JJ Javier
Justin Nguyen

Intro:
The big O notation is used in computer science to determine the performance or complexity of an algorithm. In this case, we test the two algorithms quick sort and insertion sort in order to find out which of the two is more efficient. This of course is also determined by how many elements are being sorted as well.

Hypothesis:
The quick sort algorithm will have a faster run time than the insertion sort algorithm for larger data sizes, while insertion sort will have a faster run time than the quick sort algorithm for smaller data sizes.

Experimental Design:
Our design is based on our assignment 4 quick sort and insertion sort algorithms. We used the code from assignment 4, test 2 in order to get the runtime from the arrays that are already sorted in ascending order, descending order, and in a random order. We modified the code in order to get the data we were looking for. To do so we created 6 arrays with all the exact same set array size. Arrays 1 and 2 were filled with elements going in already sorted ascending order, arrays 3 and 4 were filled with elements in already sorted descending order, and arrays 5 and 6 were filled with elements at random. Each pair of arrays had the same values in the same spots, but were tested using the two algorithms. The odd numbered arrays were tested using the insertion sort while the even numbered arrays were tested using quick sort.

Data Acquisition Procedure:
During the processes of acquiring the data we desired, we tested different array sizes in order to determine which algorithm was best. We would base this off of the randomly generated arrays 5 and 6. Doing so, we would start at a small array size of 10 then slowly make our way up in order to find out desired runtime of when both algorithms would be very close in their runtime. We were to run our code with the same set array size multiple times in order to really understand what was happening and why each algorithm took as long or as short as they did.

Data Collection:

Data type and size	Insertion Sort (ET) seconds	Quick Sort (ET) seconds
Ascending, Size=10	2.4e-7	4.91e-7
Descending, Size=10	2.2e-7	4.29e-7
Random, Size=10	2.11e-7	2.6e-7
Ascending, Size=75	3.6e-7	4.07e-7
Descending, Size=75	4.96e-6	4.22e-6
Random, Size=75	3.16e-6	2.78e-6
Ascending, Size=100,000	0.00012058	10.6248
Descending, Size=100,000	16.8332	12.0838
Random, Size=100,000	7.74716	.0398222

Data Analysis:
For the sample size of 10 we saw that insertion was faster for all of the different types of data types, this changed when we tested for the sample size of 75 and 100000 where quicksort is faster for the Descending and Random arrays. However, for all of these sample sizes we can see that insertion is quicker with every ascending array. We can see that the difference between ascending and random insertion sort and ascending and random quick sort for the 100000 sample size is much more vast than the difference for the 10 sample size. We can also see that the difference for descending arrays for each sample size doesn't differ too much.

Data Interpretation:
When interpreting the results, we come to find out that when it comes to insertion sort, best case scenario the complexity of it becomes O(n), while worst case scenario it becomes O(n^2). However, with quick sort, the best case scenario, its complexity becomes O(n log(n)), while the worst case scenario becomes O(n^2).

Conclusion:
In conclusion, in general quick sort is faster when dealing with a larger data size while insertion sort is faster when dealing with a smaller data size. However, when the data is already in ascending order, the runtime of insertion sort, no matter the data size, will always be faster than the runtime of quicksort. But when dealing with the worse case scenario, it is always best to use the quick sort algorithm rather than insertion sort.

Appendix

print_array.cpp:

```
#include "print_array.h"

/**
 * @brief Convert an array of integers to a string.
 *
 */
string ArrayToString(int array[], int lowindex, int highindex) {
  // Special case for empty array
  if (0 > highindex - lowindex) {
    return string("[]");
  }

  // Start the string with the opening '[' and element 0
  string result = "[" + to_string(array[lowindex]);

  // For each remaining element, append comma, space, and element
  for (int i = lowindex+1; i <= highindex; i++) {
    result += ", ";
    result += to_string(array[i]);
  }

  // Add closing ']' and return result
  result += "]";
  return result;
}

/**
 * @brief Print an array of integers.
 *
 */
void printArray(int array[], int lowindex, int highindex) {
  cout << ArrayToString(array, lowindex, highindex) << endl;
}
```

