Unique Karanjit

Department of Computer Science, University of the Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Dr. Satish Penmatsa

Jan 26, 2025

Heapsort is an algorithm that uses a binary heap data structure to sort elements efficiently. It is called an in-place algorithm which means that no extra memory is required. It has a time complexity $O(n\ logn)$ in all cases. The time complexity can be calculated as follows.

1. Building the Max-Heap: $\boldsymbol{O(n)}$
2. Heapify Operation: $\boldsymbol{O(\ logn)}$**,** which is repeated n times during extraction.
3. Total: $\boldsymbol{O(n\ logn)}$

The time complexity can be analyzed as follows. The algorithm consists of two main phases.

1. Building a max heap from the array.
2. Extract the maximum element repeatedly and maintain the heap.

Building the Max Heap

We call the heapify function starting from the last non-leaf node and moving upward to the root. Each call to heapify takes $\boldsymbol{O(h)}$ time, where h is the height of the current node. The height of each node at index i is approximately $log_2(i+1)$.

The number of nodes at each height decreases as we go deeper in the tree.

i. The node at height h=0: $n/2$ nodes → heapify cost=0

ii. The node at height h=1: $n/4$ nodes → heapify cost=1

iii. The node at height h=2: $n/8$ nodes → heapify cost=2

iv. The node at height h=2: $n/16$ nodes → heapify cost=3

v. And so on.

The total cost of building the heap is expressed as follows.

$$T_{build} = \sum_{h=0}^{log_2 n} (number\ of\ node\ at\ height\ h) * (heapify\ cost\ at\ height\ h)$$

This can be summed to $\boldsymbol{O(n)}$**,** since the operation is directly proportional to the number of elements in the array.

Extract the Maximum Element

After building the max-heap, the largest element is at the root. We then swap it with the last element and reduce the size of heap by 1. Then, call heapify on the root to restore the max-heap. The heapify operation takes $\boldsymbol{O(\ logn)}$ time because it traverses the height of the tree. This process is repeated n-1 times. Therefore, the total time can be calculated as follows.

$$T_{extract} = (n-1) * O(logn) = O(nlogn)$$

The overall time complexity of heapsort is a total of Tbuild and Textract.

$$T_{total} = T_{build} + T_{extract} = O(n) + O(nlogn) = O(nlogn)$$

HeapSort is $O(nlogn)$ in All Cases. The reason can be explained as follows.

- Best Case: This is the case where the input is already sorted. But, also, we must build the heap O(n) and repeatedly extract the maximum O(nlogn). Therefore, the algorithm does not take advantage of any prior order in the data.
- Worst Case: Since the array is already sorted, the same steps need to be performed as in the best case. So, heapsort ensures consistent performance regardless of the input.
- Average Case: The heapsort performs the same operation for any input, the same steps need to be performed, so it is identical to the best and worst case.

  The O(nlogn) complexity in all cases happens because of this.

i. O (n) : Build the max-heap

ii. O (log n): For each extraction which is repeated (n-1) times.

Space Complexity

Since heapsort is an in-place sorting algorithm, it does not require additional memory for separate data structures. Hence, the space complexity for this algorithm is O(1). The reason behind this is that this algorithm arranges the element within the input itself, and only a small constant amount of extra space is needed for temporary variables during heapification.

The recursive calls to heapify require space for the function call stack. However, the iterative version of heapify avoids this overhead. This makes sure the overall space complexity O (1).

Comparison between Heapsort, Quicksort and Merge sort

The time complexity between heapsort, quicksort and mergesort for best case, worst case and average time complexity can be shown in table as follows.

