Student Coding Homework Assignment Submission (Max 10 points)

1. Student Registration Information

- **Full Name**: Pablo Guardia
- **Student ID**: pjg22b
- **Course Name**: Complexity and Analysis of Data Structures and Algorithms
- **Instructor Name**: Dr. Ahsan Abdullah
- **Date of Submission (mm-dd-yyyy)**: 02-14-2025

---

2. Assignment Overview (1 point)

- **Assignment Title**: Animated Quicksort
- **Assignment Objectives**:
    - This assignment aims to animate the Quicksort algorithm and compare different input sizes' performances among three different pivot choices. The algorithm is animated using a stack of stars with the pivot being colored in red.

---

3. Code

- **Code Implementation (2 points):**

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <random>
#include <chrono>
#include <thread>
using namespace std;
using namespace std::this_thread;
using namespace std::chrono;


// Enumerated data type for pivot selection
enum PivotSelection {FIRST, LAST, MIDDLE};


// Three QuickSort algorithms, each having a different choice of pivot
void QuickSortFirst(vector<int> &arr, int start, int end, int max, double &total);
void QuickSortMiddle(vector<int> &arr, int start, int end, int max, double &total);
void QuickSortLast(vector<int> &arr, int start, int end, int max, double &total);

// Prints a visual representation of the array by drawing stacks of stars (*)
void PrintArray(const vector<int> &arr, int pivot, int max, double duration);
```

```
int main() {
   // Random number generator from 1 to 10
   random_device rd;
   mt19937 gen(rd());
   uniform_int_distribution<> nums(1, 10);

   // Create an array of n random elements between 1 and 10
   // Also determine the max element to determine how high the stacks
   // of stars (*) should be
   // Change n to determine the size of the array
   vector<int> arr;
   int n = 10, max = -2147483648;
   arr.reserve(n);
   for (int i = 0; i < n; i++)
   {
      arr.push_back(nums(gen));
      if (arr[arr.size() - 1] > max)
         max = arr[arr.size() - 1];
   }

   // Select the pivot by changing selection to FIRST, MIDDLE, or LAST
   PivotSelection selection = LAST;
   double duration = 0;

   switch (selection) {
      // Print the initial unsorted array before executing the chosen algorithm
      case FIRST:
         PrintArray(arr, 0, max, 0);
         QuickSortFirst(arr, 0, arr.size() - 1, max, duration);
         break;
      case MIDDLE:
         PrintArray(arr, arr.size() / 2, max, 0);
         QuickSortMiddle(arr, 0, arr.size() - 1, max, duration);
         break;
      case LAST:
         PrintArray(arr, arr.size() - 1, max, 0);
         QuickSortLast(arr, 0, arr.size() - 1, max, duration);
         break;
   }

   cout << "Total time taken by QuickSort: " << fixed << setprecision(9) << duration << " seconds" << endl;
}



// QuickSort algorithm with pivot at the start of each sub-array
void QuickSortFirst(vector<int> &arr, int start, int end, int max, double &total)
{
   time_point<steady_clock> begin = high_resolution_clock::now();
```

```
    if (end <= start)
        return;

    int j = end + 1;
    int pivot = start;

    for (int i = end; i >= start; i--)
    {
        if (arr[i] >= arr[pivot])
        {
            j--;
            if (i < j)
            {
                if (i == pivot)
                    pivot = j;
                swap(arr[i], arr[j]);
            }
        }
    }

    // Calculates the time taken for each iteration and adds it to a running total
    time_point<steady_clock> finish = high_resolution_clock::now();
    double iteration_time = duration_cast<nanoseconds>(finish - begin).count() * 1e-9;
    total += duration_cast<nanoseconds>(finish - begin).count() * 1e-9;

    // Prints the visual representation after each iteration
    PrintArray(arr, pivot, max, iteration_time);

    QuickSortFirst(arr, start, pivot - 1, max, total);
    QuickSortFirst(arr, pivot + 1, end, max, total);
}