print_array.h:

```
#pragma once

#include <iostream>
using namespace std;

/**
 * @brief Convert an array of integers to a string.
 *
 * @param array Input array.
 * @param lowindex The lowest index of the array to be converted.
 * @param highindex The highest index of the array to be converted.
 * @return string Converted string
 */
string ArrayToString(int array[], int lowindex, int highindex);

/**
 * @brief Print an array of integers.
 *
 */
void printArray(int array[], int lowindex, int highindex);
```

sorting_basic.cpp:

```
/**
 * Implemention of selected sorting algorithms
 * @file sorting.cpp
 */

#include "sorting.h"
#include <iostream>
using namespace std;

/**
 * Implement the insertionSort algorithm correctly
 */
void insertionSort(int array[], int lowindex, int highindex, bool reversed) {
    if (!reversed) {
    for (int i = lowindex+1;i <= highindex;i++) {
      int curindex = i;
      while (curindex > 0 && (array[curindex] < array[curindex - 1])) {
        int temp = array[curindex];
        array[curindex] = array[curindex - 1];
        array[curindex - 1] = temp;
        curindex--;
      }
    }
  }
  else {
    for (int i = lowindex+1;i <= highindex;i++) {
      int curindex = i;
      while (curindex > 0 && (array[curindex] > array[curindex - 1])) {
        int temp = array[curindex];
        array[curindex] = array[curindex - 1];
        array[curindex - 1] = temp;
        curindex--;
      }
    }
  }
}

/**
 * @brief Implementation of the partition function used by quick sort
 *
 */
int partition(int array[], int lowindex, int highindex, bool reversed) {
```

```
    // TODO: Add your code here

  if(!reversed){

    int pivot = array[highindex], i = (lowindex - 1),j;
    for(j = lowindex;j<highindex;j++){

      if(array[j]<=pivot){
        i++;
        swap(array[i],array[j]);
      }

    }

    swap(array[i+1], array[highindex]);

    return(i+1);

  }else if(reversed){

    int pivot = array[highindex], i = (lowindex-1),j;

    for(j=lowindex;j<highindex;j++){

      if(array[j]>=pivot){
        i++;
        swap(array[i],array[j]);
      }

    }

    swap(array[i+1],array[highindex]);

    return (i+1);
  }
}

/**
 * Implement the quickSort algorithm correctly
 */
void quickSort(int array[], int lowindex, int highindex, bool reversed) {
    // TODO: Add your code here

  if(lowindex >= highindex){
```

```
    return;
  }
  int p = partition(array, lowindex, highindex, reversed);
  quickSort(array, lowindex, p-1, reversed);
  quickSort(array, p+1, highindex, reversed);

}
```

sorting_basic.h:

```
/**
* @brief Header file for various sorting functions
*/

#ifndef ASSIGN_3_SORTING_H
#define ASSIGN_3_SORTING_H


/**
 * @brief Insertion sort algorithm
 * @param array Array to be sorted. The array is modified in place.
 * @param lowindex Lowest index of the array
 * @param highindex Highest index of the array
 * @param reversed If reversed = true, the array should be sorted in descending order,
otherwise in ascending order
 */
void insertionSort(int array[], int lowindex, int highindex, bool reversed = false);

/**
 * @brief The partition function used by quick sort
 *
 * @param array Array to be partitioned.
 * @param lowindex lowest index of the array
 * @param highindex highest index of the array
 * @param reversed If reversed = true, the array should be sorted in descending order,
otherwise in ascending order
 * @return int The pivot index
 */
int partition(int array[], int lowindex, int highindex, bool reversed);

/**
 * @brief Quick sort algorithm
 *
 * @param array Array to be sorted. The array is modified in place.
 * @param lowindex Lowest index of the array
 * @param highindex Highest index of the array
 * @param reversed If reversed = true, the array should be sorted in descending order,
otherwise in ascending order
 */
void quickSort(int array[], int lowindex, int highindex, bool reversed = false);
```

```
/**
 * @brief A hybrid of insertion sort and quick sort algorithm. The algorithm is based on the idea that if the array is short, it is better to use insertion sort.
 * It uses quicksort until the list gets small enough, and then uses insertion sort or another sort to sort the small lists
 *
 * @param array The array to be sorted. The array is modified in place.
 * @param lowindex The lowest index of the array
 * @param highindex The highest index of the array
 * @param reversed if reversed = true, the array should be sorted in descending order, otherwise in ascending order
 */
void hybridQuickSort(int array[], int lowindex, int highindex, bool reversed = false);

#endif //ASSIGN_3_SORTING_H
```

