Department of Computer Science and Engineering, AUC
CSCE 2202 - Term Project Report
Spring 2020

Experimental Verification and Analysis of Some Sorting Algorithms

Samah Ayman - 900171848
Marwah Hisham - 900172284
Sara Sherif - 900172348

Abstract

Sorting algorithms has been of the most important useful tools used in the development of many modern technologies. The need for optimization and enhancement became vital to many industries that depend on the usage of large data structure sizes. In this project, we explore the various sorting algorithms. The aim of this project is to implement, test, and verify the already existing various sorting algorithms. The presented data is compared in terms of the time of complexity taken to accomplish a specifically defined operation, which is a function of the number of array element comparisons. The mathematical model is compared to the experimental data for the purpose of verification.

Problem definition

Our problem is to compare between experimental results $T_{exp}(N)$ using counting of the number of array element comparisons done by a sorting algorithm with those expected from mathematical modeling of the algorithm $T_{model}(N)$. The obtained data is presented in a tabular as well as a graphical form. The mathematical model refers to the recurrence relation of all sorting algorithms presented. And the experimental data are obtained via coding and testing on a number of array sizes.

The sorting algorithms to be tested

A. Selection Sort: Input is a Random Permutation Array A[1..N]
B. Merge Sort: Input is a Random Permutation Array A[1..N]
C. Quick Sort 1: Input is a Random Permutation Array A[1..N], pivot is chosen to be the 1st element in the array / sub-array.
   Quick Sort 2: Input is a Random Permutation Array A[1..N], pivot is chosen to be the approximate middle of the array / sub-array.
D. Randomized Quick Sort: Input is a Random Permutation Array A[1..N]

Input data specifications

In our experiments, we will use Random Permutation Arrays (RPA) of the first N integers. Such arrays are often used in testing sorting algorithms. In these arrays, all integers from 1 to N are present but in a random order. No integer is missing and no integer is duplicated. The RPA are stored in arrays of size (N+1), starting from location 1 and ending at location N. Location (0) is left unused to be used as a correctness flag. In each experiment, 6 different array sizes (N = 1000, 2000, 3000, 5000, 7000, 10000) are used in both the mathematical model and the code testing.

Experimental Results

1. Selection sort

The mathematical model for T(n) as provided in the lecture slides is:

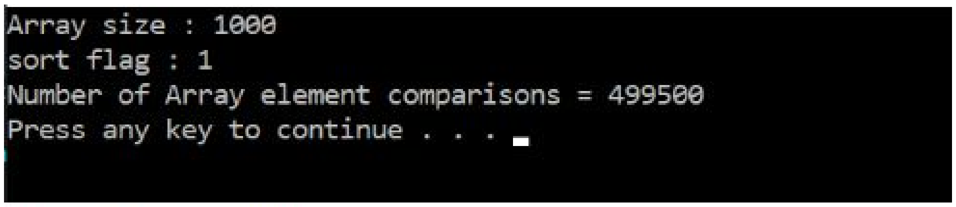
$$T_{swap}(n) = \sum_{i=0}^{n-2} 1_{swap} = (n-1)$$

$$T_{comp}(n) = \sum_{i=0}^{n-2} (n-1-i)_{comp} = \frac{n(n-1)}{2}$$

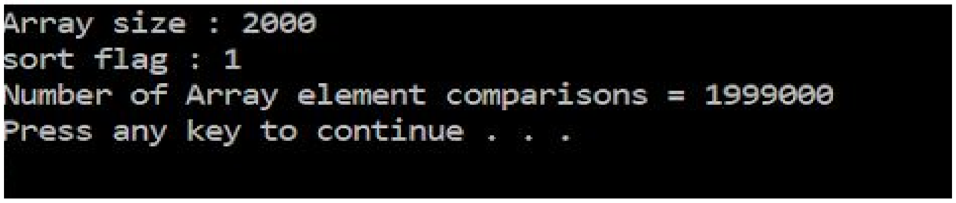
$$T(n) = 0.5n^2 + 0.5n - 1 = \Theta(n^2)$$

N	1000	2000	3000	5000	7000	10000
T(n) Model	499500	1999000	4498500	12497500	24496500	49995000
T(n) Experimental	499500	1999000	4498500	12497500	24496500	49995000

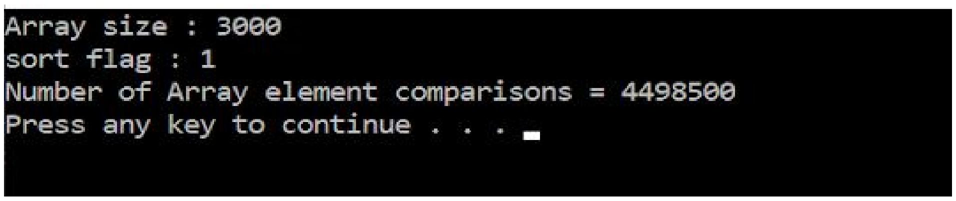
Screenshots of the experimental results:



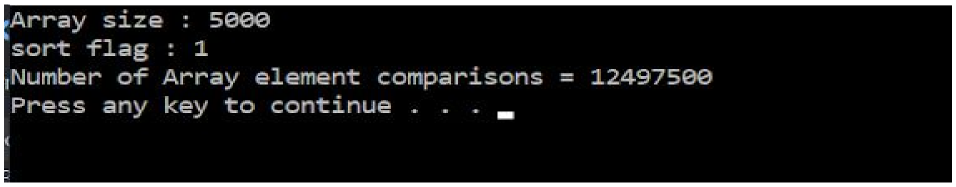
```
Array size : 1000
sort flag : 1
Number of Array element comparisons = 499500
Press any key to continue . . .
```



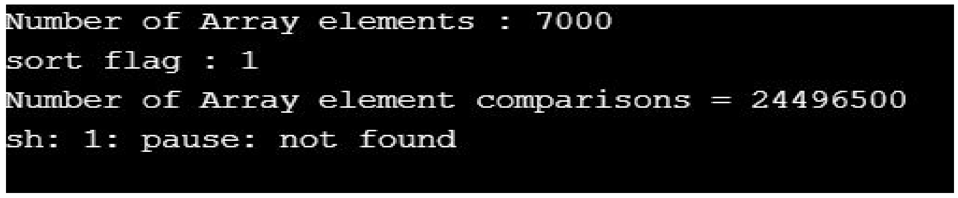
```
Array size : 2000
sort flag : 1
Number of Array element comparisons = 1999000
Press any key to continue . . .
```



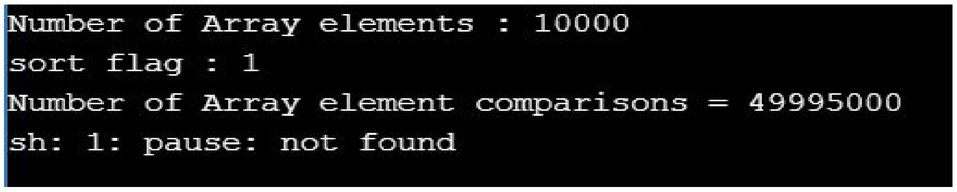
```
Array size : 3000
sort flag : 1
Number of Array element comparisons = 4498500
Press any key to continue . . .
```



```
Array size : 5000
sort flag : 1
Number of Array element comparisons = 12497500
Press any key to continue . . .
```

