———————————— Base Information ————————————

Name: Sergio Ley Languren

Language Used: Python

Platform: MacOS

CPU: Apple M1 Pro

RAM: 16gb

———————————— Methods ————————————

- **Selection Sort**:
  - We iterate through a given array k for each index i, always assuming in each iteration that the current index holds the smallest value, which we will refer to as the minimum index. We then perform an inner iteration, iterating for each index j starting from the current index to the last index. We check if the element at index j of k is less than the element at the minimum index; if so, index j becomes the minimum index. Once the inner iteration is complete, we will swap the elements at indices i and the minimum index. Repeating this process will sort the array. This algorithm is performed in-place and requires no additional arrays.
- **Mergesort**:
  - Mergesort is divided into two functions:
    - "merge", a function that is called by the primary function to merge two arrays:
      - We first initialize indices i and j and an empty array k. We are provided with left subarray x and right subarray y. We run a loop that runs while both i is less than the length of x and j is less than the length of y, meaning that if one of the conditions fails, the loop will stop. Within the loop, we check if the element of x found in index i is less than the element of y found in index j; if true, we append the element of x into array k and add a value of 1 to i, else, we append the element of y into k. Afterwards, we add the rest of the elements of array x into array k, starting at index i, followed by

the same procedure with array y and index j. We will then return array k.

- "merge_sort", a function that divides the given array into even halves and sorts each half:
    - Since this function is recursive with the given array k, we will provide the function with the following two base cases:
        - if len(k) equals 0 or len(k) equals 1:
            - return k as if the length of k is 0 or 1, the array counts as sorted
        - if len(k) equals 2:
            - We will proceed to sort array k. We check if the first index of k is greater than the second index, and if so, we swap the first and second index and return the sorted array.

      If both base cases fail, we assume array k has not been split enough to sort it properly. We calculate the midpoint of the array, rounding the result down to the nearest integer. We then split k into a left and right subarray using the midpoint, where the left takes the left half and the right takes the right half. We then proceed to set both subarrays by recursively calling "merge_sort," giving the respective subarrays as the argument. Once the function fully returns the final subarrays, we will return the result of the "merge" function, providing the left and right subarrays as the arguments for that function.

- **Data Generation**:
    - We have decided to use the following seed array: "sizes = [1000, 5000, 10000, 50000]" to collect our data. We performed 5 trials for both algorithms. Trials 1-4 got their seed in respective order, for example: "size3 = sizes[2] # seed for trial 3". The fifth trial chooses its seed randomly. To create arrays in each trial, the following code is used: "arr = [random.randint(0, <seed>) for _ in range(<seed>)]". This ensures that an array of random values of the appropriate

size is created for testing in each separate trial. To run a trial, we create an array that will hold the resulting runtime (in seconds) of the chosen algorithm when it runs each of the 5 times the trial executes the algorithm. To conclude each trial, an average of the stored times is chosen to be the time of the specific trial. Once all trials of both algorithms were run, we determined the following from both collected times: the median, mean, and standard deviation.

Results

Table 1.0

Seeds	1000	5000	10000	50000	Random
Selection Sort	0.0305723417957779	0.5129335083940532	2.013156416802667	49.41551329999929	0.4913938500045333
Mergesort	0.001497374993050471	0.01526450839592144	0.02016760000260547	0.12073962499853223	0.12928520839195698

