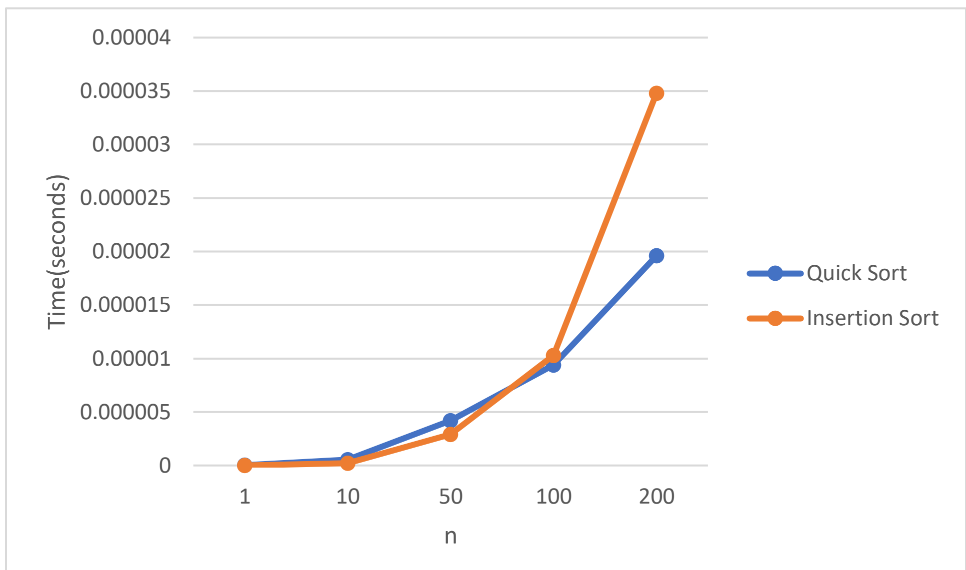
Hypothesis: If quicksort has a runtime of nlogn, and insertion sort has a runtime of n^2, then insertion sort will never be faster than quicksort.

Methods: I am using the NetBeans software version 8.2, I do not know what flags they use at compile time, I only cleaned, built, and ran the project. I took the sorting algorithms for quicksort and insertion sort from geeksforgeeks.com. Their insertion sort is pretty standard but their quicksort uses the right most element as the pivot so the algorithm is susceptible to some outlier cases. I also had a random number generator and assigned random numbers into an array using a for loop. In order to get a non-zero time for the smaller test cases, I had to put the sorting algorithm into a loop. I couldn't figure out how to do this without including the data assignment loop, so I made a loop just for data assignment and timed that. For instance, for the test case of n=1, I looped my functions 10,000,000 times. First I looped 10,000,000 times and timed only the data allocation. Then I looped a different 10,000,000 times and timed the data allocation and quicksort algorithm. I then took the total time it took for that loop and subtracted the time it calculated for just data allocation to get the time spent on the quick sorting algorithm. I repeated this same process for insertion sort. For every loop data allocation takes a different amount of time, but I found that the difference is so minimal, that averaging multiple trials overshadows the impact. I used sample sizes (n) = 1, 10, 50, 100, 200. And for every (n), I compiled once and ran 10 times and averaged out the results.

Results:

n	quick-time	insert-time
1	2.44E-08	2.3E-09
10	0.000000539	0.000000224
50	0.0000042	0.0000029
100	0.0000094	0.0000103
200	0.0000196	0.0000348



Discussion: Even though we were told that for some n, insertion sort was going to be faster, I didn't believe it because the math didn't seem like it was true. My line of thinking was if you plug in 1 into both equations (nlogn and n^2), n^2 produces the bigger number, and that continues for bigger values of n. The only time nlogn produces the bigger number is for numbers less than 1 and greater than 0. I failed to account for the fact that for smaller n's, quicksort ends up doing a lot of inefficient recursive calls. Smaller n's being for n < ~75.

Conclusions: Under the conditions tested, quicksort is a faster algorithm for n > 100, and insertion sort is a faster algorithm for n < 50, for 50<n<100, the two algorithms are indistinguistable.

```cpp
#include <time.h>
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <bits/stdc++.h>

using namespace std;

/*
 *
 */

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

//QUICK SORT FUNCTIONS START HERE
// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
```

```c
/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


int main(int argc, char** argv) {

    //Seed for random number
    srand(time(NULL));
```

```
int main(int argc, char** argv) {

    //Seed for random number
    srand(time(NULL));

    constexpr int SAMPLE_SIZE = 200;
    constexpr int NUM_TESTS = 10000;
    constexpr int RANGE = SAMPLE_SIZE*10;

    int array[SAMPLE_SIZE];


    //Time testing for data allocation itself
    clock_t ds_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
    }
    clock_t df_time = clock() - ds_time;

    //Actual time testing for quicksort
    clock_t s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        quickSort(array,0,SAMPLE_SIZE);
    }
    clock_t f_time = clock() - s_time - df_time;
    cout << "Quick Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;
    //Time testing for insertion sort
    s_time = clock();
    for(int i = 0; i < NUM_TESTS; i++){
        for (int i=0;i<SAMPLE_SIZE;i++){
            array[i] = rand() % RANGE;
        }
        insertionSort(array,SAMPLE_SIZE);
    }
    f_time = clock() - s_time - df_time;
    cout << "Insertion Sort Time: "
         << ((double) f_time) / (double) CLOCKS_PER_SEC
         << " seconds" << endl;
}
```