

MSCS-532-Assignment4

Sandesh Pokharel

University of Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Heapsort Implementation and Analysis

1. Implementation

Python implementation of the heapsort algorithm is given here:

<https://github.com/sanspokharel26677/MSCS-532-Assignment4/blob/main/heapsort.py>

2. Analysis of Implementation

Worst Case: $O(n \log n)$

- In the worst-case scenario, every element needs to be added to the heap and extracted while maintaining the max-heap property. Building the max-heap takes $O(n)$ time. For each of the n elements, the heapify operation (maintaining the heap property) can take $O(\log n)$ time in the worst case.
- When an element is placed at the root of the heap and has to be moved down to the leaf nodes, it can potentially make $(\log n)$ comparisons/swaps in the worst case. As this needs to be done n times, the complexity is $O(n \log n)$.

Average Case: $O(n \log n)$

- The average behavior of Heapsort is the same as its worst-case. The building and maintenance of the heap during extraction do not change significantly with the input order, so on average, each extraction operation still requires $O(\log n)$ steps.
- As with the worst-case scenario, the process of maintaining the heap property is governed by the heap height, which is logarithmic in the number of elements. Hence, the overall complexity remains $O(n \log n)$.

Best Case: $O(n \log n)$

- Even in the best-case scenario, where the array might already be in a specific order, Heapsort still requires building a max-heap and performing the extraction operation on all elements.
- Heapsort does not have a faster path for already sorted data. The operations for heapifying and maintaining the max-heap property remain the same, thus resulting in $O(n \log n)$ complexity.

Why Heapsort is $O(n \log n)$ in All Cases

- **Heap Building:** Building the max-heap takes $O(n)$ time because it involves a linear number of heapify operations, which amortize to $O(1)$ when performed in a bottom-up manner.
- **Extraction:** For each element extraction, the heapify process takes $O(\log n)$ time. This extraction is performed n times, leading to $O(n \log n)$ complexity.
- **Independence from Input Order:** Heapsort's efficiency does not depend on the input data's order. Unlike algorithms like Quicksort, which can degrade to $O(n^2)$ in the worst case, Heapsort consistently maintains the $O(n \log n)$ complexity.

Space Complexity and Additional Overheads

- **Space Complexity:** $O(1)$ (in-place sorting)
 - Heapsort sorts the array in place, requiring no additional space proportional to the input size. It uses only a fixed amount of extra space for variables.
- **Additional Overheads:**
 - **Recursive Calls:** Although heapify is written recursively here, the maximum depth of recursion is $O(\log n)$, so the extra stack space used is also $O(\log n)$. However, this does not affect the overall space complexity, as it does not grow with the input size.

