This report presents the comparison of different sorting algorithms. This report compares and analyzes the results of a focusing on Insertion, Merge, Quick, and Heap Sort.

Sorting Algorithms

HOMEWORK # 2

Ayushya Amitabh
CSC 22000 – ALGORITHMS
PROF. CHI HIM "TIMMY" LIU

Table of Contents

THEORY

Insertion Sort

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Quick Sort

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Heap Sort

The heap sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is O(n log n) and like insertion sort, heap sort sorts in-place. The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . n]. Since the maximum element of the array stored at the root A[1], it can be put into its correct final position by exchanging it with A[n] (the last element in A).

ALGORITHMS IN THIS REPORT

In this report, I will analyze and compare the following sorting algorithms:

1. Straight Insertion Sort
2. Merge Sort
3. Merge Sort with Insertion Sort
4. Quick Sort
5. Quick Sort with Insertion Sort
6. Heap Sort

HYPOTHESIS

I think Heap Sort will be the best sorting algorithm for the bigger arrays but Quick Sort will be better for the smaller arrays. To assess the quality of sorting algorithms we will have to compare the number of swaps made as well the number of comparisons made. So, while Heap Sort may have a significantly greater number of comparisons I think that it will be more efficient by being able to minimize the number of swaps made.

Continues on next page…

STUDYING INDIVIDUAL ARRAY SIZES

Arrays of sizes 10^2, 10^3, and 10^4 were run 10 times each. Array of size 10^6 was allowed to run for 475 hours and only completed 3 cycles through each of the sorting algorithms. Labels for the table:

1. Straight → Straight Insertion Sort
2. Merge → Merge Sort
3. Merge W/Str. → Merge Sort with Insertion Sort
4. Quick → Quick Sort
5. Quick W/Str. → Quick Sort with Insertion Sort
6. Heap → Heap Sort

The values in the right most column show the best algorithm for each case.

Array of Size 10^2

2	SWAPS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	2441	542.2	2580.6	180.4	174.3	583	QUICK W/STR.
	1 TO N	0	316	148	63	20	641	STRAIGHT
	N TO 1	4950	356	5098	112	70	517	QUICK W/STR.
	COMPARISONS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	4973	1512.2	5566.4	960	2122.8	1899	QUICK
	1 TO N	99	550	708	732	1624	2073	STRAIGHT
	N TO 1	9801	1326	10410	687	1623	1701	QUICK
TOTAL =		7414	2054.4	8147	1140.4	2297.1	2482	QUICK
		99	866	856	795	1644	2714	STRAIGHT
		14751	1682	15508	799	1693	2218	QUICK

Array of Size 10^3

3	SWAPS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	248001.3	8708.6	254574.4	2588.6	2530.9	9091	QUICK W/STR.
	1 TO N	0	4932	2980	511	255	9709	STRAIGHT
	N TO 1	499500	5044	502480	1010	754	8317	QUICK W/STR.
	COMPARISONS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	496988	21682.6	528738.9	13352.3	144782.2	28773	QUICK
	1 TO N	999	17906	25562	10031	136606	30627	STRAIGHT
	N TO 1	998001	18018	1022564	9539	136137	26451	QUICK
TOTAL =		744989.3	30391.2	783313.3	15940.9	147313.1	37864	QUICK
		999	22838	28542	10542	136861	40336	STRAIGHT
		1497501	23062	1525044	10549	136891	34768	QUICK

Array of Size 10^4

4								
	SWAPS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	25087710.8	120452.1	24979440.5	33866.2	32932.1	124185	QUICK W/STR.
	1 TO N	0	64608	44848	5904	2047	131957	STRAIGHT
	N TO 1	49995000	69008	50039848	10904	7046	116697	QUICK W/STR.
	COMPARISONS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	50185404.4	284066.1	51289659.4	166464.3	13404804.3	387555	QUICK
	1 TO N	9999	228222	1420494	137247	10355329	410871	STRAIGHT
	N TO 1	99980001	232622	101390496	132263	10350411	365091	QUICK
TOTAL =		75273115.2	404518.2	76269099.9	200330.5	13437736.4	511740	QUICK
		9999	292830	1465342	143151	10357376	542828	STRAIGHT
		149975001	301630	151430344	143167	10357457	481788	QUICK

