|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nanoseconds | N = 10^2 | N = 10^3 | N = 10^4 | N = 10^5 | N = 10^6 |
| Smooth set | 2031300 | 1825600 | 9614300 | 48806200 | 1250699000 |
| Sparse set | 1907300 | 10763000 | 21388600 | 67549600 | 1696765000 |

My first method developed, characterized by its use of heapsort and a self-adjusting BST, deviated from expected runtimes somewhat inconsistently. Compared to its pure expected runtime of O(*nlogn*) (assuming that the formula *n\*logn* denotes the expected runtime in nanoseconds), the algorithm runs at times which exceed that expectation by factors of 4141x, 92x, 180x, 44.92x, and 85x, all respective to n in ascending order. The first runtime with n=100 is consistent with its outlandish runtimes upon multiple reruns. As such, I’ve chosen to disregard it for much of the further analysis. Were the remaining runtimes to follow a consistent factor between each other, the algorithm written would reliably demonstrate itself as running at consistent factors of *nlogn.* *T*he factors of deviation, however, deviate amongst themselves by factors consistently between ½ and 2x that of a baseline factor of ~90.

Furthermore, compared to the runtime of O(n^2), however, we see runtimes of 190x, 10x, .21x, .06x, and .001x respective to N. Because these factors decrease between each other by factors of nearly 10, we can reliably assume the algorithm to NOT follow this worst-case upper bound. Compared further to O(n), we see the runtimes relate by factors of 19703x, 10763x, 2138x, 675x, and 1696x. These points seemed to vary too wildly to be considered, and given the impossibility of sorting elements below *nlogn*, O(n) can be eliminated. To obtain more data points, I further ran the algorithm at five 100 thousand milestones between N = 10^5 and 10^6 (being n=200k, 300k, 400k, 500k, 600k) and obtained relational runtimes of factors 47x, 64x, 65x, 70x, and 73x the expected *nlogn* values respectively. Because of the stricter and more consistent relation to the O(nlogn) runtimes, we can assume the algorithm to run in accordance with the expected O(nlogn).

The second method, characterized by its use of Radix sort and the vEB tree, varied between run times, something which was expected. I would like to note my confusion in the implementation of this section of the project, and the inefficient work arounds that I utilized. I assume the primary motivating factor in utilizing a vEB tree to store the sweepline status is the use of tree successor and predecessor, used to access the lines above and below the current line being analyzed. The assignment stated to use the y-coordinate of the left endpoint as the key and the Line object as the value in the structure. However, this would lead to successor and predecessor offering the lines with the most adjacent left-endpoint y-coordinate, Lines which are likely to be but not always the line directly above or below the current comparative. Furthermore, the key is used in the structure as the index, and given the variation in coordinates extending to 10\*n, the vEB tree would require more space than is necessary (an example being when n=10^2: the vEB must utilize a universe of 65536 rather 256 due to y-coordinates extending to a maximum of 10^3 – 1). Thus, in order to make the algorithm properly function, I implemented a few new functions to the structure which would inefficiently but effectively perform insertion sorts on the vEB structure, judging comparisons in order with the cross products of each line. This would promptly provide a worst-case runtime of O(n^2log(log(u)), which will be used alongside the assignment’s true worst-case runtime of O(nlog(log(u))).

Compared to the expected runtime of O(n^2log(log(u)), the algorithm runs at times which relate by factors of 80x, .58x, .031x, .0012x, .00032x, all respective to N as N increases. All factors miraculously decrease from the exponential expectation exponentially, to which we could rule out this being the de facto worst-case runtime. Furthermore, compared to the other runtime of O(nlog(log(u))), the algorithm runs at times which differ by factors of 8304x, 581x, 306x, 127x, 326x. Again, to obtain more data points, I ran the algorithm at five more milestones of N, being N=200k, 300k, 400k, 500k, and 600k. These compared to the O(nlog(log(u))) bounds by factors of 200x, 255x, 255x, 257x, and 292x, respectively. Following this same trend as the previous algorithm discussed, we might conclude this algorithm runs in accordance with the O(nlog(log(u)) runtime, despite the egregious errors in my implementation.