CS253

DESIGN AND ANALYSIS OF ALGORITHMS

PRACTICAL ASSIGNMENT

By: Vaibhav and Guruprasad

Roll no: 211CS162 211CS126

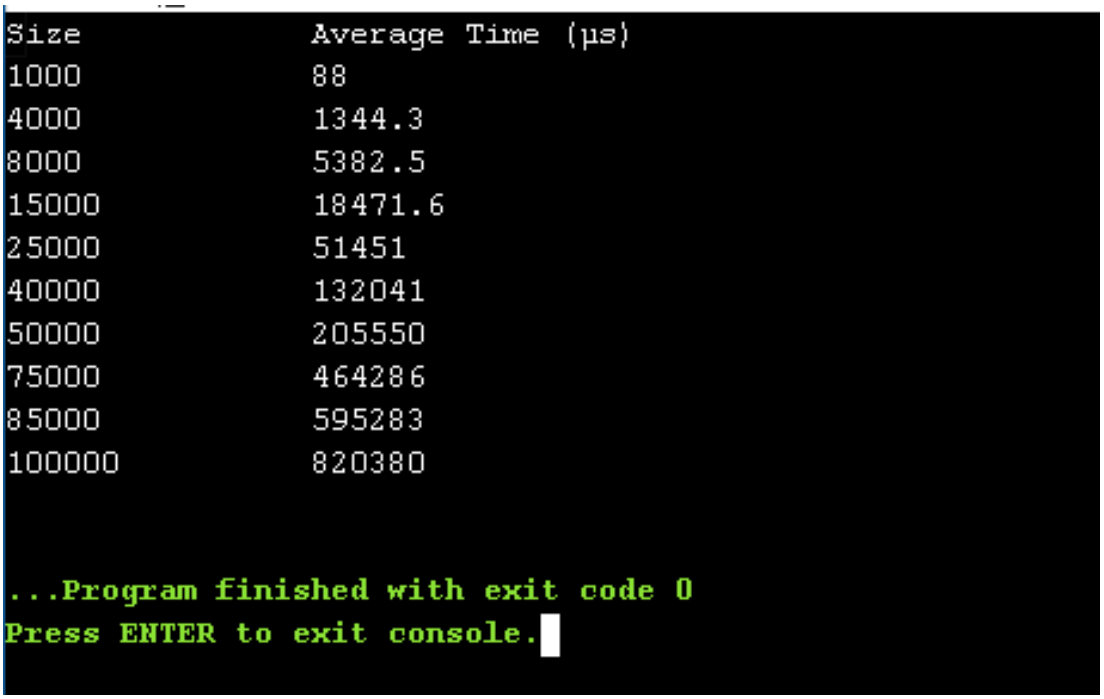
1. INSERTION SORT

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
using namespace std;
using namespace std::chrono;
void insertionSort(int arr[], int n) {
for (int i = 1; i < n; i++) {
int key = arr[i];
int j = i - 1;
while (j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
j--;
}
arr[j + 1] = key;
}
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;

int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};

int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
}
```

```
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
insertionSort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

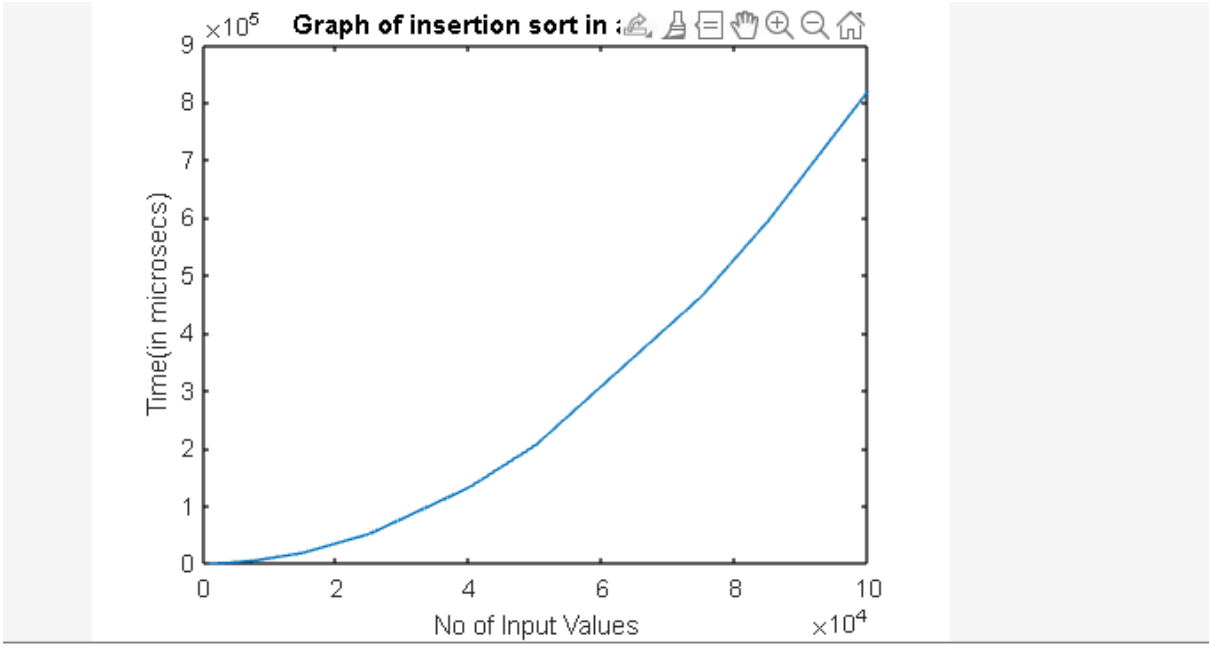
OUTPUT



MATLAB CODE:

```
% Define x and y values
x = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y=[85.3,1322.2,5234.6,15123,41994.9,107560,161393,360225,457560,640916];
% Plot the graph
plot(x, y);
```

```
% Label the axes and add a title
xlabel('No of Input Values');
ylabel('Time(in microsecs)');
title('Graph of insertion sort in average case');
```



➔ *ALREADY SORTED CASE (Best case)*

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void insertionSort(int arr[], int n) {
for (int i = 1; i < n; i++) {
int key = arr[i];
int j = i - 1;
while (j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
j--;
}
arr[j + 1] = key;
}
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;

int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};

int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
```

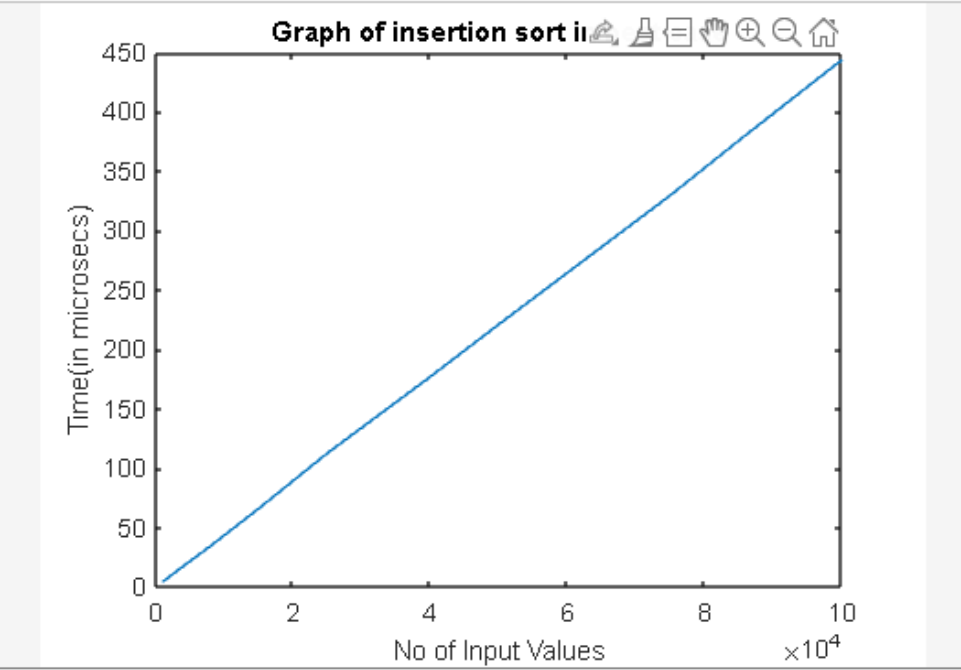
```
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
sort(arr[i],arr[i]+sizes[i]);
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
insertionSort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration =
duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) /
numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT

```
Size            Average Time (µs)
1000            4.1
4000            17.2
8000            34.3
15000           65.8
25000           112.4
40000           176.8
50000           221
75000           329.7
85000           375.9
100000          443.8


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

MATLAB CODE:

```
% Define x and y values
x = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y =[6.1,12.8,31,47.2,64.6,132.1,186.7,339.2,482.8,654.8];
% Plot the graph
plot(x, y);
% Label the axes and add a title
xlabel('No of Input Values');
ylabel('Time(in microsecs)');
title('Graph of insertion sort in best case');
```



➔ *Worst Case*

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void insertionSort(int arr[], int n) {
for (int i = 1; i < n; i++) {
int key = arr[i];
int j = i - 1;
while (j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
j--;
}
arr[j + 1] = key;
}
}
int main() {
```

