**Student ID \_\_\_\_2023337621159\_\_\_\_\_\_\_\_ name:\_\_\_\_\_代\_\_\_\_翔\_\_\_\_\_\_**

**Experiment 3:** Sorting

**Purpose:**

1. Master the concepts of sorting;

2. Master some classical sorting algorithms;

3. Understand the way of using different sorting algorithms to solve problems.

**Problems and requirements:**

【Problem 1】: Implement and analyze various sorting algorithms.

**Requirements:**

**1.** You should create a **Sort.h** file, where several sorting functions (each of which refers to one sorting algorithm) are declared. All sorting functions should be able to be applied to sort a set of data stored in an integer array.

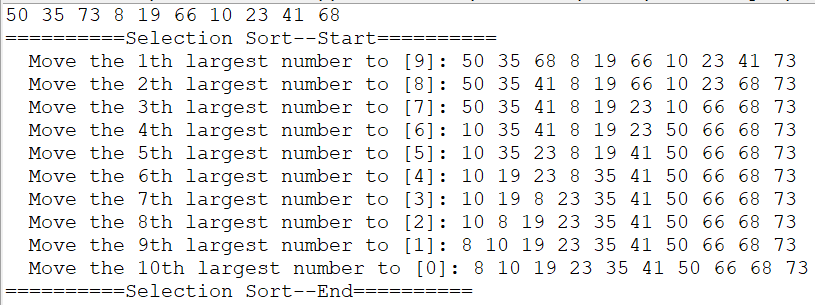
Especially, you may also need to declare a print function to print out the data stored in an integer array.

2. Then, you should implement all operations in the corresponding **Sort.c** file.

3. **Sorting algorithms must include bubble sort, selection sort, insertion sort, shellsort, heapsort, mergsort, and quicksort.**

4. In your main.c file, you can prepare the set of data to be sorted, and call different sorting function to sort your data, and print out necessary information for demonstrating your sorting results as well as showing the idea of the sorting algorithm you used.

For example, to demonstrate the sorting results and algorithm of Selection sort, you can print out the following information:



A recommendation is: providing a menu for the user to select which sort algorithm to be used.

The menu can be simply like this:

===========Menu=====

1. Insertion sort

2. Shell sort

…

0 Exit

=====================

In this way, your sorting algorithms can be tried until 0 is typed in.

5. Discuss the complexity for sorting algorithms based on your experiences and your experimental data.

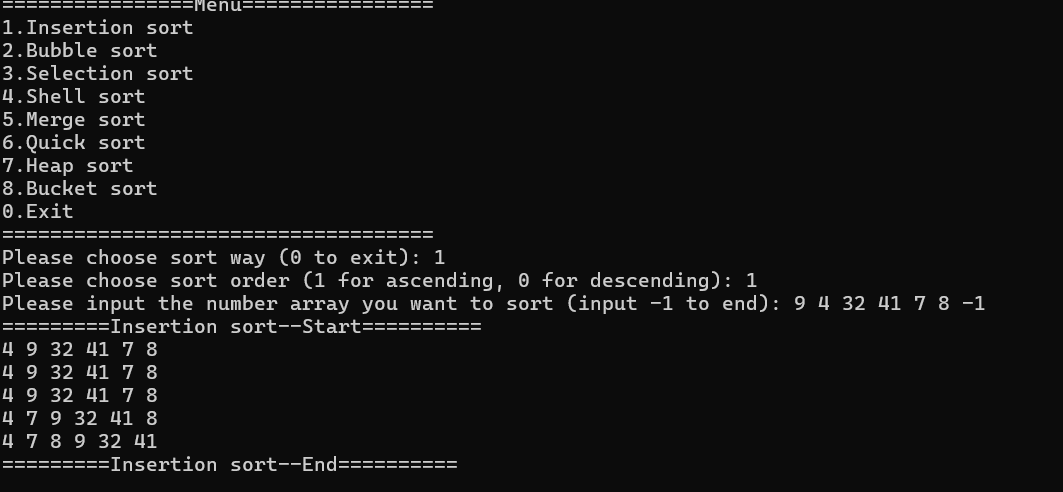
【Problem 2】: Design improved sorting algorithms.