Algorithm	**Best Case**	**Worst Case**	**Average Case**
Heap Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Quick Sort	$O(nlogn)$	$O(n^2)$	$O(nlogn)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$

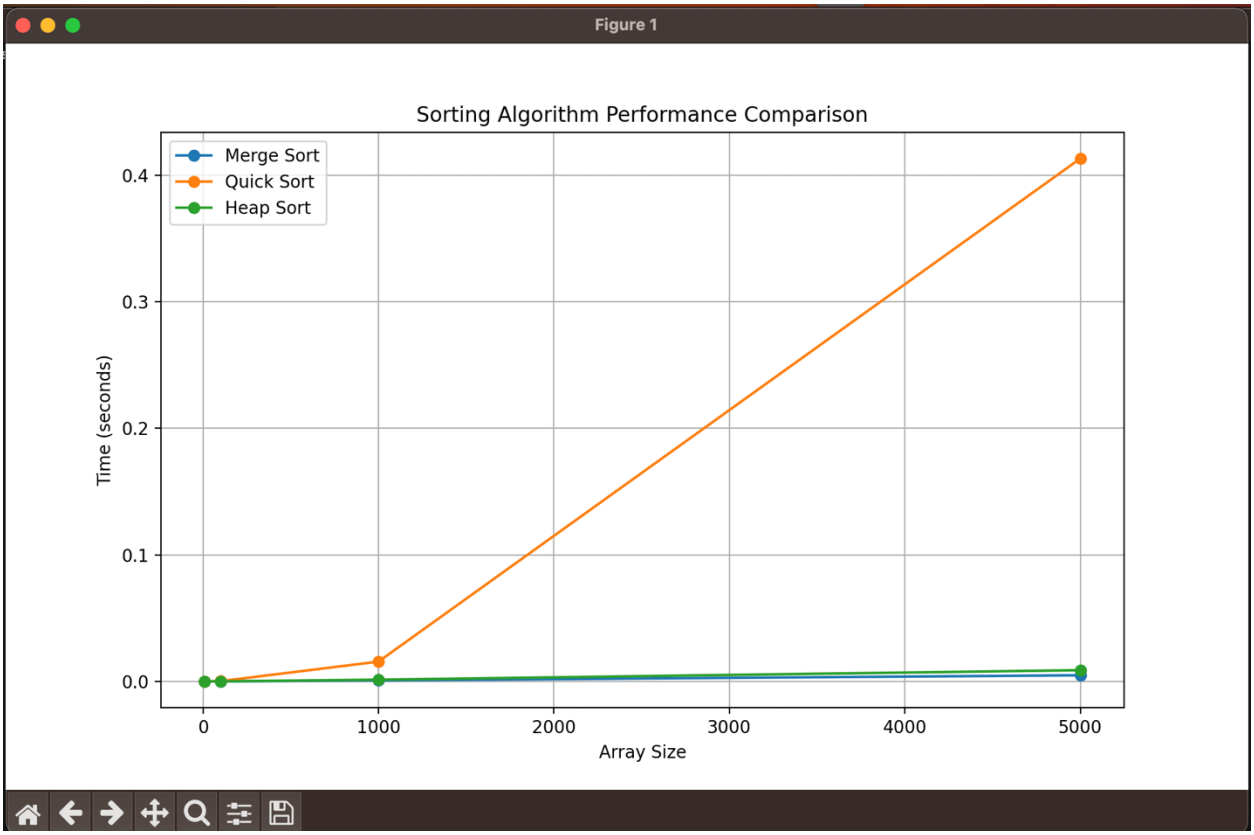

Fig: Sorting a sorted array.

This matches with the theoretical analysis.

Priority Queue Implementation

Design Decision

Data Structure: Binary Heap

A binary heap is used to achieve the priority queue because it is efficient and easy to implement. A binary heap can be represented by an array (or list), and it can be used to provide optimal performance for the main operations of a priority queue. The binary heap supports efficient insertion and extraction, which is crucial in scheduling tasks based on priority.

Task Representation

A class Task was defined to store the details of each task. The class consists of:

- task_id: An identifier for the task.
- priority: The task priority upon which the processing order will be decided (higher priority first).
- arrival_time: The time of arrival of the task in the queue.
- deadline: The deadline of the task by when it should be processed.