// QuickSort algorithm with pivot in the middle of each sub-array
void QuickSortMiddle(vector<int> &arr, int start, int end, int max, double &total)
{
    time_point<steady_clock> begin = high_resolution_clock::now();

    if (start >= end)
        return;

    int pivot = (start + end) / 2;
    int left = start;
    int right = end;

    while (left <= right)
    {
        while (arr[left] < arr[pivot])
            left++;
        while (arr[right] > arr[pivot])
            right--;
        if (left <= right)
        {
            if (right == pivot)
                pivot = left;
            else if (left == pivot)
                pivot = right;
            swap(arr[left], arr[right]);
            left++;
            right--;
        }
    }

    // Calculates the time taken for each iteration and adds it to a running total
    time_point<steady_clock> finish = high_resolution_clock::now();
    double iteration_time = duration_cast<nanoseconds>(finish - begin).count() * 1e-9;
    total += duration_cast<nanoseconds>(finish - begin).count() * 1e-9;

    // Prints the visual representation after each iteration
    PrintArray(arr, pivot, max, iteration_time);

    QuickSortMiddle(arr, start, left - 1, max, total);
    QuickSortMiddle(arr, left, end, max, total);
}

// QuickSort algorithm with pivot at the end of each sub-array
void QuickSortLast(vector<int> &arr, int start, int end, int max, double &total)
{
    time_point<steady_clock> begin = high_resolution_clock::now();

    if (start >= end)
        return;

    int j = start - 1;
    int pivot = end;

    for (int i = start; i <= end; i++)
    {
        if (arr[i] <= arr[pivot])
        {
            j++;
            if (i > j)
            {
                if (i == pivot)
                    pivot = j;
                swap(arr[i], arr[j]);
            }
        }
    }

    // Calculates the time taken for each iteration and adds it to a running total
    time_point<steady_clock> finish = high_resolution_clock::now();
    double iteration_time = duration_cast<nanoseconds>(finish - begin).count() * 1e-9;
    total += duration_cast<nanoseconds>(finish - begin).count() * 1e-9;

    // Prints the visual representation after each iteration
    PrintArray(arr, pivot, max, iteration_time);
    QuickSortLast(arr, start, pivot - 1, max, total);
    QuickSortLast(arr, pivot + 1, end, max, total);
}


// Prints a visual representation of the array by drawing stacks of stars (*)
// The pivot is represented by a red stack of stars
void PrintArray(const vector<int> &arr, int pivot, int max, double duration)
{
    // Clears the screen before printing to give the impression of a
    // frame-by-frame animation (works best if the user doesn't scroll up)
    // This is done with an ANSI escape code
    cout << "\u001b[2J";

    cout << "--------";
    for (int i = 0; i < arr.size() * 3; i++)
        cout << "-";
    cout << endl;

    for (int h = max; h > 0; h--)
    {
        cout << "       | ";
        for (int i = 0; i < arr.size(); i++)
        {
            if (i >= 10 || arr[i] == 10)
                cout << " ";
            if (h <= arr[i])
                if (i == pivot)
                    cout << "\u001b[38;5;196m*\u001b[0m "; // Red coloration is achieved with ANSI escape codes
                else
                    cout << "* ";
            else
                cout << "  ";
        }
        cout << "|" << endl;
    }

    cout << "--------";
    for (int i = 0; i < arr.size() * 3; i++)
        cout << "-";
    cout << endl;

    cout << "Index:  ";
    for (int i = 0; i < arr.size(); i++)
    {
        if (i < 10 && arr[i] == 10)
            cout << " ";
        if (i == pivot)
            cout << "\u001b[38;5;196m" << i << "\u001b[0m ";
        else
            cout << i << " ";
    }
    cout << endl;

    cout << "Value:  ";
    for (int i = 0; i < arr.size(); i++)
    {
        if (i >= 10 && arr[i] < 10)
            cout << " ";
        if (i == pivot)
            cout << "\u001b[38;5;196m" << arr[i] << "\u001b[0m ";
        else
            cout << arr[i] << " ";
    }
    cout << endl;

    // Prints the current running total for all iterations of
    // the chosen QuickSort algorithm up to this point
    cout << "Time taken for this iteration: " << fixed << setprecision(9) << duration << " seconds" << endl;

