Homework #1

In this assignment you will analyze the Big-O performance of inserts into an array by inspecting source code. You will then compare this analysis to the actual performance seen in a running program.

1. **(2 points)** Implement a function named `insert` that takes a dynamically allocated array of `int`s, 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 originally array should be `free`d. The following sections provide a detailed description of this function:

   Prototype:

   ```
   int *insert(int *array, int length, int index, int value);
   ```

   Parameters:

`array`	The array of `int`s into which a new `value` should be inserted.
`length`	The number of elements in the `array`.
`index`	The location where the `value` should be inserted into the `array`.
`value`	The value that should be inserted into the `array`.

   Return value:

   A new array of `int`s containing the contents of the original `array` plus the new `value` inserted at the given `index`. `NULL` will be returned should something goes wrong.

   Pseudocode:

   ```
   insert(array, length, index, value)
      If array is empty (i.e. length == 0)
         return a newly malloc'd array having 1 element of the given value
      End If
      Else
         Let newArray = a newly malloc'd array with length + 1 elements
         Copy array[0, index) to newArray[0, index)
         Set newArray[index] to value
         Copy array[index, length) to newArray[index + 1, length + 1)
         Free array
         Return newArray
      End Else
   ```

2. **(2 points)** 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.

Pseudocode:

```
main()
   /* Setting to allow fine-tuning the granularity of the readings */
   Let INSERTS_PER_READING = 1000

   /* Start with an empty array */
   Let array = empty array (i.e. NULL)
   Let length = 0

   /* Take 60 readings */
   Loop 60 times

      /* Each reading will be taken after INSERTS_PER_READING inserts */
      Let startTime = current time
      Loop INSERTS_PER_READING times
         Let index = random integer in range [0, length]
         Let value = random integer value
         Let array = insert(array, length, index, value)
         Let length = length + 1
      End Loop
      Let stopTime = current time
      Let timePerInsert = (stopTime – startTime) / INSERTS_PER_READING

      /* Output reading in tabular format */
      Output array length and timePerInsert

   End Loop

   /* Free the old array */
   Free array
```

Report format:

`main` should output a report similar to the format below (your values will be different). You should fine-tune the INSERTS_PER_READING constant so that none of the readings ("Seconds per insert") are zero:

```
 Array length         Seconds per insert
         1000                   0.000024
         2000                   0.000028
         3000                   0.000041
         4000                   0.000036
          ...                        ...
        57000                   0.000262
        58000                   0.000318
        59000                   0.000324
        60000                   0.000328
```

3. **(2 points)** Plot a scatter graph showing "Seconds per insert" (Y-axis) vs. "Array length" (X-axis) using the profiling data that was output by main.

4. **(2 points)** 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?

5. **(1 point)** Based on the graph does the performance of improve, degrade, or stay the same as the length of the array grows? Does your Big-O analysis of match the results of running the program?

6. **(1 point)** Make sure your source code is well-commented, consistently formatted, uses no magic numbers/values, follows programming best-practices, and is ANSI-compliant.

Turn in all source code, program output, diagrams, and answers to questions in a single Word document.