Contents

I. Introduction

1.1 Topic

Visualize and Compare the Features of Each Sorting Algorithm

1.2 Motivation

In class, we learned about various sorting algorithms such as insertion sort and bubble sort while learning the three algorithm parts. In addition, by analyzing their algorithms, it was possible to learn how to theoretically calculate time complexity. In this process, we wondered if the theoretically obtained time complexity and the time complexity of the algorithm implemented using the actual programming language would match. Therefore, we will compare and analyze the difference between theoretical and real time complexity through a program that calculates the time complexity of the sorting algorithm and visualizes it as a table.

1.3 Basic Sorting Theory

Sort refers to rearranging records in an ascending order or a descending order according to a key value with a specific item as a key among data arranged in an unordered computer storage space. Sort takes most of the time required to process data within a computer to navigate or sort data, so data should be kept sorted to facilitate the search for information. Sorts should select the optimal algorithm according to the given situation. There are various algorithms, and it is important to determine when and what algorithms should be used to achieve the best performance. To this end, the computer's system characteristics, the amount of data to be sorted, the amount of initial data, the distribution of key values, the size of the workspace, the number of key comparisons, and the number of data moves are factors to be considered when selecting a sorting algorithm.

1.4 Project Direction

The goal of the project is to visualize and verify the characteristics of each alignment algorithm. To this end, we first investigate theoretical time complexity and pros and cons as features of bubble, selection, insertion, fast, merge, and heap alignment. As an indicator for realizing these pros and cons, the amount of data to be sorted and the array state to be sorted (which affects the number of movements of the data) will be used. The amount (n) of data to sort is performed on three counts: 10, 100, and 10000 (square). The array state to be sorted is performed on three arrays: a random array, an mostly sorted array, and an inversely ordered array. Each sort function is sorted in ascending order by default and accepts duplicate element values. The implementation uses the language of C and an array of various data structures. In addition, a visual studio was used as a tool for implementing code. The results of the program implemented in this way are visualized in a table. This output will be analyzed by comparing it with the theoretical characteristics, and if the error value with the theory is large, the cause and solution will be considered.

II. Feature of Each Sorting Algorithm

Name	Best	Avg	Worst
Selection sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Bubble sort	n^2	n^2	n^2
Quick sort	$n\log_2 n$	$n\log_2 n$	n^2
Merge sort	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$
Heap sort	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$

2.1 Bubble Sort

Number of comparisons: (n-1) + (n-2) + ··· 2 + 1 = n(n-1) / 2

Number of exchanges: In the worst case where input data is sorted in reverse order, some movements (operation of the SWAP function) are required for a single exchange

In the best case where the input data is already sorted, no movement of the data occurs.

Time complexity: $O(n^2)$

Advantages – It is easy to implement.

Disadvantages – The longer the array length, the more inefficient it is.

2.2 Selection Sort

Number of comparisons: Number of runs of 2 for loops, external loops (n-1) times, internal loops (find the minimum): n-1, n-2, ···, 1 times

Number of exchanges: Equivalent to the number of runs on the external loop, i.e. constant-time operation

3 movements (operation of SWAP function) are required for a single exchange: 3(n-1)

Time complexity: (n-1) + (n-2) +···+ 2 + 1 = n(n-1)/2 = $O(n^2)$

Advantage – The exchange occurs less than the number of comparisons.

Disadvantage – The longer the array length, the more inefficient it is. It's an unstable sort.

2.3 Insertion Sort

1) Best case

Number of comparisons: Already sorted, no movement, only 1 comparison → (n-1)

Time complexity: O(n)

2) Worst case(when the array is sorted in reverse order)

Number of comparisons: i comparisons for each iteration→ (n-1) + (n-2) + ··· 2 + 1 = n(n-1) / 2

Number of exchanges: (i+2) movements occur for each iteration of the outer loop
→ $n(n-1)/2 + 2(n-1) = (n^2+3n-4) / 2$

Time complexity: $O(n^2)$

Advantage – If most elements are already sorted, it can be very efficient. Since it is an exchange method in an array to be sorted, there is no need for another memory space. → in-place sorting, It is a stable sort.