    // Delays execution of the program for fixed time in milliseconds
    // Lower the value to make the animation go faster, raise it to make it go slower
    sleep_for(milliseconds(500));
}
```

4. Results Screenshot (1 point)

- **Screenshots**:

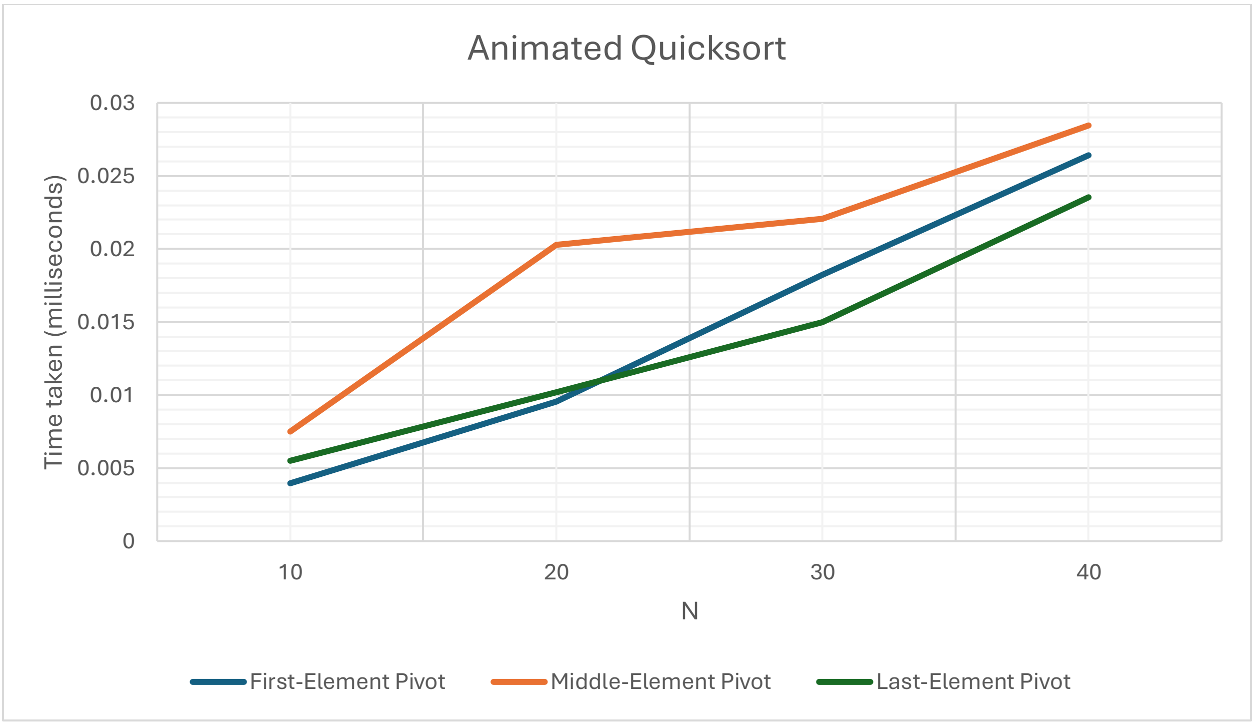
Sample run of code written in section 3. The array is of size 20 and the pivot is selected at the end of it. The animation updates whenever the array is partitioned, not every time a swap is made. However, the time taken accounts for an entire function call, from the beginning to right before the PrintArray() function is called.

---

5. Data Generated (2 points)

- **Data Output**:

N	**First-Element Pivot**	**Middle-Element Pivot**	**Last-Element Pivot**
10	0.003958	0.007499	0.0055
20	0.00957	0.020279	0.010194
30	0.018208	0.022083	0.014972
40	0.026417	0.028457	0.02354



---

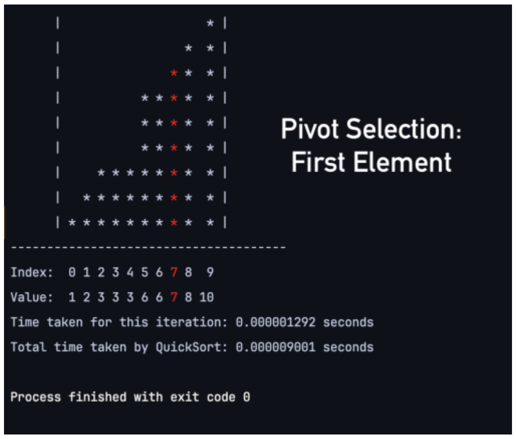
6. Analysis of Results (3 points)

1) What is the relationship between the size of the input (n) and the average completion time for quicksort?

- Quicksort has a best- and average-case time complexity of O(nlog n), meaning that in most cases, the average completion time will increase somewhat *more than linearly* but not quite quadratically the higher the input size n is. The O(nlog n) complexity represents Quicksort's divide-and-conquer structure, as it divides the array recursively after every sorting step. In an ideal scenario, we would like the array to be divided in equal parts, but if the pivot is selected badly, we could end up doing almost as many divisions as there are elements in the array. For this reason, Quicksort's worst-case time complexity is O(n^2), typically achieved when the array is already sorted or the pivot is constantly selected at the beginning or end of each sub-array.
