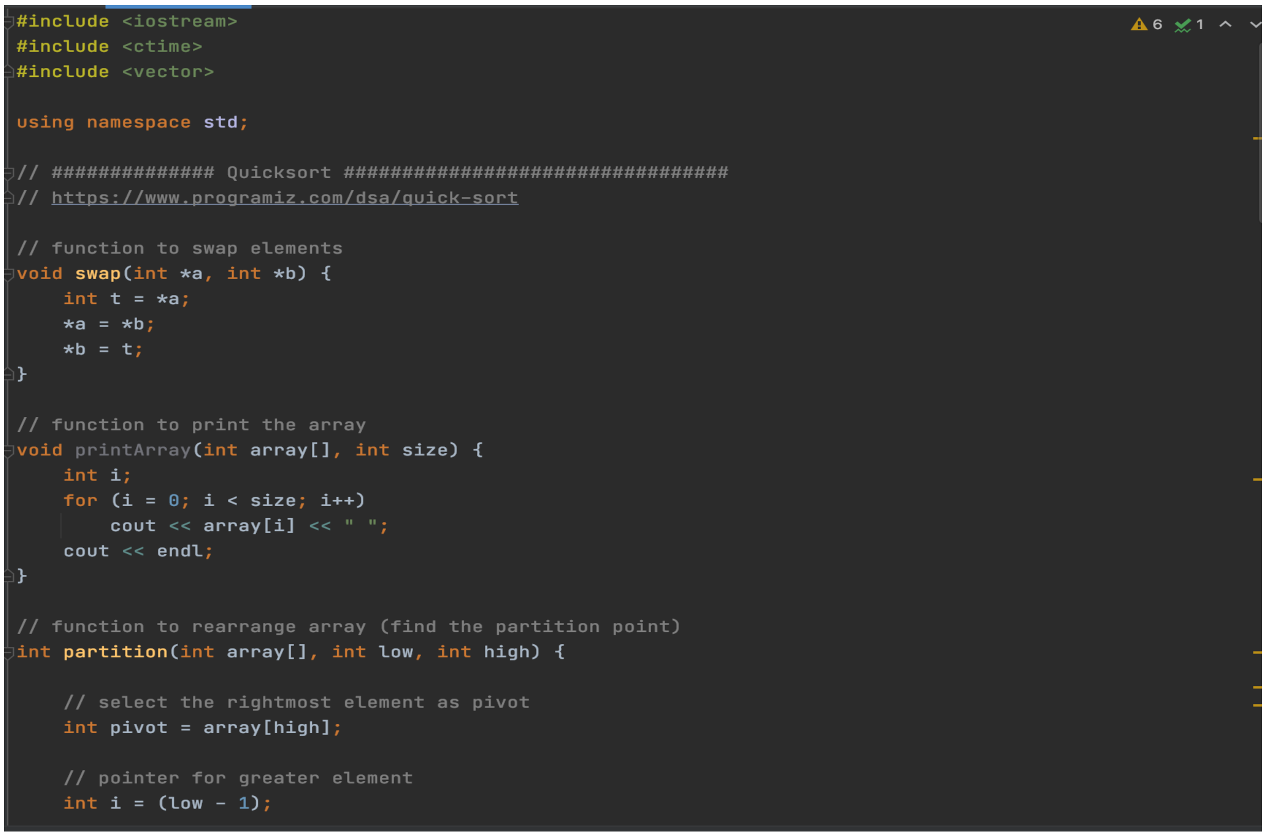
Benjamin Miller

Quicksort vs. Insertion sort

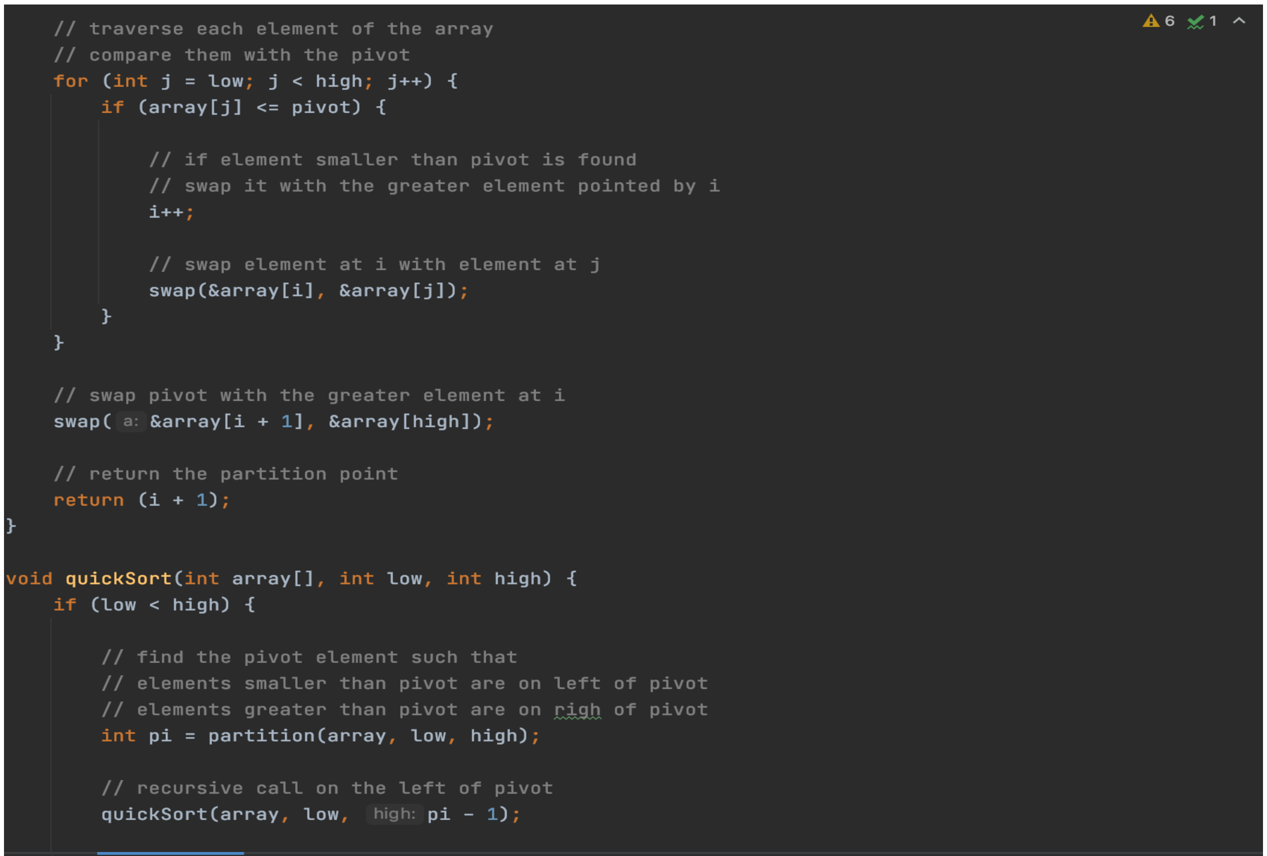
Hypothesis:
I believe that for this experiment insertion sort will be quicker for n < 10. I think I will find that for around 10 < n < 30, that the two sort methods will be similar with quick sort starting to pull away. Finally I think that for n > 30 that quicksort will be much faster.

Methods:
For this experiment I used CLion by Jetbrains to run the code. I created a new project in c++ 14 and put my code into the file that ended with ".cpp". I did not do anything special for the compiler or any optimizations. Once the code was added, I simply hit the run button to get the outputs. For the code in the main function I did these steps:

