Assignment 4

Prashanna Acharya

Algorithms and Data Structures (MSCS-532-M80)

University of the Cumberlands

GitHub Link: <https://github.com/prashcodes10>

**Heap Sort**

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements efficiently. It works by first converting the input list into a max-heap, where the largest element is at the root. Then, it repeatedly swaps the root with the last element, removes it from the heap, and re-heapifies the remaining elements. This process continues until all elements are sorted. Heap Sort is known for its consistent O(n log n) time complexity and its in-place sorting nature, meaning it doesn't require extra memory, making it both efficient and memory-friendly.

A screenshot of a computer program

AI-generated content may be incorrect.

The above code implements the Heap Sort algorithm in Python to sort a list of numbers in ascending order. It uses a helper function called maintain\_heap to ensure that a given portion of the list maintains the max-heap property, where each parent node is greater than or equal to its children. First, the heap\_sort function builds a max-heap from the unsorted list by calling maintain\_heap on all non-leaf nodes. Then, it repeatedly swaps the root of the heap (the maximum element) with the last element of the heap, reduces the heap size, and calls maintain\_heap again to restore the heap structure. This process continues until the entire list is sorted. The sorted list is returned at the end. The if \_\_name\_\_ == "\_\_main\_\_": block demonstrates the algorithm using a sample list.

**Analysis of Implementation**

Time Complexity Analysis of Heap Sort

Heap Sort has a consistent time complexity of O(n log n) in the worst, average, and best cases. This uniformity arises from the algorithm’s two main phases: heap construction and heap extraction. During the first phase, the algorithm builds a max-heap from the unsorted list by calling the maintain\_heap function (heapify) on all non-leaf nodes in a bottom-up manner. Although it might appear that this would require O(n log n) operations, this phase runs in O(n) time because most of the heapify calls are on nodes near the bottom of the tree, which take significantly less time to process.

The second phase of Heap Sort involves removing the maximum element (root) and placing it at the end of the list, then calling maintain\_heap again to restore the heap structure. This step is repeated for each element in the array, and each heapify call takes up to O(log n) time because it may traverse from the root to the leaf of the heap (which has height log n). Since this operation is done n times, the total complexity for this phase is O(n log n). Therefore, while the heap-building step is O(n), it is dominated by the O(n log n) extraction phase, resulting in an overall time complexity of O(n log n) in all cases.

The reason Heap Sort doesn’t have a better best-case time than O(n log n) is because it always performs the same extraction and re-heapifying steps regardless of the input’s initial order, whether sorted, reversed, or random. Unlike algorithms like quicksort, which can have better or worse performance based on the pivot choice and input distribution, heap sort follows a fixed structure.

**Space Complexity and Overheads**

Heap Sort is an in-place sorting algorithm, meaning it doesn’t require any significant additional memory. The space complexity is O(1), and only a constant amount of extra space is used for temporary variable swaps and loop control, aside from the input array itself. There is no need for auxiliary arrays, which makes it more memory-efficient than algorithms like Merge Sort, which uses O(n) extra space.

As for overheads, the algorithm makes frequent swaps, which can be relatively expensive on systems with high memory latency or when sorting large data types. Heap Sort is also not a stable sort, meaning it doesn't guarantee the preservation of the relative order of equal elements. This can be a drawback in applications where stability is important.

In summary, Heap Sort is a reliable and space-efficient algorithm with a predictable O(n log n) time complexity across all input types. However, it may be outperformed in practice by other algorithms like quicksort or timsort (used in Python's built-in sort) due to lower constant factors and better cache performance.

**Runtime Comparisons of Heapsort, Quick Sort, and Merge Sort**

**Quick Sort**

The Quick Sort algorithm is a highly efficient, comparison-based sorting method that follows the divide-and-conquer strategy. It works by selecting a pivot element from the list and partitioning the remaining elements into two sub-lists: those less than the pivot and those greater than the pivot. These sublists are then recursively sorted using the same approach. Once the sublists are sorted, they are combined with the pivot to form the final sorted list. Quick Sort performs well on average with a time complexity of *O(n log n)*, but its worst-case time complexity is *O(n²)*, which can occur if poor pivot choices lead to highly unbalanced partitions. Despite that, its speed and in-place nature make it one of the most commonly used sorting algorithms. A screenshot of a computer program

AI-generated content may be incorrect.

**Merge Sort**

Likewise, Merge Sort is another reliable and efficient sorting algorithm based on the divide-and-conquer principle. It works by recursively dividing the input list into two halves until each sublist contains only one element, which is naturally sorted. Then, it repeatedly merges these sorted sublists back together in the correct order to produce a single sorted list. Unlike Quick Sort, Merge Sort guarantees a consistent time complexity of *O(n log n)* in the best, worst, and average cases, making it very predictable in performance. However, it requires additional space for the merging process, so its space complexity is *O(n)*. Because it maintains the original order of equal elements, Merge Sort is also a stable sorting algorithm, which is useful in scenarios where order matters. A screenshot of a computer program

AI-generated content may be incorrect.

