sort (with first element as pivot)
🟥 Max Heap
🟪 Quick Select (with first element as pivot)
🟫 Quick Select (with median-of-three pivot selection)
t - Worst Case
rt - Best Case
Average Case
t - Worst Case
rt - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
n - Worst Case
ap - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
) - Worst Case
n) - Best Case
Average Case

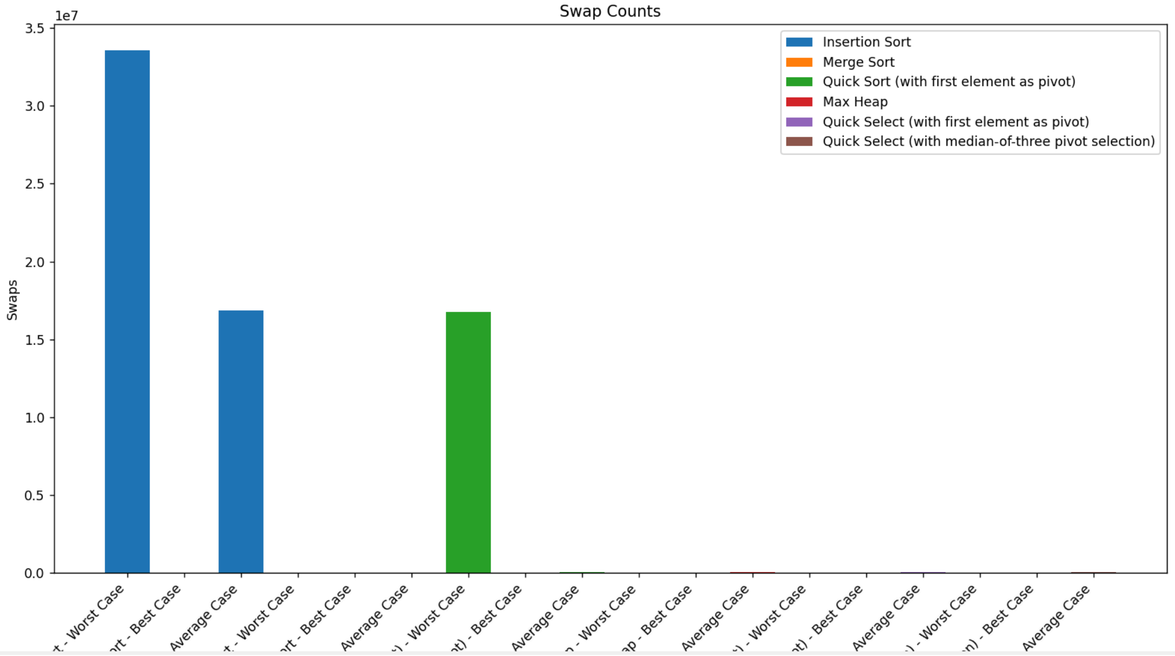
For input 8192:



