Name: Shimon Kumar Bhandari

ID: 005030903

Course: Algorithms and Data Structures (MSCS-532-A01)

Assignment 4

Heapsort

Time Complexity Analysis of Heapsort

The time complexity of heap sort for worst-case, average-case, and best-case performance is O(nlogn). This uniform performance arises from the two main stages of the algorithm: Constructing a max-heap at first followed by continual maximum value removals and preserving the heap structure. To build the heap we apply heapify from the bottom-most non-leaf node ascending to the root node. Heapify has the worst time complexity of O(logn) which is also the height of Heap but as most of the nodes are at the lower level, it takes less time. The complete heap building process operates within O(n) time constraints.

Once the heap construction process is complete the algorithm continues by exchanging the heap's root element with its final node then shrinks the heap by one position. The heapify procedure restores the max-heap condition after each swap and operates with a time complexity of O(logn) for each extraction step. The extraction phase requires n−1 repetitions of the process leading to an execution time of O(nlogn). Combining heap construction at O(n) with extraction at O(nlogn) produces an overall time complexity of O(n+nlogn) which simplifies to O(nlogn). Regardless of sequence orders for data input this algorithm maintains its O(nlogn) performance efficiency which means Heapsort achieves such performance uniformly across cases.

No matter the order of its elements, Heapsort runs with O(nlogn) time complexity because it leverages heap operations which always take O(logn) time for insertion or deletions. The heap data structure maintains Heapsort's steady O(nlogn) performance because it dictates both extraction and construction operations which together maintain this time complexity. Heap insertions and deletions take logarithmic time because their performance depends solely on the

tree's height traversals. The process of heapifying nodes during heap construction has costs which increase with node height. Higher levels of the tree contain fewer nodes which results in an overall heap construction cost those averages to O(n). O(logn) operations extract each of the n−1 elements during this phase which produces a time complexity of O(nlogn). This dual-step process maintains an O(nlogn) complexity independent of both input size and arrangement.

Space Complexity and Overheads

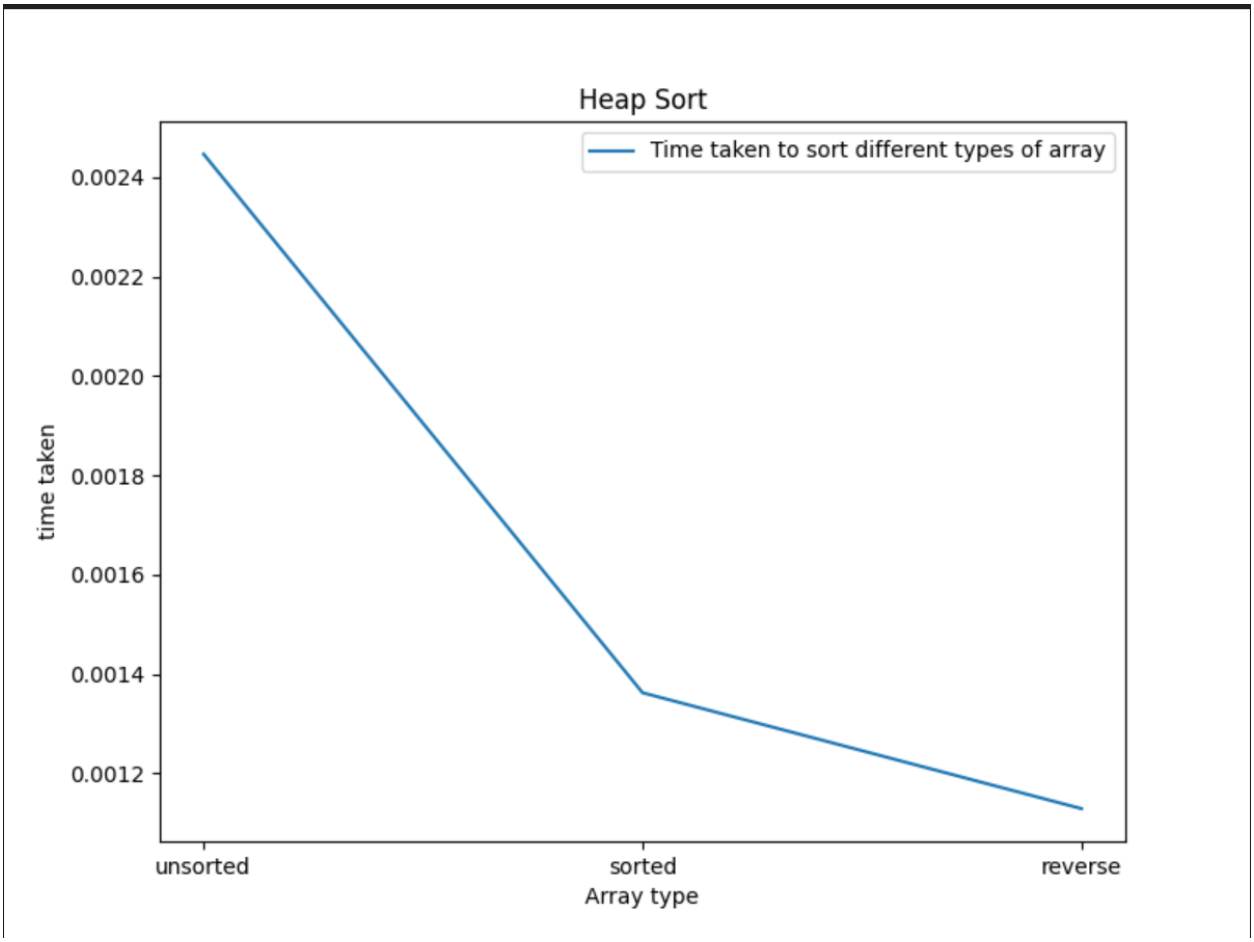
While performing its sorting operations Heapsort maintains efficiency by using the array it sorts to avoid extra memory allocation for other data structures. The space complexity of this algorithm amounts to O(1) when you don’t include the memory used by the input array. When the heapify function uses recursion the algorithm experiences recursion stack overhead that grows proportionally to the heap height measured in O(logn). An iterative heapify implementation prevents recursion stack overhead during heapsort execution. When compared with Merge Sort that demands O(n) extra space, Heapsort proves more space efficient.

