

# 项目说明文档

## 数据结构课程设计

### ——排序算法比较

作者姓名 \_\_\_\_\_ 林继申

学 号 \_\_\_\_\_ 2250758

指导教师 \_\_\_\_\_ 张 颖

学院专业 \_\_\_\_\_ 软件学院 软件工程



同濟大學  
TONGJI UNIVERSITY

二〇二三年十二月十三日

# 目录

1 项目分析.....	1
1.1 项目背景分析.....	1
1.2 项目需求分析.....	1
1.3 项目功能分析.....	1
1.3.1 冒泡排序算法.....	1
1.3.2 选择排序算法.....	2
1.3.3 插入排序算法.....	2
1.3.4 希尔排序算法.....	2
1.3.5 快速排序算法.....	2
1.3.6 堆排序算法.....	2
1.3.7 归并排序算法.....	3
1.3.8 基数排序算法.....	3
1.3.9 算法性能评测功能.....	3
1.3.10 异常处理功能.....	3
2 项目设计.....	3
2.1 数据结构设计.....	3
2.2 泛型化设计.....	4
2.2.1 泛型化设计的实现.....	4
2.2.2 泛型化设计的优势.....	4
2.2.3 泛型化排序算法函数原型.....	4
2.3 项目主体架构设计.....	5
3 项目功能实现.....	6
3.1 项目主体架构的实现.....	6
3.1.1 项目主体架构实现思路.....	6
3.1.1.1 项目主体架构实现流程图.....	6
3.1.1.2 <b>SortFunction</b> 结构体类型定义.....	6
3.1.1.3 <b>SortOption</b> 结构体类型定义.....	7
3.1.1.4 <b>sortOptions</b> 数组定义.....	7
3.1.1.5 项目主体架构实现思路.....	7
3.1.2 项目主体架构核心代码.....	8
3.1.3 项目主体架构示例.....	9
3.2 冒泡排序算法的实现.....	9
3.2.1 冒泡排序算法实现思路.....	9

3.2.2 冒泡排序算法核心代码 .....	10
3.2.3 冒泡排序算法功能示例 .....	10
3.2.4 冒泡排序算法性能分析 .....	10
3.2.4.1 冒泡排序算法时间复杂度分析 .....	10
3.2.4.2 冒泡排序算法空间复杂度分析 .....	10
3.3 选择排序算法的实现 .....	10
3.3.1 选择排序算法实现思路 .....	10
3.3.2 选择排序算法核心代码 .....	11
3.3.3 选择排序算法功能示例 .....	11
3.3.4 选择排序算法性能分析 .....	11
3.3.4.1 选择排序算法时间复杂度分析 .....	11
3.3.4.2 选择排序算法空间复杂度分析 .....	11
3.4 插入排序算法的实现 .....	12
3.4.1 插入排序算法实现思路 .....	12
3.4.2 插入排序算法核心代码 .....	12
3.4.3 插入排序算法功能示例 .....	12
3.4.4 插入排序算法性能分析 .....	12
3.4.4.1 插入排序算法时间复杂度分析 .....	12
3.4.4.2 插入排序算法空间复杂度分析 .....	13
3.5 希尔排序算法的实现 .....	13
3.5.1 希尔排序算法实现思路 .....	13
3.5.2 希尔排序算法核心代码 .....	13
3.5.3 希尔排序算法功能示例 .....	13
3.5.4 希尔排序算法性能分析 .....	14
3.5.4.1 希尔排序算法时间复杂度分析 .....	14
3.5.4.2 希尔排序算法空间复杂度分析 .....	14
3.6 快速排序算法的实现 .....	14
3.6.1 快速排序算法实现思路 .....	14
3.6.2 快速排序算法核心代码 .....	14
3.6.3 快速排序算法功能示例 .....	15
3.6.4 快速排序算法性能分析 .....	15
3.6.4.1 快速排序算法时间复杂度分析 .....	15
3.6.4.2 快速排序算法空间复杂度分析 .....	15
3.7 堆排序算法的实现 .....	15
3.7.1 堆排序算法实现思路 .....	15

