**Hypothesis:** The big-o notation describes the time complexity of a function as the input tends to infinity. This means we care about performance in cases when dealing with large amounts of data. For the merge sort algorithm, this guarantees a worst-case performance of n\*log(n) as n approaches infinity, which is the fastest known notation. However, for smaller N values there may be faster algorithms. Insertion sort is an intuitive method which has a big-o of n^2. The hypothesis is that if N is less than or equal to 50, Insertion Sort will finish in faster time than Merge Sort.

**Methods:**  
 To conduct this experiment a C++ project was created using CMake. To view the project in its entirety, including the full data set, view the repo at: <https://github.com/ephraimbennett/SortingExperiment>.

The first step was to pick the compiler and set up the environment. The compiler chosen was GNU Compiler Collection (GCC) 13.2.0, which was set up through the development environment MinGW. For optimization purposes, the compiler flags used were -O3 (for aggressive optimization), -march=native (to utilize the full capability of the CPU), -flto (link-time optimization), -fopenmp (to enable parallelism if used).

Next, the code was developed. There is a main.cpp file which starts the program. In it are three functions, main, test, and mergeWrapper. The function main loads the numbers used to sort from a txt file and pushes them into a vector. It then calls the test function, which has two parameters: the vector to sort, and the number of times to run each sort. This is important because when sorting with relatively small vector sizes, the speeds will be too fast to measure properly. So, each respective sort call was run for a specified number of times, allowing the magnitude of the time taken to be seen. The output time is then divided by the number of calls to determine the average speed of a single call. For this experiment, each sort was run 10,000 times. That means the output, which is in seconds, can be multiplied by 10^-4 to calculate the time of a single call. To measure time, the c library’s time header was included, and the program measures the difference between the time before the functions were called and after.

In test, the method mergeWrapper is called. Since the mergeSort function is recursive, the vector must be passed by reference to properly sort. So, to ensure that each call to mergeSort is the unsorted vector, we wrap the call in another method which passes the vector by copy.

The mergeSort and insertionSort functions are declared in their respective header files and defined in their respective .cpp files.

**Results:** Several tests were conducted for this experiment, across different vector sizes. It began at 10, and increased to 50, then 100, then to 250, and increased as such. These points were chosen to analyze carefully at the hypothesis (50). At each size N, the test was run 5 times and the average of these was taken. The results can be seen below:

Between a range of N=1000 and N=1200, Merge Sort became faster than Insertion Sort. The test was also run at N=1100, however the difference is too close to make a definitive claim at this point, as some runs have Insertion Sort faster, and some have Merge Sort faster. The growth of Merge Sort’s time complexity can be seen as initially fast before becoming nearly linear which tracks for a time complexity of N\*log(N). Since the hypothesis was making claims about small values, this is sufficient for the experiment. However, a test on a large N value (250,000) was done out of curiosity that can be seen below.

Insertion Sort’s average time was 5.499 seconds, while Merge Sort’s was just 0.02 seconds. Clearly for large numbers, Merge vastly outperforms Insertion.

**Discussion:** Based on the results found, we can conclude that Merge Sort performs faster than Insertion Sort on lists of numbers greater than 225-250. The hypothesis that the threshold exists at 50 is incorrect. This is surprising, since colloquially it is understood that the threshold is generally around 50. There are many reasons why this could be the case. The first reason being the fast optimizations used during compilation which made sequential tasks such as comparison much faster than accessing memory. That means even if more “instructions” were being done by Insertion Sort, those were less expensive than the one’s merge does. This is because each call to the “merge” function requires allocating two different vectors, which can take a while. However, past a point the benefits of logarithmic time does tend to pay off regardless of this.

In terms of challenges, the only issue that was covered was ensuring that the vector remained unsorted after each call. This was touched on above in methods, the solution being to use a wrapper function. During initial testing, confusing results were given where Insertion was much faster than Merge, since it was covering sorted arrays after the first pass. When the array is pre-sorted, Insertion Sort functionally runs in linear time, whereas Merge always requires the splitting and merging of the array.

**Conclusion:** Under the conditions tested, Insertion Sort produces a faster algorithm for N <= 1000, while Merge Sort is faster for N >= 1200. For N in between 1000 and 1200, the algorithms are indistinguishable.