These fields enable sophisticated task management in scheduling systems where the tasks can be of varying urgency levels, arrival times, and deadlines.

```python
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 __repr__(self):
        return f"Task ID: {self.task_id}, Priority: {self.priority}, Arrival Time: {self.arrival_time}, Deadline: {self.deadline}"
```

Max-Heap Choice: Given that tasks with higher priority need to be processed first, a **max-heap** was chosen for this implementation. This means that the task with the highest priority will always be at the root of the heap, allowing for efficient extraction of the highest-priority task.

Core Operation and Implementation

- Insert Operation: The insert() operation adds a new task to the heap. It is accomplished by inserting the task at the bottom of the heap and then heapify-up (bubbling up) it to ensure the heap property.

```
# Insert a new task into the heap
def insert(self, task):
    if task is None:  # Check if task is None before inserting
        print("Cannot insert None as a task.")
        return
    self.heap.append(task)
    index = len(self.heap) - 1
    while index > 0:
        parent_index = (index - 1) // 2
        if self.heap[parent_index].priority < self.heap[index].priority:
            self.heap[parent_index], self.heap[index] = self.heap[index], self.heap[parent_index]
            index = parent_index
        else:
            break
```

- Increase key Operation: The increase_key() operation allows us to alter the priority of an existing task in the heap. If the new priority is greater than the current one, the task is upgraded in the heap so that the heap property is still upheld.

```
# Increase the priority of a given task (bubble up)
def increase_key(self, task, new_priority):
    if task is None:  # Ensure task is valid before modifying
        print("Cannot increase key for None.")
        return
    task.priority = new_priority
    index = self.heap.index(task)
    while index > 0:
        parent_index = (index - 1) // 2
        if self.heap[parent_index].priority < self.heap[index].priority:
            self.heap[parent_index], self.heap[index] = self.heap[index], self.heap[parent_index]
            index = parent_index
        else:
            break
```

- Extract Max Operation: The extract_max() operation removes and returns the highest priority task (i.e., the root of the heap). Once the root is removed, the last task is moved to the root position, and the heap is then heapified down to maintain the heap property.

```
# Extract the task with the highest priority (max-heap)
def extract_max(self):
    if not self.heap:
        print("Priority Queue is empty, cannot extract.")
        return None

    root = self.heap[0]
    self.heap[0] = self.heap[-1]
    self.heap.pop()
    self.heapify_down(0)
    return root

# Helper function to maintain heap property by bubbling down
def heapify_down(self, index):
    left_child = 2 * index + 1
    right_child = 2 * index + 2
    largest = index

    if left_child < len(self.heap) and self.heap[left_child].priority > self.heap[largest].priority:
        largest = left_child
    if right_child < len(self.heap) and self.heap[right_child].priority > self.heap[largest].priority:
        largest = right_child
    if largest != index:
        self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]
        self.heapify_down(largest)
```

- Is Empty Operation: The is_empty() operation checks whether the priority queue is empty. The operation simply looks at the length of the heap.

```
# Check if the priority queue is empty
def is_empty(self):
    return len(self.heap) == 0
```

Time Complexity Analysis:

- Insert Operation: $\boldsymbol{O(logn)}$

  The insertion of a task is done by inserting the task at the end of the heap and then calling the heapify-up operation to ensure that the heap property is satisfied. This operation requires logarithmic time as each level of the heap is halved, and in the worst-case scenario, the task would involve traversing from the leaf node to the root node.

- Extract Max Operation: $\boldsymbol{O(logn)}$

  The choice of the highest-priority task involves replacing the root node with the terminal node, followed by a heapify-down operation. This process has comparison and node exchange down to the height of the heap in the worst case, with logarithmic complexity.

- Insert Key Operation: $\boldsymbol{O(logn)}$

  Task priority increase is accomplished by modifying the priority and then pushing the task up the heap. The operation is also logarithmic time since the task can potentially have to travel from a leaf node to the root.

- Is Empty Operation: $\boldsymbol{O(1)}$

  This is simply checking the heap length, an operation which takes constant time.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Lin, K. (n.d.). *Heaps and hashing*. CSE 373. https://courses.cs.washington.edu/courses/cse373/23wi/lessons/heaps-and-hashing