# Analysis of Heapsort and Empirical Comparison with Merge and Quick Sort

## Time Complexity

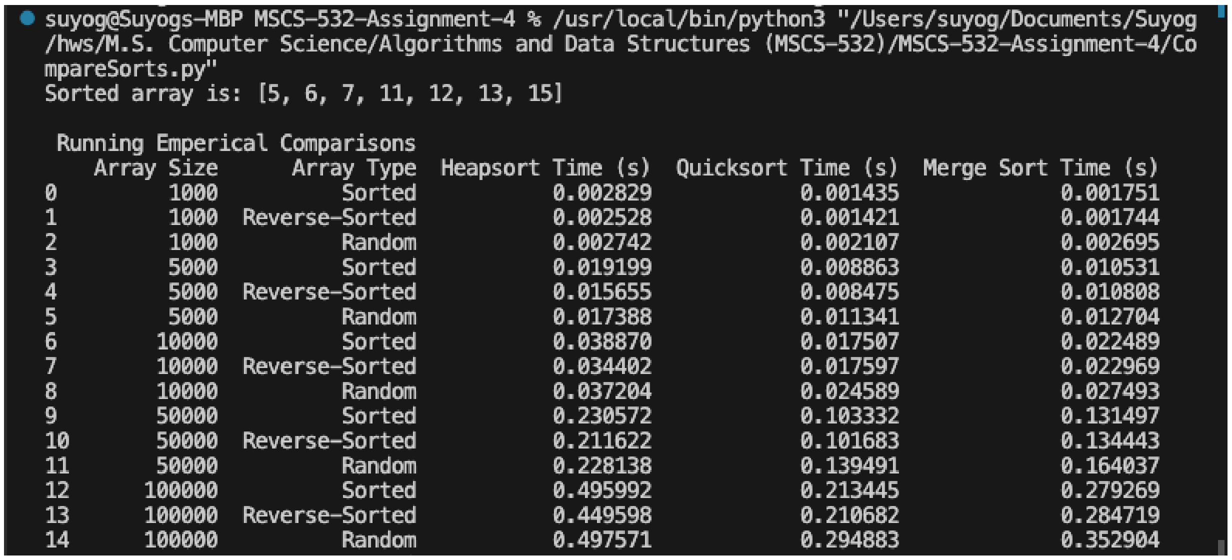
- **Building the Heap**: The process of building the max-heap is O(n). This is because heapifying each element takes O(log n) time, and we are doing it for n/2 nodes, but the time spent decreases as we move up the tree.
- **Heapify Process**: Extracting the maximum element requires O(log n) time for each of the n elements. Thus, extracting elements from the heap takes O(n log n) in total.
- **Best, Average, and Worst Cases**: The Heapsort algorithm always has a time complexity of O(n log n) because the heap structure requires log n comparisons in every step to maintain the heap property. The heap construction and extraction steps take O(n log n) time regardless of the input arrangement.

## Space Complexity

- Heapsort is in-place and requires O(1) auxiliary space apart from the input array. The space complexity is O(n) due to the input array itself, but the algorithm doesn't use additional space for recursion like Merge Sort or Quicksort.
- Overhead: Heapsort has no significant recursion or extra storage overhead, as all operations are done within the input array.

## Empirical Comparison with Quicksort and Merge Sort

Let’s compare the performance of Heapsort, Quicksort, and Merge Sort empirically by testing them on different input distributions (sorted, reverse-sorted, random) and input sizes. The input sizes for this comparison are: 1000, 5000, 10000, 50000, 100000. Below is the screenshot of  the result.



```
suyog@Suyogs-MBP MSCS-532-Assignment-4 % /usr/local/bin/python3 "/Users/suyog/Documents/Suyog
/hws/M.S. Computer Science/Algorithms and Data Structures (MSCS-532)/MSCS-532-Assignment-4/Co
mpareSorts.py"
Sorted array is: [5, 6, 7, 11, 12, 13, 15]

 Running Emperical Comparisons
    Array Size     Array Type  Heapsort Time (s)  Quicksort Time (s)  Merge Sort Time (s)
0         1000         Sorted           0.002829            0.001435             0.001751
1         1000  Reverse-Sorted          0.002528            0.001421             0.001744
2         1000         Random           0.002742            0.002107             0.002695
3         5000         Sorted           0.019199            0.008863             0.010531
4         5000  Reverse-Sorted          0.015655            0.008475             0.010808
5         5000         Random           0.017388            0.011341             0.012704
6        10000         Sorted           0.038870            0.017507             0.022489
7        10000  Reverse-Sorted          0.034402            0.017597             0.022969
8        10000         Random           0.037204            0.024589             0.027493
9        50000         Sorted           0.230572            0.103332             0.131497
10       50000  Reverse-Sorted          0.211622            0.101683             0.134443
11       50000         Random           0.228138            0.139491             0.164037
12      100000         Sorted           0.495992            0.213445             0.279269
13      100000  Reverse-Sorted          0.449598            0.210682             0.284719
14      100000         Random           0.497571            0.294883             0.352904
```

**Discussions of Result**

- Heapsort shows consistent times across all types of input (sorted, reverse-sorted, and random), and the performance does not degrade noticeably based on the input type. This matches the theoretical expectation of O(n log n). It is robust with consistent performance across all input types but is generally slower than Quicksort for random data due to the overhead of maintaining the heap property.
- Quicksort is typically faster than Heapsort and Merge Sort for random inputs due to its low overhead and better cache utilization. However, it may show degraded performance for sorted or reverse-sorted arrays unless randomized pivot selection is used. This aligns with the theoretical worst-case behavior of $O(n^2)$. It tends to be the fastest in practice for most random input cases, though it can suffer greatly from poor pivot selection on sorted or reverse-sorted arrays.
- Merge Sort performs consistently regardless of the input type. It generally takes slightly longer than Quicksort for random data but does not suffer from worst-case performance

issues like Quicksort. This matches its predictable O(n log n) time complexity but has higher memory overhead compared to the in-place algorithms.

**Heapsort vs. Quicksort vs. Merge Sort**

- Heapsort is suitable when:
    - Memory usage is a critical factor, as it operates in-place.
    - Consistent performance is required across all input types.
- Quicksort is suitable when:
    - Random inputs are the norm, and an optimized or randomized version is used to avoid worst-case $O(n^2)$ behavior.
    - Cache performance is a priority, as Quicksort often benefits from better cache locality.
- Merge Sort is suitable for:
    - Scenarios where a stable sort is required (e.g., when equal elements should maintain their relative order).
    - Linked lists, where Merge Sort can perform well without the additional memory overhead for arrays.