Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

COMP 314

Algorithms and Complexity

Lab Report 1

Comparison of Sorting Algorithms

Submitted By:

Aayush Man Shakya(48)

Submitted To:

Dr. Prakash Poudyal

Department of Computer Science and Engineering

Date of Submission:

Jan 30, 2024

Comparison of Sorting Algorithms

1. Selection Sort

It sorts an array by repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted. This is a brute force algorithm.

Time Complexity: $O(n^2)$

The time complexity of the Selection sort remains constant regardless of the input array's initial order. At each step, the algorithm identifies the minimum element and places it in its correct position. However, the minimum element cannot be determined until the entire array is traversed.

```
def selection_sort(arr):
    for i in range (len(arr)):
        min_index = i
        for j in range(i+1,len(arr)):
            if arr[min_index] > arr[j]:
                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr
```

2. Insertion Sort

It works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

Best case: O(n), If the list is already sorted, where n is the number of elements in the list.

Average case: $O(n^2)$, If the list is randomly ordered

Worst case: $O(n^2)$, If the list is in reverse order

```
def insertion_sort(arr):
    for i in range(len(arr)):
        for j in range(i, 0, -1):
            if arr[j] < arr[j-1]:
                arr[j], arr[j-1] = arr[j-1], arr[j]
    return arr
```

3. Merge Sort

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Time Complexity: O(n log n) for all cases

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    else:
        arr1 = arr[:len(arr)//2]
        arr2 = arr[len(arr)//2:]

        arr1 = merge_sort(arr1)
        arr2 = merge_sort(arr2)

        return merge(arr1, arr2)

def merge(arr1, arr2):
    result = []
    i = j = 0

    # Merge the arrays by comparing the elements
    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            result.append(arr1[i])
            i += 1
        else:
            result.append(arr2[j])
            j += 1

    # If any elements are left in arr1, add them
    result.extend(arr1[i:])

    # If any elements are left in arr2, add them
    result.extend(arr2[j:])
    return result
```

4. Heap Sort

We first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. We use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n).

Time Complexity: O(n log n) for all cases

```
def heap_sort(arr):
    n = len(arr)
    # Step 1: Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)

    # Step 2: Extract elements from the heap one by one
    for i in range(n - 1, 0, -1):
        # Swap the root (largest element) with the last element
        arr[0], arr[i] = arr[i], arr[0]

        max_heapify(arr, i, 0)
    return arr

def max_heapify(arr, n, i):
    largest = i  # Initialize largest as the root
    left = 2 * i + 1  # Left child
    right = 2 * i + 2  # Right child