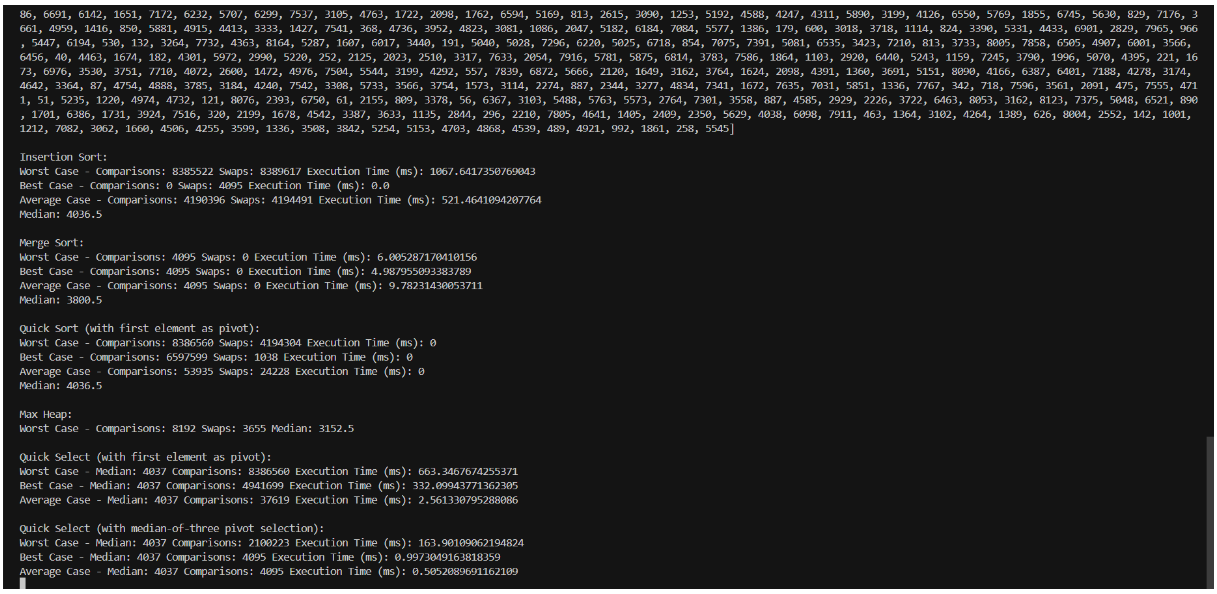
```
871, 2580, 6611, 2236, 2870, 1113, 7678, 4269, 7839, 3967, 7386, 3469, 7673, 1463, 5017, 3245, 1792, 5726, 6151, 7040, 154, 5451, 6326, 3702, 7326, 1660, 6447, 4022, 790, 5370, 5422, 3379, 3516, 4360,
5199, 1310, 709, 3591, 6995, 3560, 5503, 1081, 1760, 7548, 6851, 954, 3151, 2445, 501, 4335, 8165, 4110, 4014, 4884, 7854, 6827, 7601, 7504, 8053, 3092, 3020, 1417, 3044, 2388, 3875, 3479, 8129, 7424,
3178, 1782, 465, 7884, 3777, 610, 5792, 6115, 4550, 7916, 7579, 6115, 1795, 2207, 2060, 5153, 1116, 4930, 2551, 3734, 4991, 893, 7225, 3188, 7454, 769, 6864, 5518, 1095, 3794, 6403, 398, 3741, 1750, 19
95, 6309, 7736, 6450, 6360, 7276, 2819, 123, 2597, 603, 5037, 336, 3123, 653, 4216, 7811, 3704, 6429, 4133, 7696, 930, 6853, 1520, 1517, 4458, 6366, 3826, 2762, 4908, 3855, 1500, 2082, 7417, 7172, 6398
, 2894, 871, 1629, 6519, 2373, 5771, 4144, 1228, 1645, 4682, 3962, 869, 272, 4103, 6076, 6675, 260, 4682, 7130, 5891, 4188, 2484, 5029, 5755, 1556, 3395, 1647, 464, 4660, 3493, 2648, 2630, 1620, 5750,
844, 7079, 5893, 1811, 4141, 8073, 8012, 2934, 5492, 2377, 268, 3752, 4390, 1232, 3995, 2311, 6084, 3208, 1686, 6494, 540, 1832, 4904, 578, 2283, 5453, 5765, 1624, 3586, 3180, 2353, 6210, 6075, 7028, 3
353, 1516, 2913, 4668, 1051, 7414, 5836, 7553, 4528, 8011, 4970, 5648, 5405, 2698, 5913, 7820, 863, 838, 5614, 6836, 5276, 7600, 1426, 539, 6622, 8094, 2994, 6830, 5955, 5415, 357, 4682, 3775, 2388, 26
75, 3043, 7887, 6316, 2512, 4358, 2547, 8160, 5366, 3457, 1118, 4754, 6396, 5925, 3358, 4343, 428, 5652, 7145, 7365, 5868, 6975, 1710, 6805, 7528, 8135, 2828, 3130, 4329, 8019, 3992, 2615, 7058, 5448,
2903, 4920, 595, 190, 1518, 7511]

Insertion Sort:
Worst Case - Comparisons: 33546207 Swaps: 33554398 Execution Time (ms): 4313.04669380188
Best Case - Comparisons: 0 Swaps: 8191 Execution Time (ms): 0.9958744049072266
Average Case - Comparisons: 16831892 Swaps: 16840083 Execution Time (ms): 2468.1193828582764
Median: 4112.0

Merge Sort:
Worst Case - Comparisons: 8191 Swaps: 0 Execution Time (ms): 13.10586929321289
Best Case - Comparisons: 8191 Swaps: 0 Execution Time (ms): 14.555692672729492
Average Case - Comparisons: 8191 Swaps: 0 Execution Time (ms): 24.030208587646484
Median: 5442.0

Quick Sort (with first element as pivot):
Worst Case - Comparisons: 33550336 Swaps: 16777216 Execution Time (ms): 0
Best Case - Comparisons: 21079715 Swaps: 4129 Execution Time (ms): 0
Average Case - Comparisons: 131775 Swaps: 60266 Execution Time (ms): 0
Median: 4112.0

Max Heap:
Worst Case - Comparisons: 16384 Swaps: 7283 Median: 2229.5

Quick Select (with first element as pivot):
Worst Case - Median: 4112 Comparisons: 33550336 Execution Time (ms): 2470.590114593506
Best Case - Median: 4112 Comparisons: 15777790 Execution Time (ms): 1096.1220264434814
Average Case - Median: 4112 Comparisons: 41250 Execution Time (ms): 1.9919872283935547

Quick Select (with median-of-three pivot selection):
Worst Case - Median: 4112 Comparisons: 12288 Execution Time (ms): 1.9941329956054688
Best Case - Median: 4112 Comparisons: 12287 Execution Time (ms): 0.5042552947998047
Average Case - Median: 4112 Comparisons: 12287 Execution Time (ms): 1.0077953338623047
```