- Quicksort also has a best- and average-case space complexity of O(log n), signifying that in the best cases, each recursive division divides the array in half. In the worst-case scenario, though, if the pivot is always chosen at the start or end of the array and/or the array is already sorted, the space complexity would be O(n), as there would be many function calls as there is elements in the array, excessively populating the call stack.

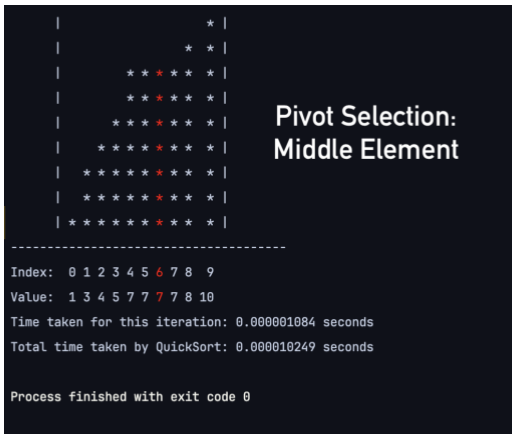
2) How does the choice of pivot affect the performance of the quicksort algorithm for different values of n?

- According to my test runs, as the input size n increases, last-element choice increases in running time the slowest out of the three pivot choices, followed by first-element choice, and lastly middle-element choice taking the most time. The latter particularly takes a long time, though the n = 20 average seems like somewhat of an outlier caused by worst-case, unbalanced partitioning, or simply function call overhead from an excessively populated call stack. In all three pivot choices, there are times when the order of the elements doesn't change at all for multiple steps while the pivot index keeps changing position. This can be the result of us seeing the "unwinding" of recursive function calls in action, which, if there were a lot of calls, could be contributing to the extra time.



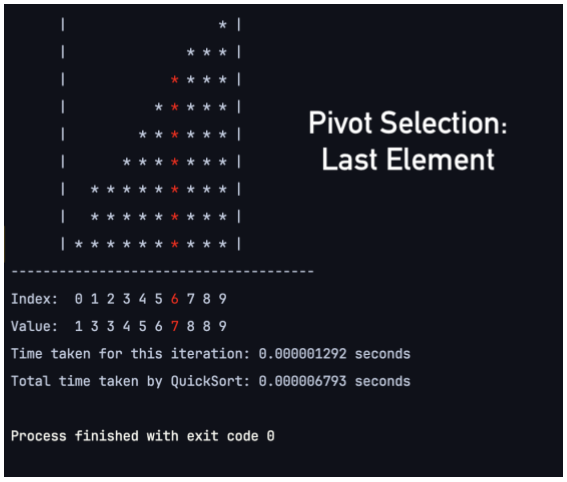
```
      |                    * |
      |                 *  * |
      |               * *  * |
      |           * * * *  * |
      |           * * * *  * |
      |           * * * *  * |
      |     * * * * * * *  * |
      |   * * * * * * * *  * |
      | * * * * * * * * *  * |
--------------------------------------
Index:  0 1 2 3 4 5 6 7 8  9
Value:  1 2 3 3 3 6 6 7 8 10
Time taken for this iteration: 0.000001292 seconds
Total time taken by QuickSort: 0.000009001 seconds