3.7.2	堆排序算法核心代码 .....	16
3.7.3	堆排序算法功能示例 .....	16
3.7.4	堆排序算法性能分析 .....	16
3.7.4.1	堆排序算法时间复杂度分析 .....	16
3.7.4.2	堆排序算法空间复杂度分析 .....	17
3.8	归并排序算法的实现 .....	17
3.8.1	归并排序算法实现思路 .....	17
3.8.2	归并排序算法核心代码 .....	17
3.8.3	归并排序算法功能示例 .....	18
3.8.4	归并排序算法性能分析 .....	18
3.8.4.1	归并排序算法时间复杂度分析 .....	18
3.8.4.2	归并排序算法空间复杂度分析 .....	18
3.9	基数排序算法的实现 .....	19
3.9.1	基数排序算法实现思路 .....	19
3.9.2	基数排序算法核心代码 .....	19
3.9.3	基数排序算法功能示例 .....	20
3.9.4	基数排序算法性能分析 .....	20
3.9.4.1	基数排序算法时间复杂度分析 .....	20
3.9.4.2	基数排序算法空间复杂度分析 .....	20
3.10	算法性能评测功能的实现 .....	20
3.10.1	算法性能评测功能实现思路 .....	20
3.10.2	算法性能评测功能核心代码 .....	21
3.10.3	算法性能评测功能示例 .....	22
3.11	异常处理功能的实现 .....	23
3.11.1	动态内存申请失败的异常处理 .....	23
3.11.2	输入非法的异常处理 .....	23
3.11.2.1	要生成随机数的个数输入非法的异常处理 .....	23
3.11.2.2	排序算法类型输入非法的异常处理 .....	24
4	项目测试 .....	25
4.1	输入要生成随机数的个数功能测试 .....	25
4.2	排序算法功能正确性测试 .....	26
5	集成开发环境与编译运行环境 .....	26

# 1 项目分析

## 1.1 项目背景分析

排序算法是计算机科学中的基本概念，对理解数据结构和算法非常重要。通过比较不同的排序算法，可以加深对它们性能差异的理解。排序是数据处理的基础操作之一，不同的排序算法在不同应用场景下有着不同的适用性和效率。通过比较算法的时间复杂度、空间复杂度和稳定性，可以选择适合特定应用场景的排序方法。随着硬件和编程语言的发展，旧算法可能被优化或新算法可能被发明，因此持续的性能评估是必要的。

## 1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

(1) 多种排序算法：本项目需要实现多种排序算法，如冒泡排序（Bubble Sort）、选择排序（Selection Sort）、插入排序（Insertion Sort）、希尔排序（Shell Sort）、快速排序（Quick Sort）、堆排序（Heap Sort）、归并排序（Merge Sort）、基数排序（Radix Sort）等；

(2) 算法性能评测：本项目需要能够测量和比较每种排序算法的性能，包括但不限于运行时间和比较次数等；

(3) 用户交互：提供用户友好的界面，允许用户选择排序算法，输入数据规模等；

(4) 数据生成：能够生成不同规模和特性的数据以进行排序；

(5) 异常处理：需要妥善处理异常情况，如内存分配失败和输入数据非法等情况。

## 1.3 项目功能分析

本项目旨在通过实现多种排序算法，并进行算法性能评测，进行排序算法的比较，同时需要考虑用户交互、数据生成和异常处理等功能。通过本项目，可以更好地理解不同排序算法在处理大量数据时的效率和适用性，同时也促进了对数据结构和算法原理更深入的理解。下面对项目的功能进行详细分析。

### 1.3.1 冒泡排序算法

冒泡排序（Bubble Sort）是一种简单的排序算法，它属于比较排序算法的

一种。它的基本思想是重复地遍历待排序的元素列表，比较相邻的两个元素，并根据比较结果交换它们的位置，直到整个列表变得有序为止。

### 1.3.2 选择排序算法

选择排序 (Selection Sort) 是一种简单的排序算法，它属于比较排序算法的一种。选择排序的基本思想是将待排序的元素分成两部分：已排序部分和未排序部分。初始时，已排序部分为空，而未排序部分包含所有待排序的元素。在每一轮中，选择排序会从未排序部分中找到最小（或最大）的元素，然后将其与未排序部分的第一个元素交换位置，将其放入已排序部分的末尾。