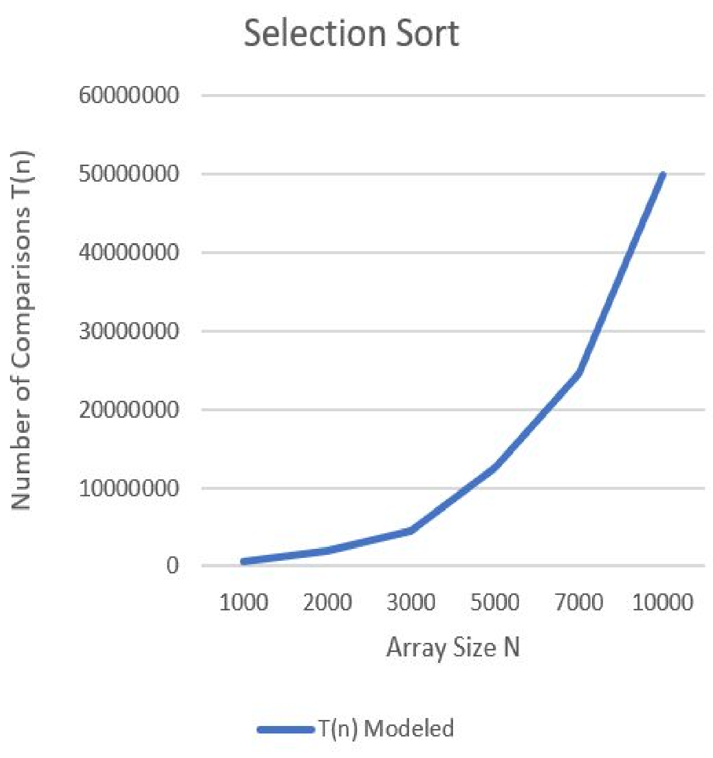


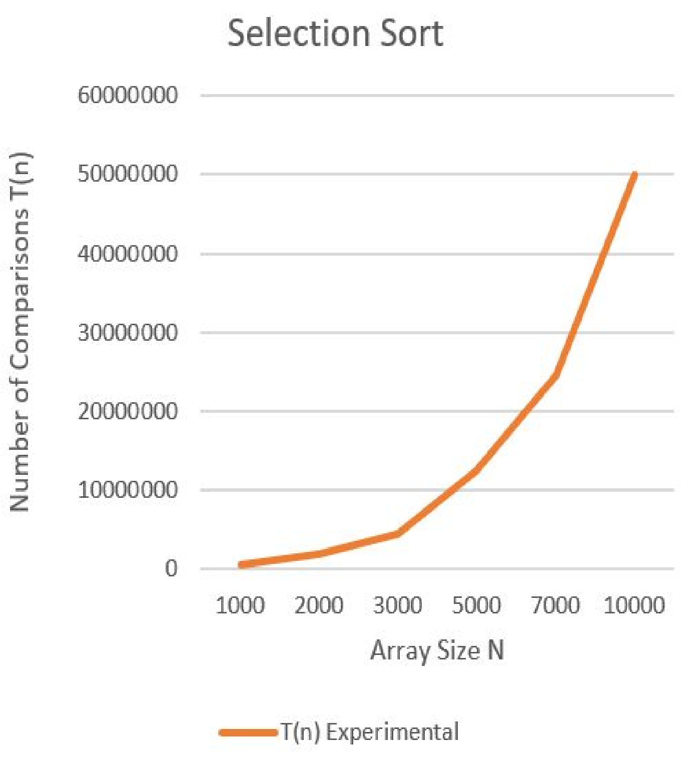
```
Number of Array elements : 7000
sort flag : 1
Number of Array element comparisons = 24496500
sh: 1: pause: not found
```



```
Number of Array elements : 10000
sort flag : 1
Number of Array element comparisons = 49995000
sh: 1: pause: not found
```

Graphical representation of MergeSort experimental vs. modeled results





2. Merge sort

The mathematical model for T(n) as provided in the lecture slides is:

$$T(n) = 2T(n/2) + n \quad with\ T(1) = 0$$

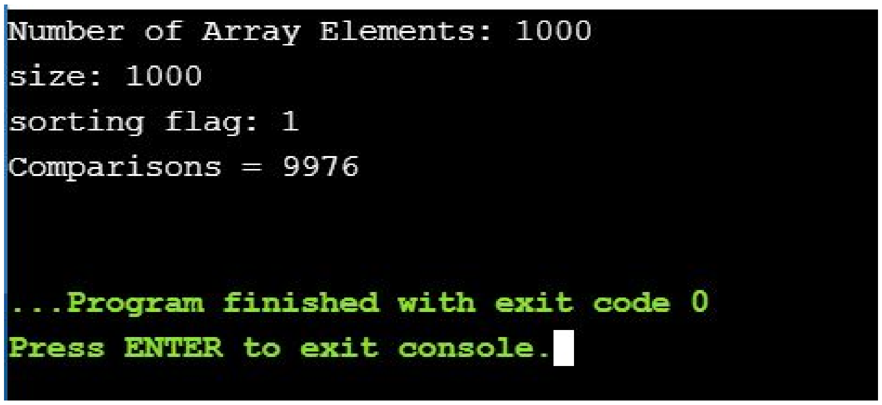
$$Here \quad a = 2, b = 1, c = 2, x = 1, a = c^x$$

$$T(n) = a^m T(1) + bn^x \log n$$

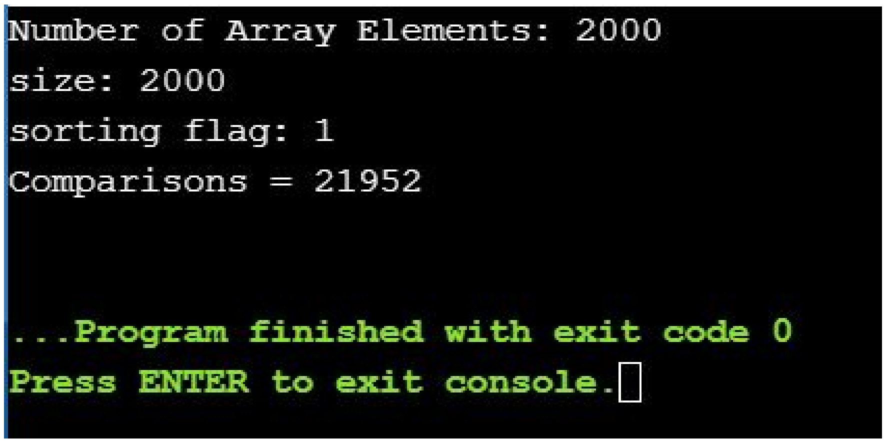
$$Hence \quad T(n) = n \log n = O(n \log n)$$

N	1000	2000	3000	5000	7000	10000
T(n) Model	9966	21932	34652	61439	89412	132877
T(n) Experimental	9976	21952	34904	61808	89808	133616