Array of Size 10^6

6								
	SWAPS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	896248957	18674306	1052879814	5607350	4841748	19047369	QUICK W/STR.
	1 TO N	0	9884992	7982144	524287	262143	19787793	STRAIGHT
	N TO 1	1783293664	10066432	1791275808	1024286	762142	18333409	QUICK W/STR.
	COMPARISONS MADE							
		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	1793497881	41625728	1318463387	21285507	1480606936	58642107	QUICK
	1 TO N	999999	32836414	771326786	20048593	2053632868	60863379	STRAIGHT
	N TO 1	729379967	33017854	1501706752	19548611	2054130535	56500227	QUICK
TOTAL =		2689746838	60300034	2371343201	26892857	1485448684	77689476	QUICK
		999999	42721406	779308930	20572880	2053895011	80651172	STRAIGHT
		2512673631	43084286	3292982560	20572897	2054892677	74833636	QUICK

STUDYING OVERALL ALGORITHM PERFORMANCES

Overall		STRAIGHT	MERGE	MERGE W/STR.	QUICK	QUICK W/STR.	HEAP	
	RANDOM	691443089.1	15184249.5	612100940.3	6777567.2	374759007.7	19560390.5	QUICK
	1 TO N	252774	10759485	195200917.5	5181842	516097723	20309262.5	STRAIGHT
	N TO 1	666040221	10852665	861488364	5181853	516347179.5	18838102.5	QUICK

ANALYSIS VS HYPOTHESIS

My hypothesis was incorrect – Heap Sort was not the best sorting algorithms. Quick Sort showed the best performance overall. Quick Sort with Insertion Sort consistently showed best results for numbers of swaps made while simple Quick Sort showed best results for the number of comparisons made. I think the fact the array needs to be heapified after every element is popped – this may lend itself to my implementation of the comparison counter rather than the performance of the algorithm itself.

Continues on next page…

LOOKING AT THE CODE

Swap Element Function

```
void swapElement(int *_array, int _currentPos, int _newPos) {
	int _storedValue = _array[_newPos];       //Saves value of the new position
	_array[_newPos] = _array[_currentPos];    //Put value into new position
	_array[_currentPos] = _storedValue;       //Puts saved value into old position
}
```

Straight Insertion Sort

```
void straightInsertionSort(int *_array, int _sizeOfArray) {
	//*_array is the array being passed to the function
	//_sizeOfArray is the size of the array being passed
	int _currentPointer=0; //Used to back track through the sub-array of sorted elements
	for (int follow = 1; follow < _sizeOfArray; follow++) {
		COMPARISON_COUNT++;  //Comparison is made... increment by one
		//IF ELEMENTS IS LESS THAN ELEMENT BEFORE
		if (_array[follow] < _array[follow - 1]) {
			SWAP_COUNT++; //Swap is made... increment by one
			swapElement(_array, follow, follow - 1);
			_currentPointer = follow - 1;
			//GOING THROUGH SORTED SUB-ARRAY
			while (_currentPointer > 0) {
				COMPARISON_COUNT += 2;
				//IF SMALLER ELEMENT IS FOUND
				if (_array[_currentPointer] < _array[_currentPointer - 1]) {
					SWAP_COUNT++; //Swap is made... increment by one
					swapElement(_array, _currentPointer, _currentPointer - 1);
					_currentPointer--;
				}
				else {
					_currentPointer = 0;
				}
			}

		}
	}
}
```

