# Ass 4

Part 1

Upon receiving an empty input array, the function promptly terminates without doing any actions. Likewise, if the array consists of a single member, prior sorting is unnecessary, and the array is delivered in its original form. These checks serve to optimize calculations and enhance the resilience of the system in real-world situations characterized by fluctuating input sizes. The fundamental sorting algorithm stays unchanged, where the array is first converted into a max-heap and then sorted by iteratively exchanging the root element with the final element and reapplying the heapify function to preserve the heap characteristic. Furthermore, this variant preserves the time complexity of O(n log n) by ensuring that the processes of constructing the heap and extracting items remain directly proportional to the logarithmic height of the heap. Inconsequential to the total time complexity, the supplementary tests for edge situations are executed in constant time, O(1). In-place operation of the method, similar to the first version, maintains a space complexity of O(1). Nevertheless, the more comprehensive error handling in this version enhances its suitability for real-world applications where input arrays may not always adhere to optimal circumstances. Implementing early returns for empty and single-element arrays prevents superfluous heap operations, thereby enhancing the efficiency of the code for certain scenarios. Furthermore, this version exemplifies a defensive programming approach, guaranteeing the algorithm's proper behavior even in the presence of atypical or unforeseen inputs. In summary, the second iteration of Heapsort offers a refined and adaptable implementation that maintains the fundamental concepts of the algorithm. This makes it a superior option for practical applications that need elegant handling of edge circumstances.

A close-up of numbers

Description automatically generated

Part 2

The priority queue functionality is implemented by using a max-heap, where the heap is represented as either an array or a list. The fundamental principle of the heap is to guarantee that the job with the greatest importance is consistently positioned at the bottom of the heap, therefore facilitating the extraction and efficient processing of the most pressing tasks. The prioritization queue architecture incorporates a Task class that retains details about each tasks, including their ID and priority. These capabilities enable the priority queue to handle a diverse range of jobs, each of which may be assigned a priority value. The priority queue actions are executed inside a class that oversees the heap and offers essential features such the insertion of new tasks, extraction of the highest priority job, and adjustment of the priority of current tasks. When a job is inserted into the heap, it is first appended to the end of the array. Following insertion, the location of the job is modified via a procedure known as "heapify up." This action evaluates the recently added job by comparing it with its parent node and exchanging them if the child task has a higher priority. The aforementioned procedure is iterated until the job attains a threshold where the heap property is fulfilled. Consequently, the heap maintains its validity and guarantees that the job with the greatest priority is consistently placed at the root. During the extract\_max operation, the root element, which represents the job with the greatest priority, is removed and substituted with the last element in the heap. Following this, the heap is reinstated to its original condition by executing a "heapify down" operation, which evaluates the new root in relation to its offspring and, if needed, replaces it with the most significant offspring. The aforementioned procedure persists until the heap property is reinstated, therefore guaranteeing that the subsequent job with the greatest priority is assigned to the root. Furthermore, the solution incorporates operations to adjust the priority of jobs currently stored in the heap, in addition to fundamental insertion and extraction activities. The increase\_key function enables the elevation of a task's priority, a scenario that may occur in a system where tasks get more urgent as time progresses. Once the priority is raised, the task's location in the heap is modified by the heapify up process to maintain the heap's validity. Furthermore, the decrease\_key method reduces the priority of a job and modifies its position by heapify down, which is advantageous in situations where tasks become less crucial. Such processes are crucial for practical applications in which the priority of tasks may vary dynamically. The temporal complexity of core operations, including insertion, extraction, and priority adjustment, is O(log n) due to the inherent logarithmic height of the heap involved in each operation. Given that the heap is represented as an array and no extra memory structures are needed, the space complexity stays economical. The proposed implementation achieves a harmonious equilibrium between simplicity and functionality, enabling effective task management while also accommodating flexible adjustments to work priorities. These characteristics render it appropriate for real-time scheduling systems, where jobs may need insertion, extraction, and modification in response to evolving requirements.

A close-up of a computer screen

Description automatically generated