بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

Palestine Polytechnic University

College of Information Technology and Computer Engineering

Design and analysis of algorithms

Dr.Nabil Arman

Aram Nasser Al-shahateet

221020

In this project, we implemented three sorting algorithms: Insertion Sort, Merge Sort, and Quick Sort.

Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is generally less efficient on large lists compared to more advanced algorithms like Quick Sort, Heap Sort, or Merge Sort. However, Insertion Sort offers several advantages:

Simple implementation: Jon Bentley demonstrates a three-line C version and a five-line optimized version.

Efficient for small datasets: It performs well on small datasets, similar to other quadratic sorting algorithms.

More efficient in practice than other simple quadratic algorithms**: It outperforms Selection Sort and Bubble Sort in practical scenarios.

Adaptive: It is efficient for datasets that are already mostly sorted, with a time complexity of O(n) when each element is no more than k positions away from its sorted position.

Stable: It maintains the relative order of elements with equal keys.

In-place: It requires only a constant amount of additional memory space (O(1)).

Online: It can sort a list as it receives it.

Insertion Sort works by iterating through the input list, consuming one element at a time, and growing a sorted output list. At each iteration, it removes one element from the input data, finds the appropriate location within the sorted list, and inserts it there. This process is repeated until no input elements remain.

Sorting is typically done in-place by iterating up the array and growing the sorted list behind it. At each array position, the algorithm compares the current value with the largest value in the sorted list (which is next to it, in the previous array position). If the current value is larger, it leaves the element in place and moves to the next position. If the current value is smaller, it finds the correct position within the sorted list, shifts all larger values up to create a space, and inserts the element into the correct position.

After k iterations, the first k + 1 entries in the array are sorted. In each iteration, the first remaining entry of the input is removed and inserted into the result at the correct position, thereby extending the sorted list.

The pseudocode for the complete algorithm, assuming zero-based arrays, is as follows:

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

The time complexity of the Insertion Sort algorithm is Θ(n^2).

Merge Sort

Merge Sort is an efficient, general-purpose, comparison-based sorting algorithm. It is known for producing a stable sort, which means it maintains the input order of equal elements in the sorted output. Merge Sort is a divide and conquer algorithm, originally invented by John von Neumann in 1945. A detailed description and analysis of bottom-up Merge Sort appeared in a report by Goldstine and von Neumann as early as 1948.

Conceptually, Merge Sort works as follows:

1. Divide the unsorted list into (n) sublists, each containing one element (a list of one element is considered sorted).

2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

The pseudocode for Merge Sort, sorting elements from `lo` to `hi` (exclusive) of array `A`, is:

```
algorithm mergesort(A, lo, hi) is

   if lo + 1 < hi then  // Two or more elements.

      mid = ⌊(lo + hi) / 2⌋

      fork mergesort(A, lo, mid)

      mergesort(A, mid, hi)

      join

      merge(A, lo, mid, hi)
```

The time complexity of Merge Sort is (Θ(nlog n)).

Quick Sort

Quick Sort is an efficient sorting algorithm used to order the elements of an array systematically. Developed by Tony Hoare in 1959 and published in 1961, Quick Sort remains a widely used algorithm. When implemented effectively, it can be about two or three times faster than its main competitors, Merge Sort and Heap Sort.

Quick Sort is a comparison sort, meaning it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. In efficient implementations, it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quick Sort can operate in-place on an array, requiring only small additional amounts of memory to perform the sorting. It is very similar to Selection Sort, except that it does not always choose the worst-case partition.

Mathematical analysis of Quick Sort shows that, on average, the algorithm takes (O(n log n)) comparisons to sort (n) items. In the worst case, it makes (O(n^2)) comparisons, though this behavior is rare.