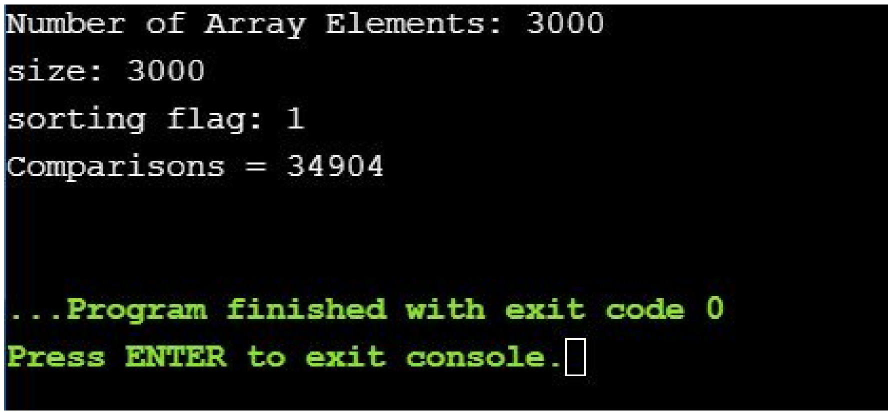
Screenshots of the experimental results:



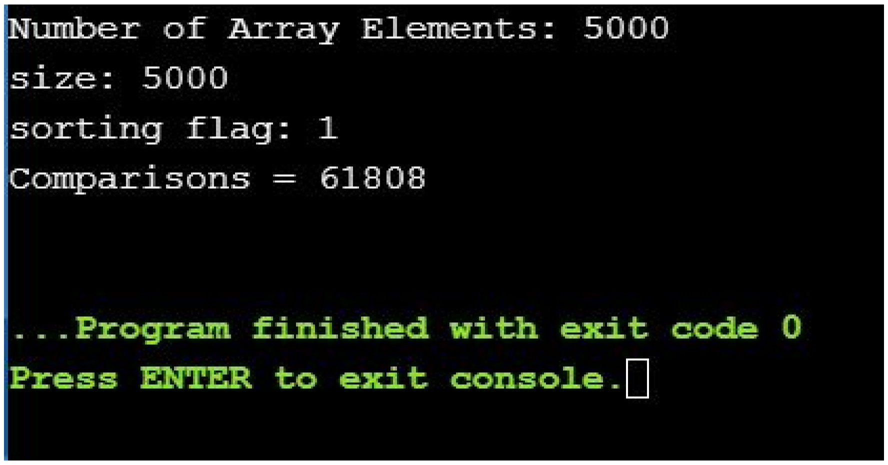
```
Number of Array Elements: 1000
size: 1000
sorting flag: 1
Comparisons = 9976


...Program finished with exit code 0
Press ENTER to exit console.
```



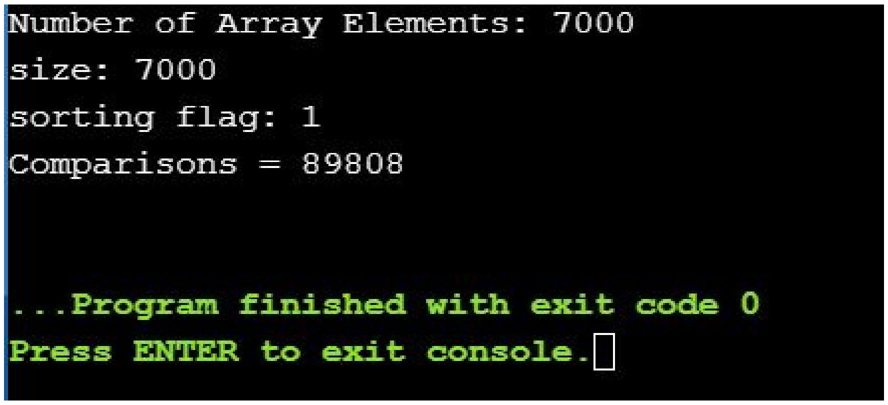
```
Number of Array Elements: 2000
size: 2000
sorting flag: 1
Comparisons = 21952


...Program finished with exit code 0
Press ENTER to exit console.
```



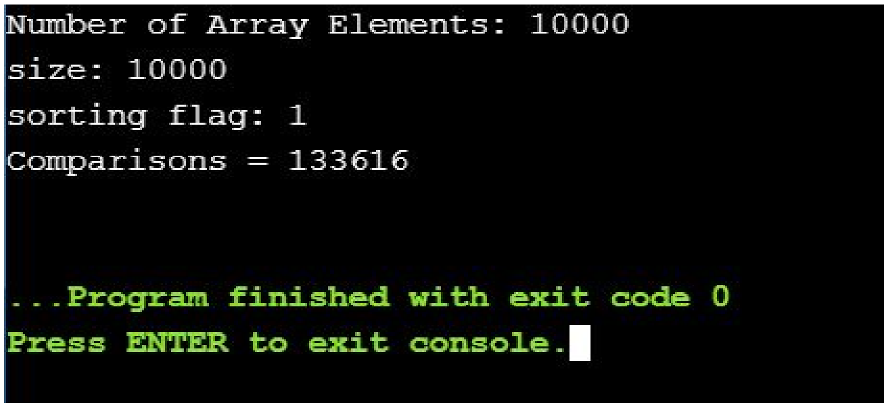
```
Number of Array Elements: 3000
size: 3000
sorting flag: 1
Comparisons = 34904


...Program finished with exit code 0
Press ENTER to exit console.
```



```
Number of Array Elements: 5000
size: 5000
sorting flag: 1
Comparisons = 61808


...Program finished with exit code 0
Press ENTER to exit console.
```

