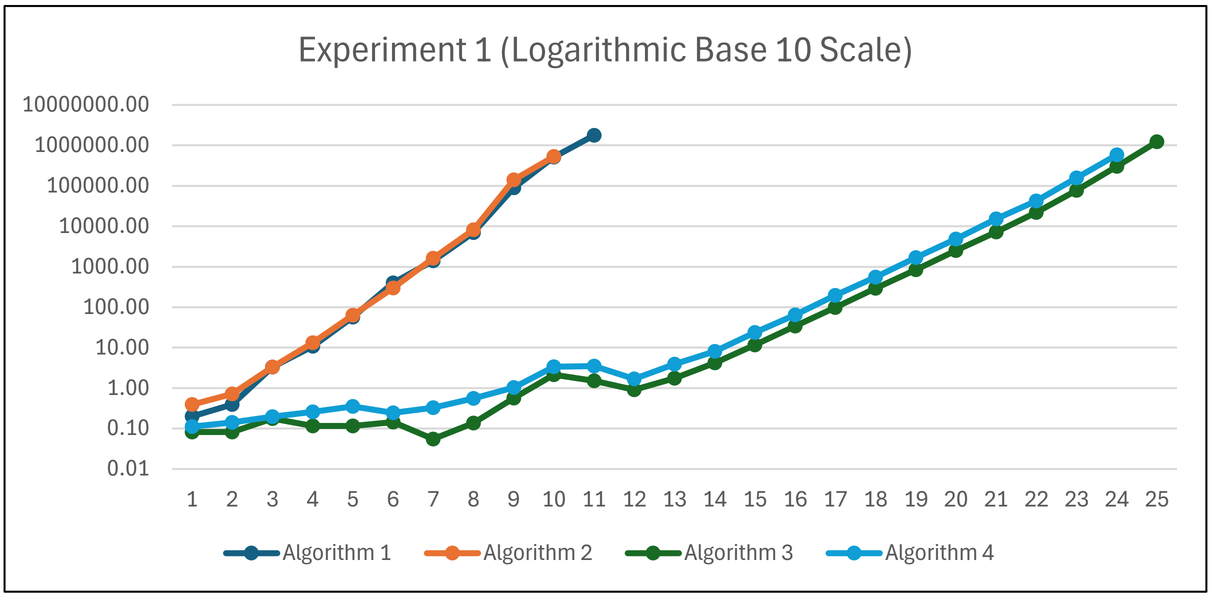
Experimental Evaluation:

- **OS**: Microsoft Windows 11 Home, version 10.0.26100 N/A Build 26100

- **CPU**: AMD Ryzen 7 5700U with Radeon Graphics, 1801 Mhz, 8 Core(s), 16 Logical Processor(s)
- **RAM**: 16GB

Experiment 1:

For this experiment, when flag = 1, the experimentRun() function generated a random instance of sets using N=2000, M=100 and C=1-10 (randomly). In doing so, the probability of having a Hitting Set at K=30 would be around 1.995*10^(-19) thus almost 0, therefore the algorithms would run out of time before reaching that number. That is, because of the huge number range (N) paired with a normal number of sets (M) with very few numbers each set (C), making 100 sets up to size 10 having 30 numbers in a hitting set from a 2000 number range incredibly hard. Then, the runAlgorithmsExperimental function run each function starting at K=1, 3 times and calculate the average time in milliseconds each algorithm took to complete. The function would finish only if all 4 algorithms took more than 1 hour to complete for some K. The results that were computed follow.



Results/ Conclusions for Experiment 1 and Predictions:

The results, excluding the one for N and algorithm 2, confirmed my initial predictions. In all cases, no hitting set was found by any algorithm from K = 1 to K = 25, showing that my choice of parameters was correct. According to the Logarithmic scale plot with all the runtimes of each algorithm, information about time, scalability (how well each algorithm handles increasing K without causing more runtime cost), efficiency and performance can be concluded:

Algorithm 3 was the fastest algorithm due to the choice of the smallest set without putting any extra overhead sorting its elements, handing up to K = 25 within the hour. It had the exponentially slowest increase in its run time, as shown by the logarithmic plot above, showing speed and scalability, as it could handle the most K without draining the CPU with recursions, especially during the smaller K runs (K = 1 – 8) where due to the slow exponential growth of the runtimes and the aimed choice of the smallest sets, the total computation time was almost the same. There were a few outliers from K = 7 and 10 iterations, where in some levels of the recursion the algorithm picked sets from the current smallest sized sets, (there were multiple sets with the smallest size) that happened to include numbers with either high or low frequencies across all sets, leading to more or less sets being removed and thus finishing faster or slower. Overall, algorithm 3 showed the best speed, performance and high growth rate, thus being the most efficient algorithm during this experiment.