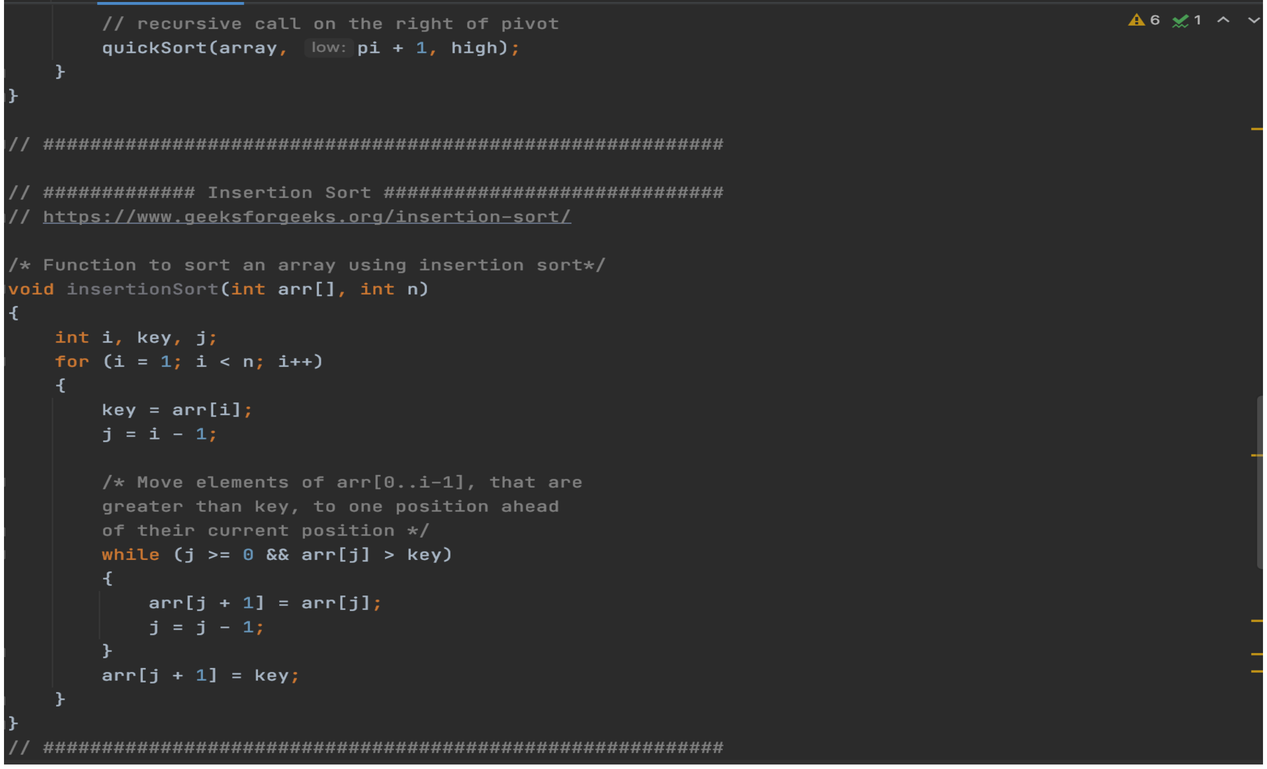
1. Created a loop to run 1,000 times and started the timer, so each test would sort 1,000 arrays
2. Created an array of n size, and filled it with random numbers from 1-100
3. Called the quicksort function from size n = 1, until size n = 80, manually changing the size after each run, and recorded the time output
4. Repeat step 3 for insertion sort