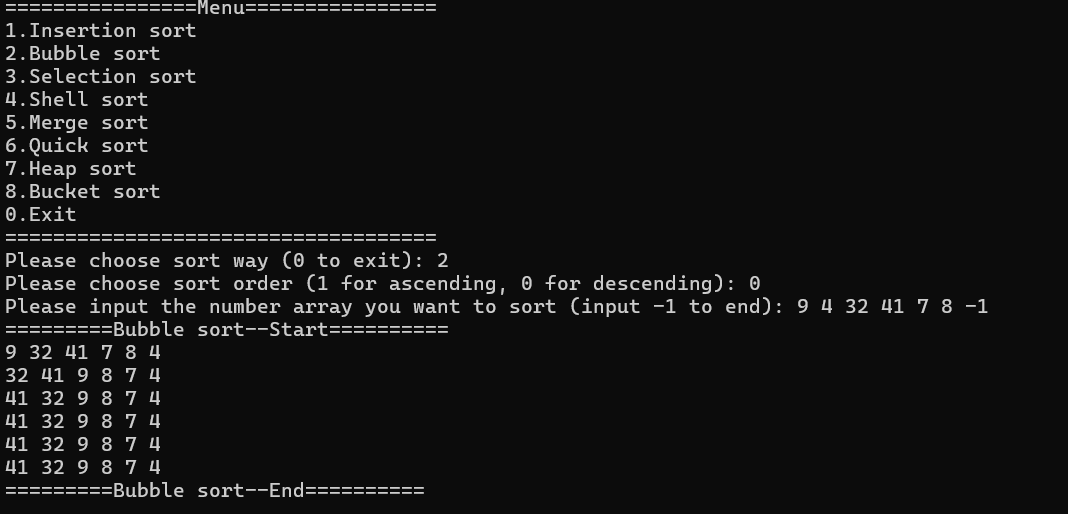
**Requirements:**

1. Select **two or more** sorting algorithms (**insertion sort, bubble sort, selection sort, shellsort, mergsort, and quicksort, and ..**), try to design some strategies to improve existing algorithms.

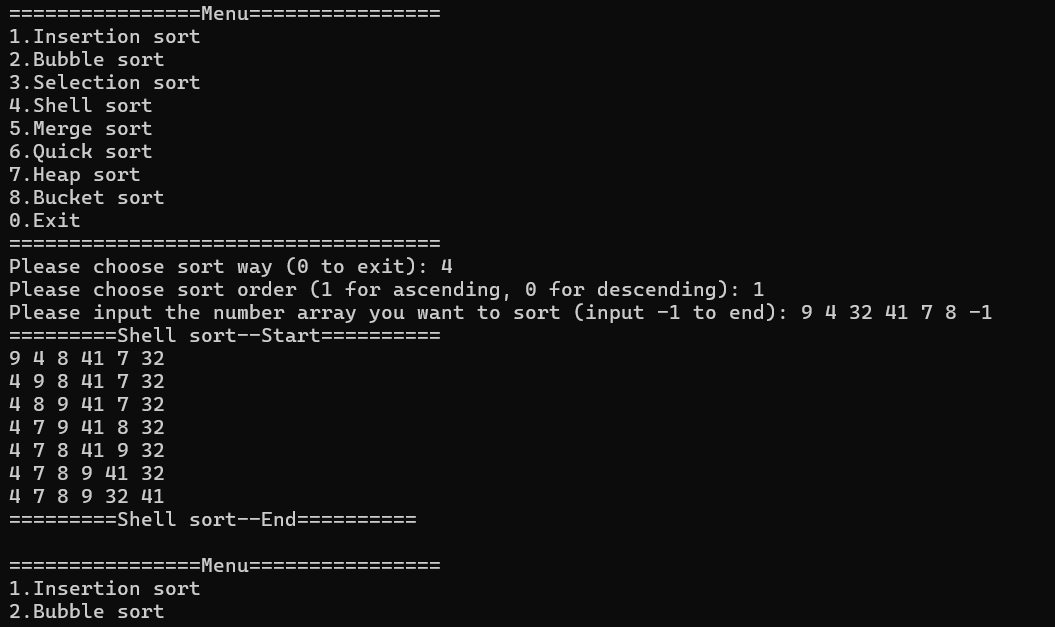
For example, you can design a new insertion sort (that is, the improved version) by changing some lines of code of the original insertion sort.