Comparisons

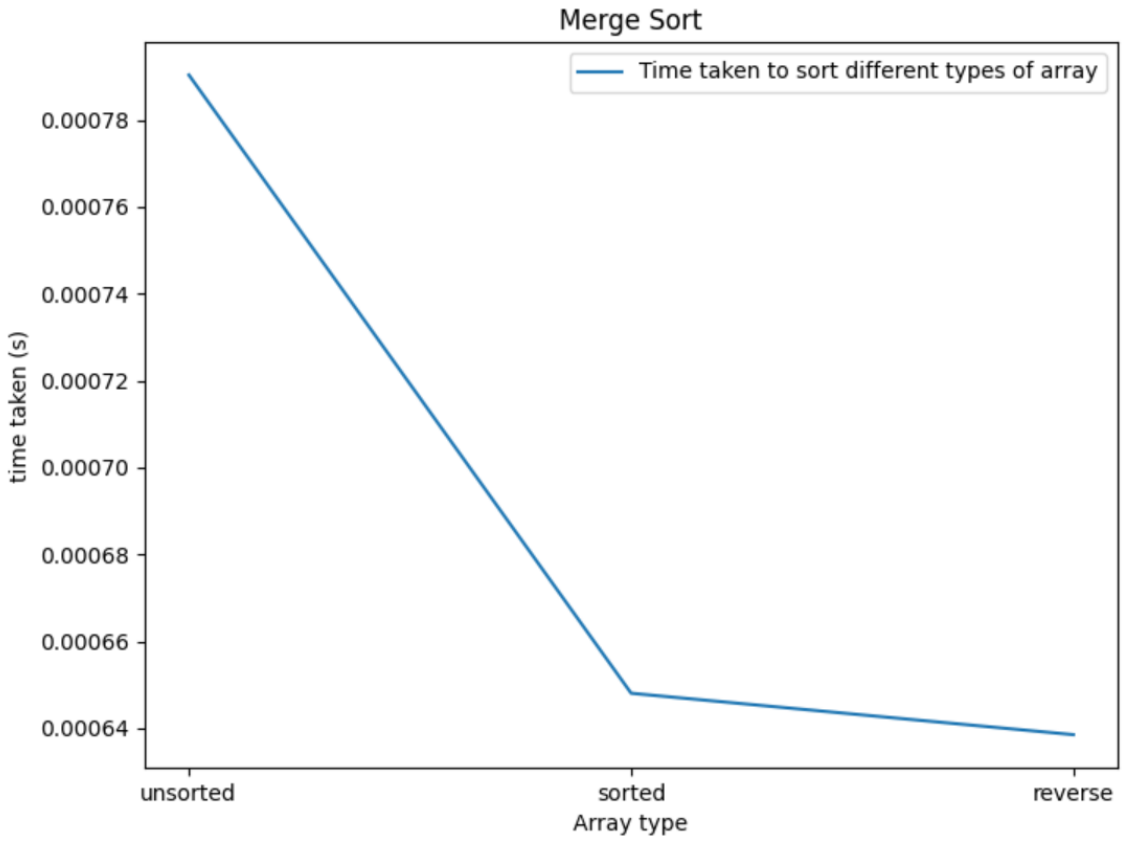
Because of its space-efficient properties together with constant O(nlogn) performance Heapsort becomes a dependable selection for memory-restricted systems. Heapsort maintains consistent performance throughout any input set even though Quick Sort's efficiency can plummet to O(n^2) during its worst performance scenarios. Heapsort operates inefficiently with cache memory compared to Quick Sort because of its random-access pattern yet it continues to be a useful method for systems that demand both time and space efficiency.