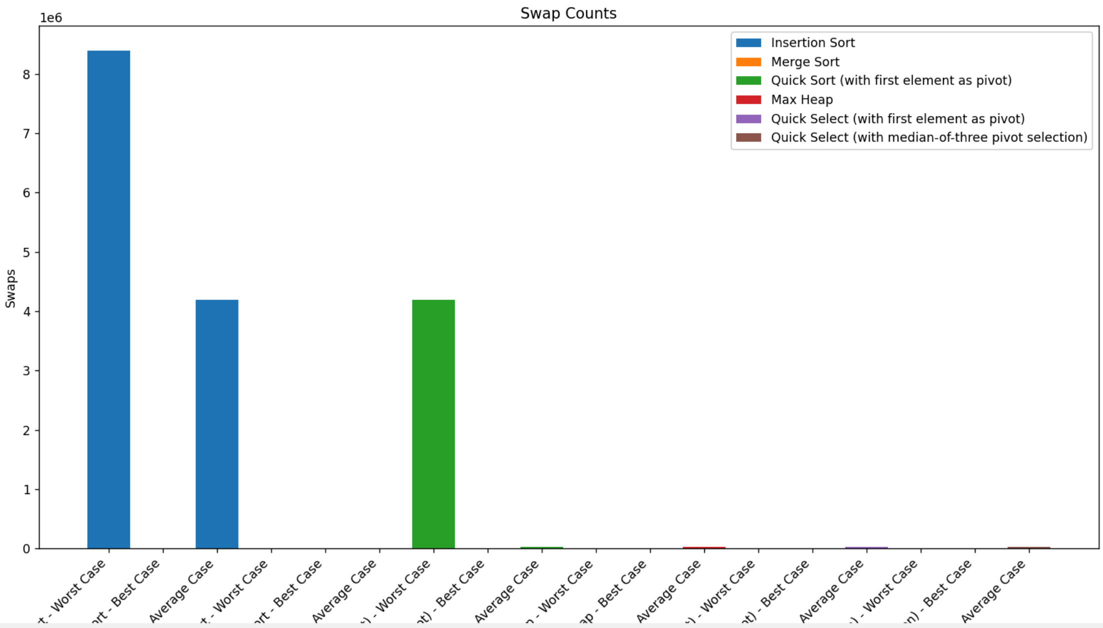

Comparison Counts
1e7
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Comparisons
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
rt - Worst Case
rt - Best Case
Average Case
t - Worst Case
rt - Best Case
Average Case
) - Worst Case
nt) - Best Case
Average Case
p - Worst Case
ap - Best Case
Average Case
) - Worst Case
nt) - Best Case
Average Case
) - Worst Case
n) - Best Case
Average Case