Disadvantage – In the worst case, performance varies greatly depending on the state of the data. The longer the array length, the more inefficient it is.

2.4 Heap Sort

Since the entire height of the heap tree is almost $\log_2 n$ (complete binary tree), it takes as much time to rearrange the heap as $\log_2 n$ when inserting or deleting an element in the heap.

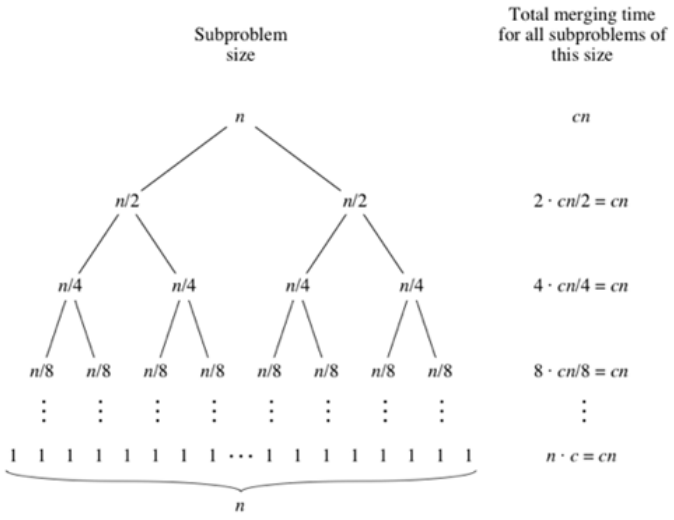
Number of elements: n

Time complexity: $O(n\log_2 n)$

Advantages – Useful when obtaining the largest or smallest value.

Disadvantages – The actual time takes longer than other $O(n\log_2 n)$ sort methods.

2.5 Merge Sort



Merge step: When the number of data is n → $\log_2 n$

Performance time(required for the sort in each step): n

Time complexity = $n+2c(n/2) = n + 2(n/2 + 2c(n/4) = n + 2(n/2 + 2(n/4 + 2c(n/8)) = ... = n\log_2 n$

Advantages – It is less affected by the distribution of data. That is, no matter what the input data is, the sort time is the same. (equivalent to $O(n\log_2 n)$)

Disadvantages – requires additional memory space. It is not an in-place sorting. When the size of the records is large, the number of movements is large, resulting in a great waste of time.

2.6 Quick Sort

1) Best case

Assuming that $n = 2^K$ elements are sorted,

Number of comparisons to correctly position pivot at each step: n times on average

Total number of operations: O(kn)

We do the pivot setting randomly in this project, so $O(n\log_2 n)$ is guaranteed.

Time complexity: $k = \log_2 n$, so $O(kn) = O(n\log_2 n)$

2) Worst case (when the array is already sorted)

Number of comparisons: n

Since the comparison operation is performed N times on average in each step, the total number of operations is $O(n^2)$

Time complexity: n^2

Advantages – In addition to reducing unnecessary data movement and exchanging long-distance data, the time complexity is fastest compared to other sorting algorithms with $O(n\log_2 n)$ due to the feature that once determined pivots are excluded from future operations. Since it is an exchange method in an array to be sorted, there is no need for another memory space.

Disadvantages – It is an unstable sort. For an ordered sort, it takes more time to perform due to the unbalanced segmentation of the quick Sort.

III. Implementation of Feature visualization Program

We have implemented several sort methods. The array size was set to 10, 100, and 10000 in order to obtain values according to the sample size. In addition, all of the array data were stored in the original_ar array through random value assignment. The elements of the Random, some sort, and Inverse arrays were all used by copying origin_ar. The random array was used as it was, mostly sorted array was partially aligned using shell sort, and the Inversely sorted array was sorted using Bubble sort and then the index was stored upside down to create a descending sort arrangement.