Merge Sort

```
void merge(int *_array, int _left, int _middle, int _right) {
	//*_array is the array being passed to the function
	//_left is the left end of the array being passed
	//_middle is the middle point of the array being passed
	//_right is the right end of the array being passed
	int _leftSubArraySize = _middle - _left + 1, _rightSubArraySize = _right - _middle;
	//  ^ Size of the left half ---------------  ^ Size of the right half after dividing
	int *_leftSubArray, *_rightSubArray;
	//   ^ Left array    ^ Right Array
	_leftSubArray = new int[_leftSubArraySize]; // Creating left sub array
	_rightSubArray = new int[_rightSubArraySize]; // Creating right sub array

	for (int i = 0; i < _leftSubArraySize; i++) { // Left sub-array
		_leftSubArray[i] = _array[_left + i];
	}
	for (int j = 0; j < _rightSubArraySize; j++) { // Right sub-array
		_rightSubArray[j] = _array[_middle + 1 + j];
	}
	int _leftCursor=0, _rightCursor=0, _mergedCursor=_left;
	// ^^ Cursors of the sub-arrays
	// Compares and merges the two sub-arrays
	while (_leftCursor < _leftSubArraySize && _rightCursor < _rightSubArraySize) {
		COMPARISON_COUNT += 2; //Comparisons are made... increment by two
		// IF FIRST ELEMENT OF LEFT SUB ARRAY IS LESS THAN RIGHT SUB ARRAY
		if (_leftSubArray[_leftCursor] < _rightSubArray[_rightCursor]) {
			_array[_mergedCursor] = _leftSubArray[_leftCursor];
			_leftCursor++;
			SWAP_COUNT++; //Swap is made... increment by one
		}
		else {
			_array[_mergedCursor] = _rightSubArray[_rightCursor];
			_rightCursor++;
			SWAP_COUNT++;
		}
		_mergedCursor++;
	}
	// INSERTS REMAINING ELEMENTS OF LEFT SUB ARRAY
	while (_leftCursor < _leftSubArraySize)  {
		COMPARISON_COUNT++; //Comparison is made... increment by one
		_array[_mergedCursor] = _leftSubArray[_leftCursor];
		_leftCursor++;
		_mergedCursor++;
	}
	// INSERTS REMAINING ELEMENTS OF RIGHT SUB ARRAY
	while (_rightCursor < _rightSubArraySize) {
		COMPARISON_COUNT++; //Comparison is made... increment by one
		_array[_mergedCursor] = _rightSubArray[_rightCursor];
		_rightCursor++;
		_mergedCursor++;
	}
}
```

Above is the sub method of Merge Sort – the merge function works from bottom up, combining smaller arrays to form the sorted array.

```
void mergeSort(int *_array, int _left, int _right, bool withIndexed = false) {
	//          ^ ARRAY      |^LEFT END |^RIGHT END|      ^ WITH INSERTION SORT
	COMPARISON_COUNT++; //Comparison is made... increment by one
	if (_left < _right) {
		int _middle = (_left + (_right - 1)) / 2;
		COMPARISON_COUNT++; //Comparison is made... increment by one
		if ((_right - _left) < 40 && withIndexed) {
			// WITH INSERTION SORT
			straightInsertionSort(_array, _right + 1);
		}
		else {
			// RECURSIVE MERGE SORT CALL
			mergeSort(_array, _left, _middle, withIndexed);
			mergeSort(_array, _middle + 1, _right, withIndexed);
		}
		merge(_array, _left, _middle, _right);
	}
}
```

Quick Sort

```
void quickSort(int *_array, int left, int right, bool withInsertion = false) {
	//          ^ ARRAY      |^LEFT END |^RIGHT END|      ^ WITH INSERTION SORT
	COMPARISON_COUNT++; // Comparison is made... increment by one
	if (right - left <= 4 && withInsertion) { // WITH INSERTION SORT
		straightInsertionSort(_array, right+1);
	}
	else {
		int i = left, j = right;
		int tmp;
		int pivot = _array[(left + right) / 2];
		while (i <= j) {
			while (_array[i] < pivot) {
				i++;
				COMPARISON_COUNT++; // Comparison is made... increment by one
			}
			while (_array[j] > pivot) {
				j--;
				COMPARISON_COUNT++; // Comparison is made... increment by one
			}
			if (i <= j) { // IF LEFT CURSOR HASN'T CROSSED OVER RIGHT CURSOR
				tmp = _array[i]; // SWAP ELEMENTS...
				_array[i] = _array[j];
				_array[j] = tmp;
				i++; // INCREMENT & DECREMENT LEFT AND RIGHT CURSORS...
				j--;
				SWAP_COUNT++; // Swap is made... increment by one
			}
			COMPARISON_COUNT++; // Comparison is made... increment by one
		}
		COMPARISON_COUNT += 2; // Comparisons are made... increment by two
		// RECURSIVE QUICK SORT CALL
		if (left < j) quickSort(_array, left, j, withInsertion);
		if (i < right) quickSort(_array, i, right, withInsertion);
	}
}
```