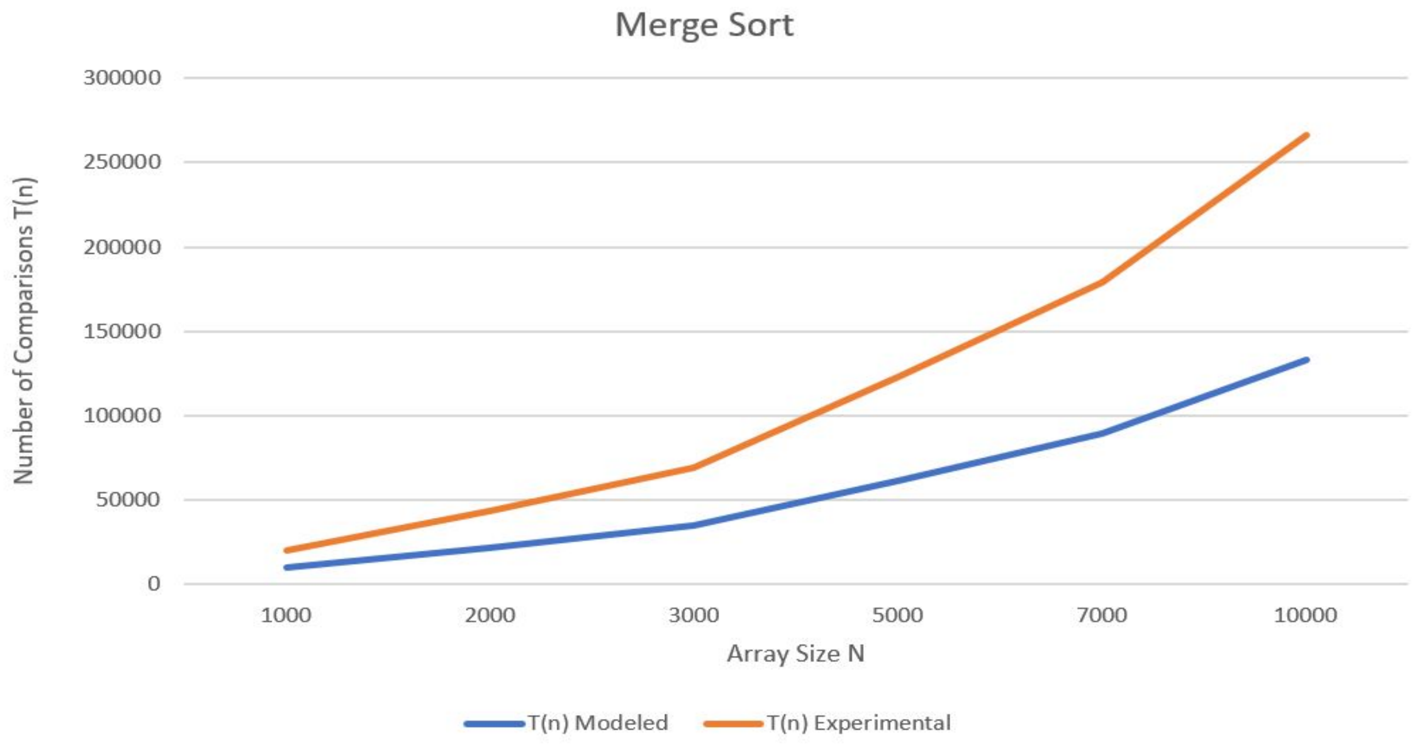


```
Number of Array Elements: 7000
size: 7000
sorting flag: 1
Comparisons = 89808


...Program finished with exit code 0
Press ENTER to exit console.
```



```
Number of Array Elements: 10000
size: 10000
sorting flag: 1
Comparisons = 133616


...Program finished with exit code 0
Press ENTER to exit console.
```

Graphical representation of MergeSort experimental vs. modeled results



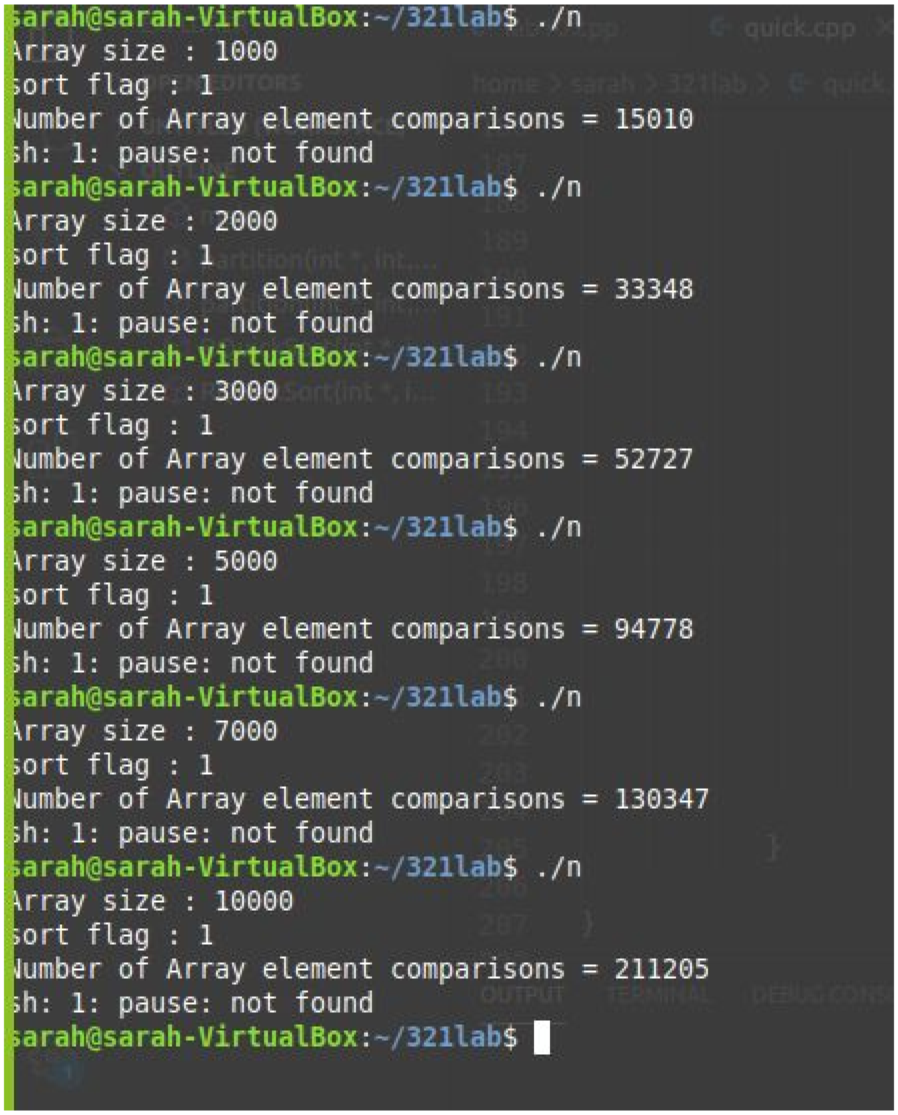
3. Quick sort 1 (worst case)

The mathematical model for T(n) is:

$$T(n) = n(n+1)/2 - 1 = O(n^2)$$

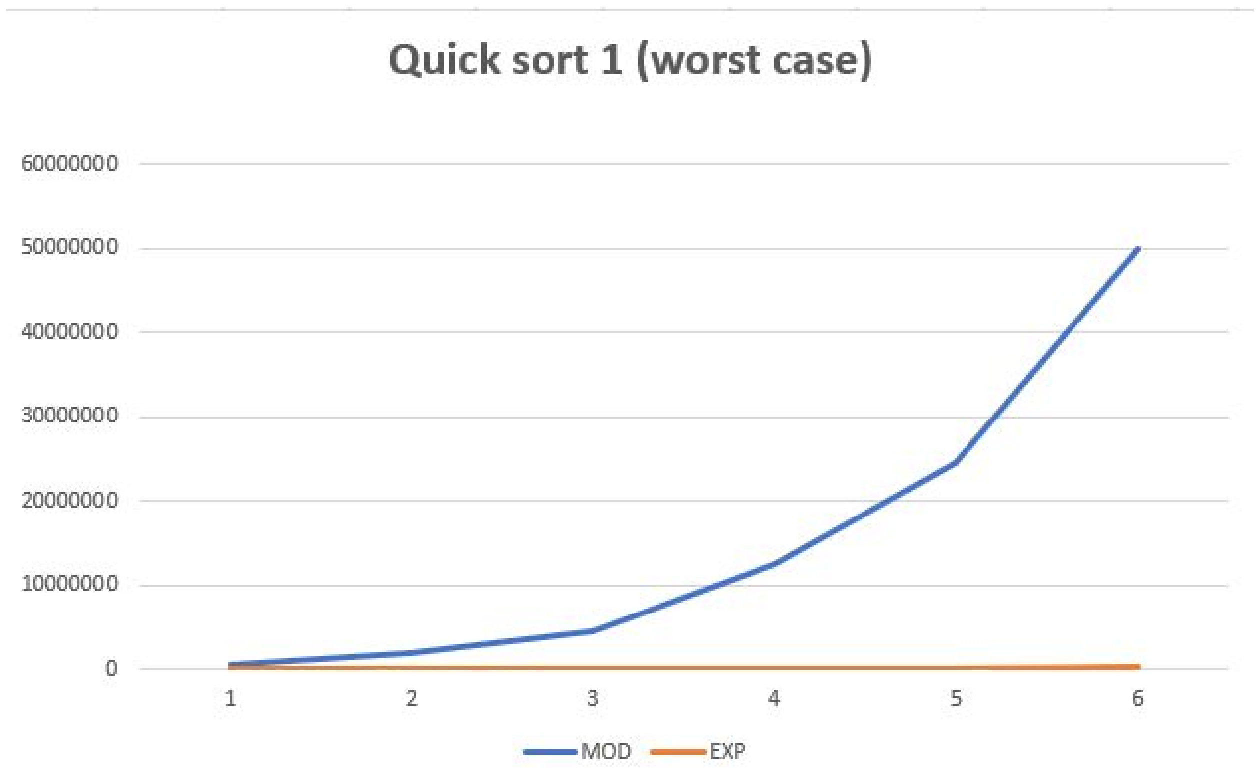
N	1000	2000	3000	5000	7000	10000
T(n) Model	500499	2000999	4501499	12502499	24503499	50004999
T(n) Experimental	15010	33348	52727	94778	130347	211205

