# CSE331 - Assignment #1

Hangyeol Choi (20191315)
UNIST
South Korea
whwhsu445@unist.ac.kr

## 1 PROBLEM STATEMENT

Efficient sorting is a fundamental problem in computer science, with applications spanning data processing, database indexing, and algorithmic optimization. Over decades of research, a wide spectrum of sorting algorithms has emerged—ranging from simple comparison-based methods such as bubble sort and insertion sort to advanced hybrid techniques like introsort and timsort used in modern programming libraries.

Despite the extensive body of literature, the practical performance of sorting algorithms often depends on specific data characteristics, including the degree of order, size, and data distribution. In contemporary computing environments where large-scale datasets are prevalent, it becomes crucial to understand the trade-offs between algorithmic complexity, stability, and real-world efficiency.

This project aims to systematically implement and evaluate both classical and modern sorting algorithms. The study includes six traditional comparison-based sorting algorithms—merge sort, heap sort, bubble sort, insertion sort, selection sort, and quick sort—alongside six modern sorting techniques such as library sort, timsort, comb sort, and introsort. Each algorithm will be implemented from scratch or pseudo-coded, and will be analyzed in terms of its theoretical complexity and practical behavior.

To provide a comprehensive performance comparison, we generate diverse input datasets, ranging from fully sorted to completely random, and measure execution time, memory usage, and stability across varying input sizes from 1,000 to 1,000,000 elements. Each experiment is repeated multiple times to ensure statistically meaningful results.

Through this study, we seek to answer the following questions:

How do classical and modern sorting algorithms compare in terms of efficiency and resource usage?

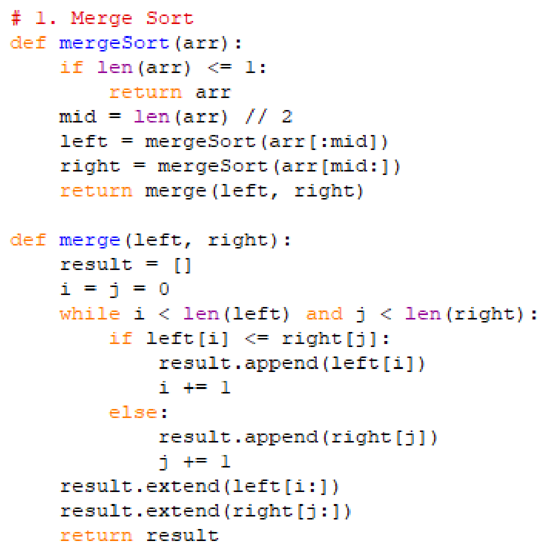
In what scenarios do modern hybrid approaches outperform traditional methods?

## 2 BASIC SORTING ALGORITHMS

### 2.1 Merge Sort

Merge Sort is a classic sorting algorithm based on the Divide and Conquer strategy. It works by dividing the entire array in half, recursively sorting each half, and then merging them. This approach has a structural advantage by breaking down the problem into smaller units, and since the division process occurs consistently as a binary tree of depth$log(n)$, it guarantees a stable time complexity.