Process finished with exit code 0
```



```
      |                    * |
      |                 *  * |
      |         * * * * *  * |
      |         * * * * *  * |
      |       * * * * * *  * |
      |     * * * * * * *  * |
      |   * * * * * * * *  * |
      |   * * * * * * * *  * |
      | * * * * * * * * *  * |
--------------------------------------
Index:  0 1 2 3 4 5 6 7 8  9
Value:  1 3 4 5 7 7 7 7 8 10
Time taken for this iteration: 0.000001084 seconds
Total time taken by QuickSort: 0.000010249 seconds

Process finished with exit code 0
```



```
|                   * |
|               * * * |
|             * * * * |
|           * * * * * |
|         * * * * * * |
|       * * * * * * * |
|   * * * * * * * * * |
|   * * * * * * * * * |
| * * * * * * * * * * |
-------------------------------------
Index:  0 1 2 3 4 5 6 7 8 9
Value:  1 3 3 4 5 6 7 8 8 9
Time taken for this iteration: 0.000001292 seconds
Total time taken by QuickSort: 0.000006793 seconds

Process finished with exit code 0
```

- While the assignment asked for us to compare running time for choosing the first, middle, or last index in the array as a pivot, there are many other ways to select a pivot that could ultimately end up being more efficient for randomized arrays. An example would be the median-of-three method, which takes the first, middle, and last index of the array and chooses whichever value is in the middle of the three as the pivot (e.g., if 2, 1, 9, 3, 6 was an array, we would analyze 2, 9, and 6, and choose the last index as our pivot since 6 is between 2 and 9).

3) How consistent are the results across multiple runs, and what might account for any variability in the average completion time?

- All three pivot choices seem to be consistent across multiple runs on their own, however there may be times, such as one time with middle pivot selection with n = 20, that outliers may happen, which I believe can happen due to the elements contained in the array itself. The array is always randomized every run with uniform distribution (see the first 3 lines inside the main() function). However, there are many runs where some duplicate numbers are encountered in the array.
- Also, if the array is already sorted to begin with or an iteration works too well to the point that most (say, 90%) of the array is sorted in one go, Quicksort may reach a worst-case time complexity of O(n^2), especially when choosing the first and/or last elements as pivots. Overall, despite its quick reputation, Quicksort is a very unstable algorithm that is not that suitable for small-sized arrays.
- Another factor that may contribute to vulnerability is the call stack. Quicksort is recursive in nature, meaning that it performs many function calls after each iteration to divide and conquer the array into smaller subproblems. The “unwinding” of these recursive function calls can provide a significant contribution to excessive running time, and it is visualized in the animation whenever the highlighted pivot index gets tossed around while the elements themselves don’t change position.

7. Conclusion (1 point)

- **Summary**:
    - In this assignment, we animated the Quicksort algorithm with three different pivot selections: first, middle, and last element, in order to visualize how pivot selection works as well as the algorithm's recursive nature. As we did so, we recorded the average time taken across multiple runs and compared the running times, finding that middle-element selection takes the longest time, followed by first-element and last-element selection. While the algorithm may have an average time complexity of O(nlog n), its recursive style causes it to have a larger space complexity that is around O(log n) on average, due to the amount of function calls present, the "unwinding" of which can result in large running times. This unwinding can be seen in the animation with the right array elements.

8. Additional Comments (Optional) (bonus 1 point)

- **Feedback/Comments**:
    - https://www.youtube.com/watch?v=WprjBK0p6rw (Video used as a guide to write the first- and last-element pivot Quicksort functions)
    - https://www.youtube.com/watch?v=pM-6r5xsNEY (Video used as a guide to write the middle-pivot Quicksort function)

Submission Checklist:

- ✅ Student Registration Information
- ✅ Assignment Overview
- ✅ Code
- ✅ Results Screenshot(s)
- ✅ Data Generated
- ✅ Analysis of Results
- ✅ Conclusion
- ✅ Additional Comments (if applicable)