### 1.3.3 插入排序算法

插入排序 (Insertion Sort) 是一种简单的排序算法，它属于比较排序算法的一种。插入排序的基本思想是将待排序的元素逐个插入到已经排好序的部分中，从而逐步构建有序的序列。

### 1.3.4 希尔排序算法

希尔排序 (Shell Sort) 是一种改进的插入排序算法，也被称为缩小增量排序。希尔排序通过将待排序的元素分成多个子序列，对每个子序列进行插入排序，然后逐渐缩小子序列的间隔，最终完成整体的排序。希尔排序的主要思想是通过较大的步长将元素移动到合适的位置，从而提前部分有序性，最终减少插入排序的工作量。

### 1.3.5 快速排序算法

快速排序 (Quick Sort) 是一种高效的分治排序算法，它常用于对大规模数据集进行排序。快速排序的基本思想是选择一个元素作为基准（通常选择最后一个元素），然后将数组分成两部分，使得左边的部分都小于基准，右边的部分都大于基准。接着，递归地对左右两部分进行排序，最终得到整个数组有序。

### 1.3.6 堆排序算法

堆排序 (Heap Sort) 是一种基于二叉堆数据结构的排序算法，它具有稳定的时间复杂度和较好的性能，通常用于对大规模数据集进行排序。堆是一种特殊的树形数据结构，分为最大堆和最小堆两种类型。最大堆要求父节点的值大于或等于其子节点的值，而最小堆则要求父节点的值小于或等于其子节点的值。在堆排序中，通常使用最大堆。

### 1.3.7 归并排序算法

归并排序（Merge Sort）是一种分治排序算法，它将待排序的数组分成两个部分，分别对这两部分进行排序，然后将它们合并成一个有序的数组。归并排序的核心思想是分而治之，通过将问题拆分为子问题并解决子问题，最终达到整体有序的目标。

### 1.3.8 基数排序算法

基数排序（Radix Sort）是一种非比较排序算法，它通过将待排序的元素按照各个位数的值（或者其他进制的位数）进行分组和排序，最终得到有序的结果。基数排序通常用于处理整数或字符串等数据类型。

### 1.3.9 算法性能评测功能

本项目需要能够测量和比较每种排序算法的性能，包括但不限于运行时间和比较次数等，同时需要考虑用户交互等功能。

### 1.3.10 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

## 2 项目设计

### 2.1 数据结构设计

基于项目分析，在本项目中选择使用传统的 C 语言数组形式实现各个排序算法，主要基于以下几个考虑：

- (1) 性能因素：C 语言数组是一种性能高效的数据结构，对于排序算法来说，数组的随机访问速度非常快，这有助于提高排序算法的性能；
- (2) 通用性：C 语言数组是一种通用的数据结构，几乎可以用于任何数据类型的排序，只需要根据需要进行模板化或者修改比较和交换操作；
- (3) 低级别控制：C 语言数组允许开发者对内存和数据的低级别控制，这对于一些排序算法的实现非常重要，例如快速排序和堆排序；
- (4) 理解性：使用传统的 C 语言数组利于从底层深入理解数据结构和排序算法的原理知识。