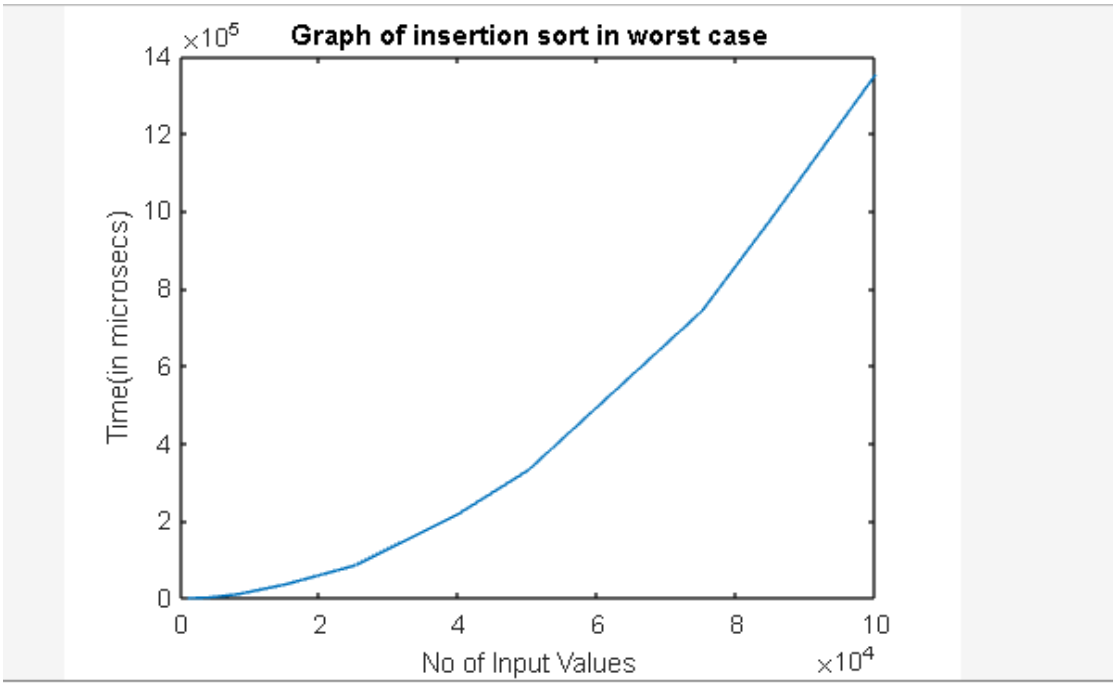
```
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1,
sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
sort(arr[i],arr[i]
+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start =
high_resolution_clock::now();
insertionSort(arr[i], sizes[i]);
auto stop =
high_resolution_clock::now();
auto duration =
duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" <<
endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average =
static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average
<< endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT

```
Size            Average Time (µs)
1000            134.4
4000            2099.8
8000            9722
15000           35454.5
25000           84418.2
40000           218396
50000           330875
75000           741701
85000           980704
100000          1.35281e+06


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

MATLAB CODE:

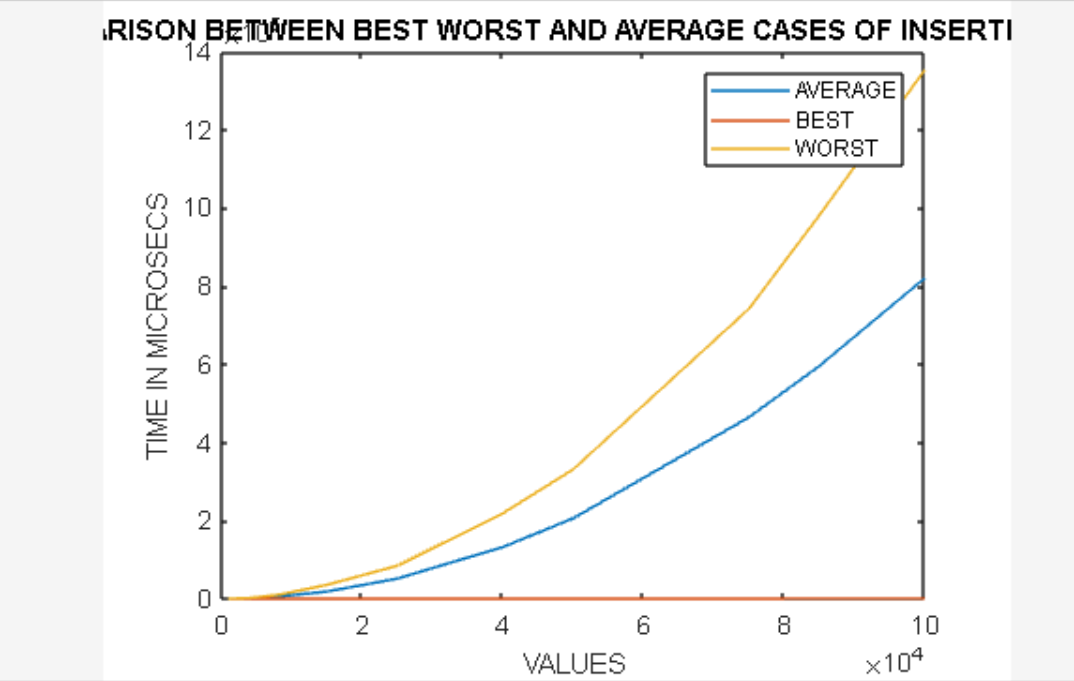
```
% Define x and y values
x = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y=[166.8,663.3,4191,9222,16347,70102,154455,411285,928276,1.6367e+06];
% Plot the graph
plot(x, y);
% Label the axes and add a title
xlabel('No of Input Values');
ylabel('Time(in microsecs)');
title('Graph of insertion sort in worst case');
```



➔ *COMPARISION*

```
% Define the x and y values for the three line graphs
x1 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y1 = [85.3,1322.2,5234.6,15123,41994.9,107560,161393,360225,457560,640916];
x2 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y2 =[6.1,12.8,31,47.2,64.6,132.1,186.7,339.2,482.8,654.8];
x3 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
```

```
y3 =[166.8,663.3,4191,9222,16347,70102,154455,411285,928276,1.6367e+06];
% Plot the three line graphs on the same figure
figure;
plot(x1, y1);
hold on; % Use hold on to keep the current plot active
plot(x2, y2);
plot(x3, y3);
hold off; % Use hold off to release the current plot
% Add title, labels and legend
title('COMPARISON BETWEEN BEST WORST AND AVERAGE CASES OF INSERTION SORT');
xlabel('VALUES');
ylabel('TIME IN MICROSECS');
legend('AVERAGE', 'BEST', 'WORST');
```



2. MERGE SORT

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void merge(int array[], int const left, int const mid,
int const right)
{
auto const subArrayOne = mid - left + 1;
auto const subArrayTwo = right - mid;
// Create temp arrays
auto *leftArray = new int[subArrayOne],
*rightArray = new int[subArrayTwo];
```

```
// Copy data to temp arrays leftArray[] and rightArray[]
for (auto i = 0; i < subArrayOne; i++)
leftArray[i] = array[left + i];
for (auto j = 0; j < subArrayTwo; j++)
rightArray[j] = array[mid + 1 + j];
auto indexOfSubArrayOne
= 0, // Initial index of first sub-array
indexOfSubArrayTwo
= 0; // Initial index of second sub-array
int indexOfMergedArray
= left; // Initial index of merged array
// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
&& indexOfSubArrayTwo < subArrayTwo) {
if (leftArray[indexOfSubArrayOne]
<= rightArray[indexOfSubArrayTwo]) {
array[indexOfMergedArray]
= leftArray[indexOfSubArrayOne];
indexOfSubArrayOne++;
}
else {
array[indexOfMergedArray]
= rightArray[indexOfSubArrayTwo];
indexOfSubArrayTwo++;
}
indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
array[indexOfMergedArray]
= leftArray[indexOfSubArrayOne];
indexOfSubArrayOne++;
indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any
while (indexOfSubArrayTwo < subArrayTwo) {
array[indexOfMergedArray]
= rightArray[indexOfSubArrayTwo];
indexOfSubArrayTwo++;
indexOfMergedArray++;
}
delete[] leftArray;
delete[] rightArray;
}
void mergeSort(int array[], int const begin, int const end)
{
if (begin >= end)
return; // Returns recursively
auto mid = begin + (end - begin) / 2;
```

```
mergeSort(array, begin, mid);
mergeSort(array, mid + 1, end);
merge(array, begin, mid, end);
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
// sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
mergeSort(arr[i],0,sizes[i] - 1);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT

```
Size            Average Time (µs)
1000            156.1
4000            656.3
8000            1351.2
15000           2514.2
25000           3313.7
40000           5262.3
50000           7361.2
75000           9333.2
85000           11708
100000          17821.6


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

➔ ***ASCENDING ORDER***

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void merge(int array[], int const left, int const mid,
int const right)
{
auto const subArrayOne = mid - left + 1;
auto const subArrayTwo = right - mid;
// Create temp arrays
auto *leftArray = new int[subArrayOne],
*rightArray = new int[subArrayTwo];
// Copy data to temp arrays leftArray[] and rightArray[]
for (auto i = 0; i < subArrayOne; i++)
leftArray[i] = array[left + i];
for (auto j = 0; j < subArrayTwo; j++)
rightArray[j] = array[mid + 1 + j];
auto indexOfSubArrayOne
= 0, // Initial index of first sub-array
indexOfSubArrayTwo
= 0; // Initial index of second sub-array
int indexOfMergedArray
= left; // Initial index of merged array
// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
&& indexOfSubArrayTwo < subArrayTwo) {
if (leftArray[indexOfSubArrayOne]
<= rightArray[indexOfSubArrayTwo]) {
```

```
array[indexOfMergedArray]
= leftArray[indexOfSubArrayOne];
indexOfSubArrayOne++;
}
else {
array[indexOfMergedArray]
= rightArray[indexOfSubArrayTwo];
indexOfSubArrayTwo++;
}
indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
array[indexOfMergedArray]
= leftArray[indexOfSubArrayOne];
indexOfSubArrayOne++;
indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any
while (indexOfSubArrayTwo < subArrayTwo) {
array[indexOfMergedArray]
= rightArray[indexOfSubArrayTwo];
indexOfSubArrayTwo++;
indexOfMergedArray++;
}
delete[] leftArray;
delete[] rightArray;
}
void mergeSort(int array[], int const begin, int const end)
{
if (begin >= end)
return; // Returns recursively
auto mid = begin + (end - begin) / 2;
mergeSort(array, begin, mid);
mergeSort(array, mid + 1, end);
merge(array, begin, mid, end);
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
```

```
}
sort(arr[i],arr[i]+sizes[i]);
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
mergeSort(arr[i],0,sizes[i] - 1);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
```