3.1 Implementing Sort Functions

3.1.1 Bubble Sort

Starting with the first element, adjacent elements continue to exchange places and align themselves from the very end

```
void bubbleSort(int* A) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - 1 - i; j++) {
            if (A[j] > A[j + 1]) {
                swap(&A[j + 1], &A[j]);
            }
        }
    }
}
```

3.1.2 Selection Sort

Sort by finding the minimum value in a given list and replacing it with the value at the beginning

```
void selectionSort(int* A) {
    int i, j, * min;
    for (i = 0; i < n - 1; i++) {
        min = &A[i];
        for (j = i + 1; j < n; j++) {
            if (*min > A[j]) {
                min = &A[j];
            }       }
        swap(&A[i], min);
    }
}
```

3.1.3 Insertion Sort

Alignment method that completes alignment by comparing all elements of a data array to the array part that is already aligned from the front and by locating and inserting its own location

```
void insertionSort(int* A) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = A[i];
        for (j = i - 1; j >= 0 && A[j] > key; j--) {
            A[j + 1] = A[j];
        }
        A[j + 1] = key;
    }
}
```

3.1.4 Heap Sort

To organize and sort a maximum or minimum heap tree

```
void heapSort(int* A) {
    int i;
    for (i = (n / 2) - 1; i >= 0; i--) {
        heapify(A, n, i);
    }
    for (i = n - 1; i > 0; i--) {
        swap(&A[0], &A[i]);
        heapify(A, i, 0);
    }
}
void heapify(int* A, int n, int i) {
    int parentNode = i;
    int leftChildNode = i * 2 + 1;
    int rightChildNode = i * 2 + 2;
    if (leftChildNode < n && A[parentNode] < A[leftChildNode]) {
        parentNode = leftChildNode;
    }
    if (rightChildNode < n && A[parentNode] < A[rightChildNode]) {
```

```
        parentNode = rightChildNode;
    }
    if (i != parentNode) {
        swap(&A[parentNode], &A[i]);
        heapify(A, n, parentNode);
    }
}
```

3.1.5 Merge Sort

A method of dividing into smaller units, sorting from smaller units, and continuing to merge the ordered units

```
void mergeSort(int* A, int start, int end) {
    int middle;
    if (start < end) {
        middle = (start + end) / 2;
        mergeSort(A, start, middle);
        mergeSort(A, middle + 1, end);
        merge(A, start, middle, end);
    }
}
void merge(int* A, int start, int middle, int end) {
    int i = start, j = middle + 1, k = start;
    while (i <= middle && j <= end) {
        if (A[i] <= A[j]) tmpArray[k++] = A[i++];
        else tmpArray[k++] = A[j++];
    }
    while (i <= middle) {
        tmpArray[k++] = A[i++];
    }
    while (j <= end) {
        tmpArray[k++] = A[j++];
    }
    for (int a = start; a <= end; a++) {
        A[a] = tmpArray[a];
```

```
    }
}
```

3.1.6 Quick Sort

After setting the reference axis (Pivot), the value smaller than the value of this axis is positioned to the right, and the numbers on the left and right are divided again by each axis to align until the value of the axis is 1

```
void quickSort(int* A, int left, int right) {
    int L = left, R = right;
    int pivotIdx = rand() % (R - L + 1) + L;
    swap(&A[(left + right) / 2], &A[pivotIdx]);
    int pivot = A[(left + right) / 2];
    do {
        while (A[L] < pivot)
            L++;
        while (A[R] > pivot)
            R--;
        if (L <= R) {
            swap(&A[R], &A[L]);
            L++;
            R--;
        }
    } while (L <= R);
    if (left < R)
        quickSort(A, left, R);
    if (L < right)
        quickSort(A, L, right);
}
```

3.2 Full Visualization Code

The program calculates the execution time for each sorting function and outputs tables. It is performed in each of three arrays: a random array, a nearly sorted array, and an inversely sorted array, and it is repeated for size n of the array 10, 100 and 10000. As a result, the program outputs three tables.