使用传统的 C 语言数组需要注意数组的内存分配和释放，并且避免出现内存泄漏和数组访问越界等问题。

## 2.2 泛型化设计

在各个排序算法的实现中，使用了泛型化的设计。泛型化是指算法可以处理不同数据类型的数据，而不仅仅是特定类型的数据。

### 2.2.1 泛型化设计的实现

下面是泛型化设计的实现方法：

(1) 使用 C++ 的模板 (template) 特性，这允许定义一个通用的排序函数，该函数可以处理不同类型的数据；

(2) `<typename Type>` 是一个模板参数，它表示排序函数将使用一个名为 `Type` 的类型参数来表示数组中的元素类型；

(3) `Type arr[]` 是一个类型为 `Type` 的数组；

(4) 通过使用泛型类型参数 `Type`，可以在不改变排序算法的基本逻辑的情况下，轻松地对不同类型的数组进行排序，例如整数、浮点数等。

### 2.2.2 泛型化设计的优势

下面是泛型化设计的优势：

(1) 代码重用性：泛型化的排序算法可以在不同的数据类型上重复使用，而不需要为每种数据类型编写一个新的排序函数，这减少了代码的重复性；

(2) 统一的接口：通过使用泛型化的方式，可以为不同数据类型提供统一的排序接口，使代码更加清晰和可维护；

(3) 提高开发效率：不必为每种数据类型都编写排序算法，可以节省开发时间，并降低出错的可能性；

(4) 更广泛的适用性：泛型化的排序算法可以用于各种应用程序，包括通用库、数据结构和算法的实现，从而增加了代码的灵活性与适用性；

(5) 降低维护成本：一旦实现了泛型化的排序算法，就可以轻松地适应未来的需求变化，而无需大规模修改现有代码。

综上所述，通过实现泛型化的排序算法，可以提高代码的可重用性、可维护性和适用性，从而更加高效地开发和维护代码。这种方法可以在处理不同类型的数据时提供更大的灵活性，并减少了代码冗余，有助于提高代码质量和开发效率。

### 2.2.3 泛型化排序算法函数原型

```
template <typename Type>
void sortFunction(Type arr[], int n)
{
    // Statements
}
```



## 2.3 项目主体架构设计

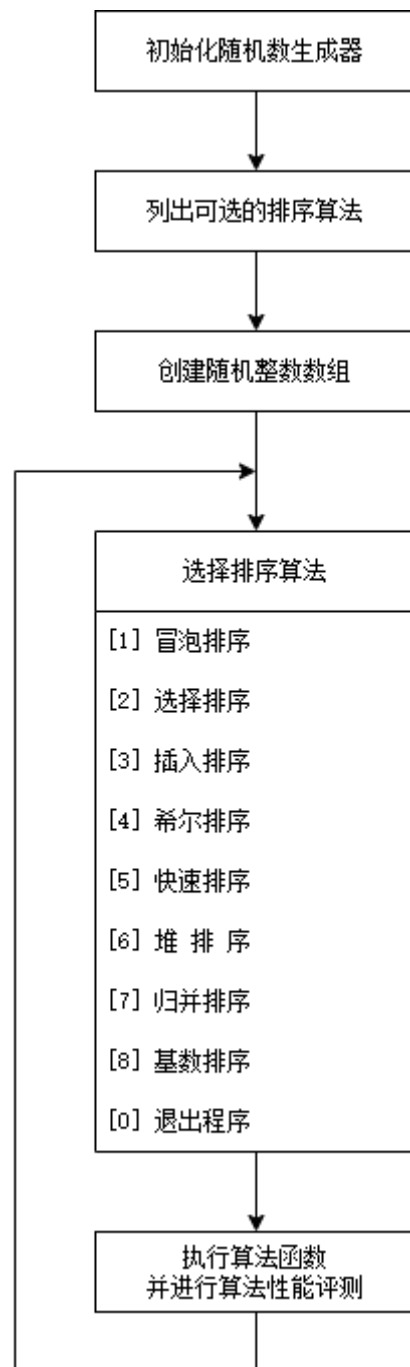


图 2.3.1 项目主体架构设计流程图

项目主体架构设计为：

- (1) 初始化随机数生成器；
- (2) 列出所有可选的排序算法；
- (3) 获取用户希望生成的随机数数量并创建一个整数数组并填充随机数；
- (4) 通过循环结构允许用户选择不同的排序算法进行比较；
- (5) 根据用户的选择执行相应的排序算法，并显示排序的时间和比较次数。

### 3 项目功能实现

#### 3.1 项目主体架构的实现

##### 3.1.1 项目主体架构实现思路

###### 3.1.1.1 项目主体架构实现流程图



图 3.1.1.1.1 项目主体架构实现流程图

###### 3.1.1.2 SortFunction 结构体类型定义

```
typedef void (*SortFunction)(int*, int);
```

这是一个函数指针类型定义，用于指向不同的排序函数。每个排序函数都接受一个整数数组和数组的长度作为参数。

#### 3.1.1.3 SortOption 结构体类型定义

```
struct SortOption {  
    SortFunction func;  
    const char* description;  
};
```