**Code Snippet to Compare these three Algorithms**

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer screen

AI-generated content may be incorrect.

This script showcases the performance of Heap Sort, Merge Sort, and Quick Sort on lists of varying sizes and distributions (sorted, reverse, and random). It generates input data, times how long each algorithm takes to sort the lists and stores the results. After running the tests, it plots the timing results using line graphs to visually compare algorithm performance. The code runs all benchmarks when executed and displays both the timing output and performance plots.

**Results and Graphs**

A graph of a graph

AI-generated content may be incorrect.A graph of a graph with different colored lines

AI-generated content may be incorrect.A graph of different colored lines

AI-generated content may be incorrect.

**Analysis:**

The three graphs provided compare the running times of Heap Sort, Merge Sort, and Quick Sort on datasets of varying order: Sorted Data, Reverse Data, and Random Data. These analyses helps in understanding the performance of each sorting algorithm in different data scenarios.

In the Sorted Data scenario, all three algorithms show an increase in running time as the input size grows. Heap Sort's time increases steadily in a linear fashion, suggesting that its performance is relatively unaffected by the initial order of the data. Merge Sort behaves similarly, with a slight edge in performance compared to Heap Sort, as it shows a slightly slower increase in time. However, Quick Sort stands out as the fastest among the three. Its time complexity is , which generally leads to faster performance than both heap sort and merge sort, especially on sorted data.

When applied to Reverse Data, Heap Sort and Merge Sort exhibit similar behaviors to those seen with sorted data, with both showing a linear increase in running time. However, Quick Sort’s performance degrades in this scenario. This is because Quick Sort may encounter its worst-case time complexity of if it consistently picks poor pivot elements (which can happen with reverse-sorted data), leading to a noticeable increase in time. Therefore, while Merge Sort and Heap Sort maintain stable and predictable performance, Quick Sort's efficiency is heavily impacted by the reverse order of the input.

In the Random Data scenario, the performance of all three algorithms follows a similar pattern to that seen in the sorted data case, but with some variations. Heap Sort’s running time increases steadily, slightly higher than when the data is sorted but still showing a linear growth pattern. Merge Sort performs similarly, exhibiting a stable linear increase in time, unaffected by the data's order. Quick Sort, on the other hand, performs the best in this scenario. As expected, Quick Sort's time complexity follows an pattern, making it the most efficient algorithm for random data. This is because Quick Sort typically avoids the worst-case scenario when working with random datasets.

In summary, Heap Sort and Merge Sort show relatively consistent performance across all three types of data, both demonstrating linear growth in time complexity. While Heap Sort is slightly faster than Merge Sort on sorted and reverse data, both algorithms are stable and predictable, regardless of the data order. Quick Sort, however, excels on random data, displaying its characteristic performance. However, its performance degrades significantly when working with reverse-sorted data due to the worst-case time complexity. The overall takeaway is that Quick Sort is ideal when working with random or mostly sorted data, Merge Sort is a reliable and stable choice for all input types, and Heap Sort, while consistent, is generally slower than Quick Sort in random or sorted data scenarios.

**Priority Queue Implementation**

A binary heap is utilized for implementing a priority queue due to its efficiency and simplicity. It can be represented using an array (or list) and offers optimal performance for the key operations of a priority queue. The binary heap allows for fast insertion and removal, which is essential for managing tasks according to their priority.

In this code, the Task class stores key details for each task, including a unique ID, priority level (with higher priority processed first), arrival time, and deadline. These attributes enable effective management and scheduling of tasks with varying urgency, arrival times, and deadlines. A max heap was chosen because tasks with higher priority must be processed first. This ensures the highest-priority task is always at first, enabling efficient extraction.

A screen shot of a computer program

AI-generated content may be incorrect.A computer screen shot of a program code

AI-generated content may be incorrect.

**Insert Operation (𝑂(log n))**  
When a task is inserted, it is first placed at the end of the heap. To maintain the heap property, a heapify-up operation is performed, which may move the task up toward the root. Since this process traverses the height of the heap, it takes logarithmic time in the worst case.

**Extract Max Operation (𝑂(log n))**  
To extract the highest-priority task, the root node is replaced with the last node in the heap, followed by a heapify-down operation. This re-establishes the max-heap structure by comparing and swapping nodes down the tree, which also takes logarithmic time in the worst case.

**Insert Key Operation (𝑂(log n))**  
When a task’s priority is increased, the key is updated, and the heapify-up operation is used to restore the max-heap property. As the task may need to move from a leaf node up to the root, the time complexity remains logarithmic.

**Empty Operation: 𝑶(𝟏)**

It simply checks the heap length, which takes constant time.

**References**

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

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 Java* (6th ed.). Wiley.