Screenshots of the experimental results:



```
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 1000
sort flag : 1
Number of Array element comparisons = 15010
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 2000
sort flag : 1
Number of Array element comparisons = 33348
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 3000
sort flag : 1
Number of Array element comparisons = 52727
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 5000
sort flag : 1
Number of Array element comparisons = 94778
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 7000
sort flag : 1
Number of Array element comparisons = 130347
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 10000
sort flag : 1
Number of Array element comparisons = 211205
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$
```

Graphical representation of Quick sort 1 experimental vs. modeled results



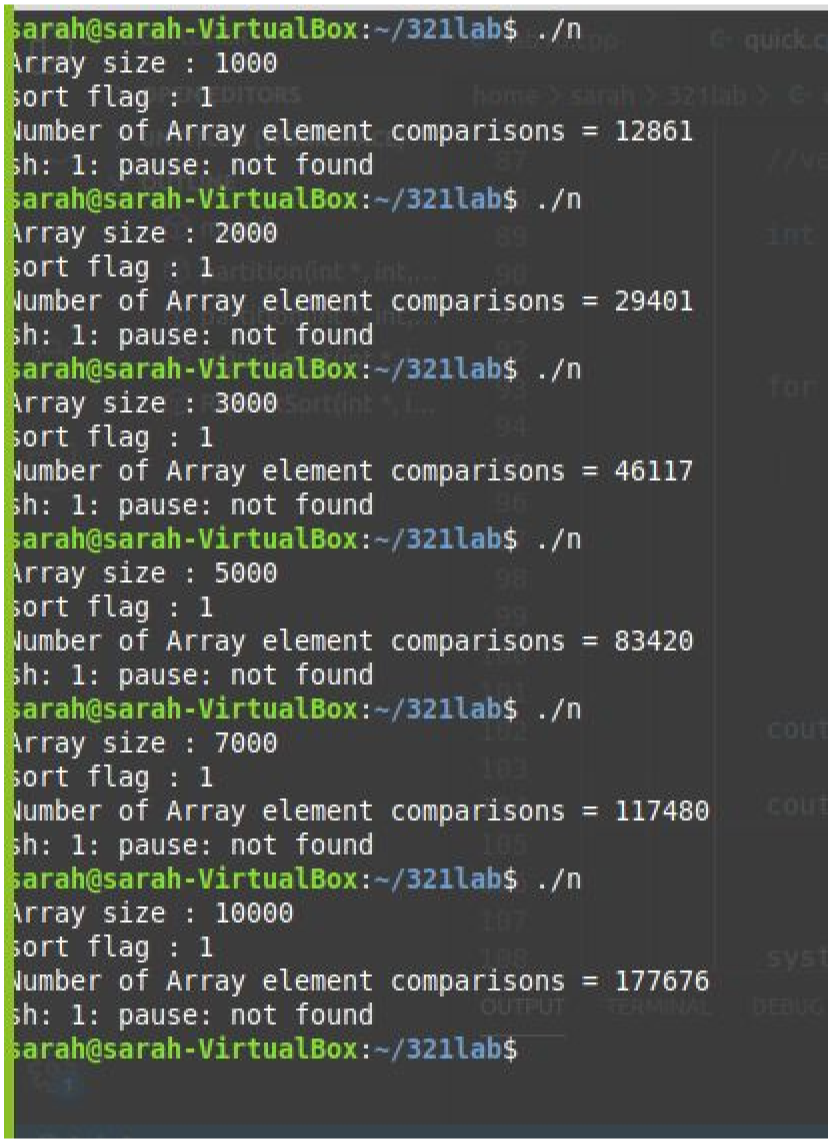
4. Quick sort 2 (best case)

The mathematical model for T(n) is:

T(n) = n log(n)= O(n log(n))

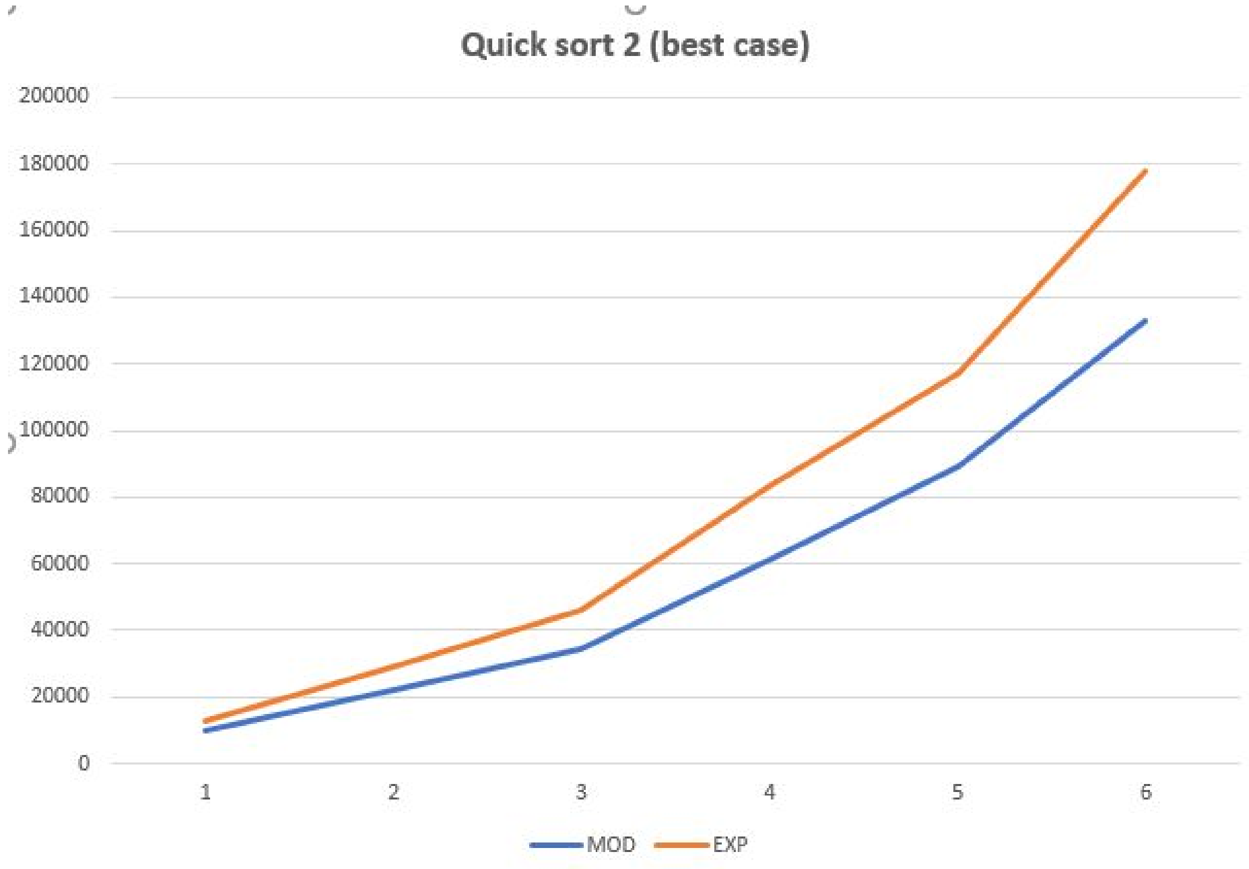
N	1000	2000	3000	5000	7000	10000
T(n) Model	9965	21931	34652	61438	89411	132877
T(n) Experimental	12861	29401	46117	83420	117480	177676