OUTPUT

```
Size            Average Time (µs)
1000            185.5
4000            834.8
8000            1388.9
15000           2200.8
25000           3933.5
40000           6870.8
50000           11483
75000           16882.8
85000           19622.2
100000          23152.9


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

➔ *DESCENDING ORDER*

CODE:

```
#include <iostream>
```

```
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void merge(int array[], int const left, int const mid,
int const right)
{
auto const subArrayOne = mid - left + 1;
auto const subArrayTwo = right - mid;
// Create temp arrays
auto *leftArray = new int[subArrayOne],*rightArray = new int[subArrayTwo];
// Copy data to temp arrays leftArray[] and rightArray[]
for (auto i = 0; i < subArrayOne; i++)
leftArray[i] = array[left + i];
for (auto j = 0; j < subArrayTwo; j++)
rightArray[j] = array[mid + 1 + j];
auto indexOfSubArrayOne
= 0, // Initial index of first sub-array
indexOfSubArrayTwo
= 0; // Initial index of second sub-array
int indexOfMergedArray
= left; // Initial index of the merged array
// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
&& indexOfSubArrayTwo < subArrayTwo) {
if (leftArray[indexOfSubArrayOne]
<= rightArray[indexOfSubArrayTwo]) {
array[indexOfMergedArray]
= leftArray[indexOfSubArrayOne];
indexOfSubArrayOne++;
}
else {
array[indexOfMergedArray]
= rightArray[indexOfSubArrayTwo];
indexOfSubArrayTwo++;
}
indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
array[indexOfMergedArray]
= leftArray[indexOfSubArrayOne];
indexOfSubArrayOne++;
indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any
while (indexOfSubArrayTwo < subArrayTwo) {
array[indexOfMergedArray]
```

```
= rightArray[indexOfSubArrayTwo];
indexOfSubArrayTwo++;
indexOfMergedArray++;
}
delete[] leftArray;
delete[] rightArray;
}
void mergeSort(int array[], int const begin, int const end)
{
if (begin >= end)
return; // Returns recursively
auto mid = begin + (end - begin) / 2;
mergeSort(array, begin, mid);
mergeSort(array, mid + 1, end);
merge(array, begin, mid, end);
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
mergeSort(arr[i],0,sizes[i] - 1);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
```

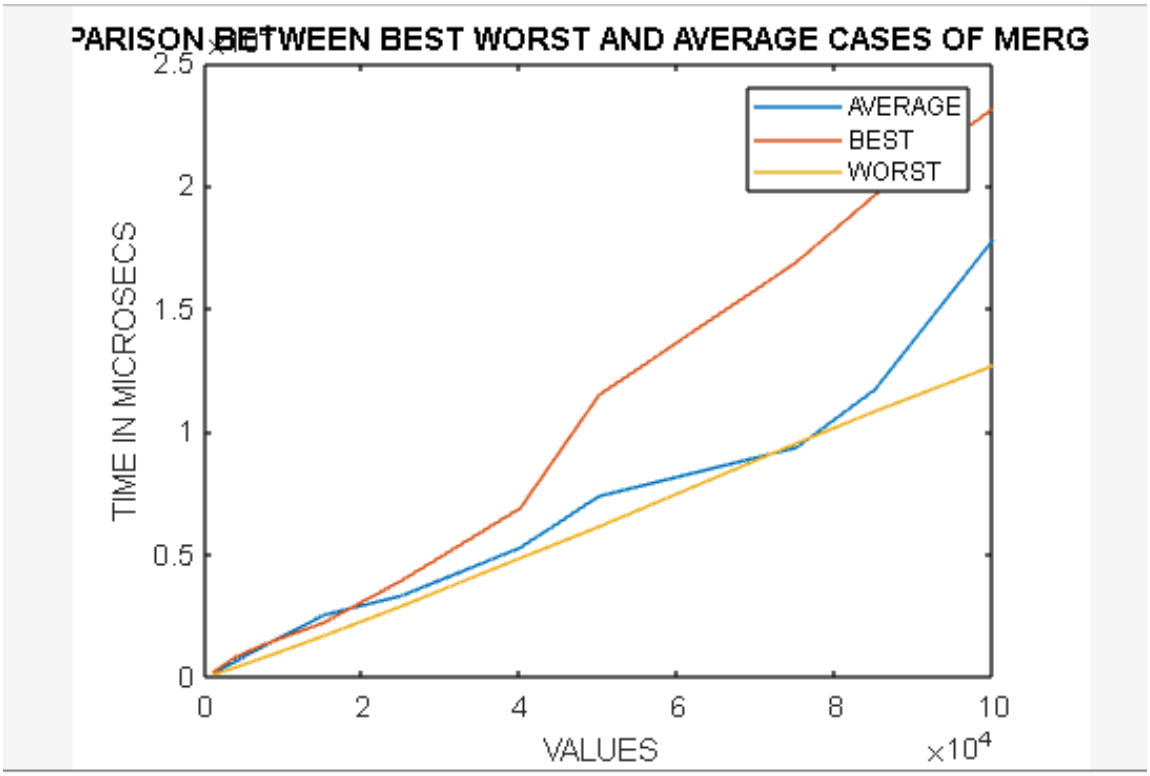
```
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT

```
Size            Average Time (µs)
1000            92.7
4000            404.7
8000            850
15000           1666.9
25000           2893.2
40000           4842.2
50000           6120.7
75000           9510.7
85000           10818.8
100000          12674.9


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

MATLAB CODE:

```
% Define the x and y values for the three line graphs
x1 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y1 =[156.1,656.3,1351.2,2514.2,3313.7,5262.3,7361.2,9333.2,11708,17821.6];
x2 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y2 =[185.5,834.8,1388.9,2200.8,3933.5,6870.8,11483,16882.8,19622.2,23152.9];
x3 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y3 =[92.7,404.7,850,1666.9,2893.2,4842.2,6120.7,9510.7,10818.8,12674.9];
% Plot the three line graphs on the same figure
figure;
plot(x1, y1);
hold on; % Use hold on to keep the current plot active
plot(x2, y2);
plot(x3, y3);
hold off; % Use hold off to release the current plot
% Add title, labels and legend
title('COMPARISON BETWEEN BEST WORST AND AVERAGE CASES OF MERGE SORT');
xlabel('VALUES');
ylabel('TIME IN MICROSECS');
legend('AVERAGE', 'BEST', 'WORST');
```



3. HEAP SORT

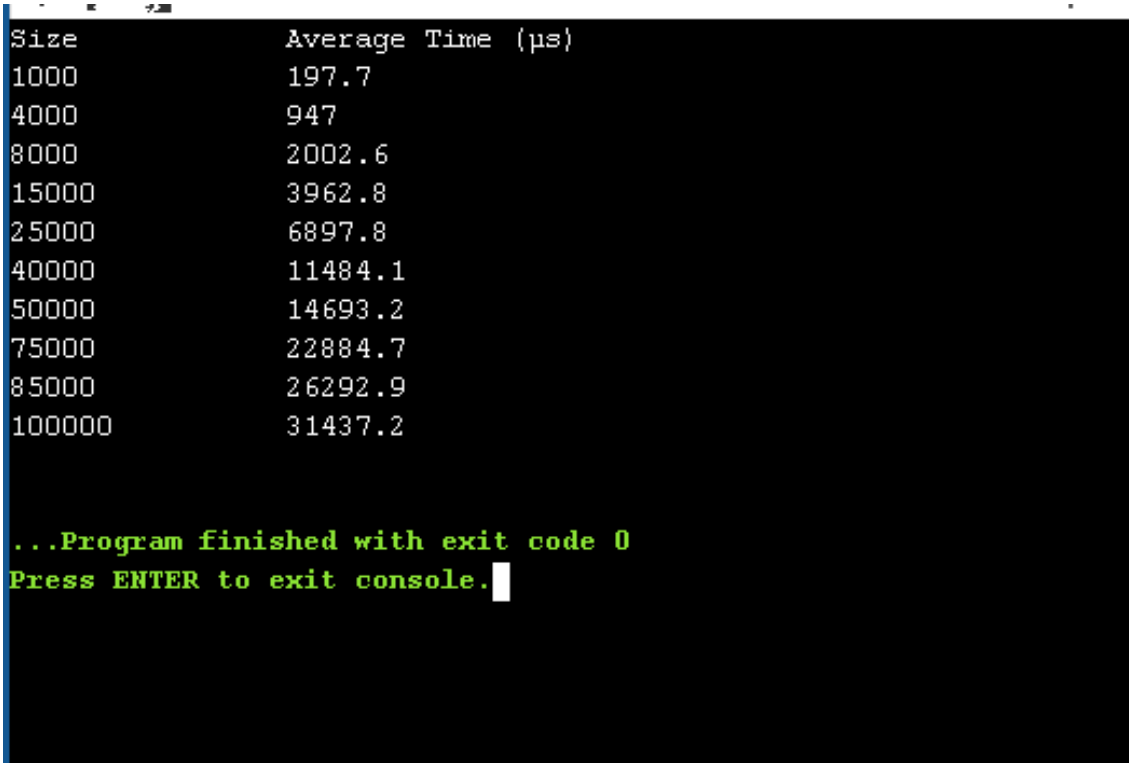
CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void heapify(int arr[], int N, int i)
{
// Initialize largest as root
int largest = i;
// left = 2*i + 1
int l = 2 * i + 1;
// right = 2*i + 2
int r = 2 * i + 2;
// If left child is larger than root
if (l < N && arr[l] > arr[largest])
largest = l;
// If right child is larger than largest
// so far
if (r < N && arr[r] > arr[largest])
largest = r;
// If largest is not root
if (largest != i) {
swap(arr[i], arr[largest]);
// Recursively heapify the affected
```

```
// sub-tree
heapify(arr, N, largest);
}
}
// Main function to do heap sort
void heapSort(int arr[], int N)
{
// Build heap (rearrange array)
for (int i = N / 2 - 1; i >= 0; i--)
heapify(arr, N, i);
// One by one extract an element
// from heap
for (int i = N - 1; i > 0; i--) {
// Move current root to end
swap(arr[0], arr[i]);
// call max heapify on the reduced heap
heapify(arr, i, 0);
}
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
// sort(arr[i],arr[i]+sizes[i]);
// sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
heapSort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
```

```
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT



➔ *ASCENDING ORDER*

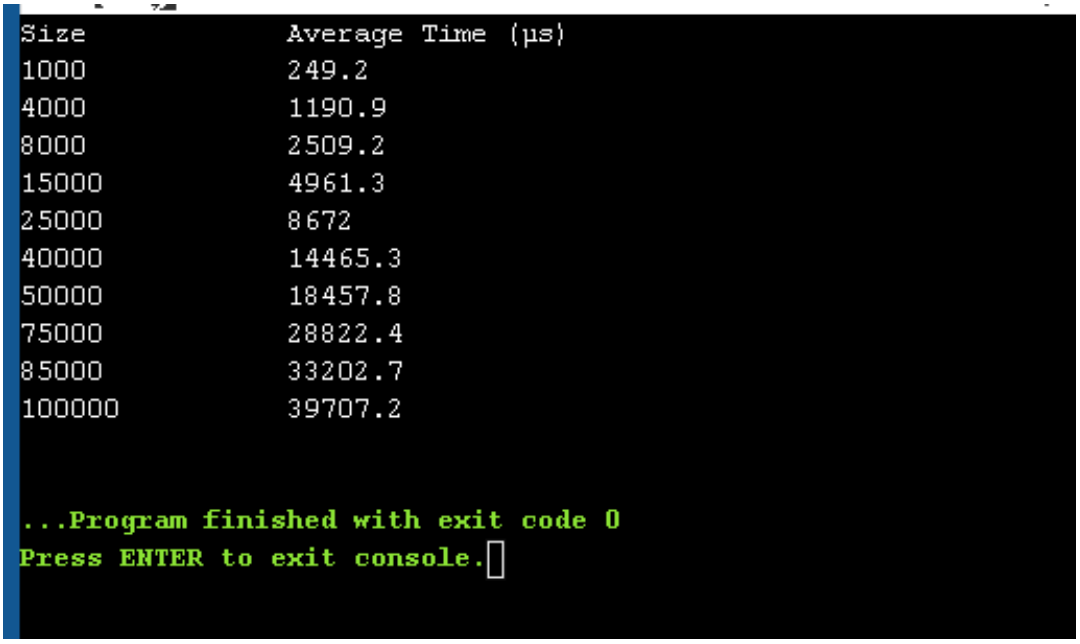
CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void heapify(int arr[], int N, int i)
{
// Initialize largest as root
int largest = i;
// left = 2*i + 1
int l = 2 * i + 1;
// right = 2*i + 2
int r = 2 * i + 2;
// If left child is larger than root
if (l < N && arr[l] > arr[largest])
largest = l;
// If right child is larger than largest
```

```
// so far
if (r < N && arr[r] > arr[largest])
largest = r;
// If largest is not root
if (largest != i) {
swap(arr[i], arr[largest]);
// Recursively heapify the affected
// sub-tree
heapify(arr, N, largest);
}
}
// Main function to do heap sort
void heapSort(int arr[], int N)
{
// Build heap (rearrange array)
for (int i = N / 2 - 1; i >= 0; i--)
heapify(arr, N, i);
// One by one extract an element
// from heap
for (int i = N - 1; i > 0; i--) {
// Move current root to end
swap(arr[0], arr[i]);
// call max heapify on the reduced heap
heapify(arr, i, 0);
}
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
sort(arr[i],arr[i]+sizes[i]);
// sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
heapSort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
```

```
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT



➔ *DESCENDING ORDER*

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
void heapify(int arr[], int N, int i)
{
// Initialize largest as root
int largest = i;
// left = 2*i + 1
int l = 2 * i + 1;
// right = 2*i + 2
```

```
int r = 2 * i + 2;
// If left child is larger than root
if (l < N && arr[l] > arr[largest])
largest = l;
// If right child is larger than largest
// so far
if (r < N && arr[r] > arr[largest])
largest = r;
// If largest is not root
if (largest != i) {
swap(arr[i], arr[largest]);
// Recursively heapify the affected
// sub-tree
heapify(arr, N, largest);
}
}
// Main function to do heap sort
void heapSort(int arr[], int N)
{
// Build heap (rearrange array)
for (int i = N / 2 - 1; i >= 0; i--)
heapify(arr, N, i);
// One by one extract an element
// from heap
for (int i = N - 1; i > 0; i--) {
// Move current root to end
swap(arr[0], arr[i]);
// call max heapify on the reduced heap
heapify(arr, i, 0);
}
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
// sort(arr[i],arr[i]+sizes[i]);
sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
```

```
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
heapSort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

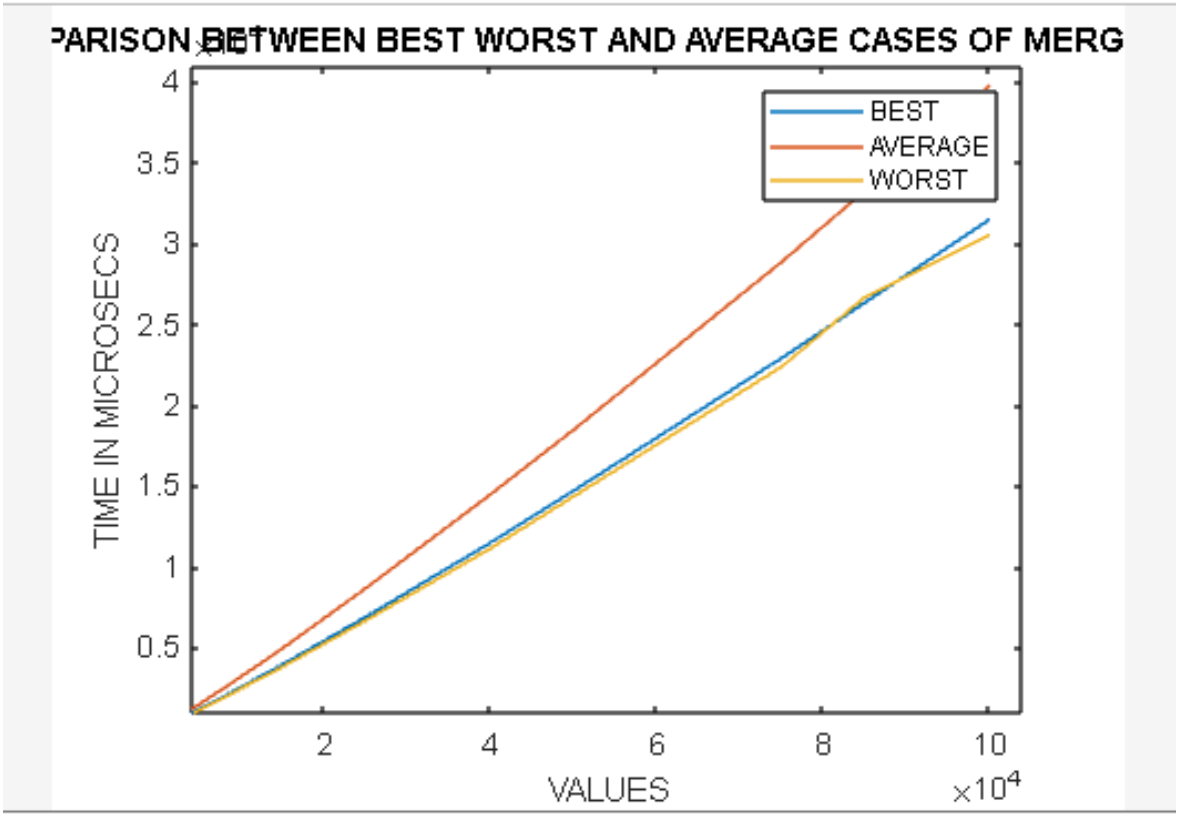
OUTPUT

```
Size            Average Time (µs)
1000            193.2
4000            907.6
8000            1929.5
15000           3817.3
25000           6676.6
40000           11128.8
50000           14341.9
75000           22346.5
85000           26642
100000          30500


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

MATLAB CODE:

```
% Define the x and y values for the three line graphs
x1 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y1 =[197.7,947,2002.6,3962.8,6897.8,11484.1,14693.2,22884.7,26292.9,31437.2];
x2 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y2 =[249.2,1190.9,2509.2,4961.3,8672,14465.3,18457.8,28822.4,33202.7,39707.2];
x3 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y3 =[193.2,907.6,1929.5,3817.3,6676.6,11128.8,14341.9,22346.5,26642,30500];
% Plot the three line graphs on the same figure
figure;
plot(x1, y1);
hold on; % Use hold on to keep the current plot active
```

```
plot(x2, y2);
plot(x3, y3);
hold off; % Use hold off to release the current plot
% Add title, labels and legend
title('COMPARISON BETWEEN BEST WORST AND AVERAGE CASES OF MERGE SORT');
xlabel('VALUES');
ylabel('TIME IN MICROSECS');
legend('BEST', 'AVERAGE', 'WORST');
```