这个结构体用于存储排序函数及其描述。每个 `SortOption` 对象包含一个 `SortFunction` 函数指针和一个指向其描述的字符串。

#### 3.1.1.4 sortOptions 数组定义

```
SortOption sortOptions[] = {  
    { bubbleSort, "冒泡排序 Bubble Sort" },  
    { selectionSort, "选择排序 Selection Sort" },  
    { insertionSort, "插入排序 Insertion Sort" },  
    { shellSort, "希尔排序 Shell Sort" },  
    { quickSort, "快速排序 Quick Sort" },  
    { heapSort, "堆排序 Heap Sort" },  
    { mergeSort, "归并排序 Merge Sort" },  
    { radixSort, "基数排序 Radix Sort" }  
};
```

这个数组包含了多个 `SortOption` 对象，每个对象代表一个排序算法及其描述。这个数组是程序中排序算法选择的基础。

#### 3.1.1.5 项目主体架构实现思路

项目主体架构实现思路为：

- (1) 调用 `srand` 函数初始化随机数生成器；
- (2) 打印系统进入提示，并打印所有可选的排序算法；
- (3) 调用 `inputInteger` 函数获取用户希望生成的随机数数量，并创建一个整数数组并填充随机数，随机数生成成功后打印给出提示信息；
- (4) 通过循环结构调用 `selectOptn` 函数允许用户选择不同的排序算法进行比较。如果用户选择选项[0]，则退出程序；
- (5) 根据用户的选择调用 `performSort` 函数，执行相应的排序算法，并显示排序的时间和比较次数。`performSort` 函数原型为：

```
void performSort(SortFunction sortFunc, Type arr[], int n,
```

```
const char* prompt);
```

这个模板函数是排序操作的核心。它接受一个排序函数、数组、数组长度和排序算法的描述。函数内部调用所选的排序函数对数组进行排序，实现了不同算法函数的调用一致性，并记录及显示排序时间和比较次数。在 `performSort` 函数内，使用以下方式调用排序函数：

```
sortFunc(sortArr, n);
```

### 3.1.2 项目主体架构核心代码

```
int main()
{
    /* Generate random number seed */
    srand((unsigned int)(time(0)));

    /* System entry prompt */
    std::cout << "+-----+" <<
std::endl;
    std::cout << "|          排序算法比较          |" << std::endl;
    std::cout << "| Comparison of Sorting Algorithms |" << std::endl;
    std::cout << "+-----+" <<
std::endl << std::endl;
    std::cout << ">>> 排序算法:" << std::endl;
    for (int i = 1; i <= 8; i++)
        std::cout << "          [" << i << "] " << sortOptions[i -
1].description << std::endl;
    std::cout << "          [0] 退出程序 Quit Program" << std::endl <<
std::endl;

    /* Generate random numbers */
    int num = inputInteger(1, INT_MAX, "要生成随机数的个数");
    int* arr = new(std::nothrow) int[num];
    if (arr == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < num; i++)
        arr[i] = rand();
    std::cout << std::endl << ">>> 随机数生成成功（随机数数量: " << num
<< ") " << std::endl;

    /* Sorting algorithm */
    while (true) {
        int optn = selectOptn();
        if (optn == 0)
```

```

        return 0;
    else
        performSort(sortOptions[optn - 1].func, arr, num,
sortOptions[optn - 1].description);
    }
}

```

### 3.1.3 项目主体架构示例

```

-----+
|               排序算法比较               |
|      Comparison of Sorting Algorithms      |
|-----+-----+
>>> 排序算法:
[1] 冒泡排序 Bubble Sort
[2] 选择排序 Selection Sort
[3] 插入排序 Insertion Sort
[4] 希尔排序 Shell Sort
[5] 快速排序 Quick Sort
[6] 堆排序 Heap Sort
[7] 归并排序 Merge Sort
[8] 基数排序 Radix Sort
[0] 退出程序 Quit Program

请输入要生成随机数的个数 [整数范围: 1~2147483647]: 10000

>>> 随机数生成成功 (随机数数量: 10000)

请选择排序算法: [0]