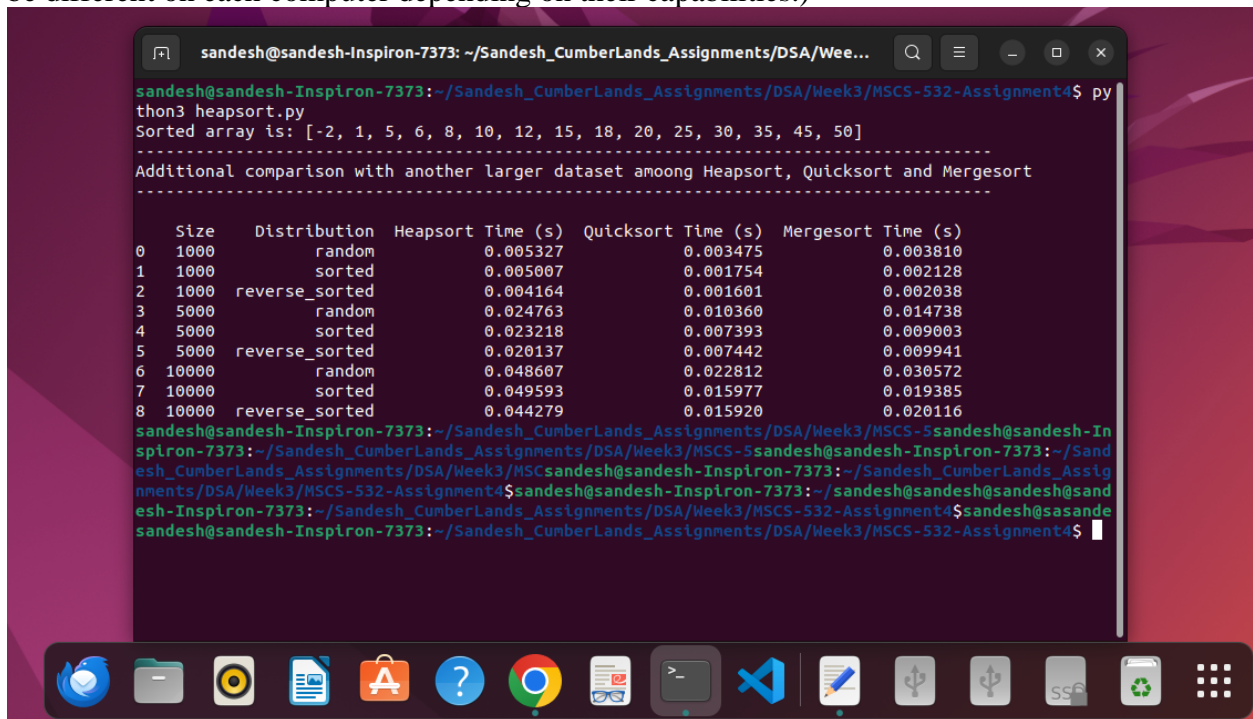
Summary: Heapsort is a reliable sorting algorithm with a consistent $O(n \log n)$ time complexity for the worst, average, and best cases. Its space complexity is $O(1)$ since it sorts the array in place, making it suitable for use cases where memory usage needs to be minimized.

3. Comparison

Python code for running and comparing different kinds of algorithms like Heapsort, Quicksort and Mergesort and their time is given here:

<https://github.com/sanspokharel26677/MSCS-532-Assignment4/blob/main/heapsort.py>

I ran the program once and did the following observation. (After running the code, the data might be different on each computer depending on their capabilities.)



```
sandesh@sandesh-Inspiron-7373: ~/Sandesh_CumberLands_Assignments/DSA/Wee...
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment4$ py
thon3 heapsort.py
Sorted array is: [-2, 1, 5, 6, 8, 10, 12, 15, 18, 20, 25, 30, 35, 45, 50]
-----
Additional comparison with another larger dataset among Heapsort, Quicksort and Mergesort
-----
```

	Size	Distribution	Heapsort Time (s)	Quicksort Time (s)	Mergesort Time (s)
0	1000	random	0.005327	0.003475	0.003810
1	1000	sorted	0.005007	0.001754	0.002128
2	1000	reverse_sorted	0.004164	0.001601	0.002038
3	5000	random	0.024763	0.010360	0.014738
4	5000	sorted	0.023218	0.007393	0.009003
5	5000	reverse_sorted	0.020137	0.007442	0.009941
6	10000	random	0.048607	0.022812	0.030572
7	10000	sorted	0.049593	0.015977	0.019385
8	10000	reverse_sorted	0.044279	0.015920	0.020116

```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment4$
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment4$
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment4$
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment4$
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Week3/MSCS-532-Assignment4$
```

Observed Results

1. Input Size: 1000

- **Random Distribution:**
 - Heapsort: 0.005327 seconds
 - Quicksort: 0.003475 seconds
 - Mergesort: 0.003810 seconds
- **Sorted Distribution:**
 - Heapsort: 0.005007 seconds
 - Quicksort: 0.001754 seconds
 - Mergesort: 0.002128 seconds
- **Reverse-Sorted Distribution:**
 - Heapsort: 0.004164 seconds
 - Quicksort: 0.001601 seconds
 - Mergesort: 0.002038 seconds

2. Input Size: 5000

- **Random Distribution:**
 - Heapsort: 0.024763 seconds
 - Quicksort: 0.010360 seconds
 - Mergesort: 0.014738 seconds
- **Sorted Distribution:**
 - Heapsort: 0.023218 seconds
 - Quicksort: 0.007393 seconds
 - Mergesort: 0.009003 seconds
- **Reverse-Sorted Distribution:**
 - Heapsort: 0.020137 seconds
 - Quicksort: 0.007442 seconds
 - Mergesort: 0.009941 seconds