3.3 Visualization result

N = 10	Random Array	Mostly Sorted Array	Inversely Sorted Array
BubbleSort	0.001500ms	0.000700ms	0.002100ms
SelectionSort	0.001300ms	0.001100ms	0.001300ms
InsertionSort	0.000500ms	0.000300ms	0.000700ms
HeapSort	0.003700ms	0.003600ms	0.002700ms
MergeSort	0.002500ms	0.002300ms	0.002200ms
QuickSort	0.003300ms	0.003200ms	0.002400ms

N = 100	Random Array	Mostly Sorted Array	Inversely Sorted Array
BubbleSort	0.143700ms	0.031000ms	0.229000ms
SelectionSort	0.039100ms	0.035600ms	0.051400ms
InsertionSort	0.019200ms	0.003300ms	0.036500ms
HeapSort	0.063300ms	0.066600ms	0.047700ms
MergeSort	0.031300ms	0.026600ms	0.026200ms
QuickSort	0.053300ms	0.043300ms	0.039000ms

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array
BubbleSort	2007.619900ms	304.190300ms	2731.522900ms
SelectionSort	303.392600ms	215.558400ms	434.655400ms
InsertionSort	176.172400ms	0.288300ms	225.285700ms
HeapSort	10.352500ms	13.151100ms	11.797300ms
MergeSort	4.220900ms	2.665800ms	2.665200ms
QuickSort	7.199900ms	5.721200ms	5.444300ms

IV. Analysis

Previously, the results were derived through actual implementation based on theoretical theorem and sort principle for each sort. In this chapter, we will analyze the differences between the size of the array and the sort of the data in the array for the result values. Statistically, the larger the sample, the higher the confidence level, so the first results presented in the analysis of the results are when N=10000, and other results can be added as needed.

4.1 bubbleSort

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array
BubbleSort	2007.619900ms	304.190300ms	2731.522900ms

Bubble sort should be compared from the first element to the end while continuously exchanging positions between adjacent elements. 3.3 When checking the image, the inefficiency was confirmed through the fact that bubble sort was significantly slower than all arrays.

Next, we will compare the bubble sort itself according to the arrangement state. For random sorting, swaps must be made when large values are met, and for reverse sorting, swaps always become ar[i]>ar[j] because they are in descending order. Therefore, the run time is the largest when it is sorted in reverse direction. When mostly sorted, the run time is the shortest because the comparison definitely reduces the number of swap operations compared to the previous array sort.

4.2 selectionSort

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array
BubbleSort	2007.619900ms	304.190300ms	2731.522900ms
SelectionSort	303.392600ms	215.558400ms	434.655400ms
InsertionSort	176.172400ms	0.288300ms	225.285700ms

Selection sort is the process of searching from all i (current index) to N-1 (array size) and retrieving the minimum value, resulting in a single swap, so it exhibits similar performance in normal situations. Therefore, the time difference (random / mostly sorted / inversely) depending on the sort state is the smallest compared to bubble sort and insertion sort with the same expected execution time $O(n^2)$.

Next, we will compare the Selection sort itself according to the arrangement state. It can be seen that there is a slight difference in Inverse when the array size is the largest (N=contact point) and when it is random and some types. For random and some sort, no swap occurs when the first value is the minimum value. However, since the stations are already ordered in descending order, swaps occur in all cases, so the execution time is relatively large.

4.3 insertionSort

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array
BubbleSort	2007.619900ms	304.190300ms	2731.522900ms
SelectionSort	303.392600ms	215.558400ms	434.655400ms
InsertionSort	176.172400ms	0.288300ms	225.285700ms

Insertion sort is a method designed to efficiently reduce the number of comparisons of bubble sort. Swap occurs only when you find something smaller than yourself, so the number of comparisons is less than other sorts. Therefore, it can be confirmed that it is significantly smaller compared to bubble sort and selection sort that hold the same execution time $O(n^2)$.

Next, we will compare the insertion sort itself according to the arrangement state. Insertion sort results in swaps if it finds a value smaller than itself from i (current index) to the first array value. This has the smallest execution time because the number of swaps is small when the state is mostly sorted, and the inverse-sorted state has the largest execution time because the number of swaps is always performed as i-1. It may be seen that as the sample size increases, the difference in execution time according to the arrangement sort state increases.

4.4 heapSort

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array
HeapSort	10.352500ms	13.151100ms	11.797300ms

Heap sort has been implemented as maximum heap. We always presupposed exchanging the root and the last element and reconstructing it via heapify. Therefore, it can be seen that the execution time is relatively large compared to the merge sort and the fast sort with the same execution time O(nlogn).

Next, we will compare the heap sort itself according to the arrangement state. Since heap sort always has O(nlogn), there is no significant difference in execution time under any sort.

