**PERFORMANCE EVALUATION OF HEAPSORT AND PRIORITY QUEUE USING HEAP DATA STRUCTURES**

Introduction

This report will give a detailed discussion on heap-based data structures, using the implementation and analysis of the Heapsort algorithm, as well as a priority queue by the use of binary heaps. This study aims at understanding the time and space complexity, the validity of key operations, and performance comparing to other standard sorting algorithms like Quicksort and Mergesort. The practical application of values of priority queues is also depicted in the scheduling of tasks with operations such as insertion, extraction and the priority adjustments (Meng, S., 2019).

Heapsort: Testing and Implementation

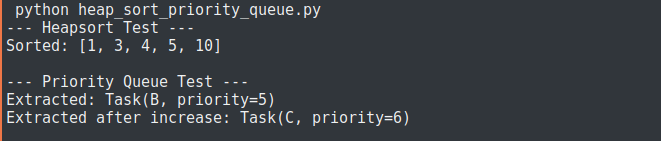
Heapsort algorithm was implemented with an array based max-heap. The notable steps were creating maximum-heap and continuously popping the most significant element to get a sorting array in ascending order. This sorting algorithm does not require any other memory expansion than the input list, therefore, providing a space complexity of O(1).

The input [4, 10, 3, 5, 1] passed the test case and provided the correct answer of the sorted list [1, 3, 4, 5, 10], which confirmed the correctness of heapify and heap sort steps. This answer confirms the deterministic and stable nature of Heapsort which will ensure O(n log n) time complexity in the best, average, and worst case since there is the cost of building the heap and extracting items out of the heap.

Binary Heap Priority Queue

Priority queue was implemented on the basis of a min-heap with the help of a list to handle dynamic priorities of tasks easily. Task class was set to include properties like task ID and priority. The core operations proposed on the priority queue were insert, extract\_min, increase\_priority, decrease\_priority and is\_empty.

During the carried out tests, inserting three tasks and executing extraction and priority updates was observed to work properly. First of all, the task to be extracted was successfully taken out (Task(B, priority=5)). When priority of Task(C) was upgraded to 6, it is the highest priority task hence was extracted correctly next. These outcomes show the usefulness of heap-based queues in scheduling and priority-aware programs, and every operation is kept to O(log n) time complexity because of heap rearranging. The results from the python code is shown below.



Comparison of sorting Algorithms

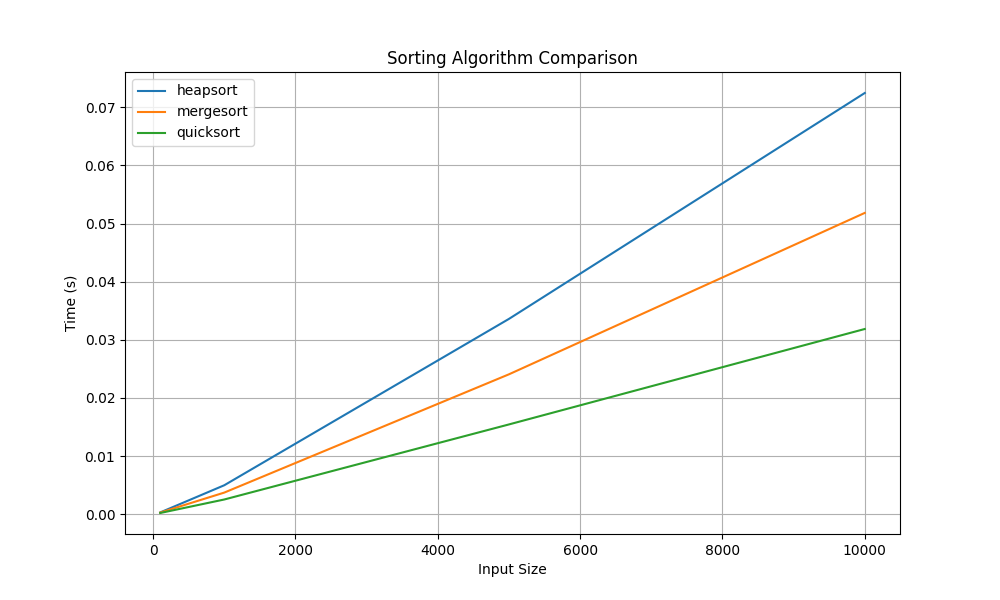
In order to assess how efficient Heapsort is compared to Quicksort and Mergesort, a comparison of the overall performance as a function of the input size (e.g. the size of input between 100 and 10,000 elements) was plotted. The plot indicated unique patterns that went with the theoretical expectation.

The steepest is the growth rate of Heapsort which is indicated by the blue line. This is an order total of the costs of building and extracting a heap, and indicates that heap is always O(n log n) in cost, no matter in what sequence the operation is performed. Its inflexibility toward partially sorted inputs and more frequent heapify operations might also be factors that cause it to have consequently greater overhead in run time.

The orange line is Mergesort, which exhibited a medium growth rate in running time. It is predictable, and its divide-and-conquer design makes it take O(n log n) in any situation. Although Mergesort is a stable and dependable algorithm, its recursive duality in splitting and merging grants an extra demand of memory, which may affect the capacity in memory-conscious scenarios.

Quicksort, represented by the green line, had the least run time irrespective of the size of the inputs. This stresses its higher average case performance, mainly because it is in-place and has less overhead. Avoiding worst-case scenarios (by randomly selecting the pivot element to use) means that Quicksort will beat the others in practice even though it has a quadratic complexity in the worst-case scenario.

The theoretical analysis of complexity is confirmed by comparison plot and demonstrates the effect of algorithmic decisions on the real results. Quicksort is a sort which performs well on random data, Mergesort is consistently stable, and Heapsort may trade better bounds on time complexity, but at the expense of increased overhead. The comparison plot is shown below.



Conclusion

The empirical and theoretical reviews reveal that although the three algorithms discussed in sorting, Heapsort, Mergesort, and Quicksort have the same average time complexity using O(n log n), they differ largely based on their implementation and the nature of the data entering the algorithm. Because of its low overhead and in-place design, Quicksort proves to be the fastest with random inputs. Mergesort is stable, although it has disadvantageous increased memory footprint. Heapsort has deterministic time performance and is space-efficient but tends to be slower compared to other sorting algorithms even though it does not require much memory because it is often reheapifying.

The implementation of the priority queue by means of binary heap is efficient in the management of dynamic priorities of tasks with predictable performance. It aids indeed- or-necessary applications such as insertion and extraction within logarithmic time and is applicable in applications such as CPU scheduling, event simulation and real-time systems.

On the whole, this assignment affirms the relevance of being familiar with data structure insides in order to make informed choices of the algorithms to implement in your coding depending on what you are computing (Tjernström, K. J., 2025).

Reference

Tjernström, K. J., & Paulsson, V. (2025). A Performance Study of Priority Queues: Binary Heap, Fibonacci Heap, Hollow Heap. *LU-CS-EX*.

Meng, S., Zhu, Q., & Xia, F. (2019). Improvement of the dynamic priority scheduling algorithm based on a heapsort. *IEEE Access*, *7*, 68503-68510.