Algorithm Runtimes in seconds

Figure 1.1

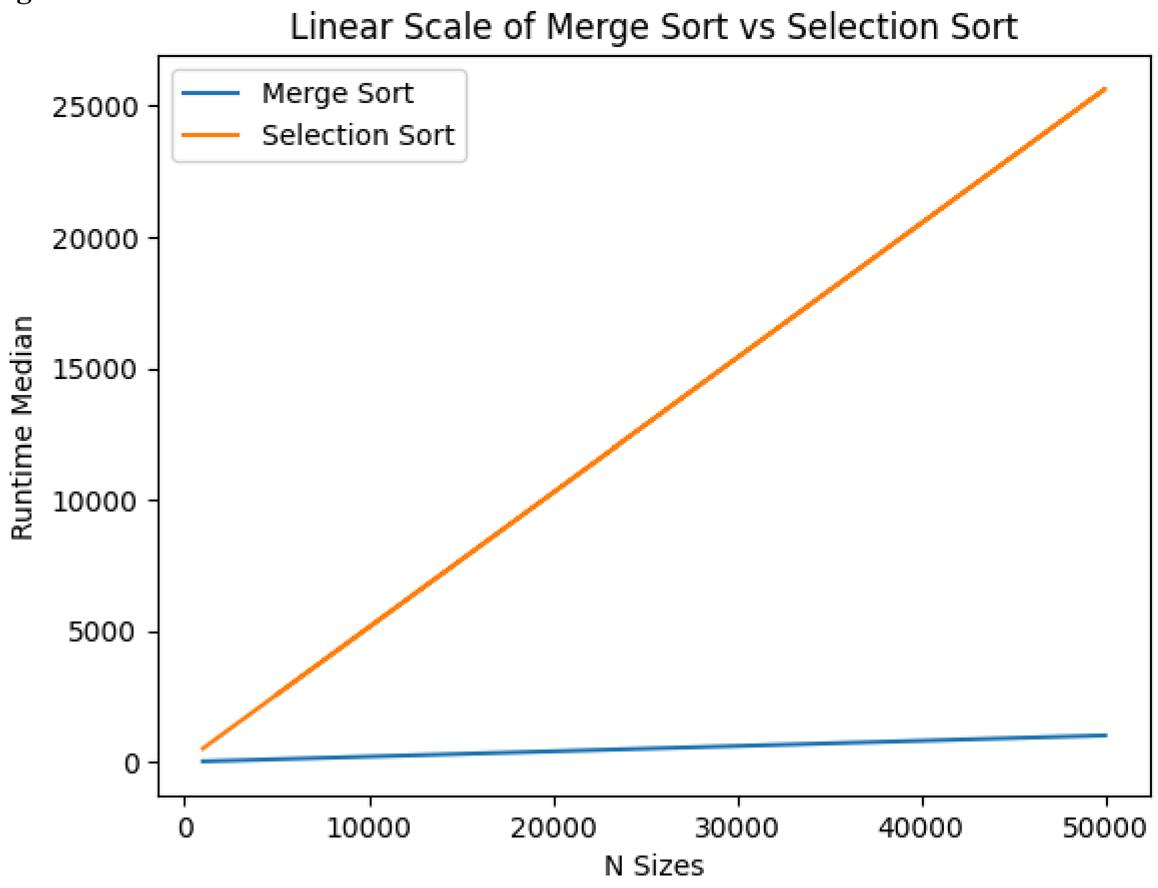
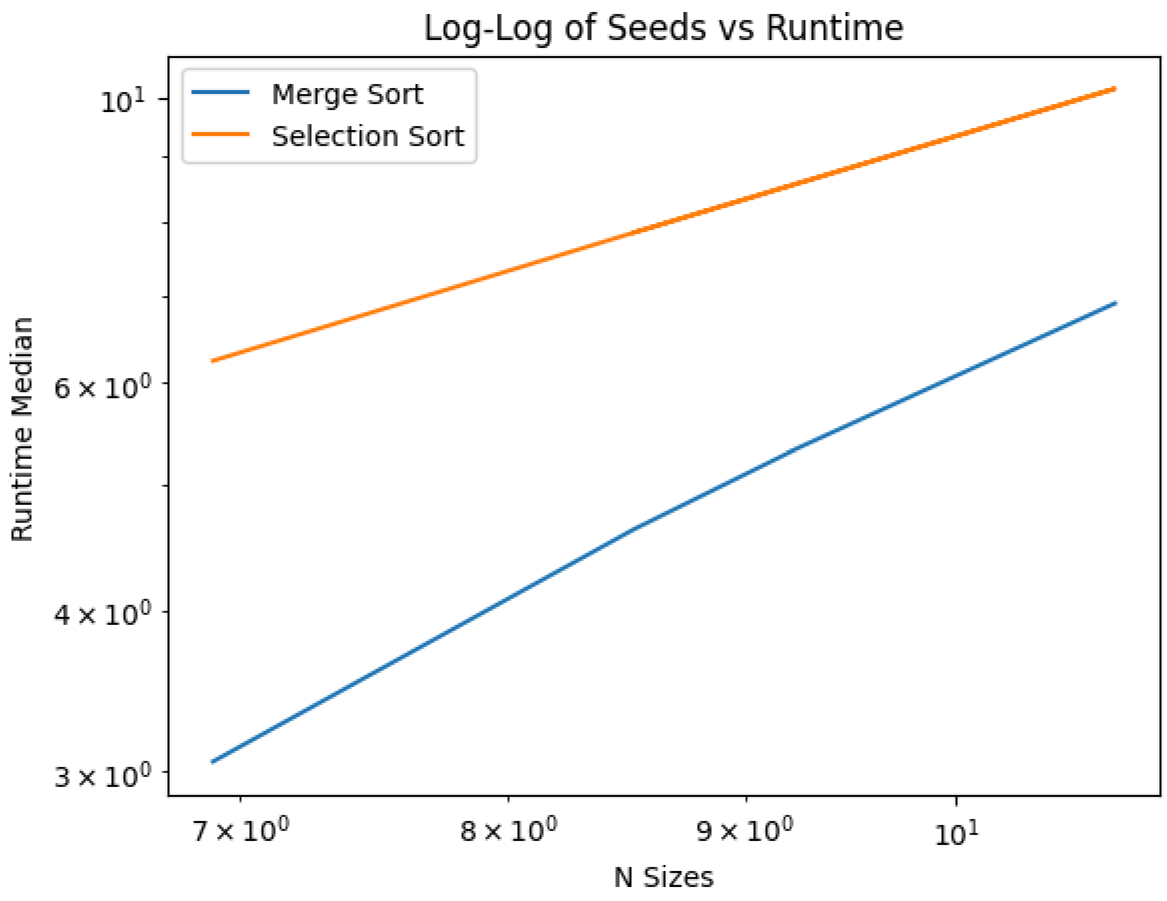


Figure 1.2



Reflection

The curves from Figure 1.2 do align with the theoretical predictions of $O(n^2)$ vs O(n log n) and thus, do not diverge. When comparing constant factors, merge sort tends to have a higher constant factor due to its recursive behavior and array copying compared to selection sort. It is also important to note that modern CPUs like Apple's M1 Pro use cache to speed up memory access. Thus, due to algorithms with good locality benefiting more from this, merge sort's recursive nature would see more cache misses than selection sort. According to our results in Table 1.0, we saw mergesort starting to pull away at Trial #2 when the seed was set to 5000, with a runtime of 0.015 seconds compared to the 0.51 seconds runtime made by selection sort. Selection sort after Trial #2 increased its runtime exponentially compared to mergesort. We could improve our experimental rigor by running more than the 5 trials with a more expansive set of seeds to test from smaller seeds to larger ones.

——————————— Appendix ———————————

1. Github Link: https://github.com/SLey3/CS-351-Assignments/tree/master
2. Reproduction Steps:
    2.1. Run the following script: ‘pip install -r requirements.txt && cd “Assignment 1”’

    2.2. To run algorithm timing: “python prob.py --seed <insert seed number>” (if you simply provide no arguments to prob.py, the default seed will be 100)