4.5 mergeSort

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array
MergeSort	4.220900ms	2.665800ms	2.665200ms

Merge sort always performs O(nlogn) because the array is always divided into two and the two are combined by linear time. Therefore, there is no significant difference depending on the sort state. However, the merge sort includes a big drawback that requires a lot of space. Additional experiments were conducted to confirm this. As an experiment, the execution time was obtained including space allocation.

-> Running time, including additional memory usage (The following execution method allocated additional memory inside the merge sort function. However, to improve this point, it was possible to allocate additional memory outside the function to obtain only the execution time of the sort part.)

mergeSort 56.629800ms 48.323700ms 59.025600ms

From the above results, it was confirmed that the merge sort always has the advantage of performing O(nlogn), but since more memory as big as the entire data is needed, if the time according to the additional size allocation is included, it has a large execution time.

4.6 quickSort

The quick sequence must be continuously divided into small and large parts based on the pivot.

N = 10000	Random Array	Mostly Sorted Array	Inversely Sorted Array	
QuickSort	7.199900ms	5.721200ms	5.444300ms	– pivot random set
QuickSort	4.258500ms	1.633200ms	1.389800ms	– pivot median set
QuickSort	4.443100ms	5.035000ms	26.026300ms	– pivot first value set

The disadvantage of the quick arrangement is that the time complexity varies according to the reference value Pivot. In fact, when the reference value was set differently, it was confirmed that the execution time was all different.

V. Conclusion

When the various sorting algorithms learned in class were actually implemented, it was questioned whether the time complexity and features would have a result value consistent with the theory. To solve this question, we planned to calculate the actual time complexity using programming languages. According to this direction, we took time to find out the theories and characteristics of various sort functions and to analyze them with practical results. To prove the features of each sort by giving the array to be sorted a variety of states, a random arrangement was set using a random number generator in C language, and after sorting, an inverse sort was created through an exchange process. However, we were faced with the question of "how to make a mostly sorted list." While looking for a solution, we found a new sort called shell sort, and we learned that it is sort that uses insertion sort to make a quick time after making a moderately sorted list by setting a gap. If this gap stopped at 1, we could get a moderately sorted list. The program was implemented with these arrays, each function call, and repetition according to N. The parts obtained from the output result of the program implemented in this way are as follows.

First, the theory is simply summarized within the sort of O(n^2) and compared with the actual implementation content. Bubble sort always compares two elements with or without sort, so in any case, the time complexity is O(n^2) and is the least efficient algorithm of the three sorts. Since the selection sort finds the smallest value within the range and exchanges it with the corresponding element, the number of comparisons is relatively small compared to the bubble sort. The insertion sort is exchanged when elements of smaller values are found, so the larger the number of sorted elements, the smaller the number of exchanges except for the search, so it has the fastest execution time among the three. Based on the above theories, it can be seen that the performance of bubble sort degrades because the more elements the more comparative operations the more elements are. It was confirmed that the bubble sort had the largest execution time, the selection sort had a smaller execution time compared to the bubble sort, and the insertion sort had the smallest execution time. In addition, in the case of insertion sort, theoretically, the ordered algorithm has a time complexity of O(n), and in fact, if you look at the (N=10000)mostly sorted array part of the insertion sort, you can see that it has the smallest execution time compared to all the sort.

Within the O(nlogn) sorts, the theory is simply summarized and compared with the actual implementation. The heap sort always maintains O(nlogn) because it uses trees, and the merge sort always maintains O(nlogn) because it divides n elements into two by n/2 size in all cases. On the other hand, the expected execution time of Quick Sort is also O(nlogn), but the size of the divided may be uneven depending on the reference point (pivot), so the execution time according to the pivot appears irregularly. These three sorts were found to be significantly faster in the actual performance time as the sample became larger compared to the arrangements (bubble sort, selection sort, and insertion sort) with the expected performance time of O(n^2). It was found that the heap sort was a little slower in the case of actual time measurements compared to the merge sort and quick sort with the same O(nlogn). In addition, it was also confirmed that the heap sort uses binary trees, merge sort, and quicksort use divide, so the time difference between the array states is not larger than that of the O(n^2) sort algorithm.