

**North South University**

**Department Of Electrical and Computer Science and Engineering**

**Student Name:** Touhid Ara Himu

**Student ID:**2312187642

**Course Name**: CSE225.11

**Faculty Name**: Mohammad Rezwanul Huq

**Assignment Topic:** Performance Comparison of Sorting Algorithms

**Introduction**

This report presents a performance comparison of five different sorting algorithms Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort applied to datasets of various sizes. Each algorithm's execution time was measured on randomly generated datasets to analyze their efficiency and suitability for different dataset sizes.

**Computer Configuration**

To ensure reproducibility of the performance results, the following computer configuration was used for testing:

* Processor: Intel Core i3-7100, 2 Cores, 3.9 GHz
* RAM: 8 GB DDR4
* Operating System: Windows 10 Pro, 64-bit
* IDE: CodeBlocks 20.03
* Compiler: GCC included with CodeBlocks
* Time Measurement Function: clock() function in C++ for recording execution time

This configuration allows for a clear assessment of sorting performance across different dataset sizes.

**Pseudocodes and Codes**

**Bubble Sort**

void bubbleSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

}

}

}

}

**Selection Sort**

void selectionSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

int index = i;

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

if (arr[j] < arr[index]) {

index = j;

}

}

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

}

}

**Insertion Sort**

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 = j - 1;

}

arr[j + 1] = key;

}

}

**Merge Sort**

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

**Quick Sort**

int quick(int arr[], int low, int high) {

int sort = arr[high];

int i = (low - 1);

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

if (arr[j] < sort) {

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 = quick(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

**Created Random Array**

void randomArray(int arr[], int n) {

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

arr[i] = rand() % 100000;

}

}

**Execution Time**

void executionTime(void (\*sortFunction)(int[], int), int arr[], int n, const string &algorithmName) {

int\* tempArr = new int[n];

copy(arr, arr + n, tempArr);

clock\_t start = clock();

sortFunction(tempArr, n);

clock\_t end = clock();

double timeTaken = double(end - start) / CLOCKS\_PER\_SEC;

cout << algorithmName << " Time " << timeTaken << " seconds." << endl;

delete[] tempArr;

}

**Overload Function for merge sort and quick sort**

void executionTime(void (\*sortFunction)(int[], int, int), int arr[], int n, const string &algorithmName) {

int\* tempArr = new int[n];

copy(arr, arr + n, tempArr);

clock\_t start = clock();

sortFunction(tempArr, 0, n - 1);

clock\_t end = clock();

double timeTaken = double(end - start) / CLOCKS\_PER\_SEC;

cout << algorithmName << " Time " << timeTaken << " seconds." << endl;

delete[] tempArr;

}

**Main Function**

int main() {

srand(time(0));

const int sizes[] = {100, 2000, 40000, 100000};

for (int size : sizes) {

cout << "\nDataset Size: " << size << endl;

int\* dataset = new int[size];

randomArray(dataset, size);

executionTime(bubbleSort, dataset, size, "Bubble Sort");

executionTime(selectionSort, dataset, size, "Selection Sort");

executionTime(insertionSort, dataset, size, "Insertion Sort");

executionTime(mergeSort, dataset, size, "Merge Sort");

executionTime(quickSort, dataset, size, "Quick Sort");

delete[] dataset;

}

return 0;

}

**Output**

Dataset Size: 100

Bubble Sort Time 0 seconds.

Selection Sort Time 0 seconds.

Insertion Sort Time 0 seconds.

Merge Sort Time 0 seconds.

Quick Sort Time 0 seconds.

Dataset Size: 2000

Bubble Sort Time 0.015 seconds.

Selection Sort Time 0.016 seconds.

Insertion Sort Time 0 seconds.

Merge Sort Time 0 seconds.

Quick Sort Time 0 seconds.

Dataset Size: 40000

Bubble Sort Time 5.686 seconds.

Selection Sort Time 1.625 seconds.

Insertion Sort Time 0.89 seconds.

Merge Sort Time 0 seconds.

Quick Sort Time 0.016 seconds.

Dataset Size: 100000

Bubble Sort Time 34.945 seconds.

Selection Sort Time 10.169 seconds.

Insertion Sort Time 5.561 seconds.

Merge Sort Time 0 seconds.

Quick Sort Time 0.016 seconds.

Process returned 0 (0x0) execution time : 59.564 s

Press any key to continue.

**Full Code Link**:[2312187\_A01.cpp](file:///Users/himu/Downloads/2312187_A01.cpp)

**Comparison Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data size | Bubble | Selection | Insertion | Merge | Quick |
| 100 | 0sec | 0sec | 0sec | 0sec | 0sec |
| 2000 | 0.015sec | 0.016sec | 0sec | 0sec | 0sec |
| 40000 | 5.686sec | 1.625sec | 0.89sec | 0sec | 0sec |
| 100000 | 34.945sec | 10.169sec | 5.561sec | 0sec | 0.016sec |

**Discussion:**From the data collected, Bubble Sort, Selection Sort, and Insertion Sort show higher execution times for larger datasets due to Ocomplexity, making them unsuitable for large datasets. Merge Sort and Quick Sort, with Ocomplexity, are far more efficient on large datasets.

**Conclusion:** This report analyzed five sorting algorithms by implementing them in C++ and measuring their performance on datasets of increasing size. While Bubble Sort, Selection Sort, and Insertion Sort have high time complexity and were inefficient on large datasets, Merge Sort and Quick Sort proved to be faster and more suited for larger datasets.