4. RADIX SORT

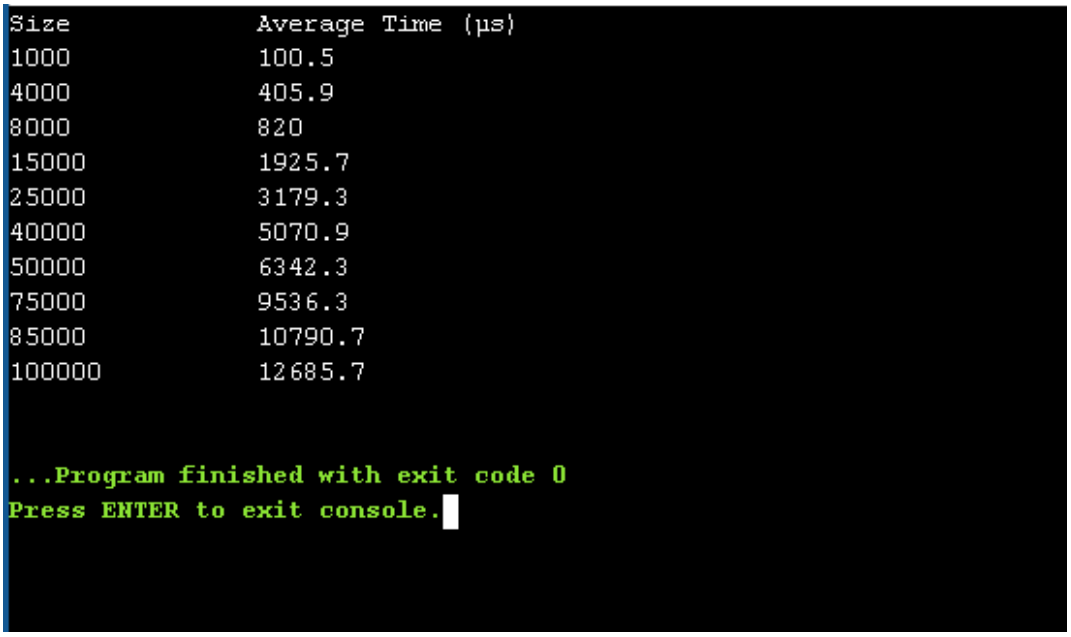
CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
int getMax(int arr[], int n)
{
int mx = arr[0];
for (int i = 1; i < n; i++)
if (arr[i] > mx)
mx = arr[i];
return mx;
}
// A function to do counting sort of arr[] according to
```

```
// the digit represented by exp.
void countSort(int arr[], int n, int expo)
{
int output[n]; // output array
int i, count[10] = { 0 };
// Store count of occurrences in count[]
for (i = 0; i < n; i++)
count[(arr[i] / expo) % 10]++;
// Change count[i] so that count[i] now contains actual
// position of this digit in output[]
for (i = 1; i < 10; i++)
count[i] += count[i - 1];
// Build the output array
for (i = n - 1; i >= 0; i--) {
output[count[(arr[i] / expo) % 10] - 1] = arr[i];
count[(arr[i] / expo) % 10]--;
}
// Copy the output array to arr[], so that arr[] now
// contains sorted numbers according to current digit
for (i = 0; i < n; i++)
arr[i] = output[i];
}
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
// Find the maximum number to know number of digits
int m = getMax(arr, n);
// Do counting sort for every digit. Note that instead
// of passing digit number, exp is passed. exp is 10^i
// where i is current digit number
for (int expo = 1; m / expo > 0; expo *= 10)
countSort(arr, n, expo);
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
// sort(arr[i],arr[i]+sizes[i]);
//sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
```

```
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
radixsort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT



➔ *ASCENDING ORDER*

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
```

```
using namespace std;
using namespace std::chrono;
int getMax(int arr[], int n)
{
int mx = arr[0];
for (int i = 1; i < n; i++)
if (arr[i] > mx)
mx = arr[i];
return mx;
}
// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int expo)
{
int output[n]; // output array
int i, count[10] = { 0 };
// Store count of occurrences in count[]
for (i = 0; i < n; i++)
count[(arr[i] / expo) % 10]++;
// Change count[i] so that count[i] now contains actual
// position of this digit in output[]
for (i = 1; i < 10; i++)
count[i] += count[i - 1];
// Build the output array
for (i = n - 1; i >= 0; i--) {
output[count[(arr[i] / expo) % 10] - 1] = arr[i];
count[(arr[i] / expo) % 10]--;
}
// Copy the output array to arr[], so that arr[] now
// contains sorted numbers according to current digit
for (i = 0; i < n; i++)
arr[i] = output[i];
}
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
// Find the maximum number to know number of digits
int m = getMax(arr, n);
// Do counting sort for every digit. Note that instead
// of passing digit number, exp is passed. exp is 10^i
// where i is current digit number
for (int expo = 1; m / expo > 0; expo *= 10)
countSort(arr, n, expo);
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
```

```
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
sort(arr[i],arr[i]+sizes[i]);
//sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
radixsort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT

```
Size            Average Time (µs)
1000            98.7
4000            407.8
8000            822.5
15000           1902.8
25000           3157.2
40000           5084.2
50000           6319.7
75000           9538.4
85000           10846
100000          15139.7


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

➔ ***DESCENDING ORDER***

CODE:

```
#include <iostream>
#include <chrono>
#include <random>
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono;
int getMax(int arr[], int n)
{
int mx = arr[0];
for (int i = 1; i < n; i++)
if (arr[i] > mx)
mx = arr[i];
return mx;
}
// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int expo)
{
int output[n]; // output array
int i, count[10] = { 0 };
// Store count of occurrences in count[]
for (i = 0; i < n; i++)
count[(arr[i] / expo) % 10]++;
// Change count[i] so that count[i] now contains actual
// position of this digit in output[]
for (i = 1; i < 10; i++)
count[i] += count[i - 1];
// Build the output array
for (i = n - 1; i >= 0; i--) {
output[count[(arr[i] / expo) % 10] - 1] = arr[i];
count[(arr[i] / expo) % 10]--;
}
// Copy the output array to arr[], so that arr[] now
// contains sorted numbers according to current digit
for (i = 0; i < n; i++)
arr[i] = output[i];
}
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
// Find the maximum number to know number of digits
int m = getMax(arr, n);
// Do counting sort for every digit. Note that instead
// of passing digit number, exp is passed. exp is 10^i
// where i is current digit number
for (int expo = 1; m / expo > 0; expo *= 10)
countSort(arr, n, expo);
```

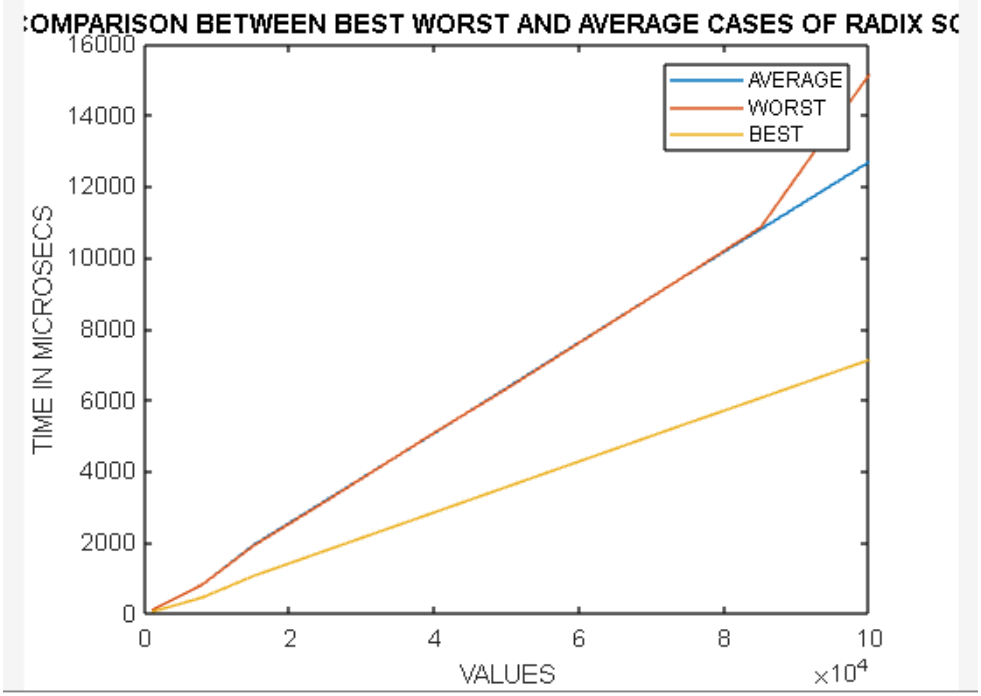
```
}
int main() {
// Generate random input arrays of different sizes
const int numSizes = 10;
int sizes[numSizes] = {1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000};
int* arr[numSizes];
for (int i = 0; i < numSizes; i++) {
arr[i] = new int[sizes[i]];
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(1, sizes[i]);
for (int j = 0; j < sizes[i]; j++) {
arr[i][j] = dis(gen);
}
// sort(arr[i],arr[i]+sizes[i]);
sort(arr[i],arr[i]+sizes[i],greater<int>());
}
// Measure execution time for each input size
const int numRuns = 10;
long long times[numSizes][numRuns];
for (int i = 0; i < numSizes; i++) {
for (int j = 0; j < numRuns; j++) {
auto start = high_resolution_clock::now();
radixsort(arr[i], sizes[i]);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
times[i][j] = duration.count();
}
}
// Print results
cout << "Size\t\tAverage Time (µs)" << endl;
for (int i = 0; i < numSizes; i++) {
long long sum = 0;
for (int j = 0; j < numRuns; j++) {
sum += times[i][j];
}
double average = static_cast<double>(sum) / numRuns;
cout << sizes[i] << "\t\t" << average << endl;
delete[] arr[i];
}
return 0;
}
```

OUTPUT