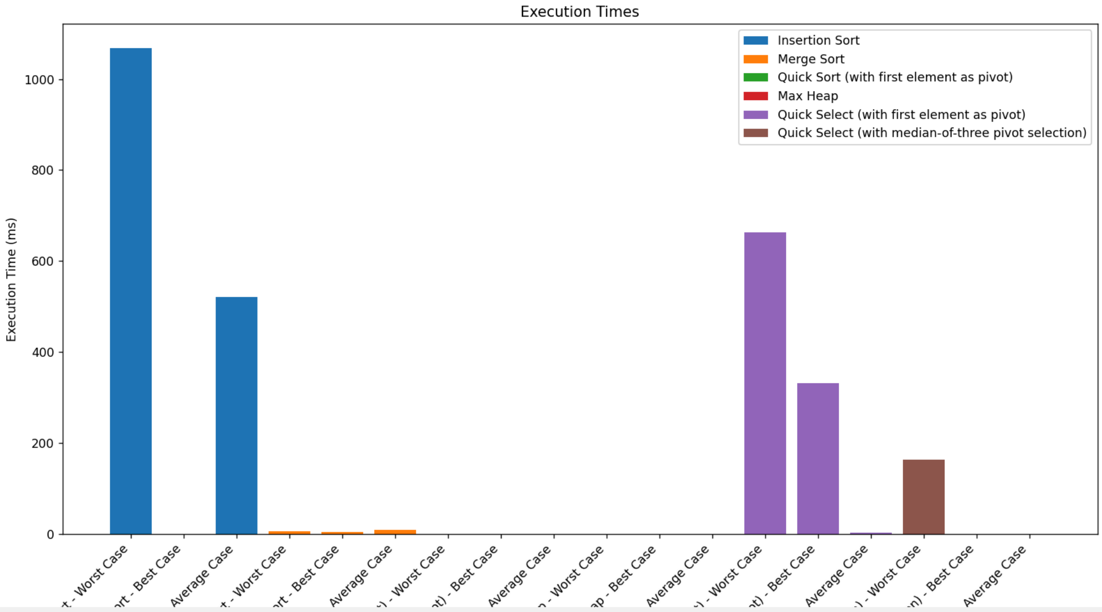

Swap Counts
1e7
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Swaps
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
t - Worst Case
rt - Best Case
Average Case
t - Worst Case
rt - Best Case
Average Case
) - Worst Case
nt) - Best Case
Average Case
p - Worst Case
ap - Best Case
Average Case
) - Worst Case
nt) - Best Case
Average Case
) - Worst Case
n) - Best Case
Average Case