main.cpp:

```
/**
 * This driver file tests the basic implementations of insertion and quick sort
 * for arrays
 */

#include "print_array.h"
#include "sorting.h"
#include <chrono>

int main() {

  cout << "**** Run Time Tests ****" << endl;
  int array_size = 100000;
  cout << "Array size: " << array_size << endl;

  int *array1 = new int[array_size];
  int *array2 = new int[array_size];
  int *array3 = new int[array_size];
  int *array4 = new int[array_size];
  int *array5 = new int[array_size];
  int *array6 = new int[array_size];


  for (int i = 0; i < array_size; i++) {
    array1[i] = i;
    array2[i] = array1[i];
  }


  int j = array_size;
  for (int i = 0; i < array_size; i++) {
    array3[i] = j;
    array4[i] = array3[i];
    j--;
  }


  for (int i = 0; i < array_size; i++) {
    array5[i] = rand() % array_size;
    array6[i] = array5[i];
  }
```

```
  // Measure the time for insertion sort
  cout << "Ascending Order:\n";
  //printArray(array1, 0, array_size - 1);
  auto t1 = std::chrono::high_resolution_clock::now();
  insertionSort(array1, 0, array_size - 1, false);
  auto t2 = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double> runtime =
     std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
  std::cout << "Time for insertion sort: " << runtime.count() << " seconds" << std::endl;
  cout << std::endl;

  // Measure the time for quick sort
  t1 = std::chrono::high_resolution_clock::now();
  quickSort(array2, 0, array_size - 1, false);
  t2 = std::chrono::high_resolution_clock::now();
  runtime = std::chrono::duration_cast<std::chrono::duration<double>>(t2 -
  t1); std::cout << "Time for quick sort: " << runtime.count() << " seconds"<< std::endl;
  cout << std::endl;

  // Measure the time for insertion sort
  cout << "Decending Order:\n";
  //printArray(array3, 0, array_size - 1);
  t1 = std::chrono::high_resolution_clock::now();
  insertionSort(array3, 0, array_size - 1, false);
  t2 = std::chrono::high_resolution_clock::now();
  runtime = std::chrono::duration_cast<std::chrono::duration<double>>(t2 -
  t1); std::cout << "Time for insertion sort: " << runtime.count() << "seconds"
          << std::endl;
  cout << std::endl;

  // Measure the time for quick sort
  t1 = std::chrono::high_resolution_clock::now();
  quickSort(array4, 0, array_size - 1, false);
  t2 = std::chrono::high_resolution_clock::now();
  runtime = std::chrono::duration_cast<std::chrono::duration<double>>(t2 -
  t1); std::cout << "Time for quick sort: " << runtime.count() << " seconds"
          << std::endl;
  cout << std::endl;


// Measure the time for insertion sort
cout << "Random Order:\n";
// printArray(array5, 0, array_size - 1);
```

```
 t1 = std::chrono::high_resolution_clock::now();
  insertionSort(array5, 0, array_size - 1, false);
 t2 = std::chrono::high_resolution_clock::now();
runtime = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
  std::cout << "Time for insertion sort: " << runtime.count() << " seconds"
            << std::endl;

  cout << std::endl;

  // Measure the time for quick sort
  t1 = std::chrono::high_resolution_clock::now();
  quickSort(array6, 0, array_size - 1, false);
  t2 = std::chrono::high_resolution_clock::now();
  runtime = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
  std::cout << "Time for quick sort: " << runtime.count() << " seconds"
            << std::endl;


  delete[] array1;
  delete[] array2;
  delete[] array3;
  delete[] array4;
  delete[] array5;
  delete[] array6;

  return 0;
}
```