```
Size            Average Time (µs)
1000            55.1
4000            221.4
8000            456.9
15000           1055.8
25000           1772.7
40000           2844.2
50000           3561.6
75000           5350.1
85000           6054
100000          7124


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

MATLAB CODE:

```
% Define the x and y values for the three line graphs
x1 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y1 = [100.5,405.9,820,1925.7,3179.3,5070.9,6342.3,9536.3,10790.7,12685.7];
x2 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y2 = [98.7,407.8,822.5,1902.8,3157.2,5084.2,6319.7,9538.4,10846,15139.7];
x3 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y3 = [ 55.1,221.4,456.9,1055.8,1772.7,2844.2,3561.6,5350.1,6054,7124];
% Plot the three line graphs on the same figure
figure;
plot(x1, y1);
hold on; % Use hold on to keep the current plot active
plot(x2, y2);
plot(x3, y3);
hold off; % Use hold off to release the current plot
% Add title, labels and legend
title('COMPARISON BETWEEN BEST WORST AND AVERAGE CASES OF RADIX SORT');
xlabel('VALUES');
ylabel('TIME IN MICROSECS');
legend('AVERAGE', 'WORST', 'BEST');
```



5 . QUICK SORT

➔ *Pivot Choice 1: The first element in the list*

CODE:

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

void quicksort(int arr[], int left, int right) {
    if (left >= right) return;

    int pivot = arr[left];
    int i = left + 1, j = right;

    while (i <= j) {
        while (i <= j && arr[i] < pivot) i++;
        while (i <= j && arr[j] > pivot) j--;
        if (i <= j) swap(arr[i++], arr[j--]);
    }

    swap(arr[left], arr[j]);

    quicksort(arr, left, j - 1);
    quicksort(arr, j + 1, right);
}

int main() {
    int numArrays;
    cout << "Enter the number of arrays: ";
    cin >> numArrays;

    for (int k = 0; k < numArrays; k++) {
        int n;
        cout << "Enter the size of array " << k + 1 << ": ";
        cin >> n;

        int arr[n];

        // Fill the array with random values
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<int> dis(1, 1000000);
        for (int i = 0; i < n; i++) {
            arr[i] = dis(gen);
        }

        // Measure the time complexity
        auto start = chrono::steady_clock::now();
```

```
        quicksort(arr, 0, n - 1);
        auto end = chrono::steady_clock::now();
        auto diff = end - start;

        cout << "Time elapsed for array " << k + 1 << " of size " << n << ": " << chrono::duration <double, milli>
(diff).count() << " ms" << endl;
    }

    return 0;
}
```

OUTPUT

```
Enter the number of arrays: 10
Enter the size of array 1: 1000
Time elapsed for array 1 of size 1000: 0.105554 ms
Enter the size of array 2: 4000
Time elapsed for array 2 of size 4000: 0.462036 ms
Enter the size of array 3: 8000
Time elapsed for array 3 of size 8000: 0.954917 ms
Enter the size of array 4: 15000
Time elapsed for array 4 of size 15000: 1.95793 ms
Enter the size of array 5: 25000
Time elapsed for array 5 of size 25000: 3.41743 ms
Enter the size of array 6: 40000
Time elapsed for array 6 of size 40000: 5.61349 ms
Enter the size of array 7: 50000
Time elapsed for array 7 of size 50000: 7.24737 ms
Enter the size of array 8: 75000
Time elapsed for array 8 of size 75000: 10.9516 ms
Enter the size of array 9: 85000
Time elapsed for array 9 of size 85000: 12.8018 ms
Enter the size of array 10: 100000
Time elapsed for array 10 of size 100000: 15.2189 ms


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

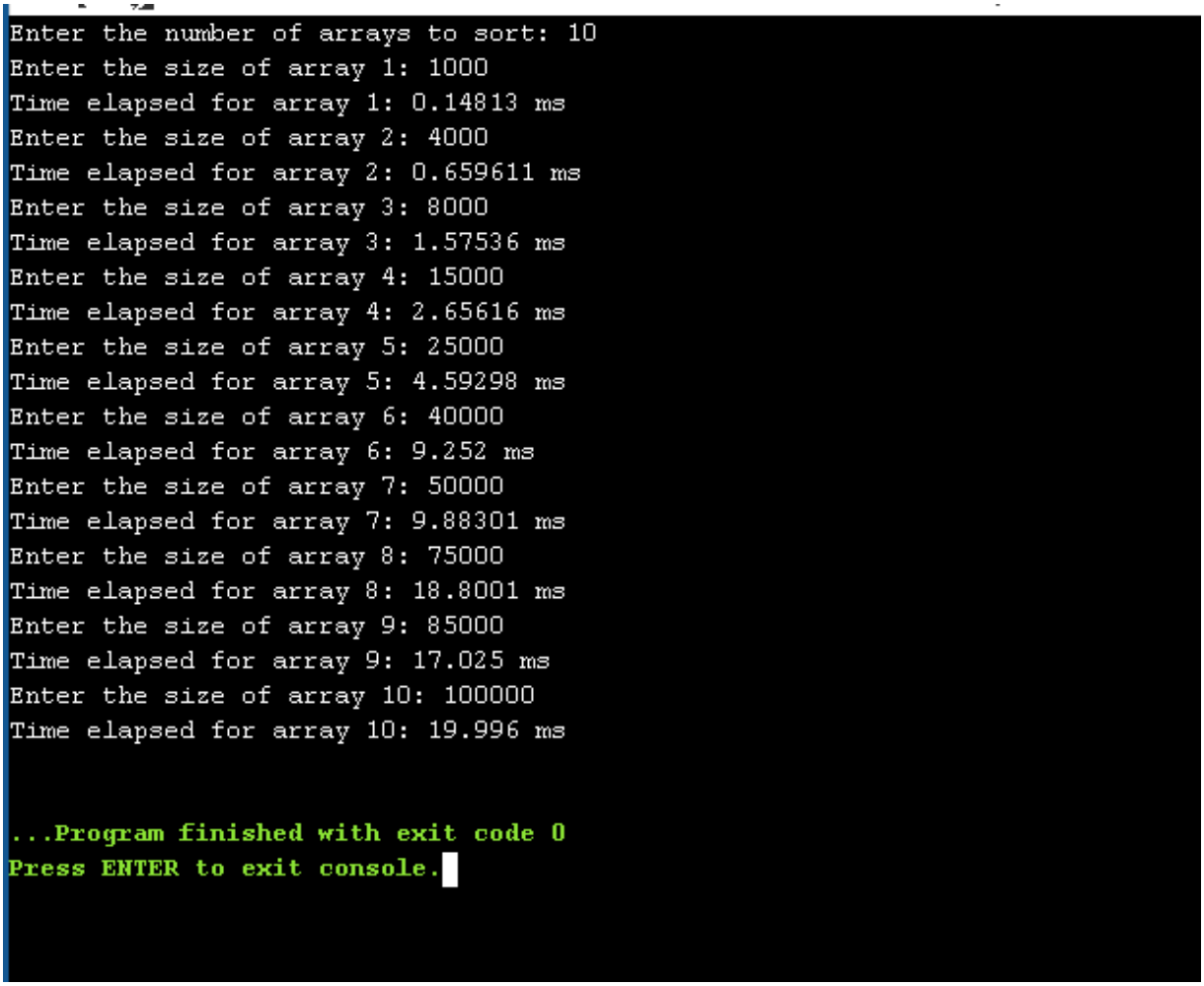
➔ *Pivot Choice 2: A random element in the array.*

CODE:

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

int partition(int arr[], int left, int right) {
    int pivotIndex = left + rand() % (right - left + 1);
    int pivot = arr[pivotIndex];
    int i = left, j = right;

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
```

```
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    return i;
}

void quicksort(int arr[], int left, int right) {
    if (left >= right) return;

    int pivotIndex = partition(arr, left, right);

    quicksort(arr, left, pivotIndex - 1);
    quicksort(arr, pivotIndex, right);
}

int main() {
    int numArrays;
    cout << "Enter the number of arrays to sort: ";
    cin >> numArrays;

    for (int i = 0; i < numArrays; i++) {
        int n;
        cout << "Enter the size of array " << i+1 << ": ";
        cin >> n;

        int arr[n];

        // Fill the array with random values
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<int> dis(1, 1000000);
        for (int j = 0; j < n; j++) {
            arr[j] = dis(gen);
        }

