**Student : Ionasco Gheorghe**

**Group :** FAF-193

**QUICK SORT**

***ANALYSIS***

Time taken by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + θ (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

***Worst Case:*** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + θ(n)

which is equivalent to

T(n) = T(n-1) + θ (n)

The solution of above recurrence is θ (n2).

***Best Case:*** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + θ (n)

The solution of above recurrence is θ (nLogn).

***Average Case:***  
To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/).  
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + θ (n)

Solution of above recurrence is also θ (nLogn)

***CODE ( C++ )***

#include <cstdlib>

#include <ctime>

#include <time.h>

#include <iostream>

using namespace std;

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

void printTimeSince(clock\_t tStart)

{

printf("Time taken: %.10fs\n", (double)(clock() - tStart)/CLOCKS\_PER\_SEC);

}

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high - 1; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void printArray(int arr[], int size)

{

int i;

for (i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;

}

int main()

{

srand(time(0));

int arr[10000];

int n = 9999;

for(int i =0; i<n; i++)

{

arr[i] = rand();

}

clock\_t tStart = clock();

quickSort(arr, 0, n - 1);

printTimeSince(tStart);

cout << "Sorted array: \n";

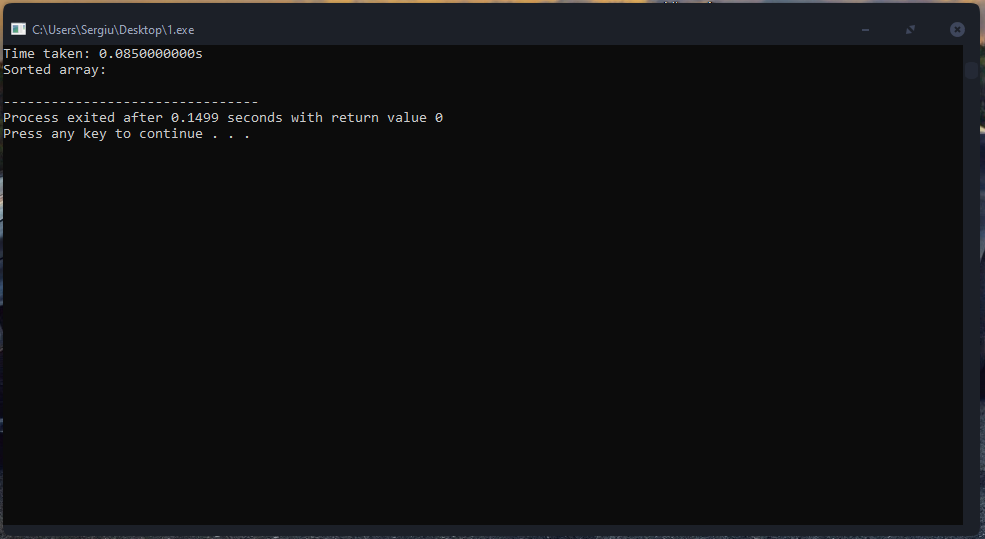
//printArray(arr, n);

return 0;