Quick Sort is a divide and conquer algorithm. It first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick Sort then recursively sorts the sub-arrays. The steps are:

1. Pick an element, called a pivot, from the array.

2. Partition the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted. The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

The pseudocode for Quick Sort is:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)


algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo - 1
    for j := lo to hi - 1 do
        if A[j] < pivot then
            i := i + 1
            swap A[i] with A[j]
    swap A[i + 1] with A[hi]
    return i + 1
```

The time complexity of Quick Sort is ($\Theta(n \log n)$).

Practical Application

Type of sorting algorithms	Data Type	Time(ms)
Processing set #1 with size 15000		
Random	Insertion sort	672
	Merge Sort	33
	Quick Sort	4
Sorted	Insertion sort	0
	Merge Sort	25
	Quick Sort	3657
Reverse Sorted	Insertion sort	1334
	Merge Sort	24
	Quick Sort	2037
Processing set #2 with size 16000		
Random	Insertion sort	813
	Merge Sort	29
	Quick Sort	8
Sorted	Insertion sort	0
	Merge Sort	40
	Quick Sort	4010
Reverse Sorted	Insertion sort	1964
	Merge Sort	28
	Quick Sort	2837
Processing set #3 with size 17000		
Random	Insertion sort	1080
	Merge Sort	47
	Quick Sort	12
Sorted	Insertion sort	0
	Merge Sort	37
	Quick Sort	5846
Reverse Sorted	Insertion sort	2206
	Merge Sort	28
	Quick Sort	3123
Processing set #4 with size 18000		
Random	Insertion sort	1009
	Merge Sort	33
	Quick Sort	7
Sorted	Insertion sort	0
	Merge Sort	31
	Quick Sort	5454
Reverse Sorted	Insertion sort	1997
	Merge Sort	29
	Quick Sort	3132
Processing set #5 with size 19000		
Random	Insertion sort	1431
	Merge Sort	46
	Quick Sort	8
Sorted	Insertion sort	0
	Merge Sort	56
	Quick Sort	10671
Reverse Sorted	Insertion sort	4298