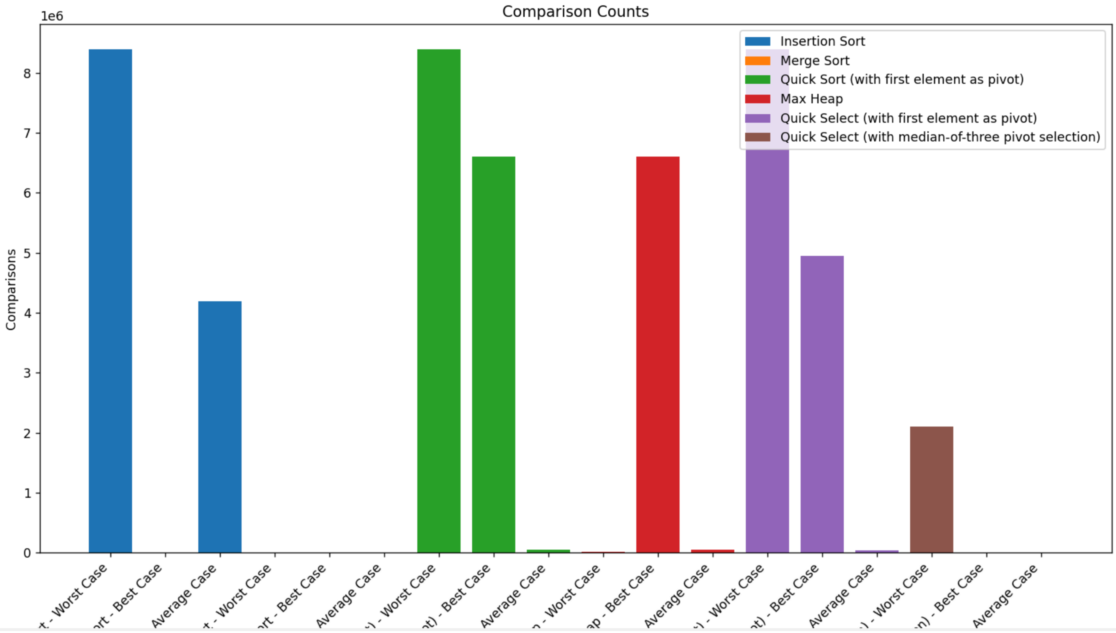
For input as 4096:



```
86, 6691, 6142, 1651, 7172, 6232, 5707, 6299, 7537, 3105, 4763, 1722, 2098, 1762, 6594, 5169, 813, 2615, 3090, 1253, 5192, 4588, 4247, 4311, 5890, 3199, 4126, 6550, 5769, 1855, 6745, 5630, 829, 7176, 3
661, 4959, 1416, 850, 5881, 4915, 4413, 3333, 1427, 7541, 368, 4736, 3952, 4823, 3081, 1086, 2047, 5182, 6184, 7084, 5577, 1386, 179, 600, 3018, 3718, 1114, 824, 3390, 5331, 4433, 6901, 2829, 7965, 966
, 5447, 6194, 530, 132, 3264, 7732, 4363, 8164, 5287, 1607, 6017, 3440, 191, 5040, 5028, 7296, 6220, 5025, 6718, 854, 7075, 7391, 5081, 6535, 3423, 7210, 813, 3733, 8005, 7858, 6505, 4907, 6001, 3566,
6456, 40, 4463, 1674, 182, 4301, 5972, 2990, 5220, 252, 2125, 2023, 2510, 3317, 7633, 2054, 7916, 5781, 5875, 6814, 3783, 7586, 1864, 1103, 2920, 6440, 5243, 1159, 7245, 3790, 1996, 5070, 4395, 221, 16
73, 6976, 3530, 3751, 7710, 4072, 2600, 1472, 4976, 7504, 5544, 3199, 4292, 557, 7839, 6872, 5666, 2120, 1649, 3162, 3764, 1624, 2098, 4391, 1360, 3691, 5151, 8090, 4166, 6387, 6401, 7188, 4278, 3174,
4642, 3364, 87, 4754, 4888, 3785, 3184, 4240, 7542, 3308, 5733, 3566, 3754, 1573, 3114, 2274, 887, 2344, 3277, 4834, 7341, 1672, 7635, 7031, 5851, 1336, 7767, 342, 718, 7596, 3561, 2091, 475, 7555, 471
1, 51, 5235, 1220, 4974, 4732, 121, 8076, 2393, 6750, 61, 2155, 809, 3378, 56, 6367, 3103, 5488, 5763, 5573, 2764, 7301, 3558, 887, 4585, 2929, 2226, 3722, 6463, 8053, 3162, 8123, 7375, 5048, 6521, 890
, 1701, 6386, 1731, 3924, 7516, 320, 2199, 1678, 4542, 3387, 3633, 1135, 2844, 296, 2210, 7805, 4641, 1405, 2409, 2350, 5629, 4038, 6098, 7911, 463, 1364, 3102, 4264, 1389, 626, 8004, 2552, 142, 1001,
1212, 7082, 3062, 1660, 4506, 4255, 3599, 1336, 3508, 3842, 5254, 5153, 4703, 4868, 4539, 489, 4921, 992, 1861, 258, 5545]

Insertion Sort:
Worst Case - Comparisons: 8385522 Swaps: 8389617 Execution Time (ms): 1067.6417350769043
Best Case - Comparisons: 0 Swaps: 4095 Execution Time (ms): 0.0
Average Case - Comparisons: 4190396 Swaps: 4194491 Execution Time (ms): 521.4641094207764
Median: 4036.5

Merge Sort:
Worst Case - Comparisons: 4095 Swaps: 0 Execution Time (ms): 6.005287170410156
Best Case - Comparisons: 4095 Swaps: 0 Execution Time (ms): 4.987955093383789
Average Case - Comparisons: 4095 Swaps: 0 Execution Time (ms): 9.78231430053711
Median: 3800.5

Quick Sort (with first element as pivot):
Worst Case - Comparisons: 8386560 Swaps: 4194304 Execution Time (ms): 0
Best Case - Comparisons: 6597599 Swaps: 1038 Execution Time (ms): 0
Average Case - Comparisons: 53935 Swaps: 24228 Execution Time (ms): 0
Median: 4036.5

Max Heap:
Worst Case - Comparisons: 8192 Swaps: 3655 Median: 3152.5

Quick Select (with first element as pivot):
Worst Case - Median: 4037 Comparisons: 8386560 Execution Time (ms): 663.3467674255371
Best Case - Median: 4037 Comparisons: 4941699 Execution Time (ms): 332.09943771362305
Average Case - Median: 4037 Comparisons: 37619 Execution Time (ms): 2.561330795288086

Quick Select (with median-of-three pivot selection):
Worst Case - Median: 4037 Comparisons: 2100223 Execution Time (ms): 163.90109062194824
Best Case - Median: 4037 Comparisons: 4095 Execution Time (ms): 0.9973049163818359
Average Case - Median: 4037 Comparisons: 4095 Execution Time (ms): 0.5052089691162109
```




