Analysis of Algorithms

Project 2 Report

Kerem Er

1. Implementation

1.1. Implementation of HeapSort

HeapSort turns the array into a max heap and then repeatedly removes the largest element and heapifies the remaining heap until the whole array is sorted.

1. Build a Max Heap: First, you rearrange the array into a max heap. A max heap is a binary tree where the parent node is always larger than its children. You do this for the entire array.
2. Sort the Array: Once you have the max heap, the largest element is at the root (beginning of the array). You swap this element with the last element in the array, then reduce the heap size by one (ignoring the last element which is now sorted).
3. Heapify the Remaining Heap: With the largest element removed, the heap property might be violated. So, you "heapify" the root again (make sure the new root is larger than its children) with the reduced heap size. This brings the next largest element to the root.
4. Repeat: Repeat the process of removing the root, reducing the heap size, and heapifying the reduced heap until all elements are sorted.

	Population1	**Population2**	**Population3**	**Population4**
HeapSort	125126100	128784200	120072500	130210600

Table 1.1: Comparison of different csv files on input data.

1.1.1. Elapsed Time Comparison

- Heapsort vs Quicksort (Last Element as Pivot): Heapsort is consistently faster than Quicksort when the last element is used as a pivot. This suggests that Heapsort may be more efficient in handling worst-case scenarios of Quicksort.
- Heapsort vs Quicksort (Random Element as Pivot): Similar trends are observed with random elements as pivots. Heapsort again shows better performance, indicating its robustness in handling various data distributions.
- Heapsort vs Quicksort (Median of 3 as Pivot): Quicksort with the median of three elements as the pivot significantly outperforms Heapsort, especially for larger populations. This indicates the efficiency of Quicksort in more balanced cases.

1.1.2. Comparison Efficiency in Terms of Comparisons

- Heapsort: Performs more consistently across different data distributions but generally involves more comparisons than Quicksort in balanced cases.
- Quicksort: Number of comparisons can vary greatly depending on the pivot choice. The median of three strategy significantly reduces the number of comparisons, leading to better performance.

1.1.3. Strengths and Weaknesses

- Heapsort:
  - Strengths: More consistent performance, better in worst-case scenarios of Quicksort.
  - Weaknesses: Generally slower than a well-optimized Quicksort in average and best cases.
  - Best Scenarios: Preferable in applications where worst-case performance is critical.
- Quicksort:
  - Strengths: Highly efficient with a good pivot choice, especially in average and best-case scenarios.
  - Weaknesses: Can degrade to O(n$\hat{2}$) in worst-case scenarios (like with the last element as a pivot).
  - Best Scenarios: Ideal for large datasets where average performance is key, particularly with a balanced pivot strategy.

1.1.4. Conclusion

In conclusion, Heapsort offers more predictability against Quicksort's worst-case scenarios, however if Quicksort is used with a good pivoting strategy it will provide a better Average-Case performance. The choice between the two should be influenced by the nature of the data and the performance requirements of the application.