Heap Sort

```
void heapify(int *_array, int n, int i) {
       //            ^ ARRAY     |^SIZE |^LEFT
       int largest = i;
       int l = 2 * i + 1;
       int r = 2 * i + 2;
       COMPARISON_COUNT += 3; // Comparisons are made... increment by three
       if (l < n && _array[l] > _array[largest]) largest = l;
       if (r < n && _array[r] > _array[largest]) largest = r;
       if (largest != i) {
              SWAP_COUNT++; // Swap is made... increment by one
              swap(_array[i], _array[largest]);
              heapify(_array, n, largest);
       }
}
void heapSort(int *_array, int n) {
       //            ^ ARRAY     | ^ SIZE OF ARRAY
       for (int i = n / 2 - 1; i >= 0; i--) heapify(_array, n, i);
       // ^^ HEAPIFY LEFT HALF OF ARRAY
       for (int i = n - 1; i >= 0; i--) {
              SWAP_COUNT++; // Swap is made... increment by one
              swap(_array[0], _array[i]);
              heapify(_array, i, 0); // HEAPIFY RIGHT HALF ARRAY
       }
}
```

TIME COMPLEXITIES

	BEST CASE	AVERAGE CASE	WORST CASE
STRAIGHT INSERTION	N	N^2	N^2
MERGE SORT	N log N	N log N	N log N
QUICK SORT	N log N	N log N	N^2
HEAP SORT	N log N	N log N	N log N

WITH STRAIGHT INSERTION SORT…

Merge Sort vs. Insertion Sort

	ARRAY	MERGE	INSERTION
40	1 to N	864	39
	N to 1	864	3198
	Random	864	1386
Average	-	864	1541
100	1 to N	2688	99
	N to 1	2688	19998
	Random	2688	9299
1000	1 to N	39904	999
	N to 1	39904	1999998
	Random	39904	1008352

Quick Sort vs. Insertion Sort

TYPE	SIZE	QUICK	INSERTION
1 TO N	80	288	79
	50	186	49
	40	144	39
RANDOM	100	1110	11272
	10	78	127
	4	18	14
N TO 1	40	264	3198
	10	66	198
	4	24	30

