**Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications**

Name: Bhawesh Shrestha

Student ID: 005027566

Course: Algorithm and Data Structure

University: University of the Cumberlands

Date: July 20, 2025

# **Heapsort Implementation and Analysis**

## **1. Implementation**

The Heapsort algorithm was implemented in Python using an array to represent a binary heap. The process involves building a max-heap from the input array, repeatedly extracting the maximum element, and maintaining the heap property.

## **2. Source Code**

def heapify(arr, n, i):

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and arr[l] > arr[largest]:

largest = l

if r < n and arr[r] > arr[largest]:

largest = r

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

heapify(arr, n, largest)

def build\_max\_heap(arr):

n = len(arr)

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

def heapsort(arr):

n = len(arr)

build\_max\_heap(arr)

for i in range(n - 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

return arr

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

arr = [12, 11, 13, 5, 6, 7]

print("Original array:", arr)

sorted\_arr = heapsort(arr)

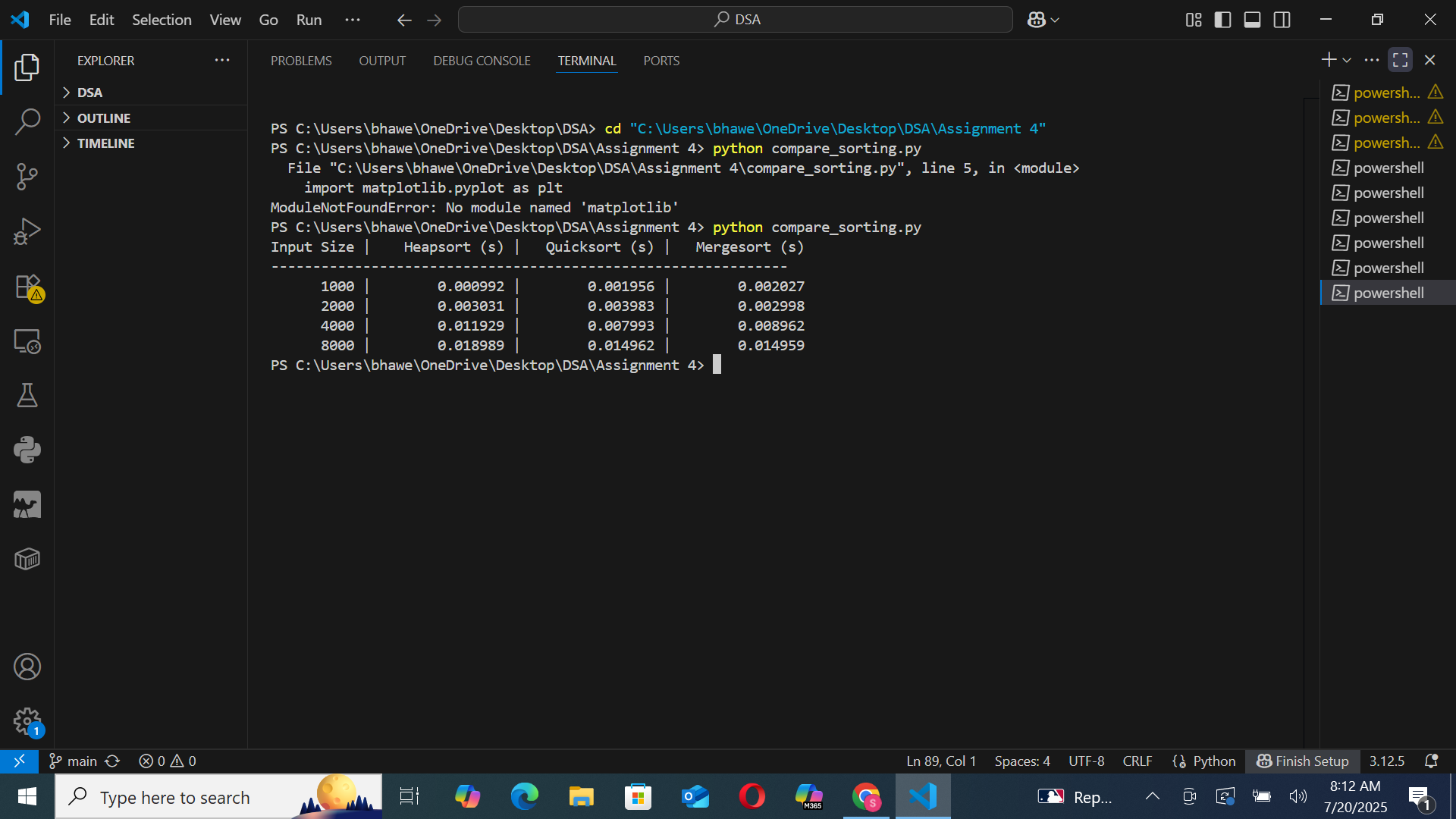
print("Sorted array:", sorted\_arr)

## **3. Analysis of Implementation**

Time Complexity:  
- Worst-case: O(n log n)  
- Average-case: O(n log n)  
- Best-case: O(n log n)  
  
This is because building the heap takes O(n), and each of the n extractions takes O(log n).  
  