Screenshots of the experimental results:



```
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 1000
sort flag : 1
Number of Array element comparisons = 12861
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 2000
sort flag : 1
Number of Array element comparisons = 29401
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 3000
sort flag : 1
Number of Array element comparisons = 46117
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 5000
sort flag : 1
Number of Array element comparisons = 83420
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 7000
sort flag : 1
Number of Array element comparisons = 117480
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 10000
sort flag : 1
Number of Array element comparisons = 177676
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$
```

Graphical representation of Quick sort 2 experimental vs. modeled results



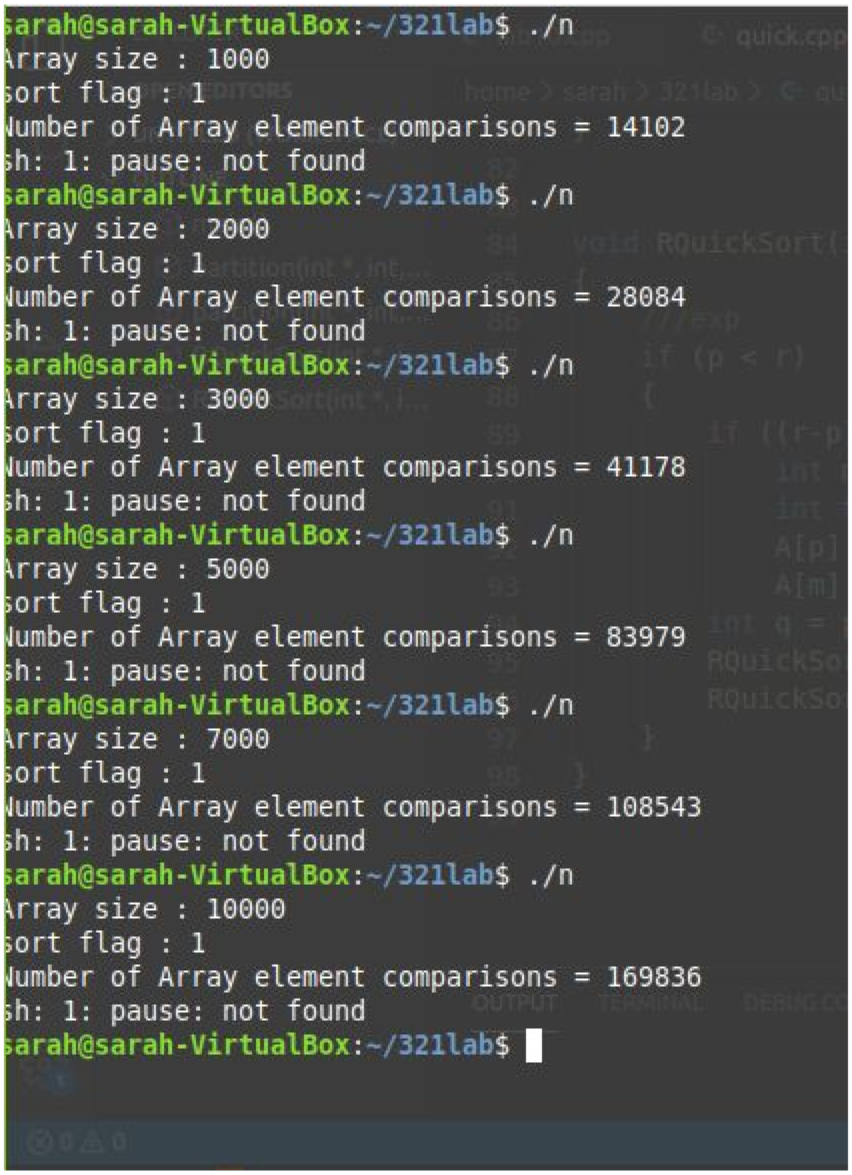
5. Randomized Quick sort

The mathematical model for T(n) is:

T(n) = 2n ln(n) = 1.39 n log(n)= O(n log(n))

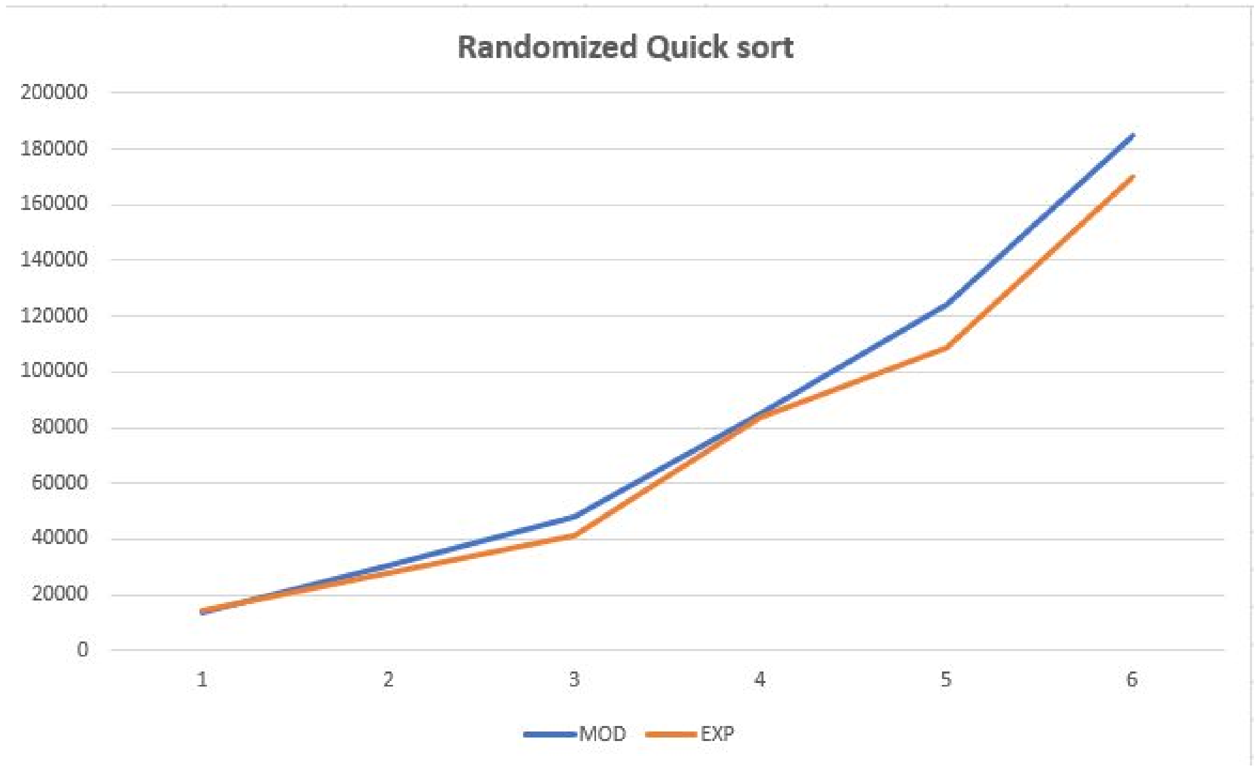
N	1000	2000	3000	5000	7000	10000
T(n) Model	13852	30484	48166	85399	124282	184699
T(n) Experimental	14102	28084	41178	83979	108543	169836