	Merge Sort	72
	Quick Sort	7623
Processing set #6 with size 20000		

Random	Insertion sort	2572
	Merge Sort	121
	Quick Sort	17
Sorted	Insertion sort	0
	Merge Sort	116
	Quick Sort	11287
Reverse Sorted	Insertion sort	4078
	Merge Sort	67
	Quick Sort	7061

Analysis of Sorting Algorithm Performance

Based on the provided data, we analyze the performance of three sorting algorithms: Insertion Sort, Merge Sort, and Quick Sort, across six different input sizes (15,000 to 20,000) and three data types (random, sorted, and reverse sorted).

Random Data

Insertion Sort: As expected, Insertion Sort performs poorly on large random datasets due to its (O(n^2)) time complexity. The processing time ranges from 672 ms (for size 15,000) to 2572 ms (for size 20,000).

Merge Sort: Merge Sort, with its (O(n log n)) time complexity, handles random data efficiently, with times ranging from 33 ms to 121 ms.

Quick Sort: Quick Sort performs the best on random data with times as low as 4 ms and up to 17 ms. This aligns with its average-case (O(n log n)) time complexity.

Sorted Data

Insertion Sort: Insertion Sort performs exceptionally well on sorted data, taking 0 ms consistently across all sizes. This is due to its adaptive nature, making it (O(n)) for nearly sorted data.

Merge Sort: Merge Sort remains efficient with times slightly increasing with input size, ranging from 25 ms to 116 ms.

Quick Sort: Quick Sort, however, struggles with sorted data due to its worst-case (O(n^2)) time complexity if the pivot is poorly chosen. The times range from 3657 ms (for size 15,000) to 11,287 ms (for size 20,000).

Reverse Sorted Data

Insertion Sort: Insertion Sort is inefficient on reverse sorted data, with times significantly higher than on random data, ranging from 1334 ms to 4078 ms.

