1) **Implement a function named insert that takes a dynamically allocated array of ints, the array's length, the index at which a new value should be inserted, and the new value that should be inserted. The function should allocate a new array populated with the contents of the original array plus the new value inserted at the given index. The original array should be freed.**

hw1 > q1.cpp > insert(int *, int, int, int)

```cpp
#include <iostream>
#include <cstdlib>
#include <chrono>
#include <iomanip>

int *insert(int *array, int length, int index, int value){
    // no index or invalid input --> return a null pointer
    if (index < 0 || index > length){
        return nullptr;
    }

    // an array doesn't already exist
    if (length == 0){
        // declare newArray and allocate memory equal to a single int variable
        int* newArray = (int*)malloc(sizeof(int));

        // return null pointer if cannot create newArray (or newArray doesn't exist)
        if (!newArray){
            return nullptr;
        }

        // place given value as first index of newArray and return it
        newArray[0] = value;
        return newArray;
    } else{
        // if length of array is not 0 --> allocate enough memory for array length + 1 (for insert)
        int* newArray = (int*)malloc(sizeof(int) * (length+1));

        if (!newArray){
            return nullptr;
        }

        // Copy original array into newArray up to where new value is inserted
        for (int i = 0; i < index; ++i){
            newArray[i] = array[i];
        }

        // Insert new value into the specified index
        newArray[index] = value;

        // Copy the rest of the array following the newly inserted value
        for (int i = index; i<length ; ++i){
            newArray[i+1] = array[i];
        }

        // Free up memory allocated (prevent memory leak) and return resulting array (with insert)
        free(array);

        return newArray;
    }
}
```

2) **Implement a main function that profiles the performance of insert and outputs a table showing the average time per insert as the length of the array increases.**

```cpp
int main(){
    // declare constant
    const int INSERTS_PER_READING = 1000;

    // set seed for random
    srand(time(0));

    // delcare empty array and length variable
    int* array = nullptr;
    int length = 0;

    // print table header
    std::cout  <<"Array Length           " << "Seconds per insert" << std::endl;

    // Loop through the insert into array 60 times
    for (int i = 0; i < 60; ++i){
        auto startTime = std::chrono::system_clock::now();

        // Insert a random value into a random index (0, length) up to specified amount of times
        for (int i =0; i < INSERTS_PER_READING; ++i){
            int index = rand() % (length + 1);
            int value = rand();
            array = insert(array, length, index, value);

            // increase length of array each time
            ++length;
        }

        auto stopTime = std::chrono::system_clock::now();

        // calculate elapsed time using a double (minimum precision of at least 15 decimals)
        std::chrono::duration<double> elapsedTime = stopTime - startTime;
        double timePerInsert = elapsedTime.count() / INSERTS_PER_READING;

        // print result line by line
        std::cout << length << "                  " <<
        std::fixed << std::setprecision(6) << timePerInsert << std::endl;
    }

    // free up allocated memory (prevent memory leak)
    free(array);
    return 0;
}
```

OUTPUT

```
[Running] cd "/Users/jeffylee/Desktop/UCSD DSA C++/hw1/" && g++ q1.cpp -o q1 && "/Users/jeffylee/Desktop/UCSD DSA C++/hw1/"q1
q1.cpp:55:9: warning: 'auto' type specifier is a C++11 extension [-Wc++11-extensions]
        auto startTime = std::chrono::system_clock::now();
        ^
q1.cpp:64:9: warning: 'auto' type specifier is a C++11 extension [-Wc++11-extensions]
        auto stopTime = std::chrono::system_clock::now();
        ^
2 warnings generated.
Array Length             Seconds per insert
1000                    0.000001
2000                    0.000003
3000                    0.000004
4000                    0.000006
5000                    0.000007
6000                    0.000007
7000                    0.000008
8000                    0.000008
9000                    0.000009
10000                    0.000010
11000                    0.000010
12000                    0.000010
13000                    0.000012
14000                    0.000013
15000                    0.000014
16000                    0.000018
17000                    0.000019
18000                    0.000017
19000                    0.000017
20000                    0.000018
21000                    0.000019
22000                    0.000020
23000                    0.000020
24000                    0.000021
25000                    0.000022
26000                    0.000023
27000                    0.000024
28000                    0.000025
29000                    0.000026
30000                    0.000027
31000                    0.000027
32000                    0.000028
33000                    0.000029
34000                    0.000029
35000                    0.000030
36000                    0.000036
37000                    0.000033
38000                    0.000034
39000                    0.000034
40000                    0.000036
41000                    0.000037
42000                    0.000040
43000                    0.000038
44000                    0.000039
45000                    0.000040
46000                    0.000041
47000                    0.000042
48000                    0.000043
49000                    0.000043
50000                    0.000044
51000                    0.000045
52000                    0.000046
53000                    0.000046
54000                    0.000047
55000                    0.000048
56000                    0.000049
57000                    0.000051
58000                    0.000051
59000                    0.000052
60000                    0.000053

[Done] exited with code=0 in 2.265 seconds
```