Screenshots of the experimental results:



```
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 1000
sort flag : 1
Number of Array element comparisons = 14102
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 2000
sort flag : 1
Number of Array element comparisons = 28084
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 3000
sort flag : 1
Number of Array element comparisons = 41178
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 5000
sort flag : 1
Number of Array element comparisons = 83979
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 7000
sort flag : 1
Number of Array element comparisons = 108543
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$ ./n
Array size : 10000
sort flag : 1
Number of Array element comparisons = 169836
sh: 1: pause: not found
sarah@sarah-VirtualBox:~/321lab$
```

Graphical representation of Randomized Quick sort experimental vs. modeled results



Analysis

- Selection sort

The outputs of the mathematical model at the 5 inputs are the same as the outputs of the experimental testing. This is because the Mathematical equation is derived from the same algorithm implemented with no approximations. Also, the algorithm performs exactly as expected with no redundant operations nor uncalculated useless operations.

- Merge sort

The data of both the mathematical model was almost matching that of the experimental results for small data sizes (even smaller than 1000). However, as the array size grows, the number of array element comparisons did not match the mathematical model quite well. As shown in the Merge Sort graph, the gap between both lines grows as the data size grows - with the experimental data always bigger than the modeled. The explanation to this gap is the duplicate count of some already-done comparisons. If we look at the code (in the appendix), we will find that duplication in function “merge”. In the first while loop, we are checking for the conditions: i < nl && j < nr but in the subsequent 2 while loops, we are checking again for the same conditions separately, which means possible duplication of the same counted comparisons. That explains the gap between the results of the mathematical model and the experimental data.

- Quick sort

It adopts the same technique that the Merge Sort uses, which is divide and conquer. But the Quick sort sorts the data in place.There are three cases of the Quick Sort each depends on the pivot position.

Regarding the graphs modelling both experimental and mathematical modelled outcomes of the 3 Quick sort algorithms, it is noticeable that the modelled data and the experimental data vary from each other in all the graphs. This is mainly because the model and the experiment are not identical in the real application. To clarify, the pivot, although it is chosen firstly to be in a certain location, the order of values in the array is what will control how the partitioning will happen in every recursive call, which will consequently control the number of comparisons.

1- Quick Sort 1:
In this algorithm, the pivot is chosen to be the first element of the array, which will require more array comparisons and time because the partitioning will be not balanced at all.

2- Quick Sort 2:
In this case of the algorithm, the pivot position is set to be exactly in the middle. This will require the least amount of recursive calls, because the partitioning will be mostly balanced. Hence the number of comparisons will be reduced dramatically and the time.
3- Randomized Quick Sort:
In this case of the algorithm, the pivot position is set to be random in the array. This will require an average amount of recursive calls, because the partitioning will not be mostly balanced nor totally unbalanced. Hence the number of comparisons and the time will be close to the best case but not exactly equal to it.

Conclusion

In conclusion, deciding the best sorting algorithm in terms of time complexity and the number of array element comparisons depends on some factors. The best two options are the quick sort and merge sort. They are both Divide and Conquer algorithms unlike the selection sort. Selection sort is not preferable since it is a brute force algorithm and it always takes quadratic time to sort. Regrading Quick sort, the time and number of comparisons depend on the location of the pivot. The best case is when the pivot is in the middle and the worst case is when the pivot is the first or last element (quadratic complexity). So, the average case is when the pivot is chosen randomly and the number of comparisons becomes on average 39% higher than the best case. Regarding merge sort, it is also a good choice since it has a fixed complexity O(nlogn) . So, the best option out of all presented algorithms is the Quick sort with the pivot to be chosen in the middle.

Appendix

Selection sort code

```
#include<iostream>
using namespace std;

int main()
{
	bool exist = false;
	int ran;
	int N;
	cout << "Number of Array elements : ";
	cin >> N;

	int * A = new int[N+1];
	for (int i = 1; i <= N; i++) // generating random permutation array
	{
		while (true)
		{
			ran = (rand() % N) + 1; //from 1 to N
			for (int j = 1; j < i; j++)
				if (A[j] == ran)
				{
					exist = true;
					break;
				}
				else
					exist = false;

			if (!exist)
			{
				A[i] = ran;
```

```
                        break;
                }
        }
    }

    //selection sort
    int m; //minimum
    int temp;
    int count=0;
    for(int i = 1;i<=N-1;i++)
    {
            m = i;
            //find minimum between i+1 and N
            for (int j = i + 1; j <= N; j++)
            {
                    if (A[j] < A[m]) //1 comparison
                    {
                            m = j;
                    }
                    count++;
            }


            //swaping
            temp = A[m];
            A[m] = A[i];
            A[i] = temp;
    }

    //verification it is sorted
    int sorted = 1;

    for (int i = 1; i <= N; i++)
            if (i != A[i]) sorted = 0;

    cout << "sort flag : " << sorted << endl;
    cout << "Number of Array element comparisons = "<<count << endl;

    system("pause");
    return 0;
}
```