Merge Sort: Merge Sort remains efficient and stable with times from 24 ms to 67 ms.

Quick Sort: Quick Sort's performance is worse on reverse sorted data compared to random data, but better than sorted data, with times ranging from 2037 ms to 7061 ms.

Summary

1. **Insertion Sort** is highly inefficient for large random or reverse sorted data but excels with pre-sorted data.

2. **Merge Sort** consistently performs well across all data types due to its stable (O(n log n)) time complexity.

3. **Quick Sort** is extremely efficient on random data but suffers with sorted and reverse sorted data due to potential poor pivot choices leading to \(O(n^2)\) time complexity.

Recommendations

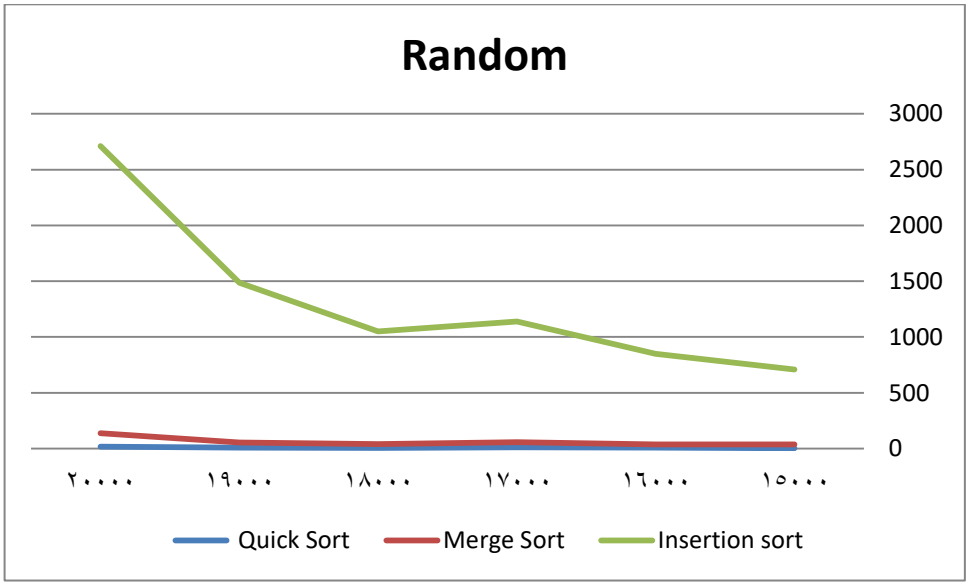
- For large datasets, **Merge Sort** is generally the most reliable choice due to its consistent performance.

- **Quick Sort** can be very effective with random data but requires careful implementation to avoid worst-case scenarios.

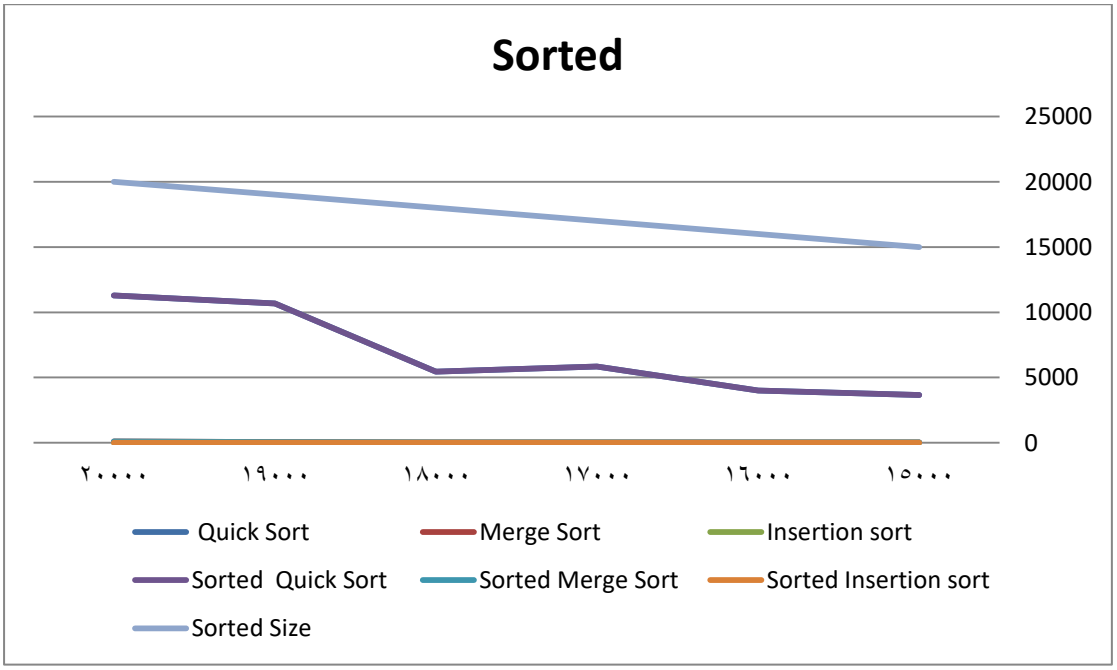
- **Insertion Sort** should be reserved for small or nearly sorted datasets where its simplicity and adaptive nature can be leveraged.

This analysis provides insights into selecting the appropriate sorting algorithm based on the nature of the input data and the dataset size.

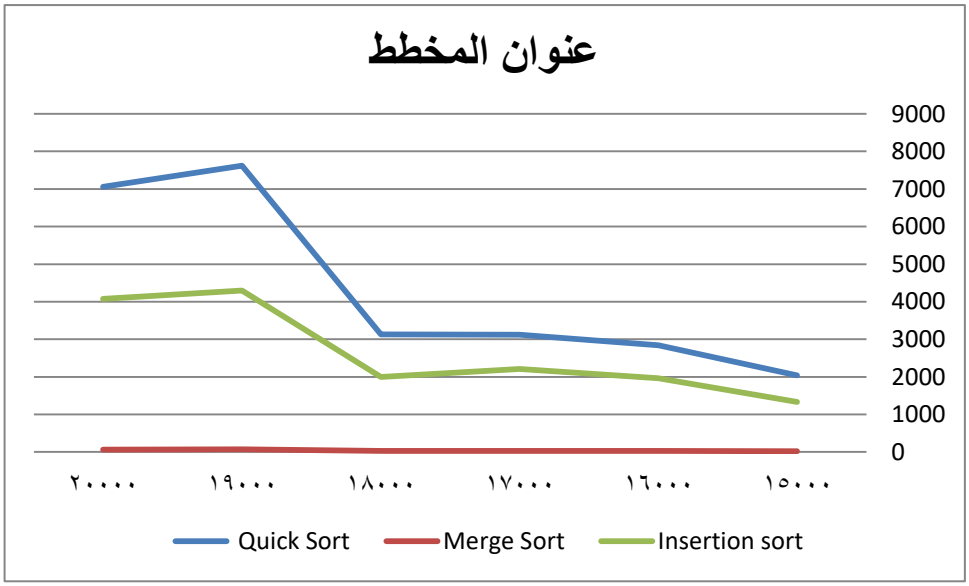
Random			
Size	**Insertion sort**	**Merge Sort**	**Quick Sort**
15000	672	33	4
16000	813	29	8
17000	1080	47	12
18000	1009	33	7
19000	1431	46	8
20000	2572	121	17



Sorted			
Size	**Insertion sort**	**Merge Sort**	**Quick Sort**
15000	0	25	3657
16000	0	40	4010
17000	0	37	5846
18000	0	31	5454
19000	0	56	10671
20000	0	116	11287



Sorted			
Size	**Insertion sort**	**Merge Sort**	**Quick Sort**
15000	1334	24	2037
16000	1964	28	2837
17000	2206	28	3123
18000	1997	29	3132
19000	4298	72	7623
20000	4078	67	7061



Here the c++ code:

