|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nanoseconds | N = 10^2 | N = 10^3 | N = 10^4 | N = 10^5 | N = 10^6 |
| Smooth set | 28745300 | 6590200 | 54833600 | 432136100 | 4652306400 |
| Sparse set | 10859500 | 7348600 | 28223000 | 86238700 | 1624891800 |

My first method developed, characterized by its use of heapsort and a self-adjusting BST, deviated from expected runtimes somewhat inconsistently. Paired with Heap sort, we would expect the runtimes to follow in accordance with O(*nlogn + nlogn*) = O(*2nlogn*) = O(*nlogn*). To obtain values ready for comparison to the pure expected runtime of *nlogn,* I’ve assumed the formula *nlogn* to denote the expected runtime in nanoseconds, then taking the true runtime in nanoseconds and dividing it by this ideal comparator. Such will allow us to analyze the factors by which the algorithm runs compared to its ideal runtime and deduce trends in these runtimes. Compared to its pure expected runtime of O(*nlogn*), the self-balancing Binary Search Tree and heap sort sweeping line algorithm runs at times which exceed that expectation by factors of 8172x, 368x, 106x, 25x, and 40x, all respective to the ascending N. The first runtime where N=100 is consistent with its outlandish runtimes upon multiple reruns and on both algorithms, and so I’ve chosen to disregard it for much of the further analysis and attribute it to a constant overhead runtime that comes with running the Java Virtual Machine.

Secondly, one might notice the runtimes to not converge to a consistent factor, at least not given this set of data inputs. To check for stabilization, I ran the algorithm at two larger data sets—being N = 10^7 and N=10^8, and received runtimes which exceeded the ideal by factors of 84x and 110x. To see these factors increase further is not necessarily terrible so long as the factors remain within a consistent range (as variability is to be expected), though it is a possible indication of failure to adhere to the expected *nlogn* runtime. To determine whether the algorithm is running worse than expected, we should compare it to the next step up in expected runtimes, O(n^2). In doing so, we receive relative factors of 1085x, 7.3x, 0.28x, .009x, .001x, .0003x, and .00005x; again respective to ascending N and inclusive of N=10^7 and N=10^8. Given the consistent departure from quadratic runtimes, we can pointedly assume the algorithm to NOT adhere to O(n^2) as a worst case.

To then confirm whether or not our runtimes are adhering to some variation O(*nlogn*) upper-bound, we must check if the factors diverge from O(*logn*) runtimes in a similar fashion to the O(*n^2*) runtimes. In doing this with respect to ascending N, we receive relative factors of 1,634,517x; 737,383x; 2,123,992x; 5,192,087x; 81,523,528x; 1,685,863,518x; and 22,056,625,179x. Here, we see the trend is similar, wherein between each step of N values (save from N=10^2 to 10^3) there is an increase of at least double the previous factor. Therefore we can rule out a possible adherence to an O(*logn*) runtimes. Because of the stricter adherence to the O(nlogn) runtimes in both relation and magnitude, we can assume the algorithm to run in accordance with the expected O(nlogn).

The second algorithm, characterized by its use of Radix sort and the Van Emde Boas tree, varied similarly between runtimes as the previous did, and so we will be employing the same method to deduce to which function the runtimes are adhering to. Ln this case, worst-case runtime of the algorithm utilizing this structure and sort will be *O(nlog(log(u)) + (d \* (n + b)))*, where *u* is the universe of the vEB Tree, *d* is the number of digits in the highest value number being sorted, and *b* indicates the base of the numbers utilized--in every case 10--therefore making the runtime *O(nlog(log(u)) + (d \* (n + 10)))*. In the previous analysis, I simplified the worst case by merging n values and removing lower factor terms, however given the increase in variables here with the introduction of *u and d,* I’ve opted to calculate runtime factors based on this larger formula as to obtain the most accurate results.

Compared to the expected runtime of *O(nlog(log(u)) + (d \* (n + 10)))*, the algorithm runs at times which relate by factors of 39377x, 819x, 548x, 392x, 387x, respective to N as N increases. To see if the factors converge on a value somewhere within the three-hundreds, I ran the algorithm at N values of N=10^7 and N=10^8 here as well, obtaining factors of 399x and 440x. It is a similar trend as we saw in the previous analysis, where the initial factors declined, only to bump upwards again in the final trials. Unlike the previous algorithm, however, we see factors consistently higher in magnitude, although that doesn’t matter as much as the relational increase between runtime factors, which appear almost identical to the self-balancing binary search tree.

To ensure adherence to the *O(nlog(log(u)) + (d \* (n + 10)))* runtime, lets compare to the steps above and below in terms of runtime equations. Given that *u* is restricted within our runtimes to a maximum of 4,294,967,296 (i.e. making *log(log(u))* = 5 at the most), our runtimes should provide a consistent departure in runtime factors as N increases when compared to O(*nlogn*). In doing this, we compute factors of this algorithm as follows: 43265x, 661x, 412x, 260x, 233x, 111x, 107x. The factors most noticeably converge here, hinting at its possible adherence to *nlogn* rather than the expected *O(nlog(log(u)) + (d \* (n + 10))).*

Given that we’ve stumbled upon a better-suited upper bound, we must test if the equation adheres more so to O(n^2). In doing this, we receive factors of 2874.53x, 6.5x, .5x, .04x, .004x, .0005x, .00005x. We can rest easy knowing that my algorithm diverges from O(n^2) as was expected. Therefore, we can pointedly assume the algorithm to adhere to a worst case upper bound of O(*nlogn*), despite the original conjecture.