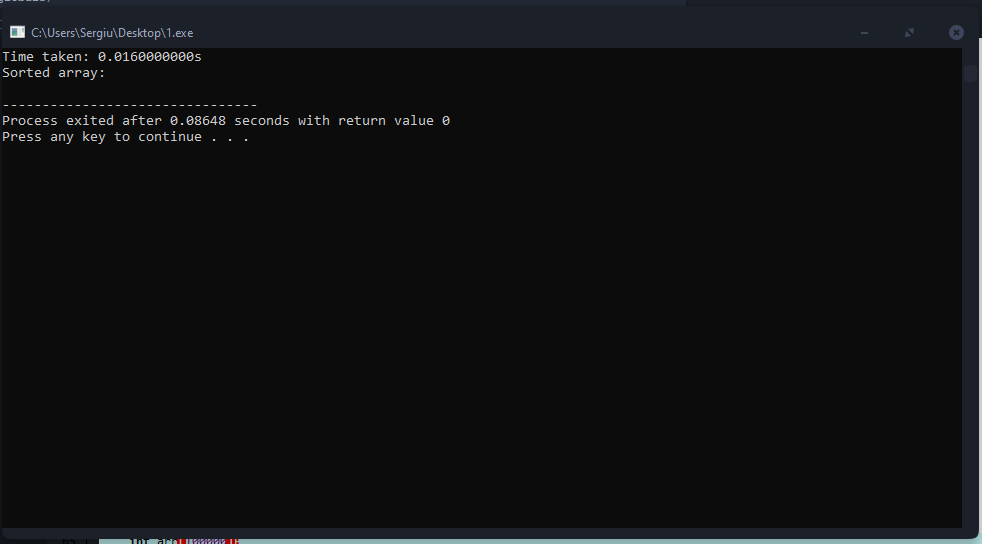
}

***OUTPUT***

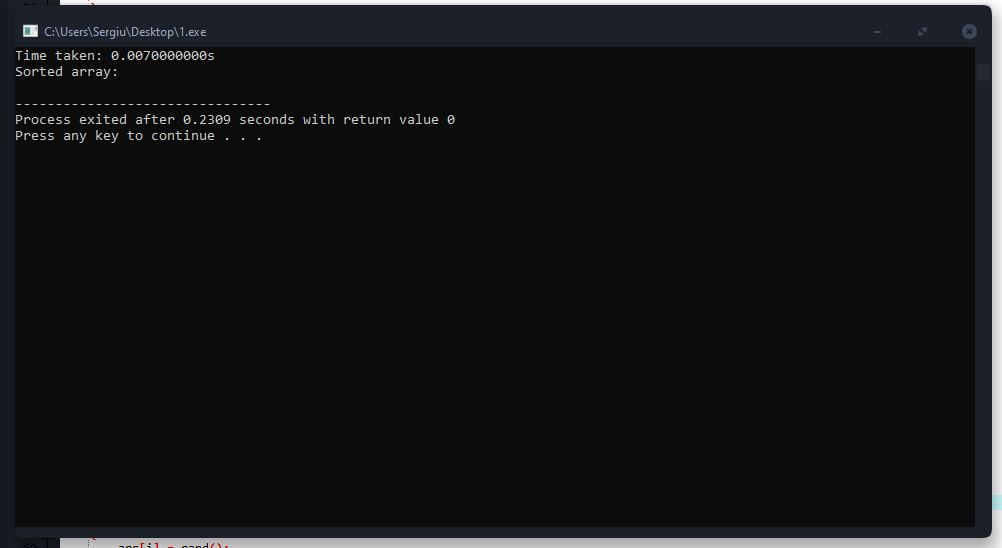
For 500000 elements to sort it took 0.085sec



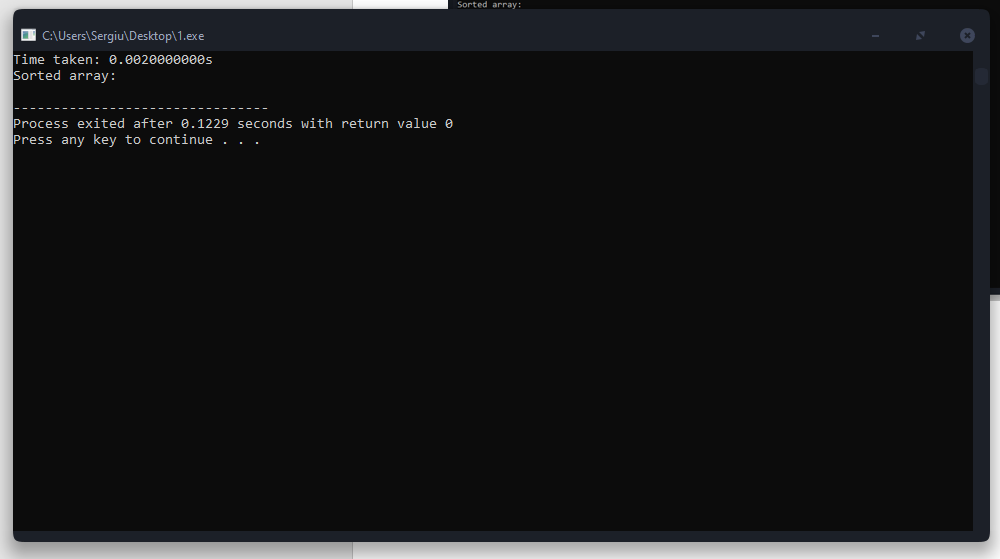
For 100000 elements to sort it took 0.016sec



For 50000 elements to sort it took 0.007sec



For 10000 elements to sort it took 0.002sec



**CONCLUSIONS**

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms , QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.