Continues on next page…

APPENDIX : MAIN.CPP

```cpp
/*
S O R T I N G      A L G O R I T H M S
Coded By: Ayushya Amitabh
HOMEWORK # 2
-------------------------
CSC 22000 - ALGORITHMS
PROF. CHI HIM "TIMMY" LIU
*/

#include <iostream>
#include <time.h>
#include <fstream>
#include <string>
#include <algorithm>

using namespace std;
ofstream file;
int COMPARISON_COUNT, SWAP_COUNT;

void swapElement(int *_array, int _currentPos, int _newPos) {
    int _storedValue = _array[_newPos]; //Saves value of the new position
    _array[_newPos] = _array[_currentPos];  //Put value into new position
    _array[_currentPos] = _storedValue; //Puts saved value into old position
}

void straightInsertionSort(int *_array, int _sizeOfArray) {
    //*_array is the array being passed to the function
    //_sizeOfArray is the size of the array being passed
    int _currentPointer=0; //Used to back track through the sub-array of sorted elements
    for (int follow = 1; follow < _sizeOfArray; follow++) {
        COMPARISON_COUNT++; //Comparison is made... increment by one
        //IF ELEMENTS IS LESS THAN ELEMENT BEFORE
        if (_array[follow] < _array[follow - 1]) {
            SWAP_COUNT++; //Swap is made... increment by one
            swapElement(_array, follow, follow - 1);
            _currentPointer = follow - 1;
            //GOING THROUGH SORTED SUB-ARRAY
            while (_currentPointer > 0) {
                COMPARISON_COUNT += 2;
                //IF SMALLER ELEMENT IS FOUND
                if (_array[_currentPointer] < _array[_currentPointer - 1]) {
                    SWAP_COUNT++; //Swap is made... increment by one
                    swapElement(_array, _currentPointer, _currentPointer - 1);
                    _currentPointer--;
                }
                else {
                    _currentPointer = 0;
                }
            }

        }
```

```
    }
}

void merge(int *_array, int _left, int _middle, int _right) {
    //*_array is the array being passed to the function
    //_left is the left end of the array being passed
    //_middle is the middle point of the array being passed
    //_right is the right end of the array being passed
    int _leftSubArraySize = _middle - _left + 1, _rightSubArraySize = _right - _middle;
    //  ^ Size of the left half ---------------  ^ Size of the right half after dividing
    int *_leftSubArray, *_rightSubArray;
    //   ^ Left array    ^ Right Array
    _leftSubArray = new int[_leftSubArraySize]; // Creating left sub array
    _rightSubArray = new int[_rightSubArraySize]; // Creating right sub array

    for (int i = 0; i < _leftSubArraySize; i++) { // Left sub-array
        _leftSubArray[i] = _array[_left + i];
    }
    for (int j = 0; j < _rightSubArraySize; j++) { // Right sub-array
        _rightSubArray[j] = _array[_middle + 1 + j];
    }
    int _leftCursor=0, _rightCursor=0, _mergedCursor=_left;
    // ^^ Cursors of the sub-arrays
    // Compares and merges the two sub-arrays
    while (_leftCursor < _leftSubArraySize && _rightCursor < _rightSubArraySize) {
        COMPARISON_COUNT += 2; //Comparisons are made... increment by two
        // IF FIRST ELEMENT OF LEFT SUB ARRAY IS LESS THAN RIGHT SUB ARRAY
        if (_leftSubArray[_leftCursor] < _rightSubArray[_rightCursor]) {
            _array[_mergedCursor] = _leftSubArray[_leftCursor];
            _leftCursor++;
            SWAP_COUNT++; //Swap is made... increment by one
        }
        else {
            _array[_mergedCursor] = _rightSubArray[_rightCursor];
            _rightCursor++;
            SWAP_COUNT++;
        }
        _mergedCursor++;
    }
    // INSERTS REMAINING ELEMENTS OF LEFT SUB ARRAY
    while (_leftCursor < _leftSubArraySize) {
        COMPARISON_COUNT++; //Comparison is made... increment by one
        _array[_mergedCursor] = _leftSubArray[_leftCursor];
        _leftCursor++;
        _mergedCursor++;
    }
    // INSERTS REMAINING ELEMENTS OF RIGHT SUB ARRAY
    while (_rightCursor < _rightSubArraySize) {
        COMPARISON_COUNT++; //Comparison is made... increment by one
```

```
        _array[_mergedCursor] = _rightSubArray[_rightCursor];
        _rightCursor++;
        _mergedCursor++;
    }
}

void mergeSort(int *_array, int _left, int _right, bool withIndexed = false) {
    //         ^ ARRAY     |^LEFT END |^RIGHT END|     ^ WITH INSERTION SORT
    COMPARISON_COUNT++; //Comparison is made... increment by one
    if (_left < _right) {
        int _middle = (_left + (_right - 1)) / 2;
        COMPARISON_COUNT++; //Comparison is made... increment by one
        if ((_right - _left) < 40 && withIndexed) {
            // WITH INSERTION SORT