When comparing heap sort with merge sort; the time complexity is O(nlogn) for worst, average, and best cases. The algorithms demonstrate substantial differences in space efficiency and practical performance because they use memory differently and have distinct caching behaviors. Heap Sort becomes the better choice in environments with limited memory despite Merge Sort delivering faster performance under most conditions because of how it utilizes the cache.

The below image is the time taken by heap sort to sort 990 elements of unsorted, sorted and reverse sorted array.



The below image is the time taken by Merge sort to sort 990 elements of unsorted, sorted and reverse sorted array.



It can be seen that both merge sort and heap sort follow the similar pattern when sorting different types of arrays.

Priority Queue Implementation

A priority queue for task scheduling was constructed through a binary heap structure which was implemented as a list. The binary heap was selected because of its effective performance when maintaining heap structure during insertion processes and priority updates which meet the demands of real-time application scheduling. Key binary heap operations achieve logarithmic time complexity O(logn) while performing task insertion and extraction of the highest-priority task together with task priority modification. The array representation of the heap was selected for its simplicity and compact memory usage, as parent-child relationships can be computed using simple arithmetic operations: (i−1)//2 for the parent and 2i+1 and 2i+2 for the left and right children, respectively. By choosing this approach we achieve less overhead than pointer-based tree implementations and benefit from Python's automatic list expansion features.

The implementation results show that the priority queue arranges task scheduling in order of priority levels effectively. Real-time adaptability comes from processing high-priority tasks first combined with seamless management of dynamic priority changes. The binary heap structure supports system scalability because its logarithmic operational performance maintains system efficiency even as task quantities increase. Architecture thrives in environments where task priorities continuously change, and rapid decision-making remains essential.

The binary heap-based priority queue serves as both a powerful and reliable framework for task scheduling operations. This system reaches ideal status for real-world applications requiring dynamic task prioritization because of its straightforward yet scalable and efficient design. The research extends fundamental algorithmic knowledge described in scholarly sources through heap-based priority queue implementations alongside real-time scheduling systems.

Time complexity:

The logarithmic height of the binary heap structure dictates the time complexity for its priority queue operations. During the insert operation the algorithm requires O(logn) time for worst-case scenarios because each insertion might require upward adjustments throughout the heap's height to maintain its structural property. With optimal placement of the new element, the binary heap operation requires only O(1) time. extract_max requires removing the root element that has the highest priority and then restores heap structure by moving the last element to root and executing a heapify_down process. Pushing to correct positions in a heap takes maximum O(logn) time because one might need to traverse from root to leaf. The change_priority operation functions with O(n

). It takes O(n) time to find the key of which the priority is to be updated. It takes O(logn) complexity when modifying a task priority since it demands heapifying up or down to maintain heap order through increased or decreased priority values. The operations is_empty and maximum/minimum retrieval without element removal execute in O(1) time because they need only one heap root access. The binary heap maintains efficient performance for priority queue tasks through logarithmic time complexity for updating operations while allowing simple queries to execute instantly.

References:

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Drozdek, A. (2013). *Data Structures and Algorithms in C++*. Cengage Learning.