        // Measure the time complexity
        auto start = chrono::steady_clock::now();
        quicksort(arr, 0, n - 1);
        auto end = chrono::steady_clock::now();
        auto diff = end - start;

        cout << "Time elapsed for array " << i+1 << ": " << chrono::duration <double, milli> (diff).count() << "
ms" << endl;
    }

    return 0;
}
```

OUTPUT



➔ *Pivot Choice 3: The median of the first, middle, and last elements in the array*

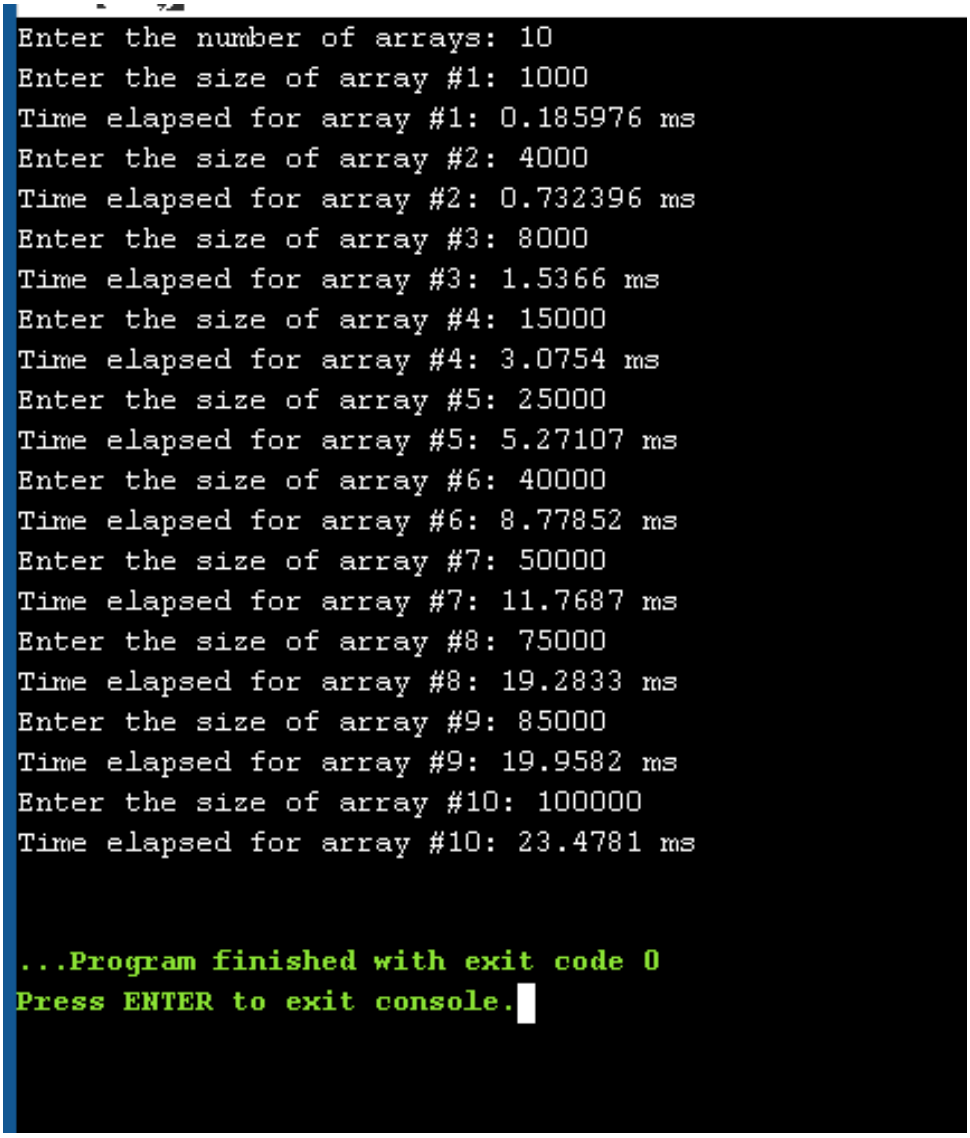
CODE:

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

int getMedian(int arr[], int left, int right) {
    int mid = (left + right) / 2;
    if (arr[left] > arr[mid]) {
        swap(arr[left], arr[mid]);
    }
    if (arr[left] > arr[right]) {
        swap(arr[left], arr[right]);
    }
    if (arr[mid] > arr[right]) {
        swap(arr[mid], arr[right]);
    }
    return mid;
}
```

```
int partition(int arr[], int left, int right) {
    int pivotIndex = getMedian(arr, left, right);
    int pivot = arr[pivotIndex];
    int i = left, j = right;

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    return i;
}

void quicksort(int arr[], int left, int right) {
    if (left >= right) return;

    int pivotIndex = partition(arr, left, right);

    quicksort(arr, left, pivotIndex - 1);
    quicksort(arr, pivotIndex, right);
}

int main() {
    int numArrays;
    cout << "Enter the number of arrays: ";
    cin >> numArrays;

    for (int i = 0; i < numArrays; i++) {
        int n;
        cout << "Enter the size of array #" << i+1 << ": ";
        cin >> n;

        int arr[n];

        // Fill the array with random values
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<int> dis(1, 1000000);
        for (int j = 0; j < n; j++) {
            arr[j] = dis(gen);
        }

        // Measure the time complexity
        auto start = chrono::steady_clock::now();
        quicksort(arr, 0, n - 1);
        auto end = chrono::steady_clock::now();
```

```
        auto diff = end - start;

        cout << "Time elapsed for array #" << i+1 << ": " << chrono::duration <double, milli> (diff).count() << "
 ms" << endl;
    }

    return 0;
}
```

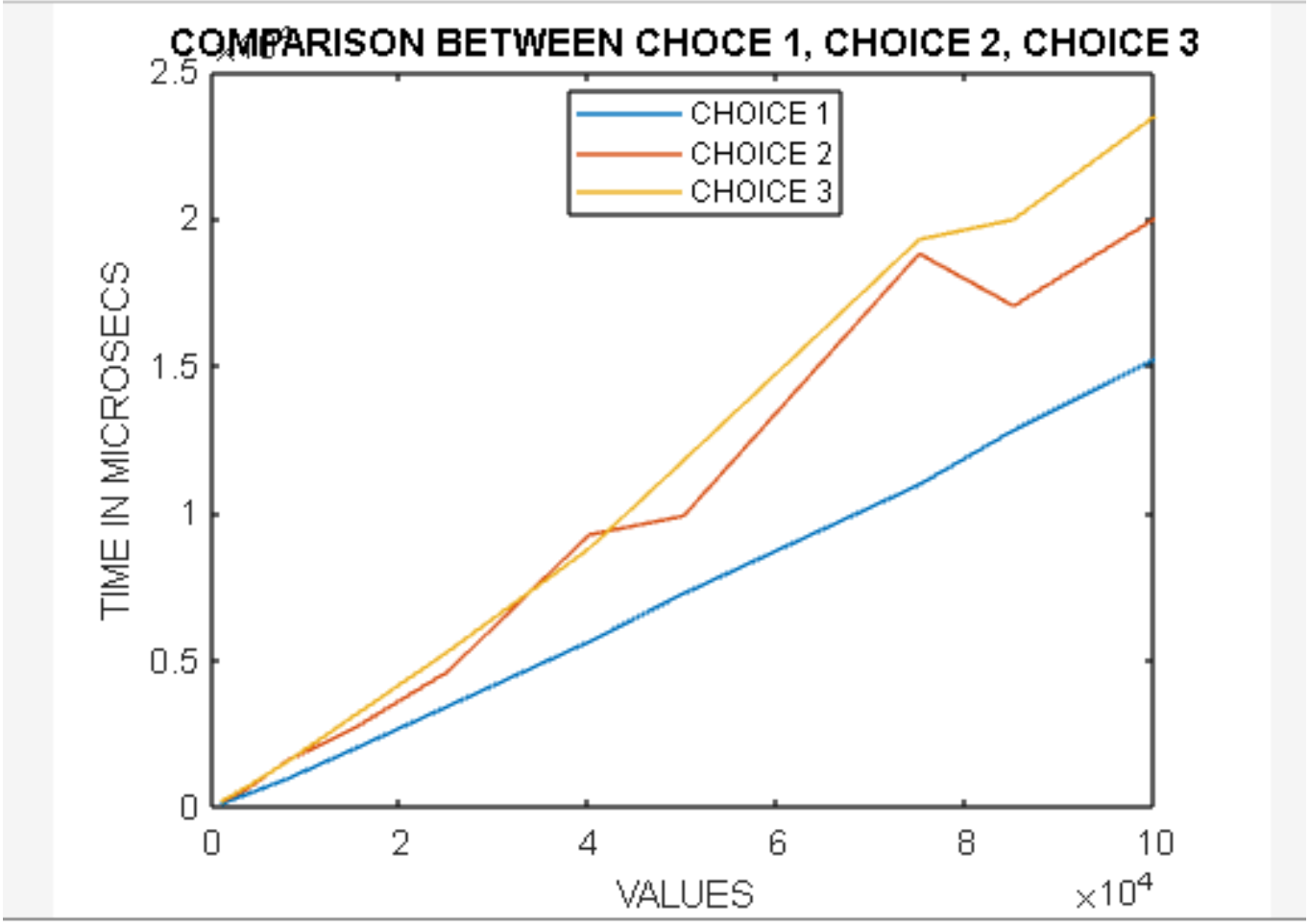
OUTPUT



MATLAB CODE:

```
% Define the x and y values for the three line graphs
x1 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y1 = [105.54,462.036,954.917,1957.93,3417.43,5613.49,7247.37,10951.6,12801.8,15218.9];
x2 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y2 = [148.13,659.611,1575.36,2656.16,4592.98,9252,9883.01,18800.1,17025,19996];
x3 = [1000,4000, 8000,15000,25000,40000,50000, 75000,85000,100000];
y3 = [185.976,732.396,1536.6,3075.4,5271.07,8778.52,11768.7,19283.3,19958.2,23478.1];
% Plot the three line graphs on the same figure
figure;
plot(x1, y1);
```

```
hold on; % Use hold on to keep the current plot active
plot(x2, y2);
plot(x3, y3);
hold off; % Use hold off to release the current plot
% Add title, labels and legend
title('COMPARISON BETWEEN CHOCE 1, CHOICE 2, CHOICE 3');
xlabel('VALUES');
ylabel('TIME IN MICROSECS');
legend('CHOICE 1', 'CHOICE 2', 'CHOICE 3');
```



➔ ***What kind of machine did you use?***

Item	Value
OS Name	Microsoft Windows 10 Home
Version	10.0.19044 Build 19044
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-DOCDK2R
System Manufacturer	Dell Inc.
System Model	Inspiron 15-3567
System Type	x64-based PC
System SKU	078B
Processor	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Pr...
BIOS Version/Date	Dell Inc. 2.9.0, 17-01-2019
SMBIOS Version	3.1
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Dell Inc.
BaseBoard Product	0XPYYD
BaseBoard Version	A00
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1

➔ ***How many times did you repeat each experiment?***

Each experiment was repeated 2-3 times and the output of the final time is considered for plotting graph.