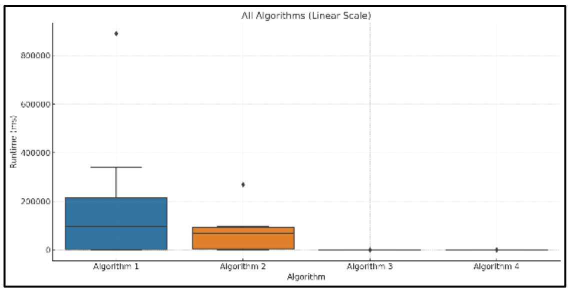
Algorithm 4 was performed consistently behind algorithm 3, reaching up to K = 24 under the hour mark timeout. That is due to the slight overhead of computing the critical order of its elements and still falls under my predictions. Because of the way the instance was generated, whatever overhead computation would not lead to a faster solution, as there was no hitting set, and that is another reason why algorithm 4 was a few ways lengthier than algorithm 3. Despite this, algorithm 4 displayed high speed, being a little slower because of the overhead, similar with algorithm 3's scalability, as both made similar choices regarding the set selections, showing that its small overhead had little to none effect on its performance. The few outlier from K = 8 and 9 iterations are explained the same way as algorithm 3's. Overall, algorithm 4 was short behind 3 with high speed and scalability, thus showing efficiency, as predicted.

Algorithm 1 and 2 performed significantly worse, as anticipated, stopping at K = 11 and 10 respectively. That is due to the random selection of the sets at each recursion as predicted and explained. Also, the random selection as well as the overhead the critical sorting, resulting in faster exponential runtime growth, as it can be observed from its steepness in the plot, made the 2nd algorithm go out first. Also, while algorithm 1 could keep up with algorithms 3 and 4, during the small K iterations, , it could not keep up much. That is thanks to its faster exponential growth, thus worse growth rate coming from randomly selecting sets that most likely were bigger than the smallest set choice of each recursion, resulting in more recursions, more computing power and more time until completion. Moreover, despite its overhead, algorithm's 2 run time, scalability and performance were relatively close to algorithm's 1, due to them both randomly choosing the set to use in each recursion and the choice of numbers not being of any help, as there were not any actual hitting sets during the iterations.

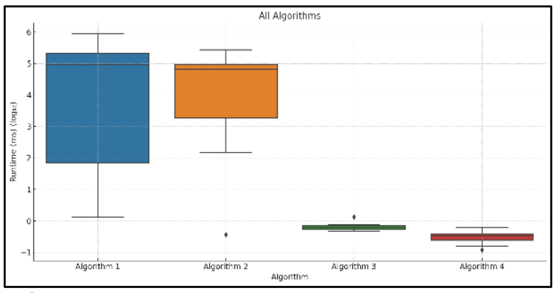
Overall, the results validated my analysis, confirming that the fastest and most scalable algorithm is 3, algorithm 4 is a strong and balanced alternative and algorithms 1 and 2 are unstable and provide poor scalability, so are usable only for small K or simple instances.

Experiment 2:

For this experiment, when flag = 2, I chose and instance with parameters N = 100, M = 40, C = 3-8 (randomly) and K = 15. The randomized instance having a lot of overlap makes it viable for the sets to have a hitting set of 15, due to my selection of 40 sets and a range of 100 numbers to choose from. The choice of a small C was done so for the probability to have overlaps but still to make the slower algorithms compute in a few minutes and not more and the M = 40 makes it so brute force will not be trivial, so they will not finish in a few seconds for every iteration. Due to this combination, a hitting set of 15 is possible but will still stress out the slower algorithm due to the exponential increase of time coming from K. Function experimentRun2 was responsible in creating the instance and then running the runAlgorithms function 10 times. The experiment results follow.