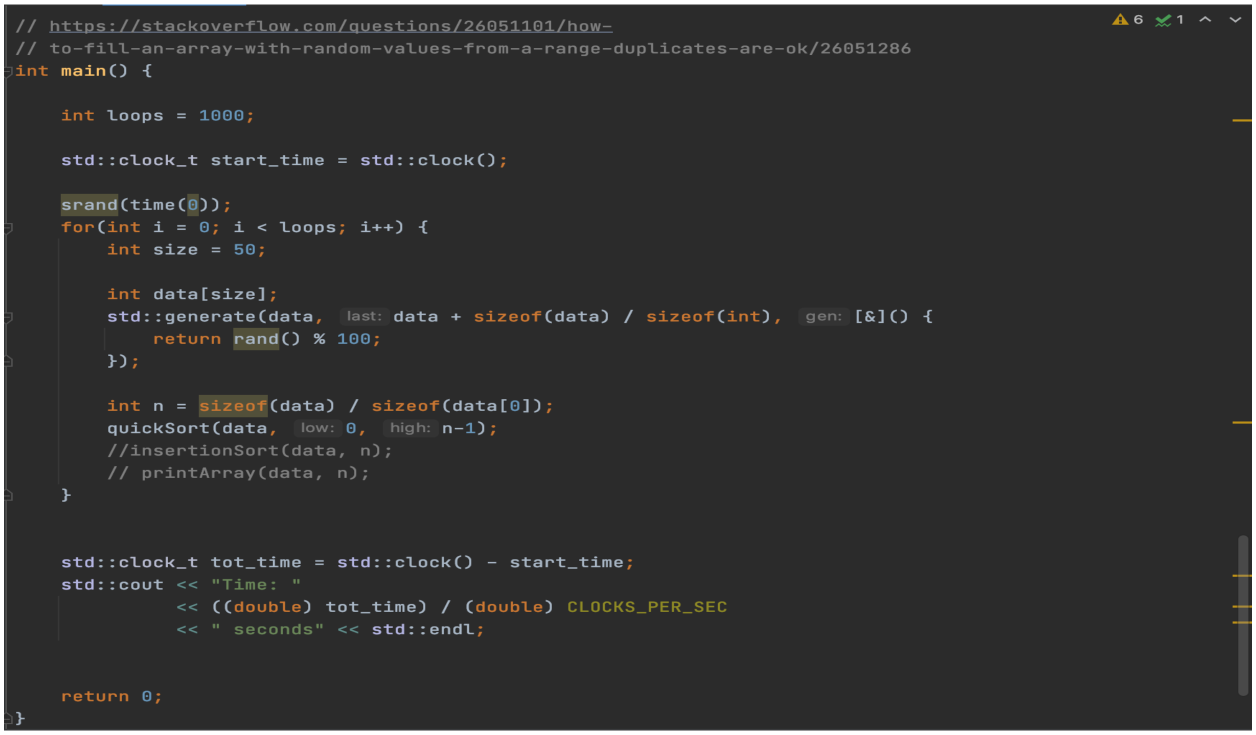
```cpp
#include <iostream>
#include <ctime>
#include <vector>

using namespace std;

// ############## Quicksort ##################################
// https://www.programiz.com/dsa/quick-sort

// function to swap elements
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// function to print the array
void printArray(int array[], int size) {
    int i;
    for (i = 0; i < size; i++)
        cout << array[i] << " ";
    cout << endl;
}

// function to rearrange array (find the partition point)
int partition(int array[], int low, int high) {

    // select the rightmost element as pivot
    int pivot = array[high];

    // pointer for greater element
    int i = (low - 1);
```



```
    // traverse each element of the array
    // compare them with the pivot
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {

            // if element smaller than pivot is found
            // swap it with the greater element pointed by i
            i++;

            // swap element at i with element at j
            swap(&array[i], &array[j]);
        }
    }

    // swap pivot with the greater element at i
    swap( a: &array[i + 1], &array[high]);

    // return the partition point
    return (i + 1);
}

void quickSort(int array[], int low, int high) {
    if (low < high) {

        // find the pivot element such that
        // elements smaller than pivot are on left of pivot
        // elements greater than pivot are on righ of pivot
        int pi = partition(array, low, high);

        // recursive call on the left of pivot
        quickSort(array, low,  high: pi - 1);
```



```
        // recursive call on the right of pivot
        quickSort(array,  low: pi + 1, high);
    }
}

// ##############################################################

// ############# Insertion Sort #############################
// https://www.geeksforgeeks.org/insertion-sort/

/* 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;
    }
}
// ##############################################################
```