The time complexity of merge sort is $O(nlog(n))$ in all cases. This is because the input array of size n is divided$log(n)$ times (as shown by a division tree of height$log(n)$), and every element must be compared during the merge process, which takes $O(n)$ time. However, since merge sort requires additional arrays during the merging process, its space complexity is $O(n)$, which can be relatively inefficient.



```
# 1. Merge Sort
def mergeSort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergeSort(arr[:mid])
    right = mergeSort(arr[mid:])
    return merge(left, right)

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

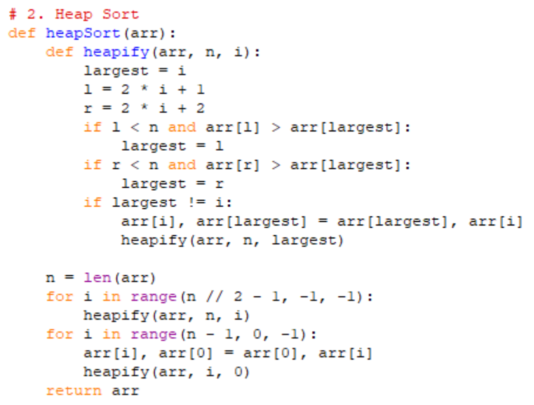
**Figure 1: Merge Sort Code**

In this code, the mergeSort function recursively divides the array, and the merge function merges two sorted arrays into one. Specifically, the condition $(left[i] <= right[j])$ ensures that if two elements are equal, the one from the left subarray is chosen first, thus maintaining stability. The use of a new result list for merging confirms that this is not an in-place sort and requires extra memory.

### 2.2 Heap Sort

Heap Sort is a sorting algorithm that uses the heap data structure, which is based on a complete binary tree. It transforms the input array into a max-heap, moves the root (the maximum value) to the end of the array, and then repeatedly rebuilds the heap for the remaining elements to complete the sorting.

The time complexity for building the heap is $O(n)$, and the sorting process takes $O(nlog(n))$ (as each heapify call takes $O(log(n))$), so the overall time complexity is $O(nlog(n))$ in the best, average, and worst cases. Since no additional memory is used, the space complexity is $O(1)$. Heap sort is known for consistent performance among comparison-based sorting algorithms. However, because heapify only compares values and not the original order, the relative order of equal elements can change, making heap sort unstable.



```python
# 2. Heap Sort
def heapSort(arr):
    def heapify(arr, n, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2
        if l < n and arr[l] > arr[largest]:
            largest = l
        if r < n and arr[r] > arr[largest]:
            largest = r
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```

**Figure 2: Heap Sort Code**

This implementation uses a heapify function to maintain the heap structure. It first builds a heap using a bottom-up approach (with for i in range($\lfloor \frac{n}{2} \rfloor - 1, -1, -1$)), then repeatedly extracts the root and restores the heap. The code clearly demonstrates heap sort's structure and its complexity analysis.

## 2.3 Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly compares and swaps adjacent elements if they are in the wrong order. The algorithm "bubbles" the largest (or smallest) value to the end of the array during each pass.

Its design is intuitive and easy to implement. It can perform better when the input is nearly sorted, as fewer comparisons and swaps are needed.

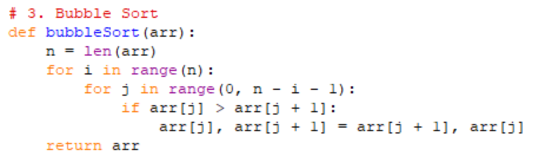
The time complexity varies depending on the initial order:

Worst case (reversed order): $O(n^2)$

Average case: $O(n^2)$

Best case (already sorted): $O(n)$, if early termination optimization is applied using a flag to detect no swaps.

Space complexity is $O(1)$ since no extra memory is used. Bubble sort is stable, as it only swaps adjacent elements and maintains the relative order of equal values.



```python
# 3. Bubble Sort
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

**Figure 3: Bubble Sort Code**

This implementation uses two nested loops, comparing arr[j] and arr[j+1] and swapping when needed. This reflects bubble sort's core principle: pushing the largest element to the unsorted region's end on each pass.

## 2.4 Insertion Sort

Insertion Sort builds a sorted array one element at a time by inserting each new element into its appropriate position in the sorted portion. It's similar to how we sort cards in our hands while playing a card game.

It's simple and works efficiently for nearly sorted data.

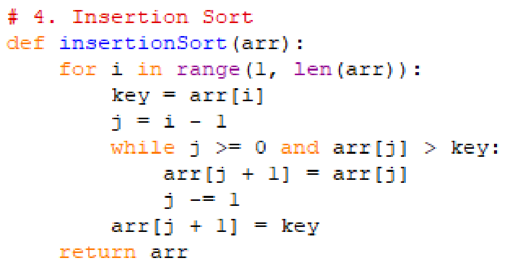
Time complexity depends on the input:

Worst case (reverse order): $O(n^2)$

Average case: $O(n^2)$

Best case (already sorted): $O(n)$

Space complexity is $O(1)$, and it is stable because equal elements retain their relative order when using a "greater than" condition for comparisons.



