Heapsort Analysis

Heapsort has the time complexity of O(n.logn) in all the cases because the sorting process begins with building a max heap from an input array. In order to build the heap, we need to rearrange elements in such a way that the largest element is always at the root. Along with that, every subtree should satisfy the heap property. This step involves starting from the last non-leaf node and applying the heapify operation to each node, which takes O(logn) for each node. Going over each node to heapify takes O(n) time for heap construction.

The next step of Heapsort is repeatedly extracting the largest element from the root. After each extraction, the last element in the heap is swapped with the root, and the heap size is reduced. The heap property is then restored using the heapify operation, which takes O(logn). This extraction and heapify process is repeated n−1 times, resulting in O(n.logn) time. Since both building and sorting steps contribute to O(n.logn), the total complexity remains the same regardless of input data order, ensuring consistent performance for sorted, reverse-sorted, or random data.

Heapsort has O(nlogn) time complexity in all cases because of how the max-heap works. A max-heap is a binary tree where each parent is larger than its children. Building the heap involves fixing each node from bottom to top, which takes O(n) in total because deeper nodes require less work. During sorting, the largest element is repeatedly moved to the end, and heapify is called on a smaller heap, taking O(logn) on each step. This gives O(nlogn) overall for any kind of data.

From the implementation, we can see heapsort is very memory-efficient because of the way it operates. It requires O(1) additional space since all operations are performed directly on the input array, without using extra data structures. Heap sort should perform better in the cases where low memory is available. However there are two different ways to implement the heapify method for heap sort: iterative and recursive. An iterative implementation keeps memory overhead minimal, but a recursive approach may use O(logn) space due to the recursion stack.

Here is the performance comparison between heapsort, quicksort and merge sort for different types of ordered and random data. The below results show time taken for different algorithms to sort different order of data for 10000 integers.

Sorting Algorithm	Sorted	Reverse sorted	Random	Random with duplicates
MergeSort	0.0126 sec	0.0160 sec	0.0120 sec	0.0121 sec
Quicksort randomized	0.0109 sec	0.0169 sec	0.0150 sec	0.0136 sec
Heapsort	0.0157 sec	0.0137 sec	0.0149 sec	0.0147 sec

Here is the performance comparison between heapsort, quicksort and merge sort for different types of ordered and random data. The below results show time taken for different algorithms to sort different orders of data for 20000 integers.

Sorting Algorithm	Sorted	Reverse sorted	Random	Random with duplicates
MergeSort	0.0240 sec	0.0237 sec	0.0277 sec	0.0273 sec
Quicksort randomized	0.0218 sec	0.0208 sec	0.0212 sec	0.0217 sec
Heapsort	0.0045 sec	0.0340 sec	0.0566 sec	0.0333 sec

From the above results, merge sort and quicksort seem to perform slightly better than heapsort as the volume of data increases. This could be because heapsort often performs more comparisons than both Quicksort and Merge Sort due to its heapify process during extraction and heap construction. However heapsort seems to perform consistently between the different types of ordered data.

Analysis for Binary heap implementation:

In order to implement the binary heap or the scheduler program, I chose the array data structure. This is because the parent-child relationship is directly mapped using array indices, making heap operations like insert and extract efficient. While a list could be used, it requires manually tracking the indices of nodes and their children, which adds complexity and reduces efficiency.

In this Heapsort implementation, I used an iterative approach for the heapify function to avoid recursion and minimize space usage. The algorithm operates in-place on the input list, ensuring minimal memory overhead. The max-heap is built bottom-up, and the sorting process involves repeatedly swapping the root with the last element, then restoring the heap property. This approach guarantees O(nlogn) time complexity for insertion and deletion. I designed a maximum heap where the task with the highest priority gets extracted first. Here are the methods available for the binary heap with their expected time complexity:

1. Insert: In this heap implementation, inserting a new element is efficient because the task can be appended to the end of the array. It takes constant time O(1)). The next step is to perform a heapify operation, which takes O(logn) to maintain the heap property. This makes the total time complexity of insertion O(logn).
2. Extract max: In the same way, removing the max priority item takes O(logn) time complexity. This can be done by swapping it with the last element O(1), followed by "heapify-down" to restore the heap property. These combined steps ensure the time complexity to be O(logn)
3. Increase/Decrease priority: This operation updates the priority of an existing task and restores the heap property by bubbling the task upwards as needed. Firstly, finding the task's index in the heap takes O(n) time, as it requires searching the entire heap. The second step is to move the item up if the priority is increasing and move the item down if

the priority is decreasing. These steps take O(logn) time, making the total time complexity O(n).

4. Check empty: To check the empty heap, the array's length is checked. This step takes constant time. Therefore, an empty heap check takes O(1) time.

In order to test the maximum heap implementation, I created an instance of the heap and created three objects with varying levels of priorities. Then, I extracted the max item from the heap and got the expected task. I also made sure to increase and decrease the priority of tasks before extracting more items. Once all the items were extracted, I made sure to check the empty function, and it returned the empty status as expected.