            straightInsertionSort(_array, _right + 1);
        }
        else {
            // RECURSIVE MERGE SORT CALL
            mergeSort(_array, _left, _middle, withIndexed);
            mergeSort(_array, _middle + 1, _right, withIndexed);
        }
        merge(_array, _left, _middle, _right);
    }
}

void quickSort(int *_array, int left, int right, bool withInsertion = false) {
    //         ^ ARRAY     |^LEFT END |^RIGHT END|     ^ WITH INSERTION SORT
    COMPARISON_COUNT++; // Comparison is made... increment by one
    if (right - left <= 4 && withInsertion) { // WITH INSERTION SORT
        straightInsertionSort(_array, right+1);
    }
    else {
        int i = left, j = right;
        int tmp;
        int pivot = _array[(left + right) / 2];
        while (i <= j) {
            while (_array[i] < pivot) {
                i++;
                COMPARISON_COUNT++; // Comparison is made... increment by one
            }
            while (_array[j] > pivot) {
                j--;
                COMPARISON_COUNT++; // Comparison is made... increment by one
            }
            if (i <= j) { // IF LEFT CURSOR HASN'T CROSSED OVER RIGHT CURSOR
                tmp = _array[i]; // SWAP ELEMENTS...
                _array[i] = _array[j];
                _array[j] = tmp;
                i++; // INCREMENT & DECREMENT LEFT AND RIGHT CURSORS...
                j--;
                SWAP_COUNT++; // Swap is made... increment by one
            }
```

```cpp
            COMPARISON_COUNT++; // Comparison is made... increment by one
        }
        COMPARISON_COUNT += 2; // Comparisons are made... increment by two
        // RECURSIVE QUICK SORT CALL
        if (left < j) quickSort(_array, left, j, withInsertion);
        if (i < right) quickSort(_array, i, right, withInsertion);
    }
}

void heapify(int *_array, int n, int i) {
    //         ^ ARRAY    |^SIZE |^LEFT
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    COMPARISON_COUNT += 3; // Comparisons are made... increment by three
    if (l < n && _array[l] > _array[largest]) largest = l;
    if (r < n && _array[r] > _array[largest]) largest = r;
    if (largest != i) {
        SWAP_COUNT++; // Swap is made... increment by one
        swap(_array[i], _array[largest]);
        heapify(_array, n, largest);
    }
}
void heapSort(int *_array, int n) {
    //         ^ ARRAY    | ^ SIZE OF ARRAY
    for (int i = n / 2 - 1; i >= 0; i--) heapify(_array, n, i);
    // ^^ HEAPIFY LEFT HALF OF ARRAY
    for (int i = n - 1; i >= 0; i--) {
        SWAP_COUNT++; // Swap is made... increment by one
        swap(_array[0], _array[i]);
        heapify(_array, i, 0); // HEAPIFY RIGHT HALF ARRAY
    }
}

void cycle(int exp) {
    int _arraySize, _sizeExp, *_searchThroughArray, *_1to_N, *_N_to1;

    _sizeExp = exp;
    _arraySize = pow(10, _sizeExp);

    //Initializing Arrays
    _searchThroughArray = new int[_arraySize]; //Array of Random Integers
    _1to_N = new int[_arraySize];   //Ascending array from 1 to n
    _N_to1 = new int[_arraySize]; //Descending array from n to 1

    // STRAIGHT INSERTION SORT ============================================
    cout << "\n\nSTRAIGHT INSERTION SORT ============================================\n";
    file << "\n\nSTRAIGHT INSERTION SORT ============================================\n";

    for (int i = 0, j = _arraySize; i < _arraySize; i++, j--) {
```

```
        _searchThroughArray[i] = rand() % 9 + 1 * rand() + rand();
        _1to_N[i] = i + 1;
        _N_to1[i] = j;
    }

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    straightInsertionSort(_searchThroughArray, _arraySize);

    file << "\n::::: RANDOM :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    straightInsertionSort(_1to_N, _arraySize);

    file << "\n::::: 1 TO N :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    straightInsertionSort(_N_to1, _arraySize);

    file << "\n::::: N TO 1 :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    // MERGE SORT ==========================================================
    cout << "\n\nMERGE SORT ==========================================================\n";

    file << "\n\nMERGE SORT ==========================================================\n";

    for (int i = 0, j = _arraySize; i < _arraySize; i++, j--) {
        _searchThroughArray[i] = rand() % 9 + 1 * rand() + rand();
        _1to_N[i] = i + 1;
        _N_to1[i] = j;
    }

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    mergeSort(_searchThroughArray, 0, _arraySize - 1);

    file << "\n::::: RANDOM :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
```