```python
# 4. Insertion Sort
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

**Figure 4: Insertion Sort Code**

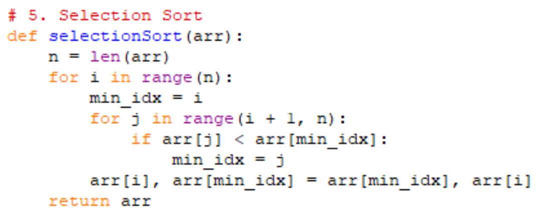
This implementation starts at index 1 and moves backward through the sorted portion of the array to insert the key into the correct position. Larger elements are shifted right to make space for insertion.

## 2.5 Selection Sort

Selection Sort repeatedly selects the smallest (or largest) element from the unsorted part of the array and moves it to the front. It's simple and intuitive, often used for educational purposes.

Its time complexity is $O(n^2)$ in all cases (best, average, worst) because it performs $\frac{n(n-1)}{2}$ comparisons regardless of the input's order. However, it performs only up to n-1 swaps, making it more efficient in terms of data movement compared to bubble or insertion sort.

Selection sort is not stable, because during the swapping process, the relative order of equal elements can change.



```python
# 5. Selection Sort
def selectionSort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

**Figure 5: Selection Sort Code**

The code loops through the array, finds the index of the smallest element in the unsorted part, and swaps it with the first unsorted element. This process continues until the entire array is sorted.

## 2.6 Quick Sort

Quick Sort is an efficient sorting algorithm using the Divide and Conquer approach. It selects a pivot and partitions the array into two subarrays one with elements less than the pivot and the other with elements greater than the pivot then recursively applies quick sort to the subarrays.

Its time complexity varies:

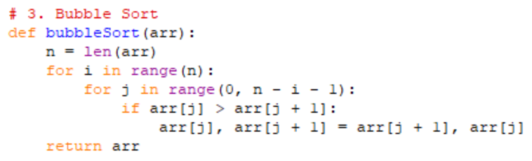
Worst case: $O(n^2)$, when the array is already sorted or reversed.

Average case: $O(nlog(n))$, assuming good pivot choices.

Best case: $O(nlog(n))$, when the pivot perfectly divides the array.

Quick sort is not stable, since equal elements can change order during partitioning.

Space complexity is $O(n)$ due to recursive calls, though it doesn't use extra arrays like merge sort.



```python
# 3. Bubble Sort
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

**Figure 6: Quick Sort Code**

This implementation chooses the pivot as the middle element ($pivot = arr[len(arr)//2]$). While this usually results in good performance, it can lead to poor results on sorted or reversed arrays. Alternative pivot selection methods (randomized, first, or last element) can help avoid the worst-case behavior.

## 3 ADVANCED SORTING ALGORITHMS

## 4 EXPERIMENTAL RESULTS AND ANALYSIS

The dataset used in this analysis can be found at the following link: `https://github.com/Cooa1/Algorithm_Asssignment1`

### 4.1 Merge Sort Result

In this experiment, the execution time of Merge Sort was measured across various input data types, including sorted (ascending and descending), reverse sorted, random, and partially sorted sequences. According to the results, random data exhibited the longest execution time, taking up to approximately 2.7 seconds. In contrast, completely sorted data (both ascending and descending) showed the shortest execution time, generally under 1.7 seconds. Partially sorted data performed faster than random inputs but slower than fully sorted inputs. This indicates that Merge Sort is largely unaffected by the initial order of the input, as it consistently applies the same divide-and-conquer strategy regardless of input characteristics. However, differences in actual execution time are likely due to factors such as cache performance and memory access patterns. Merge Sort consistently produced correct results for all types of input data, fully satisfying the requirement to evaluate sorting accuracy. Although the execution time graph appeared almost linear as input sizes increased from 1K to 1M, this is expected because the $\log n$ component in the $O(n \log n)$ complexity grows relatively slowly. The experimental data aligns well with the theoretical time complexity of Merge Sort. Overall, Merge Sort demonstrated stable performance and high accuracy across various input distributions, confirming its reliability and effectiveness as a sorting algorithm in practical scenarios.