Execution Times
Execution Time (ms)
0
200
400
600
800
1000
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
t - Worst Case
ort - Best Case
Average Case
t - Worst Case
ort - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
p - Worst Case
ap - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
) - Worst Case
on) - Best Case
Average Case


Comparison Counts
1e6
Comparisons
0
1
2
3
4
5
6
7
8
Insertion Sort
Merge Sort
Quick Sort (with first element as pivot)
Max Heap
Quick Select (with first element as pivot)
Quick Select (with median-of-three pivot selection)
t - Worst Case
ort - Best Case
Average Case
t - Worst Case
ort - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
p - Worst Case
ap - Best Case
Average Case
) - Worst Case
ot) - Best Case
Average Case
) - Worst Case
on) - Best Case
Average Case

As array sizes expand, the efficiency of sorting algorithms changes. Here’s an analysis of their time complexities and how they fare with different array sizes, specifically 10, 200, 4096, and 8192 elements:

Insertion Sort:

- **Time Complexity**:
- O(n^2)
- **Performance**: Effective for small arrays like 10 elements, but as arrays grow, especially up to 8192 elements, performance drops sharply due to the increased number of swaps and comparisons.

Merge Sort:

- **Time Complexity**:
- O(n log n)
- **Performance**: Shows consistent efficiency for all sizes. Execution time grows moderately with larger arrays. Swap count is zero; comparisons grow logarithmically with array size.

Quick Sort:

- **Time Complexity**:
- O(n log n)
- on average,
- O(n^2)
- in the worst case
- **Performance**: Varies based on pivot choice and array’s initial state. Generally efficient, but in the worst case (like a sorted array), performance for large arrays like 8192 elements can be greatly affected.

Max Heap:

- **Time Complexity**:
- O(n log n)
- **Performance**: Stability across sizes, with a moderate increase in execution time for larger arrays. Tends to have more swaps than other algorithms.

Quick Select (First Pivot):

- **Time Complexity**:
- O(n^2)
- in the worst case,
- O(n)
- on average
- **Performance**: Similar to Quick Sort, it’s efficient on average but can struggle in the worst case with large arrays.

Quick Select (Median-of-Three Pivot):

- **Time Complexity**:
- O(n)
- on average
- **Performance**: Generally outperforms the first pivot method, showing consistent efficiency even for large arrays.

Overall:

- **Insertion Sort** sees a rapid decline in efficiency with larger arrays.
- **Merge Sort** maintains stable performance.
- **Quicksort** efficiency is highly variable.
- **Max Heap** and **Quick Select** with median-of-three pivot maintain relatively steady efficiency.
- **Quick Select** with the first pivot can experience significant performance drops in the worst-case scenario, particularly with larger arrays.