Brendan Yeong

Student ID: 005025797

MSCS 532

Assignment 4

Heapsort Implementation and Analysis

Analysis of Implementation

Time Complexity:

Looking at the algorithm, to build the max-heap from an unordered array, it takes . As each node is heapified once, and each operation takes time, but with nodes closer to the leaves have smaller subtrees to heapify, reducing the total number of operations. This results in an aggregate cost of .

After the max-heap is built, we repeatedly extract the largest element and re-heapify the remaining elements. The heapify operation takes time because it maintains the heap property as it compares and possibly swap nodes along the height of the tree. So, the total time complexity is as we perform number of times.

Since both steps are independent of the input arrangement, the total complexity of Heapsort is always , making it stable across all cases.

* Best-case: Elements are sorted, the algorithm will still build the heap and re-heapify elements.
* Average-case: Elements are in random order; the algorithm will build the heap and re-heapify elements making it .
* Worst-case: Elements are in reverse order; the algorithm will build the heap by repeating extraction remains and re-heapify elements making it .

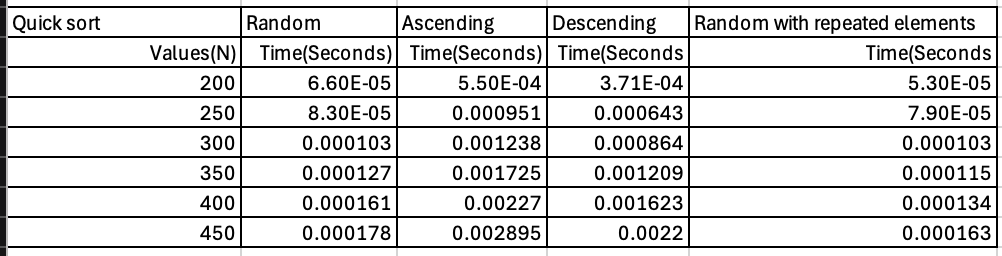
Space complexity

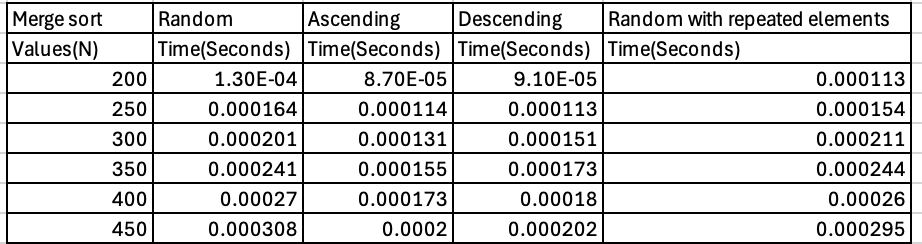
Heapsort is an in-place sorting algorithm, so it requires no additional memory to store data when it sorts the input array input. So, this give the space complexity of , making it more space efficient compared to merge sort.

