**CSCE 2202 S20 Term Project Report**

**Experimental Verification and Analysis**

**of Some Sorting Algorithms**

**Nada El-Shenawy**

**ID:900182782**

**Introduction**

There are various sorting algorithms that have different time complexities, stability and space complexities. Different input data set types and sizes may work better with different algorithms. The soring functionality is used in day-to-day life and is also needed as a part of other algorithms. Therefore, it is important to find efficient sorting algorithms.

**Problem Definition**

Given a random permutation integer array of different sizes, the purpose of each of the 5 algorithms tested is to sort the array in ascending order.

**Sorting Algorithms Tested**

The five sorting algorithms tested are selection sort, merge sort, quick sort with the pivot as the first element of the array, quick sort with the pivot as the middle element of the array and lastly quick sort with a randomly chosen pivot. The brute force approach of selection sort is used, along with divide and conquer approaches for the other methods.

**Input Data Specifications**

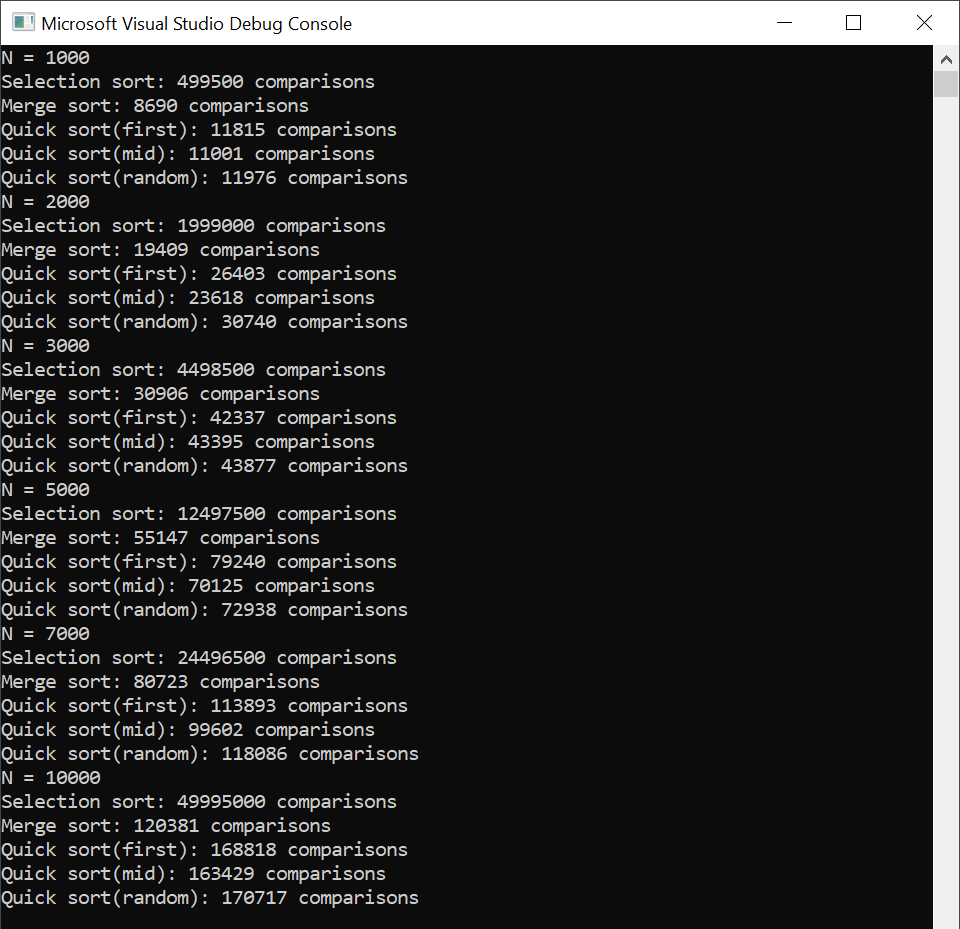
The inputs to the program are n = 1000, 2000, 3000, 5000, 7000 and 10000. For each input, a random permutation integer array of size n+1 is generated. Each array contains numbers from 1 to n inclusive.

**Methodology**

Sorting algorithms work differently with different data set types. For example, a random array would be sorted faster than an array that is in descending order. Therefore, for each of the data set sizes, the random permutation integer array generated is then copied into 5 other arrays. The 5 arrays are used to test the five algorithms with the same data in order to create a more accurate comparison. The comparison is then made by analyzing the number of array element comparisons done by each method.

**Experimental Results**

The screenshot below shows the number of comparisons made by each of the 5 algorithms when given each of the 6 array sizes N.