```

图 3.1.3.1 项目主体架构示例

## 3.2 冒泡排序算法的实现

### 3.2.1 冒泡排序算法实现思路

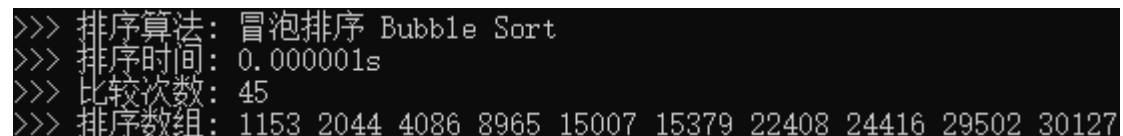
冒泡排序算法的实现思路为：

- (1) 从数组的第一个元素开始，比较相邻的元素；
- (2) 如果第一个比第二个大（小），就交换它们两个；
- (3) 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对，这一步做完后，最后的元素会是最大（或最小）的数；
- (4) 针对所有的元素重复以上的步骤，除了最后一个；
- (5) 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

### 3.2.2 冒泡排序算法核心代码

```
template <typename Type>
void bubbleSort(Type arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++) {
            compareCount++;
            if (arr[j] > arr[j + 1])
                mySwap(arr[j], arr[j + 1]);
        }
}
```

### 3.2.3 冒泡排序算法功能示例



```
>>> 排序算法: 冒泡排序 Bubble Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 45
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127
```

图 3.2.3.1 冒泡排序算法功能示例

### 3.2.4 冒泡排序算法性能分析

#### 3.2.4.1 冒泡排序算法时间复杂度分析

最佳情况： $T(n)=O(n)$ ，当输入的数据已经是正序时（假设从小到大排序），只需要进行一轮比较，就没有必要再进行交换。

最差情况： $T(n)=O(n^2)$ ，当输入的数据是反序时，需要进行  $n*(n-1)/2$  次比较和交换。

平均情况： $T(n)=O(n^2)$ ，平均情况下，比较和交换次数大致为  $n$  的平方的一半。

#### 3.2.4.2 冒泡排序算法空间复杂度分析

空间复杂度为  $O(1)$ ，因为冒泡排序在原地交换元素，除了输入数组外，只需要用到少量的临时存储空间。

## 3.3 选择排序算法的实现

### 3.3.1 选择排序算法实现思路

选择排序算法的实现思路为：

(1) 首先在未排序序列中找到最小（或最大）元素，存放到排序序列的起始位置；

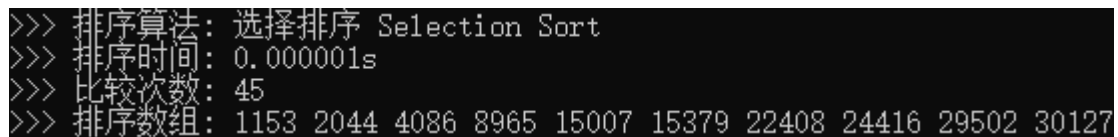
(2) 然后，再从剩余未排序元素中继续寻找最小（或最大）元素，然后放到已排序序列的末尾；

(3) 重复第二步，直到所有元素均排序完毕。

### 3.3.2 选择排序算法核心代码

```
template <typename Type>
void selectionSort(Type arr[], int n)
{
    for (int i = 0; i < n - 1; i++) {
        int k = i;
        for (int j = i + 1; j < n; j++) {
            compareCount++;
            if (arr[j] < arr[k])
                k = j;
        }
        mySwap(arr[k], arr[i]);
    }
}
```

### 3.3.3 选择排序算法功能示例



```
>>> 排序算法: 选择排序 Selection Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 45
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127
```