Merge sort code

```
#include<iostream>
using namespace std;
int Comparisons = 0;
int N;
int * A = new int[N + 1];
bool sort_flag;

/* Generate RPA */
void RPA()
{
        bool exist = false;
       int ran;
       cout << "Number of Array Elements: ";
       cin >> N;

       for (int i = 1; i <= N; i++) // generating random permutation array
       {
              while (true)
              {
                     ran = (rand() % N) + 1; //from 1 to N
                     for (int j = 1; j < i; j++)
                             if (A[j] == ran)
                             {
                                     exist = true;
                                     break;
                             }
                             else
                                     exist = false;

                     if (!exist)
                     {
                             A[i] = ran;
                             break;
                     }
              }
       }
}

void swapping(int &a, int &b)
{
       int temp;
       temp = a;
       a = b;
       b = temp;
}

void merge(int *array, int l, int m, int r)
{
       int i, j, k, nl, nr;
       nl = m - l + 1;
       nr = r - m;
       int larr[nl], rarr[nr];

       for (i = 0; i < nl; i++)
              larr[i] = array[l + i];
       for (j = 0; j < nr; j++)
              rarr[j] = array[m + 1 + j];

       i = 0;
       j = 0;
       k = l;
       while (i < nl && j < nr)
       {
              if (larr[i] <= rarr[j])
              {
                     array[k] = larr[i];
                     i++;
                     Comparisons++;
              }
              else
              {
                     array[k] = rarr[j];
                     j++;
                     Comparisons++;
              }
              k++;
       }
       while (i < nl)        //for left array
       {
              array[k] = larr[i];
              i++; k++;
              Comparisons++;
       }
       while (j < nr)      // for right array
       {
              array[k] = rarr[j];
              j++; k++;
              Comparisons++;
       }
}
void mergeSort(int *array, int l, int r)
{
       int m;
       if (l < r)
       {
              int m = l + (r - l) / 2;
              mergeSort(array, l, m);
              mergeSort(array, m + 1, r);
              merge(array, l, m, r);
       }
}
void verify(int arr[])
{
       sort_flag = 1;
       for (int i = 1; i <= N; i++)
              if (arr[i] > arr[i + 1])
                     sort_flag = 0;
}

int main()
{
       RPA();
       mergeSort(A, 0, N - 1);
       verify(A);
       cout << "size: " << N << endl;
       cout << "sorting flag: " << sort_flag << endl;
       cout << "Comparisons = " << Comparisons << endl;
}
```

Quick Sort Code

Exp1:

```
#include<iostream>

#include <math.h>

using namespace std;


void QuickSort(int* A, int p, int r, int& count);

int partition(int* A, int p, int r, int& count, int&, int&);


int main()


{


	bool exist = false;

	int ran;

	int N;

	int count = 0;

	cout << "Array size : ";


	cin >> N;


	int * A = new int[N + 1];


	for (int i = 1; i <= N; i++) // generating random permutation array

	{

		while (true)

		{

			ran = (rand() % N) + 1; //from 1 to N


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

				if (A[j] == ran)
```

```
                        {
                                exist = true;
                                break;
                        }
                        else
                                exist = false;

                if (!exist)
                {
                        A[i] = ran;
                        break;
                }
        }

}



QuickSort(A, 1, N, count);



//verification it is sorted

int sorted = 1;

for (int i = 1; i <= N; i++)

        if (i != A[i]) sorted = 0;

cout << "sort flag : " << sorted << endl;

cout << "Number of Array element comparisons = " << count << endl;
```

```
        system("pause");

        return 0;

}




int partition(int* A, int p, int r, int& count, int& i, int& j)
{
        int temp;

        int pivot = A[p]; // pivot

        i = p; //start
        j = r; //end//////

        while (i <= j) {



                while (A[i] < pivot) { i++; count++; };
                while (A[j] > pivot) { j--; count++; };
                count = count + 2;
                if (i <= j)

                { //swap

                        temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                        i++; j--;
                }
        };

        return (j);

}




void QuickSort(int* A, int p, int r, int& count)
{

        if (p < r)
        {

                int i, j;
                int q = partition(A, p, r, count, i, j);
                QuickSort(A, p, j, count);
                QuickSort(A, i, r, count);

        }

}
```

Exp2:

```
#include<iostream>

#include <math.h>

using namespace std;


void QuickSort(int* A, int p, int r, int& count);

int partition(int* A, int p, int r, int& count, int&, int&);



int main()

{


        bool exist = false;

        int ran;

        int N;

        int count = 0;


        cout << "Array size : ";

        cin >> N;


        int * A = new int[N + 1];


        for (int i = 1; i <= N; i++) // generating random permutation array

        {

                while (true)

                {

                        ran = (rand() % N) + 1; //from 1 to N

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

                                if (A[j] == ran)
                        {
                            exist = true;
                            break;
                        }
                        else
                            exist = false;

                    if (!exist)
                    {
                        A[i] = ran;
                        break;
                    }
                }
            }



            QuickSort(A, 1, N, count);

            //verification it is sorted
            int sorted = 1;
            for (int i = 1; i <= N; i++)
                if (i != A[i]) sorted = 0;

            cout << "sort flag : " << sorted << endl;
            cout << "Number of Array element comparisons = " << count << endl;
            system("pause");
            return 0;
        }



        int partition(int* A, int p, int r, int& count, int& i, int& j)
        {
```

```
        int temp;
        double f = p + r;
        int piv = floor((f) / 2);
        int pivot = A[piv]; // pivot
        i = p; //start
        j = r; //end//////

        while (i <= j) {

                while (A[i] < pivot) { i++; count++; };
                while (A[j] > pivot) { j--; count++; };
                count = count + 2;
                if (i <= j)
                { //swap
                        temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                        i++; j--;
                }
        };

        return (j);
}

void QuickSort(int* A, int p, int r, int& count)
{
        if (p < r)
        {
                int i, j;
                int q = partition(A, p, r, count, i, j);
                QuickSort(A, p, j, count);
                QuickSort(A, i, r, count);
        }
}
```

Randomized Quick Sort Code

```
#include<iostream>
#include <math.h>
using namespace std;
void RQuickSort(int* A, int p, int r,int& count);
int partition(int* A, int p, int r,int& count);

int main()
{

  bool exist = false;
  int ran;
  int N;
  int count=0;
  cout << "Array size : ";
  cin >> N;

  int * A = new int[N+1];
  for (int i = 1; i <= N; i++) // generating random permutation array
  {
    while (true)
    {
      ran = (rand() % N) + 1; //from 1 to N
      for (int j = 1; j < i; j++)
        if (A[j] == ran)
        {
          exist = true;
```

```
                    break;
                }
                else
                    exist = false;

            if (!exist)
            {
                A[i] = ran;
                break;
            }
        }
    }



    RQuickSort(A, 1, N,count);


    //verification it is sorted
    int sorted = 1;


    for (int i = 1; i <= N; i++)
        if (i != A[i]) sorted = 0;




    cout << "sort flag : " << sorted << endl;
    cout << "Number of Array element comparisons = " << count << endl;


    system("pause");
    return 0;

}
```

```
int partition(int* A, int p, int r,int& count)
{

    int pivot = A[p]; // pivot
    int i = p; //start
    int j = r; //end//////
    while (i < j) {
        do { i++; count++; } while (A[i] < pivot);
        do { j--; count++; } while (A[j] > pivot);
        if (i < j)
        { //swap
           int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    };

        A[p] = A[j];
        A[j] = pivot;


    return (j);
}


void RQuickSort(int* A, int p, int r,int& count)
{
    ///exp
    if (p < r)
    {
        if ((r-p) > 5) {
            int m = rand()%(r-p+1) + p;
            int temp = A[p];
            A[p] = A[m];
            A[m] = temp;}
        int q = partition(A, p, r+1,count);
        RQuickSort(A, p, q - 1,count);
        RQuickSort(A, q + 1, r,count);
    }
}
```