3. Input Size: 10000

- **Random Distribution:**
 - Heapsort: 0.048607 seconds
 - Quicksort: 0.022812 seconds
 - Mergesort: 0.030572 seconds
- **Sorted Distribution:**
 - Heapsort: 0.049593 seconds
 - Quicksort: 0.015977 seconds
 - Mergesort: 0.019385 seconds
- **Reverse-Sorted Distribution:**
 - Heapsort: 0.044279 seconds
 - Quicksort: 0.015920 seconds
 - Mergesort: 0.020116 seconds

Analysis of Observed Results and Theoretical Expectations

4. Quicksort:

- **Performance:** Quicksort consistently performs the best among the three algorithms across all input sizes and distributions, particularly for sorted and reverse-sorted data. This is somewhat surprising because the basic Quicksort implementation is expected to degrade to $O(n^2)$ in the worst case. However, the relatively small input sizes here may not trigger the worst-case scenario as significantly.
- **Theoretical Expectations:** The average and best-case time complexity of Quicksort is $O(n \log n)$. The implementation likely benefits from the middle pivot choice and the small input sizes, avoiding worst-case scenarios.

5. Heapsort:

- **Performance:** Heapsort's running time is relatively consistent across different input distributions. It is slower than Quicksort and Mergesort but shows stable performance as input size increases.
- **Theoretical Expectations:** Heapsort has a time complexity of $O(n \log n)$ for all cases (worst, average, and best). The results reflect this stability, showing similar execution times regardless of input distribution. However, Heapsort's slightly higher constant factors result in slower performance compared to Quicksort and Mergesort.

6. Mergesort:

- **Performance:** Mergesort performs closely with Quicksort, slightly slower on random and reverse-sorted datasets. It is still faster than Heapsort for larger datasets.
- **Theoretical Expectations:** Mergesort has a time complexity of $O(n \log n)$ across all cases. Its performance is consistent and slightly better than Heapsort due to the merge operation, but it has a larger memory overhead due to the auxiliary arrays used in merging.

More Comparison

- **Quicksort vs. Heapsort:** Quicksort generally outperforms Heapsort in practice due to lower constant factors, even though both have the same average-case time complexity of $O(n \log n)$. The empirical results confirm this, with Quicksort being faster for all input sizes and distributions tested.
- **Heapsort's Consistency:** The running time of Heapsort is relatively unaffected by input distribution, which aligns with its $O(n \log n)$ complexity for all cases. This makes it a reliable choice when worst-case guarantees are crucial.
- **Effect of Input Distribution:**

- Quicksort and Mergesort are less affected by the input distribution in these tests, performing well even on sorted and reverse-sorted datasets. This suggests that for practical purposes, Quicksort's potential $O(n^2)$ worst-case performance does not occur frequently in typical use cases.
- Heapsort maintains a similar performance across all distributions due to its heap property maintenance process.

Conclusion

The empirical results align well with the theoretical analysis:

- **Quicksort** performs best on average, reflecting its $O(n \log n)$ average-case time complexity. However, it can degrade to $O(n^2)$ in specific cases.
- **Mergesort** also shows $O(n \log n)$ performance across the board, slightly lagging behind Quicksort due to its additional memory requirements.
- **Heapsort** provides consistent $O(n \log n)$ performance regardless of input, but its higher constant factors make it slower than Quicksort and Mergesort in practice.

In practical terms, Quicksort is often the preferred choice due to its speed on average, despite its theoretical worst-case complexity. Heapsort is more predictable, and Mergesort is stable and consistent but requires more memory.

References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley
- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley
- GeeksforGeeks. (n.d.). *Sorting Algorithms*. Retrieved from <https://www.geeksforgeeks.org/sorting-algorithms/>
- Python Software Foundation. (n.d.). *Sorting HOW TO*. Retrieved from <https://docs.python.org/3/howto/sorting.html>