图 3.3.3.1 选择排序算法功能示例

### 3.3.4 选择排序算法性能分析

#### 3.3.4.1 选择排序算法时间复杂度分析

最佳情况： $T(n)=O(n^2)$ ，即便输入的数据已经是正序，选择排序仍然需要进行  $n*(n-1)/2$  次比较。

最差情况： $T(n)=O(n^2)$ ，当输入的数据是反序时，同样需要进行  $n*(n-1)/2$  次比较。

平均情况： $T(n)=O(n^2)$ ，不管初始状态如何，选择排序的比较次数大致相同。

#### 3.3.4.2 选择排序算法空间复杂度分析

空间复杂度为  $O(1)$ ，选择排序是一个原地排序算法，除了输入数组外，它不需要额外的存储空间。

## 3.4 插入排序算法的实现

### 3.4.1 插入排序算法实现思路

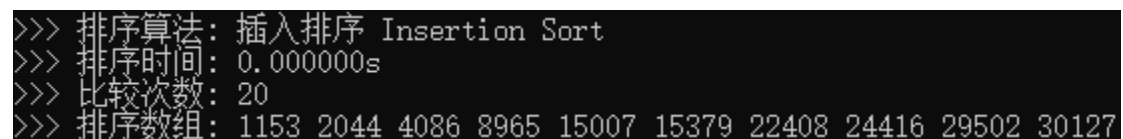
插入排序算法的实现思路为：

- (1) 从第一个元素开始，该元素可以认为已经被排序；
- (2) 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- (3) 如果该元素（已排序）大于新元素，将该元素移到下一位置；
- (4) 重复步骤 3，直到找到已排序的元素小于或等于新元素的位置；
- (5) 将新元素插入到该位置后；
- (6) 重复步骤 2 至 5。

### 3.4.2 插入排序算法核心代码

```
template <typename Type>
void insertionSort(Type arr[], int n)
{
    for (int i = 1; i < n; i++) {
        int key = arr[i], j = i - 1;
        while (j >= 0 && arr[j] > key) {
            compareCount++;
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

### 3.4.3 插入排序算法功能示例



```
>>> 排序算法: 插入排序 Insertion Sort
>>> 排序时间: 0.000000s
>>> 比较次数: 20
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127
```

图 3.4.3.1 插入排序算法功能示例

### 3.4.4 插入排序算法性能分析

#### 3.4.4.1 插入排序算法时间复杂度分析

最佳情况： $T(n)=O(n)$ ，当输入的数据已经是正序时。

最差情况： $T(n)=O(n^2)$ ，当输入的数据是反序时。

平均情况： $T(n)=O(n^2)$ ，取决于输入数据的初始顺序。



#### 3.4.4.2 插入排序算法空间复杂度分析

空间复杂度为  $O(1)$ ，插入排序是一个原地排序算法，除了输入数组外，它只需要固定数量的额外空间。

### 3.5 希尔排序算法的实现

#### 3.5.1 希尔排序算法实现思路

希尔排序算法的实现思路为：

(1) 选择一个合适的增量序列。一开始选一个大的增量（通常是数据长度的一半），逐渐减少增量，最后增量为 1；

(2) 根据当前的增量，将待排序列分割成若干个子序列，所有距离为增量的元素组成一个子序列；

(3) 对每个子序列应用直接插入排序；

(4) 减少增量，重复步骤 2 和 3，直到增量为 1，执行最后一次直接插入排序后排序完成。

#### 3.5.2 希尔排序算法核心代码

```
template <typename Type>
void shellSort(Type arr[], int n)
{
    int gap, i, j;
    for (gap = n >> 1; gap > 0; gap >= 1)
        for (i = gap; i < n; i++) {
            Type tmp = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > tmp; j -= gap) {
                compareCount++;
                arr[j + gap] = arr[j];
            }
            arr[j + gap] = tmp;
        }
}
```

#### 3.5.3 希尔排序算法功能示例