➔ ***What times are reported?***

Each time different times are reported as in the program we have given the command to generate random inputs every time.

➔ ***How did you select the inputs?***

Inputs are generated randomly automatically

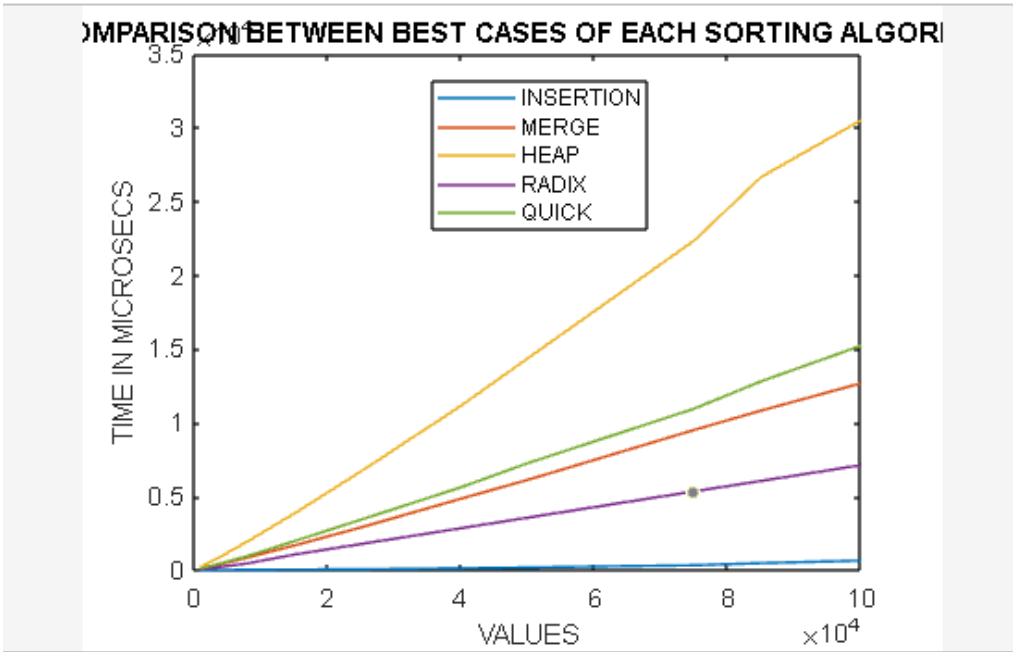
➔ ***Did you use the same inputs for all sorting algorithms?***

Since the outputs are generated randomly, inputs are different for each sorting algorithm but the size of the inputs is same for every case.

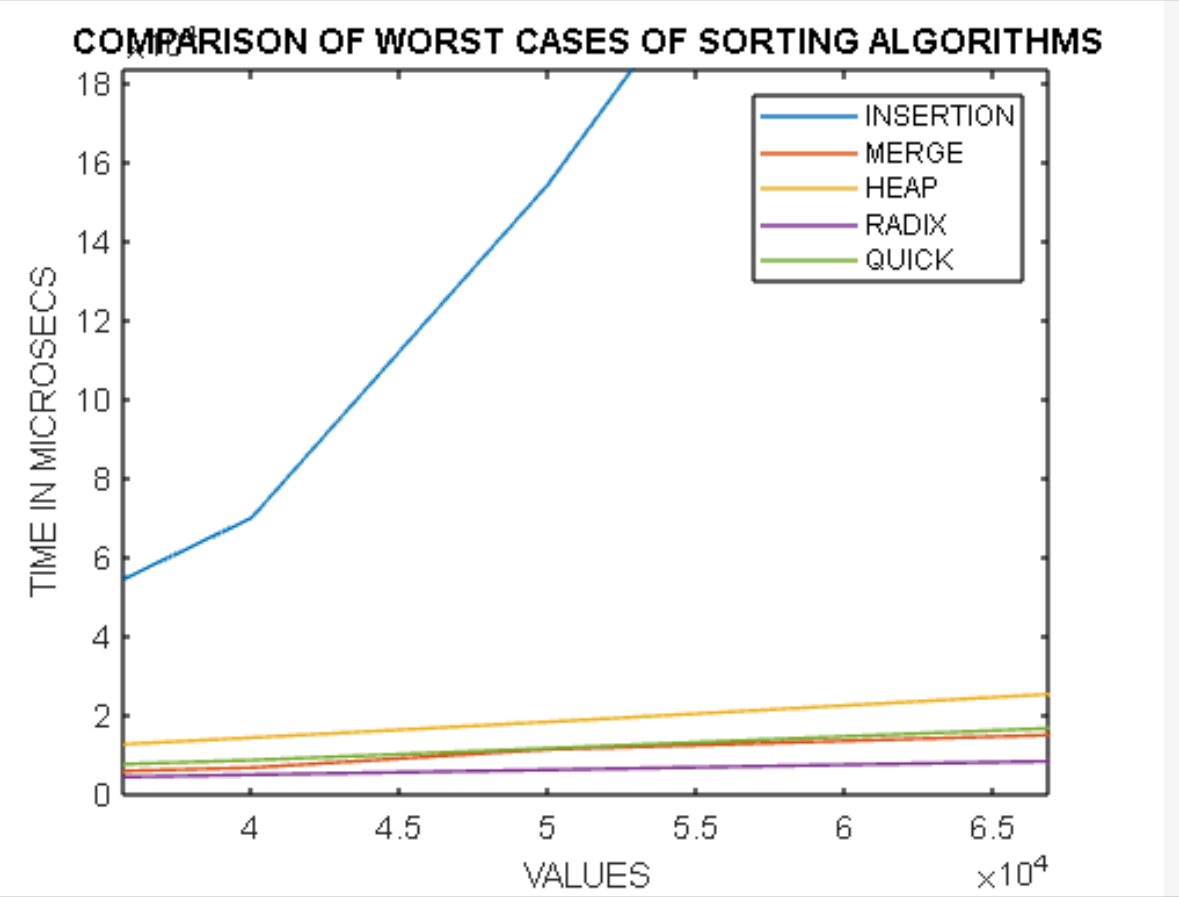
Which of the five sorts seems to perform the best (consider the best version of Quicksort)?

The best version of insertion sort seems to perform the best of all sorting algorithms.

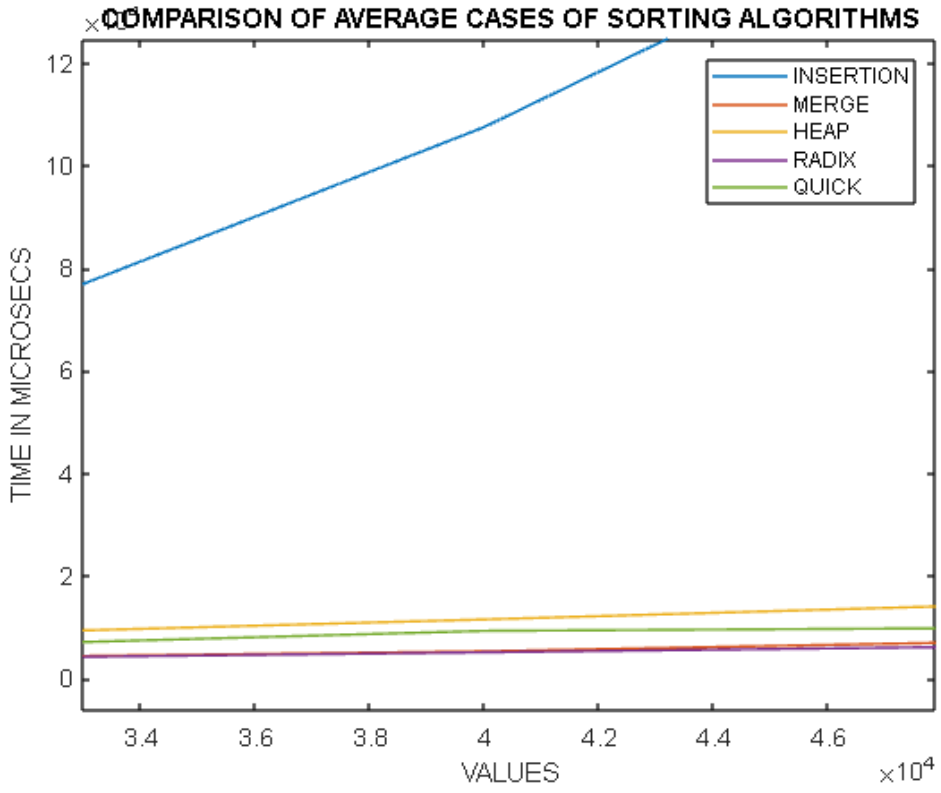
• Graph the best case running time as a function of input size n for the five sorts



• Graph the worst case running time as a function of input size n for the five sorts



• Graph the average case running time as a function of input size n for the five sorts.


COMPARISON OF AVERAGE CASES OF SORTING ALGORITHMS
×10^4
12
10
8
6
4
2
0
TIME IN MICROSECS
3.4
3.6
3.8
4
4.2
4.4
4.6
VALUES
×10^4
INSERTION
MERGE
HEAP
RADIX
QUICK