**Mathematical Modelling**

**Selection Sort**

**Merge Sort**

**Quick sort (pivot at first element)**

**Quick sort (pivot at middle element)**

**Randomized Quick Sort**

**Comparing Measured and Calculated Results**

**Table 1. Selection Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | 1000 | 2000 | 3000 | 5000 | 7000 | 10000 |
| T(n) exp | 499500 | 1999000 | 4498500 | 12497500 | 24496500 | 49995000 |
| T(n) model | 500499 | 2000999 | 4501499 | 12502499 | 24503499 | 50004999 |

**Table 2. Merge Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | 1000 | 2000 | 3000 | 5000 | 7000 | 10000 |
| T(n) exp | 8690 | 19409 | 30906 | 55147 | 80723 | 120381 |
| T(n) model | 9966 | 21932 | 34652 | 61439 | 89412 | 132877 |

**Table 3. Quick Sort (pivot at first element)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | 1000 | 2000 | 3000 | 5000 | 7000 | 10000 |
| T(n) exp | 11815 | 26403 | 42337 | 79240 | 113893 | 168818 |
| T(n) model | 500500 | 2001000 | 4501500 | 12502500 | 24503500 | 50005000 |

**Table 4. Quick Sort (pivot at middle element)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | 1000 | 2000 | 3000 | 5000 | 7000 | 10000 |
| T(n) exp | 11001 | 23618 | 43395 | 70125 | 99602 | 163429 |
| T(n) model | 9966 | 21932 | 34652 | 61439 | 89412 | 132877 |

**Table 5. Randomized Quick Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| n | 1000 | 2000 | 3000 | 5000 | 7000 | 10000 |
| T(n) exp | 11976 | 30740 | 43877 | 72938 | 118086 | 170717 |
| T(n) model | 9966 | 21932 | 34652 | 61439 | 89412 | 132877 |

**Conclusions**

The conclusions that can be drawn from the above analysis and comparison is that selection sort and quick sort with the pivot at the first element in the array would result in the largest complexity (quadratic), while the other three algorithms (merge sort, quick sort with pivot at the middle of the array and randomized quick sort) have linear-logarithmic complexities. Therefore, those 3 algorithms are more preferred as they are more efficient.

**Appendix**

int selectionSort(int \*arr, int size, int &comp) {

int m = 0;

for (int i = 1; i < size-1; i++) {

m = i;

for (int j = i + 1; j < size; j++) {

comp++;

if (arr[j] < arr[m]) m = j;

}

swap(arr[i], arr[m]);

}

return comp;

}

void merge(int \*arr, int n, int a, int mid, int b, int &comp) {

int \*B = new int[n+1];

for (int i = 1; i <= n; i++)B[i] = arr[i];

int i = a, j = mid + 1, k = a;

while (i <= mid && j <= b) {

comp++;

if (B[i] <= B[j]) { arr[k] = B[i]; i++; }

else { arr[k] = B[j]; j++; }

k++;

}

while (i <= mid){

arr[k] = B[i]; i++; k++;

}

while (j <= b) {

arr[k] = B[j]; j++; k++;

}

}

int mergeSort(int \*arr, int size, int a, int b, int &comp) {

if (a < b) {

int mid = floor((a + b) / 2);

mergeSort(arr, size, a, mid, comp);

mergeSort(arr, size, mid + 1, b, comp);

merge(arr, size, a, mid, b, comp);

}

return comp;

}

int partitionA(int \*arr, int a, int b, int &comp) {

int pivot = arr[a];

int i = a, j = b;

do {

do { i++; comp++; } while (arr[i] < pivot);

do { j--; comp++; } while (arr[j] > pivot);

if (i < j)swap(arr[i], arr[j]);

} while (i < j);

arr[a] = arr[j];

arr[j] = pivot;

return j;

}

int quickSortA(int \*arr, int a, int b, int &comp) {

if (a < b) {

int x = partitionA(arr, a, b + 1, comp);

quickSortA(arr, a, x - 1, comp);

quickSortA(arr, x + 1, b, comp);

}

return comp;

}

int partitionB(int \*arr, int a, int b, int &comp) {

int p = (a + b) / 2, pivot = arr[(a + b) / 2];

int i = a, j = b;

do {

while (arr[i] < pivot) { i++; comp++; }

while (arr[j] > pivot) { j--; comp++; }

if (i < j)swap(arr[i], arr[j]);

} while (i < j);

return j;

}

int quickSortB(int \*arr, int a, int b, int &comp) {

if (a < b) {

int x = partitionB(arr, a, b, comp);

quickSortB(arr, a, x - 1, comp);

quickSortB(arr, x + 1, b, comp);

}

return comp;

}

int RQuickSort(int \*arr, int p, int r, int &comp) {

if (p < r) {

if ((r - p) > 5) {

int m = rand() % (r - p + 1) + p; swap(arr[p], arr[m]); }

int q = partitionA(arr, p, r + 1, comp);

RQuickSort(arr, p, q - 1, comp);

RQuickSort(arr, q + 1, r, comp);

}

return comp;

}