# Assignment 4

The method begins with constructing a helper function, heapify, with three inputs: the array arr, the total element count n, and the current index i. The heapify operation guarantees the max-heap quality of the subtree anchored at index i. This is accomplished by first comparing the current node with its left and right children; if required, it is then swapped with the biggest of the three and thereafter recursively applied to the impacted subtree. Starting from the final internal node and proceeding backwards to the root, heapify on all non-leaf nodes first turns the input array into a max-heap. The heap size is one less when the biggest member—the root—is swapped with the final element of the array after heap building. Once again executed on the root, the heapify method helps to restore the heap characteristic. This procedure keeps on until every component is arranged.  
  
Since each element addition into the heap takes O(log n) time and there are n items to sort, the Heapsort method guarantees in terms of complexity an O(n log n) time complexity in the worst, average, and best circumstances. Operating in-place and needing just extra memory aside from the input array, this version's space complexity is O(1). Heapsort is an excellent contender for sorting issues when memory utilization has to be minimised as its simple and effective implementation is appropriate. The method is not stable, hence after sorting equal items could not keep their relative rank. Heapsort is a commonly used method despite non-stability because its deterministic character and consistent performance across many input kinds. Since it follows the fundamental procedures of heap formation, extraction, and re-heapification without adding more complexity, this version is particularly appropriate for learning the basic ideas of heap-based sorting.

A screenshot of a computer program

Description automatically generated

Individual assignments are shown using the class Task, which has a task\_id and a priority level. The PriorityQueue class is used to implement the priority queue itself; an internal list (or array) denotes the binary heap. The queue facilitates the basic heap operations like extract\_max and insert. When adding a new job, the heap is added to and the \_heapify\_up mechanism is run to guarantee heap property is kept. If the youngster has more priorities, this entails matching the newly assigned chore with its parent and switching them. The action keeps on until the right location for the additional work is discovered, hence preserving the max-heap attribute. Usually the root of the heap, the extract\_max method removes and returns the job with the greatest priority. The final element in the heap is relocated to the root location after root extraction; thus, the \_heapify\_down function is invoked to restore heap property. This entails contrasting the root with its offspring and, should required, switching it with the bigger kid. The procedure go on till the heap is in a legitimate condition. With an O(log n) time complexity for both insertion and extraction driven by the need to preserve the heap structure, this implementation concentrates on the fundamental operations required for a priority queue. Particularly for applications where only simple heap operations are required, the simplicity of the first version makes understanding and implementation simple. Using a max-heap guarantees that chores with greatest priority are always completed first; the internal view of the heap as an array allows effective indexing and manipulation.

A screenshot of a computer program

Description automatically generated