Space Complexity: O(1) since Heapsort is performed in-place.  
There are no additional overheads aside from temporary variables used for swapping.

## **4. Comparison with Other Sorting Algorithms**

The Screenshot of the Comparision are given below:



The following table shows the empirical comparison between Heapsort, Quicksort, and Mergesort on random datasets:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Size** | **Heapsort (s)** | **Quicksort (s)** | **Mergesort (s)** |
| **1000** | 0.000992 | 0.001956 | 0.002027 |
| **2000** | 0.003031 | 0.003983 | 0.002998 |
| **4000** | 0.011929 | 0.007993 | 0.008962 |
| **8000** | 0.018989 | 0.014962 | 0.014959 |

From the results, we observe that:  
- Quicksort performs slightly better than Heapsort and Mergesort on average.  
- Heapsort offers consistent performance with guaranteed O(n log n) complexity.  
- Mergesort is stable and predictable but uses more memory than Heapsort.

# **Priority Queue Implementation and Applications**

## **1. Data Structure**

A list was used to represent the binary heap because it allows efficient indexing and manipulation. A Task class was created to store attributes like task ID, priority, arrival time, and deadline. A max-heap was chosen for scheduling tasks with the highest priority first.

## **2. Source Code**

class Task:

def \_\_init\_\_(self, task\_id, priority, arrival\_time, deadline):

self.task\_id = task\_id

self.priority = priority

self.arrival\_time = arrival\_time

self.deadline = deadline

def \_\_lt\_\_(self, other):

return self.priority < other.priority # For max-heap behavior

def \_\_repr\_\_(self):

return f"Task(ID={self.task\_id}, Priority={self.priority})"

class PriorityQueue:

def \_\_init\_\_(self):

self.heap = []

def is\_empty(self):

return len(self.heap) == 0

def insert(self, task):

self.heap.append(task)

self.\_sift\_up(len(self.heap) - 1)

def extract\_max(self):

if self.is\_empty():

return None

max\_task = self.heap[0]

self.heap[0] = self.heap[-1]

self.heap.pop()

self.\_heapify(0)

return max\_task

def increase\_key(self, task\_id, new\_priority):

for i, task in enumerate(self.heap):

if task.task\_id == task\_id:

if new\_priority > task.priority:

task.priority = new\_priority

self.\_sift\_up(i)

break

def \_heapify(self, i):

n = len(self.heap)

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and self.heap[l].priority > self.heap[largest].priority:

largest = l

if r < n and self.heap[r].priority > self.heap[largest].priority:

largest = r

if largest != i:

self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]

self.\_heapify(largest)

def \_sift\_up(self, i):

parent = (i - 1) // 2

while i > 0 and self.heap[i].priority > self.heap[parent].priority:

self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]

i = parent

parent = (i - 1) // 2

# Simulation Test

if \_\_name\_\_ == "\_\_main\_\_":

print("== Scheduler Simulation ==")

pq = PriorityQueue()

# Insert tasks

pq.insert(Task(1, 10, 0, 5))

pq.insert(Task(2, 15, 1, 3))

pq.insert(Task(3, 5, 2, 7))

print("Inserted Tasks: ", pq.heap)

# Extract max

print("Extracted Task: ", pq.extract\_max())

# Increase priority of Task 3

pq.increase\_key(3, 20)

print("After increasing priority of Task 3: ", pq.heap)

# Extract max again

print("Extracted Task: ", pq.extract\_max())