3) **Plot a scatter graph showing “Seconds per insert” (Y-axis) vs. “Array length” (X-axis) using the profiling data that was output by main.**

Add following command:

```
5    #include <fstream>
```

.

Add following line and edit std::cout commands to store into new file:

```
// Open file to save data
std::ofstream profilingData("profiling_data.txt");

// Output header
profilingData <<"Array Length            " << "Seconds per insert" << std::endl;
```

```
// write data to file
profilingData << length << "                    " <<
std::fixed << std::setprecision(6) << timePerInsert << std::endl;
```

Then, I used python's matplotlib library to create a scatter plot of seconds per insert vs. array length. **Please let me know if I should use something else** to create the graph. My background is in Data Science and ML, so I thought python was a good tool to use.

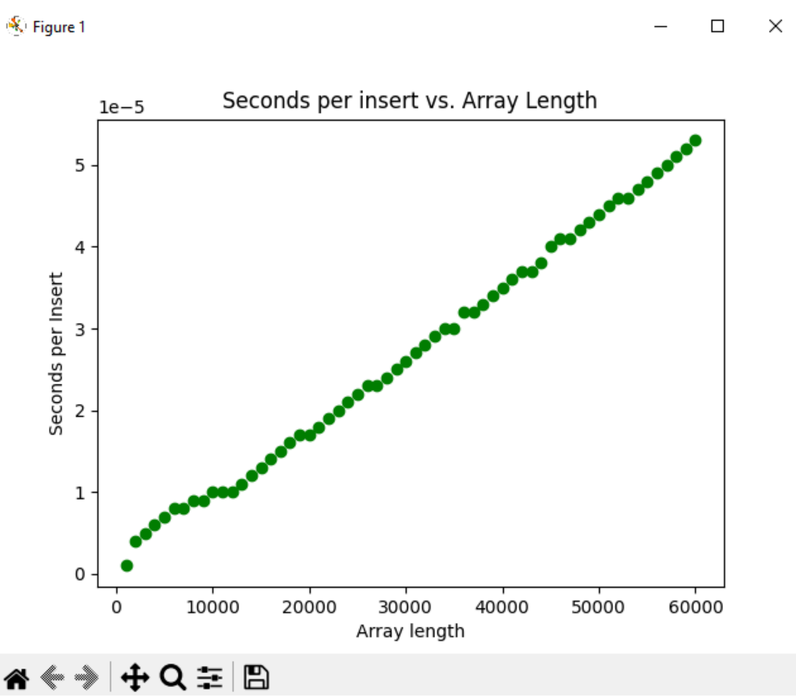
```
hw1 > q3.py > ...
import matplotlib.pyplot as plt

x = []
y = []

# Open file as read-only
with open('profiling_data.txt', 'r') as file:
    next(file)                                   # skip header line
    for line in file:
        length, timePerInsert = line.split()    # split at variables into x and y at whitespace
        x.append(int(length))
        y.append(float(timePerInsert))

# Create Scatter plot
plt.scatter(x, y, color='green', label="Seconds per Insert")

# Label the axes
plt.xlabel("Array length")
plt.ylabel("Seconds per Insert")
plt.title("Seconds per insert vs. Array Length")
plt.show()
```



4) **Provide a line-by-line Big-O analysis of your implementation of insert. You can do this by adding a comment next to each line in your source code. What is the overall Big-O performance of insert? What parts of the algorithm contribute most heavily to the overall Big-O performance?**