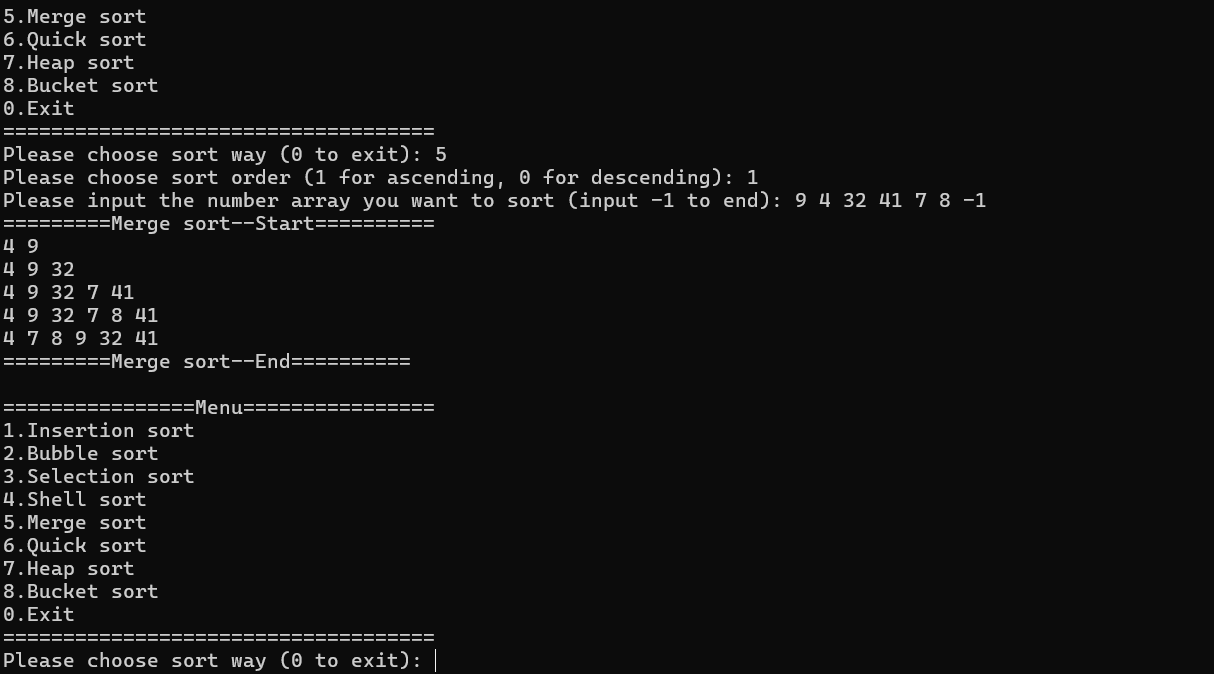
1. Clarify your algorithm, implement them, and show their sorting results.