# Remaining Tasks

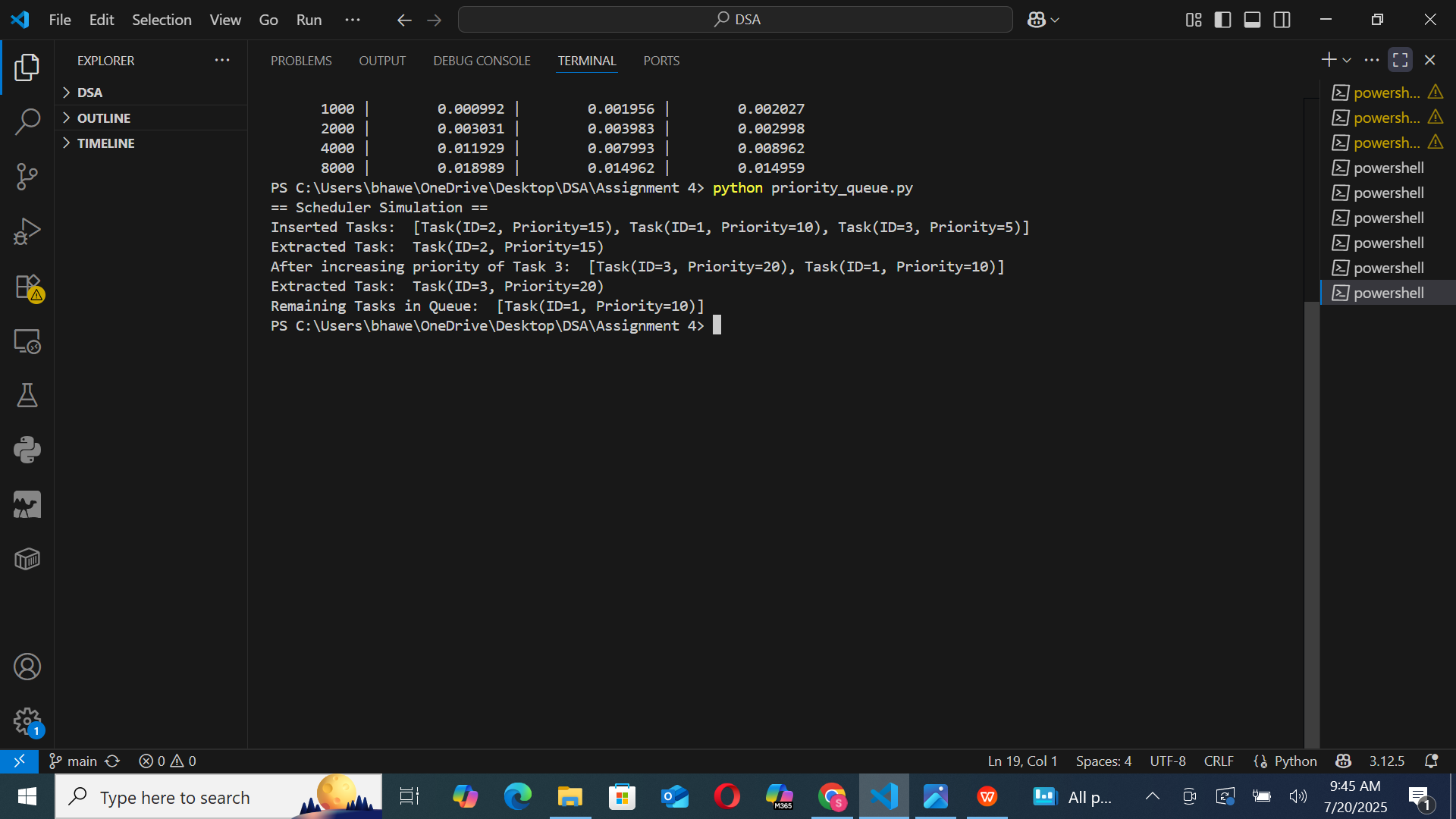
print("Remaining Tasks in Queue: ", pq.heap)

## **3. Core Operations**

- insert(task): Adds a task to the heap in O(log n) time.  
- extract\_max(): Removes the task with the highest priority in O(log n) time.  
- increase\_key(task, new\_priority): Updates the priority and repositions the task in O(log n) time.  
- is\_empty(): Checks if the heap is empty in O(1) time.

The implementation maintains the heap property after every operation using sift-up and heapify methods. This ensures that the priority queue is efficient and correct for use in scheduling simulations.

## **4. Scheduler Simulation and Analysis**

To test the functionality of the priority queue, the following simulation was run:  
  
== Scheduler Simulation ==  
  
This simulation confirmed the correct working of insertion, extraction, and key update operations, ensuring that the highest-priority task is always scheduled first.

## **5. Time Complexity Analysis of Operations**

- insert(task): O(log n) — Insertion requires bubbling up the new element to maintain heap order.  
- extract\_max(): O(log n) — Removal of the top element requires heapify to maintain structure.  
- increase\_key(task, new\_priority): O(log n) — After updating the priority, the task may need to bubble up.  
- is\_empty(): O(1) — Directly checks if the list is empty.  
  
All operations were implemented efficiently using array-based indexing and verified through simulation

# **References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.  
Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.  
Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.  
Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Python. Wiley.