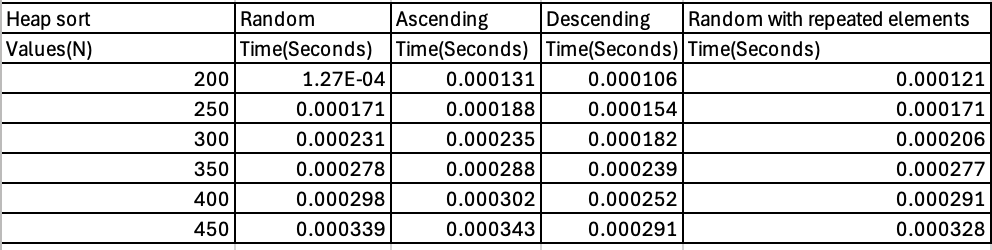
Additional overheads

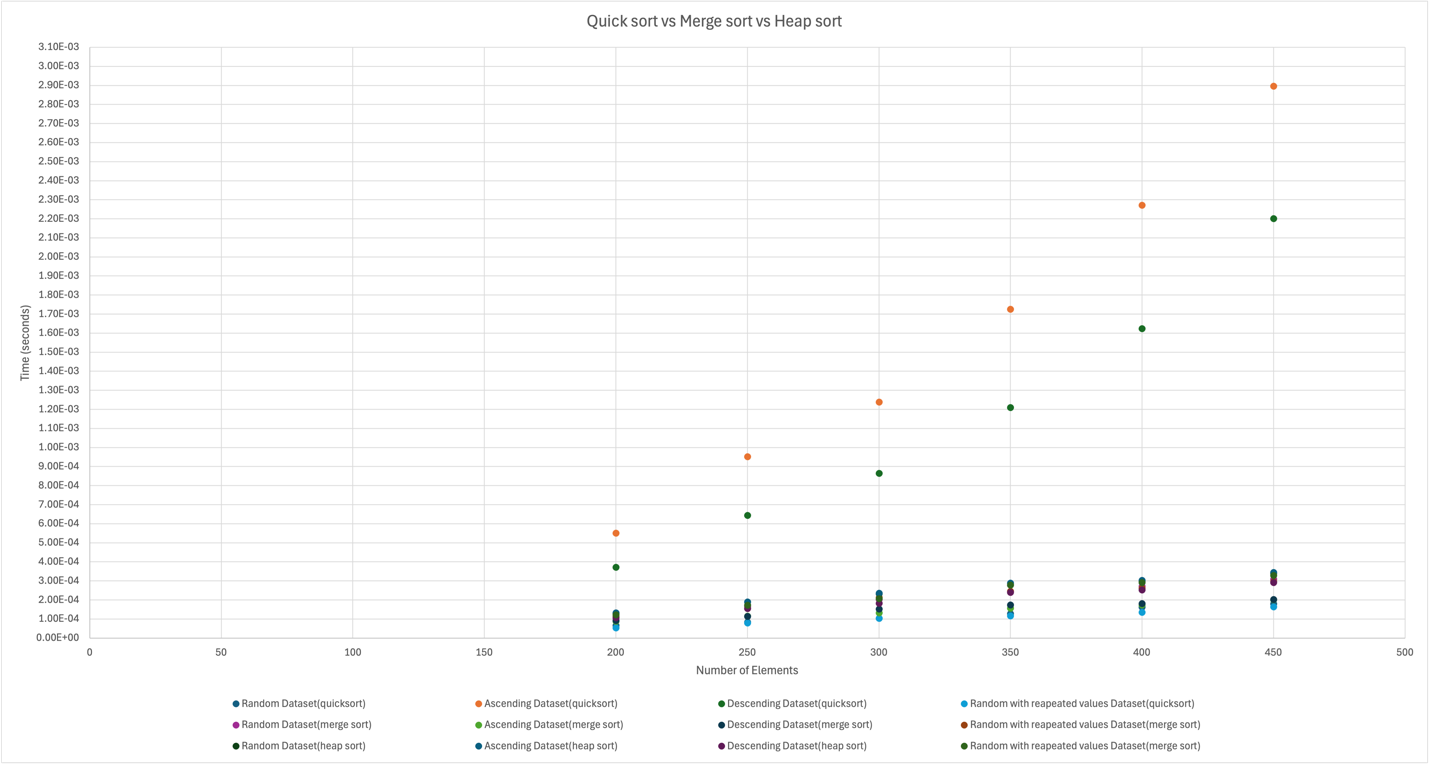
The overhead is due to the recursive calls to the heapify operations, but stack overhead do not require extra space beyond a constant amount for the variables. This makes it a highly efficient algorithm with consistent performance and low memory usage.

Comparison









These are the results from the empirical comparison of Quicksort, and Merge Sort, and Heap Sort for different input sizes and distributions have also been displayed.

Discussion based on the data:

* Quicksort: This was generally faster than heapsort on random data but when it is time to sort on sorted data or data that was assembled in descending order it suffer in the worst case .
* Merge Sort: This had consistent performance across all input types, which is expected since it always divides the array in half and merges it in time but when it comes to smaller datasets it is slower.
* Heapsort: This consistently performed well across all input sizes and distributions, with time complexities roughly aligning with its theoretical performance

Priority Queue Implementation and Applications

Data structure

For the data structure I have decided to implement the priority queue using a binary heap. Since binary heap provides efficient insertion and extraction of the highest or lowest priority element, making it ideal for a priority queue.

Justification: priority list using a binary heap can be efficiently implemented using a list, where the parent-child relationship in the heap can be derived using simple index arithmetic. For which at node, the left node is at , right node at and the parent is at . Using the heap structure the efficiency in using this is that the insertion and extraction operations have a time complexity of and in python using a list allows for dynamic resizing and can store the heap elements contiguously in memory.

Design a Task class

I will choose to implement the min-heap suitable for scheduling algorithms where tasks with the smallest priority number are processed first. Which means lower priority value will give it a higher priority.

Analysis

insert(task):

Time complexity:

Inserting into a heap by inserting at the end of the list takes constant time and then potentially "bubbling up" the element to maintain the heap property and this process take .

extract\_min():

Time complexity: Remove and return the task with the lowest priority from the heap. After removing the root, move the last element to the root and then heapify down to restore the heap property. The extraction requires constant time to remove the root, but the heapify down operation may take time to restore the heap property.

increase\_key(task, new\_priority) / decrease\_key(task, new\_priority):

Time complexity: If the new priority is smaller (decreasing priority in a min-heap), we heapify up. If the new priority is larger (increasing priority), we heapify down. Both heapify up and heapify down operations take

is\_empty():

Time complexity: This operation takes constant time.

Reference

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms

(4th ed.). Random House Publishing Services.

<https://reader2.yuzu.com/books/9780262367509>

GeeksforGeeks. (2023, January 10). *Heap and priority queue using heapq module in Python*. https://www.geeksforgeeks.org/heap-and-priority-queue-using-heapq-module-in-python/

*Heap sort algorithm*. Programiz. (n.d.). https://www.programiz.com/dsa/heap-sort