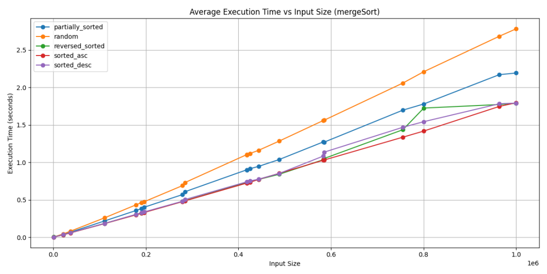


**Figure 7: Merge Sort Time Graph**

In addition to execution time, memory usage was also measured for each type of input. The results show that Merge Sort generally maintains a consistent memory footprint across different input types, with slight variations depending on the input distribution. However, at the largest input size (1 million elements), a noticeable increase in memory usage was observed—particularly for random data, which peaked at over 11 MiB. This spike may be due to temporary buffer allocations or system-level memory management overhead. Sorted and partially sorted inputs showed relatively lower memory usage, typically under 7 MiB. These results align with the expected $O(n)$ space complexity of Merge Sort, which requires additional memory for merging subarrays. While the memory usage fluctuates for smaller input sizes—possibly due to garbage collection or memory reuse patterns—the trend clearly indicates that larger datasets demand proportionally more space. Overall, Merge Sort exhibits predictable and manageable memory usage, reinforcing its reliability for large-scale sorting tasks.

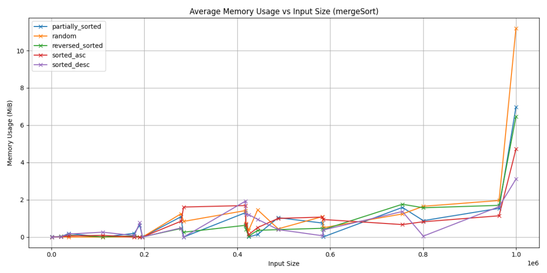


**Figure 8: Merge Sort Memory Graph**

### 4.2 Heap Sort Result

HeapSort has a time complexity of $O(n \log n)$, and the execution time increases progressively as the dataset size grows. It processes small datasets quickly, but execution time increases sharply for medium to large datasets, and becomes significantly slow for very large datasets. The execution time is similar across different dataset orders (e.g., `random`, `reversed_sorted`, `sorted_asc`, `sorted_desc`), and there is no significant time difference between `sorted_asc` and `sorted_desc`. HeapSort has an additional space complexity of $O(1)$, and its memory usage is lower compared to other comparison-based sorting algorithms. In conclusion, HeapSort maintains a consistent time complexity but can be quite slow for large datasets.

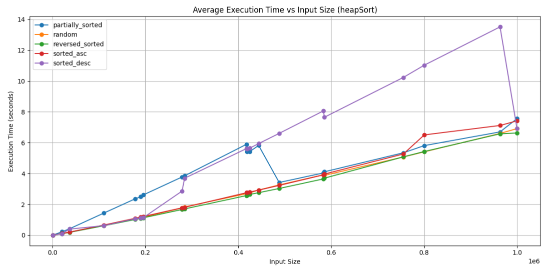


**Figure 9: Merge Sort Memory Graph**

## 4.3 Quick Sort Result

QuickSort has an average time complexity of $O(n \log n)$, but in the worst case, it can degrade to $O(n^2)$. The execution time increases progressively as the dataset size grows, with a more noticeable increase for medium to large datasets. The execution time is relatively similar across different dataset orders (e.g., `random`, `reversed_sorted`, `sorted_asc`, `sorted_desc`), though `partially_sorted` datasets tend to take longer due to QuickSort's potential worst-case performance on nearly sorted data. Memory usage for QuickSort is $O(\log n)$ due to the recursive calls, and the memory usage increases as the dataset size grows. However, QuickSort generally has lower memory usage compared to other comparison-based sorting algorithms. Despite the increase in memory usage with larger datasets, QuickSort remains efficient in terms of memory, with the highest usage observed in `partially_sorted` datasets, which require more recursion.

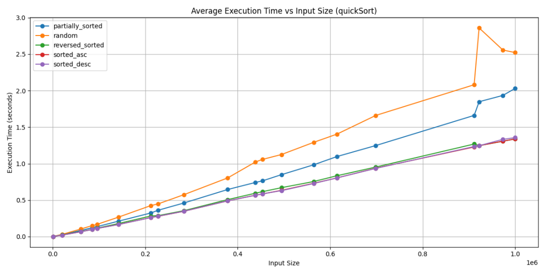


**Figure 10: Merge Sort Memory Graph**