```cpp
    mergeSort(_1to_N, 0, _arraySize - 1);

    file << "\n::::: 1 TO N :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    mergeSort(_N_to1, 0, _arraySize - 1);

    file << "\n::::: N TO 1 :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    // MERGE SORT WITH INSERTION SORT ======================================
    cout << "\n\nMERGE SORT WITH INSERTION SORT ======================================\n";

    file << "\n\nMERGE SORT WITH INSERTION SORT ======================================\n";

    for (int i = 0, j = _arraySize; i < _arraySize; i++, j--) {
        _searchThroughArray[i] = rand() % 9 + 1 * rand() + rand();
        _1to_N[i] = i + 1;
        _N_to1[i] = j;
    }

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    mergeSort(_searchThroughArray, 0, _arraySize - 1, true);

    file << "\n::::: RANDOM :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    mergeSort(_1to_N, 0, _arraySize - 1, true);

    file << "\n::::: 1 TO N :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    mergeSort(_N_to1, 0, _arraySize - 1, true);

    file << "\n::::: N TO 1 :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    // QUICK SORT ==========================================================
```

```
    cout << "\n\nQUICK SORT
      ==========================================================\n";

    file << "\n\nQUICK SORT
      ==========================================================\n";

    for (int i = 0, j = _arraySize; i < _arraySize; i++, j--) {
        _searchThroughArray[i] = rand() % 9 + 1 * rand() + rand();
        _1to_N[i] = i + 1;
        _N_to1[i] = j;
    }

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    quickSort(_searchThroughArray, 0, _arraySize - 1);

    file << "\n::::: RANDOM :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    quickSort(_1to_N, 0, _arraySize - 1);

    file << "\n::::: 1 TO N :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    quickSort(_N_to1, 0, _arraySize - 1);

    file << "\n::::: N TO 1 :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    // QUICK SORT WITH INSERTION SORT ======================================
    cout << "\n\nQUICK SORT WITH INSERTION SORT
      ======================================\n";

    file << "\n\nQUICK SORT WITH INSERTION SORT
      ======================================\n";

    for (int i = 0, j = _arraySize; i < _arraySize; i++, j--) {
        _searchThroughArray[i] = rand() % 9 + 1 * rand() + rand();
        _1to_N[i] = i + 1;
        _N_to1[i] = j;
    }

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    quickSort(_searchThroughArray, 0, _arraySize - 1, true);
```

```
    file << "\n::::: RANDOM :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    quickSort(_1to_N, 0, _arraySize - 1, true);

    file << "\n::::: 1 TO N :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    quickSort(_N_to1, 0, _arraySize - 1, true);

    file << "\n::::: N TO 1 :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    // HEAP SORT ===========================================================
    cout << "\n\nHEAP SORT ===========================================================\n";

    file << "\n\nHEAP SORT ===========================================================\n";

    for (int i = 0, j = _arraySize; i < _arraySize; i++, j--) {
        _searchThroughArray[i] = rand() % 9 + 1 * rand() + rand();
        _1to_N[i] = i + 1;
        _N_to1[i] = j;
    }

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    heapSort(_searchThroughArray, _arraySize);

    file << "\n::::: RANDOM :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
    heapSort(_1to_N, _arraySize);

    file << "\n::::: 1 TO N :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;

    COMPARISON_COUNT = 0;
    SWAP_COUNT = 0;
```

```cpp
    heapSort(_N_to1, _arraySize);

    file << "\n::::: N TO 1 :::::\n"
        << "Comparisons made = " << COMPARISON_COUNT
        << "\nSwaps made = " << SWAP_COUNT;
}

int main() {
    srand(time(NULL));
    int _sizes[3] = {3};

    for (int i = 0; i < 1; i++) {
        string _fileName = to_string(_sizes[i]) + ".txt";
        file.open(_fileName);
        for (int j = 0; j < 10; j++) {
            file <<
              "\n================================================================
              =========\n RUN "
                << j <<
                  "\n=============================================================
                  =============\n";
            cout << "\nRunning:\n\tSize : " << i << "\n\tRun : " << j << endl;
            cycle(_sizes[i]);
        }
        file.close();
    }

    return 0;
}
```