To understand the overall time complexity of the program, we should split the program into two parts. The main() function and the insert() function. The time complexity of insert function looks like this:

```
int *insert(int *array, int length, int index, int value){
    // no index or invalid input --> return a null pointer
    if (index < 0 || index > length){                                                       // O(1)
        return nullptr;                                                                     // O(1)
    }

    // an array doesn't already exist
    if (length == 0){                                                                       // O(1)
        // declare newArray and allocate memory equal to a single int variable
        int* newArray = (int*)malloc(sizeof(int));                                          // O(1)

        // return null pointer if cannot create newArray (or newArray doesn't exist)
        if (!newArray){                                                                     // O(1)
            return nullptr;                                                                 // O(1)
        }

        // place given value as first index of newArray and return it
        newArray[0] = value;                                                                // O(1)
        return newArray;                                                                    // O(1)
    } else{
        // if length of array is not 0 --> allocate enough memory for array length + 1 (for insert)
        int* newArray = (int*)malloc(sizeof(int) * (length+1));                             // O(1)

        if (!newArray){                                                                     // O(1)
            return nullptr;                                                                 // O(1)
        }

        // Copy original array into newArray up to where new value is inserted
        for (int i = 0; i < index; ++i){                                                    // O(n)
            newArray[i] = array[i];                                                         // O(1)
        }

        // Insert new value into the specified index
        newArray[index] = value;                                                            // O(1)

        // Copy the rest of the array following the newly inserted value
        for (int i = index; i<length ; ++i){                                                // O(n)
            newArray[i+1] = array[i];                                                       // O(1)
        }

        // Free up memory allocated (prevent memory leak) and return resulting array (with insert)
        free(array);                                                                        // O(1)

        return newArray;                                                                    // O(1)
    }
}
```

From this, we can see that there are 2 for-loops which are of the O(n) time complexity. However, they are not nested within each other and the overall time complexity of the insert function is O(2n), which ultimately simplifies to O(n). Now, we must take a look at the main function.

```cpp
int main(){
    // declare constant
    const int INSERTS_PER_READING = 1000;                                        // O(1)

    // set seed for random
    srand(time(0));                                                              // O(1)

    // Open file to save data
    std::ofstream profilingData("profiling_data.txt");                           // O(1)

    // Output header
    profilingData <<"Array Length           " << "Seconds per insert" << std::endl;        // O(1)

    // delcare empty array and length variable
    int* array = nullptr;                                                        // O(1)
    int length = 0;                                                              // O(1)

    // Loop through the insert into array 60 times
    for (int i = 0; i < 60; ++i){                                                // O(1)
        auto startTime = std::chrono::system_clock::now();                       // O(1)

        // Insert a random value into a random index (0, length) up to specified amount of times
        for (int i =0; i < INSERTS_PER_READING; ++i){                            // O(1)
            int index = rand() % (length + 1);                                   // O(1)
            int value = rand();                                                  // O(1)
            array = insert(array, length, index, value);                         // O(n)

            // increase length of array each time
            ++length;                                                            // O(1)
        }

        auto stopTime = std::chrono::system_clock::now();                        // O(1)

        // calculate elapsed time using a double (minimum precision of at least 15 decimals)
        std::chrono::duration<double> elapsedTime = stopTime - startTime;
        double timePerInsert = elapsedTime.count() / INSERTS_PER_READING;

        // write data to file
        profilingData << length << "                    " <<
        std::fixed << std::setprecision(6) << timePerInsert << std::endl;        // O(1)
    }

    // free up allocated memory (prevent memory leak)
    free(array);                                                                 // O(1)
    return 0;
}
```

There are for-loops which are nested within each other. However, they are both constants because they are fixed (constant) number of executions (60 for the first for-loop and 1000 for the second, nested for-loop). This ultimately simplifies to O(1) time complexity. However, we must take into account the insert function inside the second for-loop. We concluded that it is O(n) time complexity earlier, and this ultimately makes the whole program O(n) time complexity. This is also consistent with the graph provided in Question #3.

The two for-loops in the insert function contribute most heavily to the overall Big-O performance.

5) **Based on the graph does the performance improve, degrade, or stay the same as the length of the array grows/ Does your Big-O analysis match the results of running the program?**

Based on the graph from Question #3, the time it takes the program to finish its tasks grows linearly with increasing array length. This means that the program degrades as the length of the array grows. My Big-O analysis in Question #4 is consistent with my graph provided in Question #3.