```
#include <iostream<
#include <vector<
#include <algorithm<
#include <chrono<
#include <random<

using namespace std؛
using namespace std::chrono؛

void generate_data(vector<int>& data, int size, int case_type} (
    random_device rd؛
    mt19937 gen(rd؛(()
    uniform_int_distribution<> dis(1, 1000000؛(

    data.clear؛()
    for (int i = 0; i < size; ++i} (
        data.push_back(dis(gen؛((
{

    if (case_type == 2) {  // Sorted data
        sort(data.begin(), data.end؛(()
{
    else if (case_type == 3) {  // Sorted data in reverse order
        sort(data.begin(), data.end(), greater<int؛(()<
{
{

void insertion_sort(vector<int>& data} (
    int n = data.size؛()
    for (int i = 1; i < n; ++i} (
```

```
        int key = data[i؛[
        int j = i - 1؛
        while (j >= 0 && data[j] > key} (
            data[j + 1] = data[j؛[
            j = j - 1؛
{
        data[j + 1] = key؛
{
{

void merge(vector<int>& data, int left, int mid, int right} (
    int n1 = mid - left + 1؛
    int n2 = right - mid؛
    vector<int> L(n1), R(n2؛(
    for (int i = 0; i < n1; ++i(
        L[i] = data[left + i؛[
    for (int i = 0; i < n2; ++i(
        R[i] = data[mid + 1 + i؛[
    int i = 0, j = 0, k = left؛
    while (i < n1 && j < n2} (
        if (L[i] <= R[j} ([
            data[k] = L[i؛[
            i؛++
{
        else}
            data[k] = R[j؛[
            j؛++
{
        k؛++
{
    while (i < n1} (
        data[k] = L[i؛[
        i؛++
```

```
        k؛++
{
    while (j < n2} (
        data[k] = R[j؛[
        j؛++
        k؛++
{
{

void merge_sort(vector<int>& data, int left, int right} (
    if (left < right} (
        int mid = left + (right - left) / 2؛
        merge_sort(data, left, mid؛(
        merge_sort(data, mid + 1, right؛(
        merge(data, left, mid, right؛(
{
{

int partition(vector<int>& data, int low, int high} (
    int pivot = data[high؛[
    int i = (low - 1؛(
    for (int j = low; j <= high - 1; j} (++
        if (data[j] < pivot} (
            i؛++
            swap(data[i], data[j؛([
{
{
    swap(data[i + 1], data[high؛([
    return (i + 1؛(
{

void quick_sort(vector<int>& data, int low, int high} (
    while (low < high} (
```

```
        int pi = partition(data, low, high؛(

        if (pi - low < high - pi} (

            quick_sort(data, low, pi - 1؛(

            low = pi + 1؛

    {

        else}

            quick_sort(data, pi + 1, high؛(

            high = pi - 1؛

    {

    {

    {


    void display_time(const string& algorithm, int size, const string& case_name, long
long time} (

        cout << algorithm << " | Size: " << size << " | Case: " << case_name << " |
Time(ms): " << time << endl؛

    {


    int main} ()

        const int num_sets = 6؛

        vector<int> sizes(num_sets؛(

        vector<string> case_names = { "Random", "Sorted", "Reverse Sorted؛{ "


        for (int i = 0; i < num_sets; ++i} (

            cout << "Enter the size of set #" << (i + 1؛" :" >> (

            cin >> sizes[i؛[

    {


        vector<int> data؛


        for (int set_num = 1; set_num <= num_sets; ++set_num} (

            int size = sizes[set_num - 1؛[

            cout << "Processing set #" << set_num << " with size " << size << endl؛
```

```
    for (int case_type = 1; case_type <= 3; ++case_type} (
        generate_data(data, size, case_type؛(

        auto start = high_resolution_clock::now؛()
        insertion_sort(data؛(
        auto end = high_resolution_clock::now؛()
        auto duration = duration_cast<milliseconds>(end - start).count؛()
        display_time("Insertion Sort", size, case_names[case_type - 1], duration؛(

        generate_data(data, size, case_type؛(
        start = high_resolution_clock::now؛()
        merge_sort(data, 0, data.size() - 1؛(
        end = high_resolution_clock::now؛()
        duration = duration_cast<milliseconds>(end - start).count؛()
        display_time("Merge Sort", size, case_names[case_type - 1], duration؛(

        generate_data(data, size, case_type؛(
        start = high_resolution_clock::now؛()
        quick_sort(data, 0, data.size() - 1؛(
        end = high_resolution_clock::now؛()
        duration = duration_cast<milliseconds>(end - start).count؛()
        display_time("Quick Sort", size, case_names[case_type - 1], duration؛(
{
{

   return 0؛
}
```