Lazy sort analysis

1. **Implementation Analysis:**

Merge sort:

I implemented threading by assigning each half of the array at every step a new thread. Pros and cons of this approach are:

Pros:

a. Efficient utilization of resources
b. Scalability: The algorithm scales well as the input size increases because more threads can be created to handle increasingly smaller sections in parallel.

Cons:

a. High overhead for small data sets. The overhead for handling threads introduces more overhead than benefit from parallelization.
b. Increased memory usage.
c. Thread limitations. Some systems have a limit on the number of threads, which might cause problems for large data sets.

Count sort:

I implemented my distributed count sort by taking inspiration from this site. https://stackoverflow.com/questions/39903181/can-you-do-a-parallel-counting-sort-in-on-p-time

Pros and cons of this approach are:

Pros:

a. Parallel efficiency. For large datasets, dividing the array into p parts significantly reduces runtime.
b. Scalability. This approach works well with large datasets, because of the decrease of time by a factor p.

Cons:

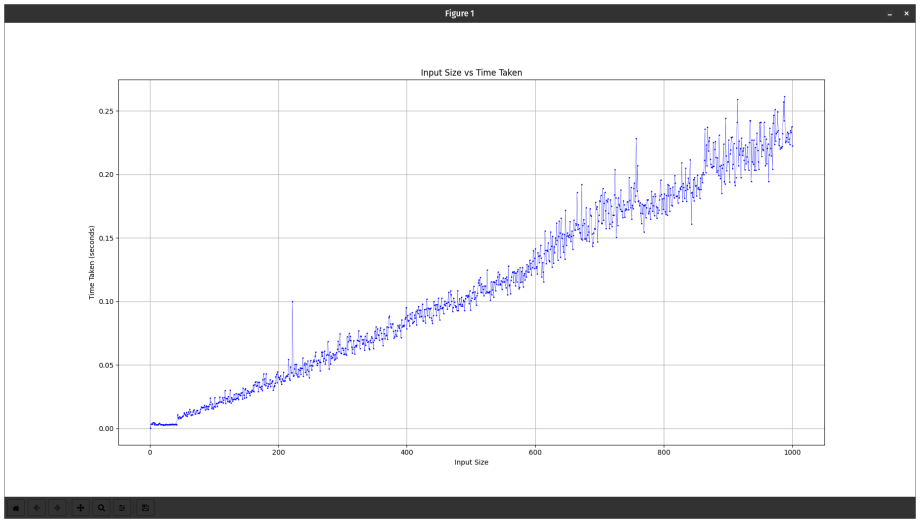
a. High memory overhead.
b. Limited benefit for small datasets.
c. Assumes uniform distribution of work among p parts of array.

Assumption:

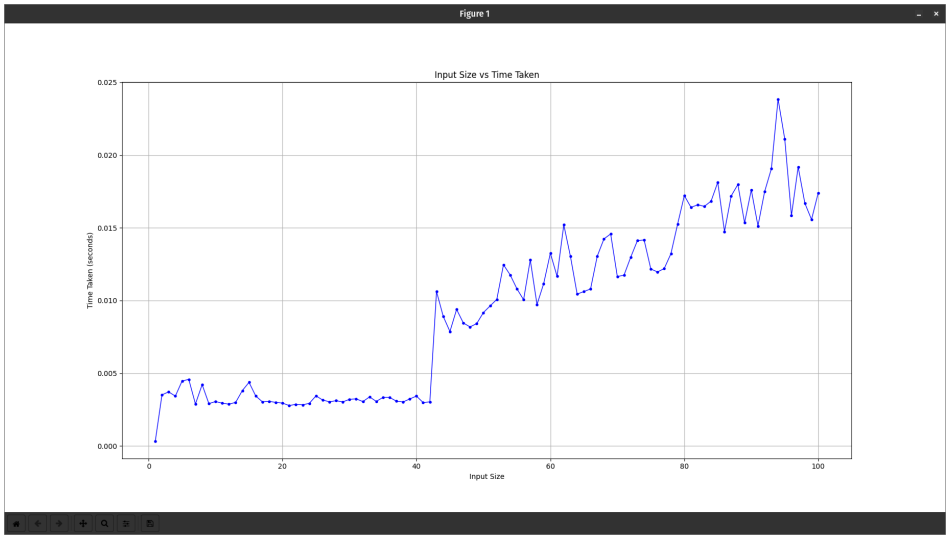
In count sort for names, I am taking only the first four characters for hashing. Because taking more than that makes the hash value too big.

Lazy sort analysis

2. Time analysis:



Large dataset (n = 1 to 1000)



Small dataset (n=1 to 100)

Observation: From n = 42 to n = 43 there is a steep increase in the time because count sort is much more efficient than merge sort.

Lazy sort analysis