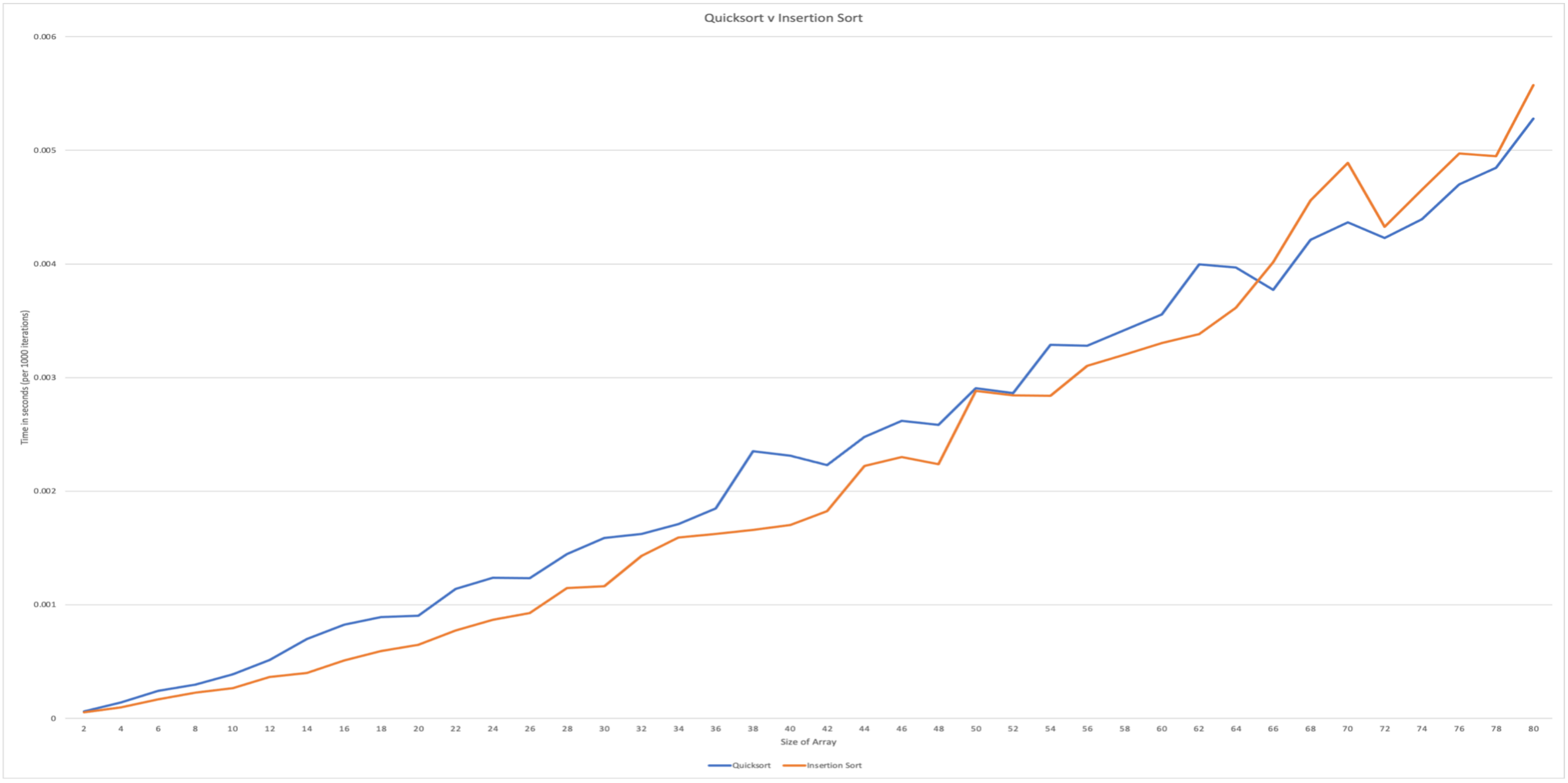


```
// https://stackoverflow.com/questions/26051101/how-
// to-fill-an-array-with-random-values-from-a-range-duplicates-are-ok/26051286
int main() {

    int loops = 1000;

    std::clock_t start_time = std::clock();

    srand(time(0));
    for(int i = 0; i < loops; i++) {
        int size = 50;

        int data[size];
        std::generate(data,  last: data + sizeof(data) / sizeof(int),  gen: [&]() {
            return rand() % 100;
        });

        int n = sizeof(data) / sizeof(data[0]);
        quickSort(data,  low: 0,  high: n-1);
        //insertionSort(data, n);
        // printArray(data, n);
    }


    std::clock_t tot_time = std::clock() - start_time;
    std::cout << "Time: "
              << ((double) tot_time) / (double) CLOCKS_PER_SEC
              << " seconds" << std::endl;


    return 0;
}
```

The auto generated makefile:

```
cmake_minimum_required(VERSION 3.17)
project(431question1)

set(CMAKE_CXX_STANDARD 14)

add_executable(431question1 main.cpp)
```

Results:



The data collected is shown above. This graph shows the comparison of quicksort (blue) vs. insertion sort (orange) for different array sizes. The time is measured in seconds, and is based on each sort program running 1000 times for the given size. Lastly, the intersection point for this data is at size 65, where the quicksort ultimately becomes more efficient.

<u>Discussion:</u>
I was very surprised at how long it took for quicksort to become more efficient. My prediction was pretty far off. A challenge that I faced was collecting the data, as I expected that by 30 I would have a clear intersection. Since I did not, I had to continue collecting data until I was able to find the cutoff point. According to my data, insertion sort was faster for n smaller than 65. I believe that this number could be higher than expected because of my hardware, a Macbook, but relatively 65 is still very small compared to 10,000+. One last thing I did to make sure my algorithms were correct was test times at array size of 10,000. I found that quicksort took 3.13775 seconds while insertion sort took 52.3384. This made me feel more confident that I implemented everything correctly.

<u>Conclusions:</u>
Under the conditions tested, insertion sort was faster for $n < 65$, and quicksort became faster for $n > 65$. At $n = 65$, the two sorting algorithms intersected.