# Ass 4

# Part 1

The Heapsort function leaves without doing any operations right away when it comes across an empty array. Likewise, should the input array have just one element, no sorting is required, and the array is returned unaltered. By eliminating needless calculations, particularly in real-world situations where input sizes may vary substantially, these first checks maximize the performance.   
  
The Heapsort method's fundamental idea is still the same: the array is first converted into a max-heap then repeatedly swapped the root (the biggest element) with the last unsorted element then heapify once more to restore the heap property. This method guarantees that both the heap building and element extraction phases are proportionate to the logarithmic height of the heap, therefore preserving the time complexity of \( O(n \log n) \).   
  
Executed in constant time, \( O(1) \), the edge case tests for empty and single-element arrays have no effect on the general time complexity. Heapsort is a in-place sorting method hence preserves a space complexity of \( O(1) \). These basic tests, however, help the function to be more robust for pragmatic uses when ideal circumstances may not always be met from input arrays.   
  
Early returns for these unique situations help the function to avoid needless heap operations, hence optimizing the performance in circumstances where tiny or empty arrays are typical. This version additionally shows defensive programming by guaranteeing appropriate behavior even with unanticipated inputs. Overall, this improved Heapsort implementation preserves the fundamental properties of the algorithm while providing flexibility and resilience, hence improving its fit for practical uses requiring consistent handling of edge situations.

A close-up of numbers

Description automatically generated

# Part 2

Using a max-heap structure, the priority queue is set wherein the heap is shown as an array or list. The highest-priority work in a max-heap is always located at the top (root), therefore enabling effective extraction and processing of the most pressing chores. This priority queue architecture incorporates a task class to store information about every job, including its ID and priority, therefore allowing the queue to handle a range of tasks, each with an allocated priority level.   
  
The actions of the priority queue are under control within a class that preserves the heap and offers necessary features such task insertion, task extraction, and task priority modification of current tasks. A job entered first finds their place at the end of the array. The "heapify up" process is used to change the location of the additional work within the heap after insertion. This operation guarantees the heap property by swapping the new task with its parent node should the child have greater priority. This approach guarantees that the highest-priority job stays at the root because it keeps on until the job finds a valid place in the heap.   
  
The final element in the heap replaces the root element (highest-priority job) eliminated during the extract\_max operation. The heap is then rebuilt using the "heapify down process, in which case the new root is swapped with the highest-priority child if necessary after being compared with its children. This operation is continued until the heap property is recovered, therefore guaranteeing that the next highest-priority job takes root position.   
  
Apart from simple insertion and extraction, the program enables priority changes for previously heap-occupied jobs. In dynamic systems where task urgency might vary with time, the increase\_key function elevates the priority of a task. The location of the job changes via the heapify up process to preserve heap order if the priority rises. When chores become less important, the decrease\_key function similarly lowers their priority and employs heapify down to rearrange them. These capabilities provide systems whose job priorities must be dynamically changed flexibility.   
  
Since each core operation—insertion, extraction, and priority adjustment—involves traversing the logarithmic height of the heap, the time complexity for those operations is \(O(\log n)\). The space complexity is maintained low as the heap is retained as an array devoid of extra data structures, therefore optimizing the implementation memory-wise.   
  
With dynamic priority changes, this priority queue strikes a mix of simplicity and utility that lets task management be effective. Real-time scheduling systems where jobs may need to be added, deleted, and altered depending on evolving needs fit this concept.

A close-up of a computer screen

Description automatically generated