    # Check if the left child exists and is greater than the root
    if left < n and arr[left] > arr[largest]:
        largest = left
    # Check if the right child exists and is greater than the largest
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If the largest is not the root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        max_heapify(arr, n, largest) #Recursively heapify affected subtree
```

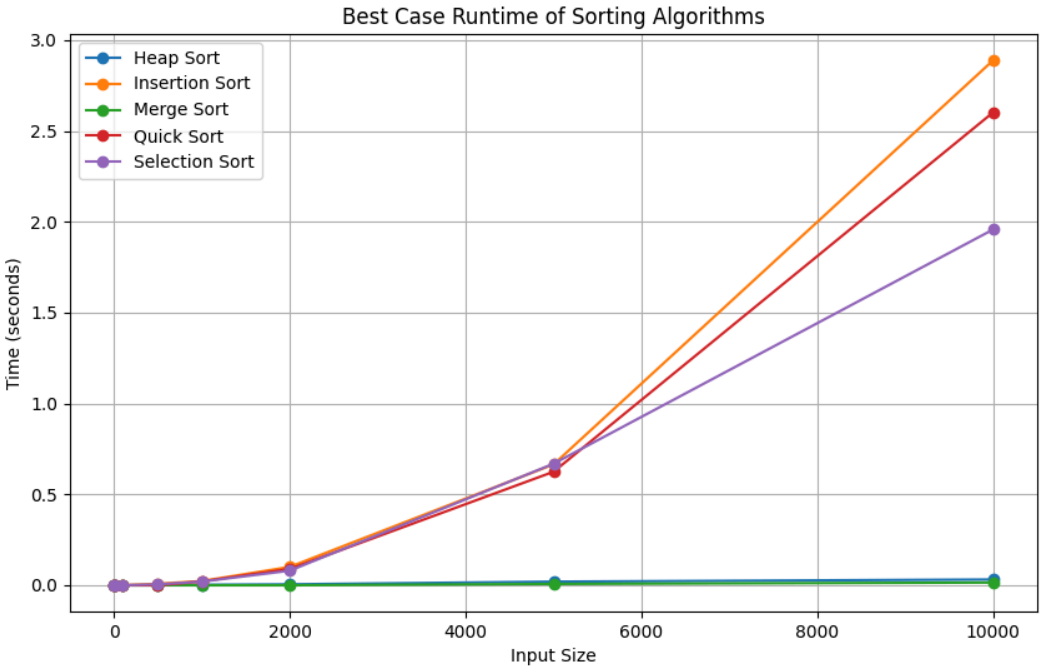
5. Quick Sort

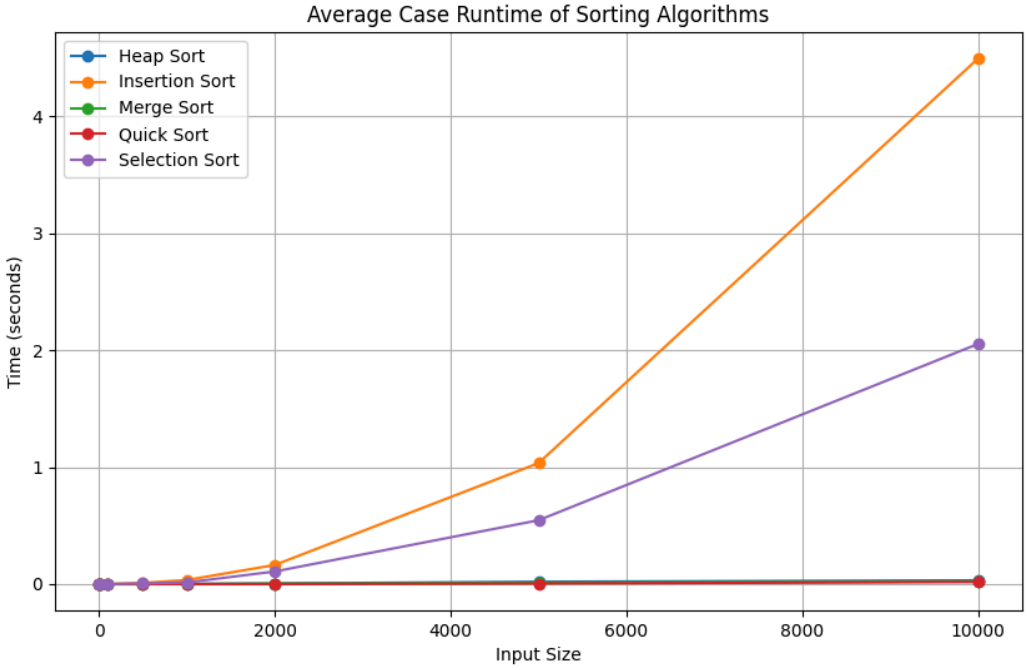
QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

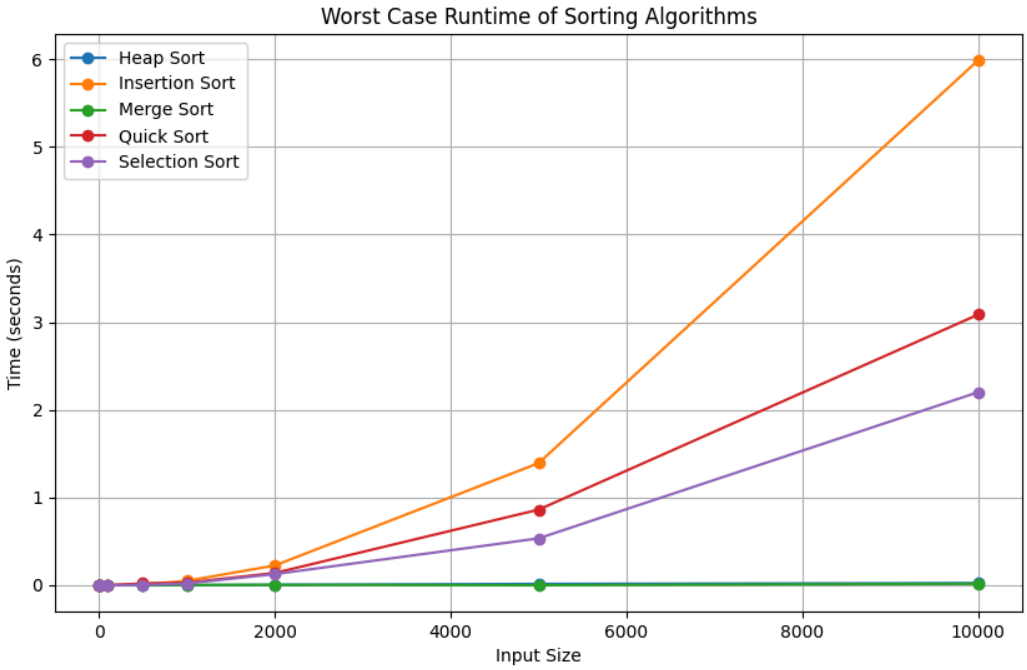
Best case: O(n log n), when the pivot element divides the array into two equal halves
Average case: O(n log n), when pivot divides the array into two parts, but not necessarily equal.
Worst case: O(n^2), when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays)

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[-1]
    smaller = []   # Elements smaller than the pivot
    bigger = []  # Elements bigger than the pivot
    equal = []

    # Partition the array
    for num in arr:
        if num < pivot:
            smaller.append(num)
        elif num > pivot:
            bigger.append(num)
        else:
            equal.append(num)

    # Recursively sort the left (smaller) and right (bigger) partitions
    return quick_sort(smaller) + equal + quick_sort(bigger)
```

Comparison Graphs:







The best, average and worst case time complexities of each algorithm was graphed using input array sizes of 0, 10, 100, 500, 1000, 2000, 5000, 10000 elements.

Conclusion

The time complexity of sorting algorithms was compared and we can see that Insertion Sort, Quick Sort and Merge Sort are slower than Heap and Merge Sort. So, they are not preferred, especially on large input sizes.