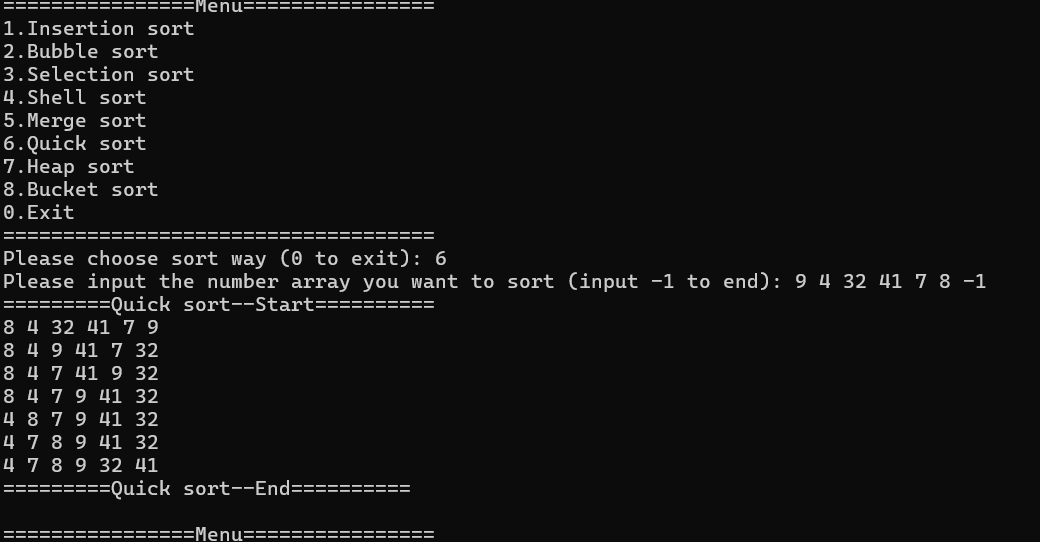


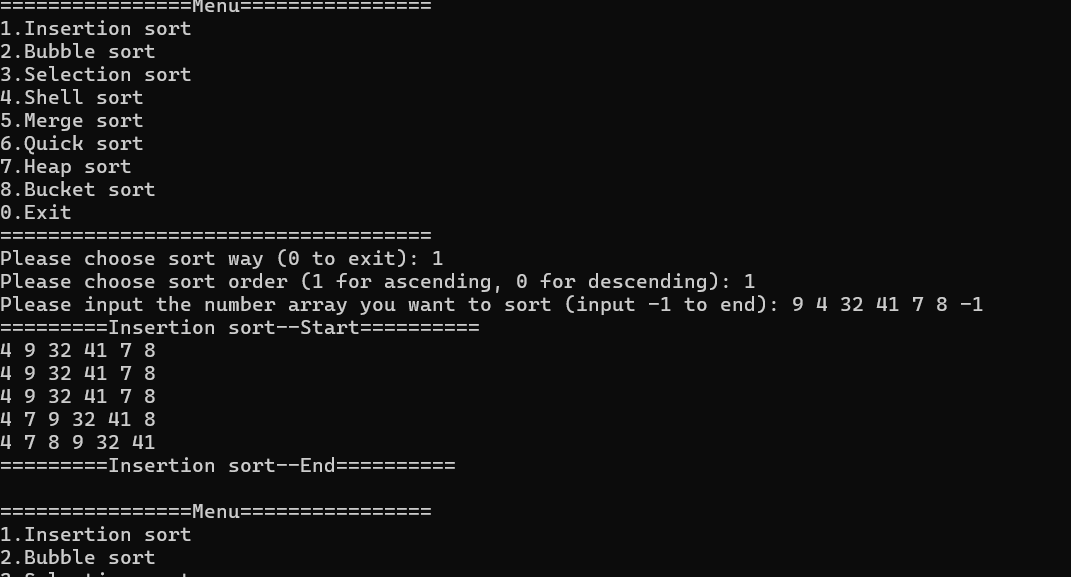


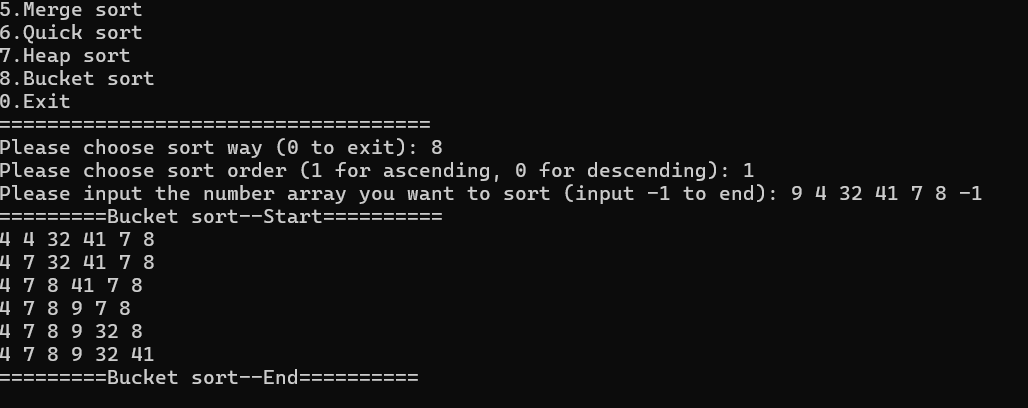












3. Discuss the difference between the original and the improved algorithms from the perspective of time and space complexity.

Insertion

Use the BinarySearch function to root number n to find the insertion location.The time complexity of the insertion sort is changed from the original n squared is optimized for fewer comparison operations, but the data movement complexity is still there n squared.

Bubble

Time complexity: In the best-case scenario (the array is already ordered), you can exit early after each round of checks, and the time complexity will start from n squared to downgrade n

Space Complexity: Still O(1) because only an additional flag variable swapped is introduced

Selection

It can be optimized to two-way selection sorting: in each outer loop, find the minimum and maximum values of the currently unsorted parts, respectively, and place them in the correct position. This allows the position of two elements to be determined in a single loop, reducing the number of overall traversals.

Time complexity: Sorts from one-way selections n squared improved to Half-n-squared

Space Complexity: Still O(1)

Shell

Optimize incremental sequences: Knuth sequences (such as 1,4,13,40,… ) has excellent performance in the experiment, which can better optimize the grouping division and improve the sorting efficiency.

Time complexity:Insert sorting is used instead, combined with efficient incremental sequences, and the time complexity is optimized to 1.3 to the 1.5 power of n

Spatial complexity:O(1)

Merge:

In-situ merge optimization

Time Complexity: n squared; Inserting moving elements of an in-place operation results in performance degradation.

Space Complexity:O(1) (no auxiliary array overhead).

Quick,Bucket,Heap can be no stable optimization.