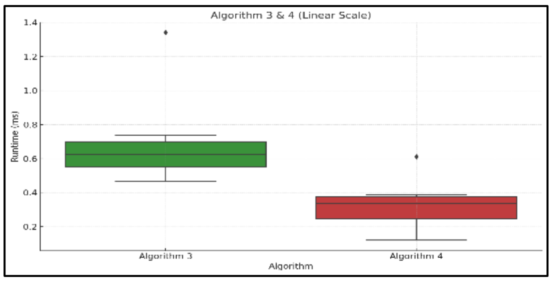


Graph 1:

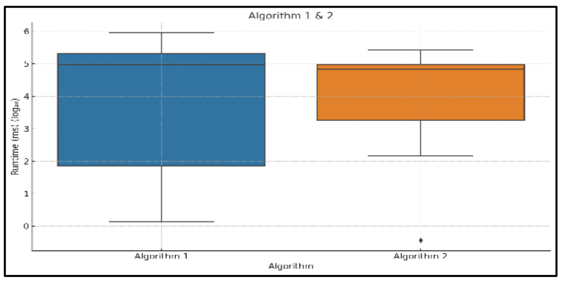


Graph 2:

Graph 3:



Graph 4:



Results/ Conclusions for Experiment 2 and Predictions:

Algorithm 3 was expected to be the faster and more stable algorithm, and it showed a lot of consistency (minimal spread of runtimes), with all its runtimes close to 0.5ms. However, **Algorithm 4** showed slight overperformance regarding speed and showed similar stability as well (a few nanoseconds difference at most, runtimes) in almost all its runs. This occurred due to Algorithm's 4 additional critical selection of the smallest set's elements. Despite algorithm 3 having no overhead, due to its random selection of elements from the smallest sets and the way the instance was created with elements appearing in higher frequencies, the choices of elements led to slightly more recursions and a few nanoseconds, at most, longer total runtimes. This shows that algorithm 4 is the most well-rounded for all parameter choices like predicted.

Algorithm 1 showed the worst consistency of all the algorithms, sometimes solving the problem in a few nanoseconds or milliseconds, while other times taking more than 10 minutes to finish. Like predicted, due to its randomization of sets and elements it is the most unpredictable one, as it is shown in the linear and logarithmic plots 1 and 2 and the logarithmic plot 4. Additionally, **algorithm 2** was predicted to be the slowest, due to the bigger overhead of critical sorting paired with the randomized selection of the sets and so it was. However, because of its more predictable choice of the most critical elements as well as the fact that the instance used had highly frequent numbers, algorithm 2 showed more stability than algorithm 1, but still had a wider spread than 3 and 4. This result comes from the random set picking of algorithm 2.

Additionally, **all algorithms had a few outliers in their runtimes**. For the 1st one, it was due to the randomized selection that happened to pick sets and elements that were included in a hitting set or continually picked sets and elements that were not in such set. For the 2nd one, the nanosecond runtime happened due to the random selection of a set with the smallest size at the current recursion so that its critical elements were highly frequent and in a hitting set, while the few minute runs occurred from the random choice of sets that had numbers that were not frequent across all sets. Algorithm 3's outlier comes from the unlucky choice of one from the smallest sets that did not help prune the instance quicky (did not have frequent numbers). The same applies for algorithm 4 but in a shorter range due to its critical sorting paired with the high frequency numbers instance.

To summarize experiment 2: In terms of speed, algorithm 4 was slightly faster than 3 due to the high frequency number instance, but both showed extremely low runtimes. In terms of stability, algorithms 3 and 4 were highly consistent with tide distributions, while algorithms 1 and 2 had high variance. Outliers occurred for every algorithm but were more noticed in algorithm 1 and 2. Algorithm 1 was the least stable due to its sensitivity to random factors. Compatibility-wise, algorithms 3 and 4 handled varying instances better (from experiment 1 as well) and solved the problem with minimal cost and more efficiency. All predictions were correct except for algorithm 4 outperforming algorithm 3 (and the clarification of algorithm 2). However, this confirms the prediction that algorithm 4 is the most well-rounded of all algorithms for every parameter, due to this instance having higher frequency numbers.