```
>>> 排序算法: 希尔排序 Shell Sort
>>> 排序时间: 0.000000s
>>> 比较次数: 8
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127
```

图 3.5.3.1 希尔排序算法功能示例

### 3.5.4 希尔排序算法性能分析

#### 3.5.4.1 希尔排序算法时间复杂度分析

最佳情况：取决于增量序列，可以达到  $O(n \log n)$ 。

最差情况： $T(n) = O(n^2)$ ，但这取决于选定的增量序列。

平均情况：时间复杂度也取决于增量序列，一般被认为是  $O(n \log^2 n)$  或  $O(n^{3/2})$ 。

#### 3.5.4.2 希尔排序算法空间复杂度分析

空间复杂度为  $O(1)$ ，希尔排序是一个原地排序算法，它不需要额外的存储空间。

## 3.6 快速排序算法的实现

### 3.6.1 快速排序算法实现思路

快速排序算法的实现思路为：

(1) 从数组中挑出一个元素，称为“基准” (pivot)；

(2) 重新排序数组，所有比基准小的元素摆放在基准前面，所有比基准大的元素摆放在基准后面（相同的数可以到任一边）。在这个分割结束之后，该基准就处于数组的中间位置。这个称为分区 (partition) 操作；

(3) 递归地把小于基准值元素的子数组和大于基准值元素的子数组排序。

### 3.6.2 快速排序算法核心代码

```
template <typename Type>
int partition(Type arr[], int low, int high)
{
    Type pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) {
        compareCount++;
        if (arr[j] < pivot)
            mySwap(arr[++i], arr[j]);
    }
    mySwap(arr[i + 1], arr[high]);
    return i + 1;
}

template <typename Type>
void quickSort(Type arr[], int low, int high)
```

```

{
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

template <typename Type>
void quickSort(Type arr[], int n)
{
    quickSort(arr, 0, n - 1);
}

```

### 3.6.3 快速排序算法功能示例

```

>>> 排序算法: 快速排序 Quick Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 20
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127

```

图 3.6.3.1 快速排序算法功能示例

### 3.6.4 快速排序算法性能分析

#### 3.6.4.1 快速排序算法时间复杂度分析

最佳情况:  $T(n)=O(n\log n)$ , 当分区操作能每次都均匀划分数组时。

最差情况:  $T(n)=O(n^2)$ , 当分区操作每次只能排除一个元素时 (例如数组已经有序或逆序)。

平均情况:  $T(n)=O(n\log n)$ , 对于大多数实际情况。

#### 3.6.4.2 快速排序算法空间复杂度分析

空间复杂度为  $O(\log n)$ , 主要是递归调用的栈空间。

## 3.7 堆排序算法的实现

### 3.7.1 堆排序算法实现思路

堆排序算法的实现思路为:

- (1) 将输入的数据数组构造成一个最大堆 (或最小堆);
- (2) 由于堆的最大 (或最小) 元素总是位于根节点, 可以通过将其与堆的最后一个元素交换并减少堆的大小, 从而将最大 (或最小) 元素移至数组末尾;
- (3) 重复这个过程, 每次从堆中移除最大 (或最小) 元素, 并减小堆的大小,

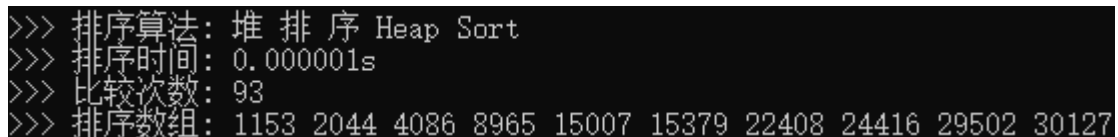
直到堆的大小为 1。

### 3.7.2 堆排序算法核心代码

```
template <typename Type>
void heapify(Type arr[], int n, int i)
{
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;
    compareCount++;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    compareCount++;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    compareCount++;
    if (largest != i) {
        mySwap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

template <typename Type>
void heapSort(Type arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        mySwap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

### 3.7.3 堆排序算法功能示例



```
>>> 排序算法: 堆 排 序 Heap Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 93
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127
```

图 3.7.3.1 堆排序算法功能示例

### 3.7.4 堆排序算法性能分析

#### 3.7.4.1 堆排序算法时间复杂度分析

堆排序的时间复杂度在所有情况下都是  $O(n \log n)$ 。这是因为创建堆的时间复杂度是  $O(n)$ ，而堆排序的时间复杂度是  $O(n \log n)$ 。

#### 3.7.4.2 堆排序算法空间复杂度分析

空间复杂度为  $O(1)$ ，堆排序是一个原地排序算法，它不需要额外的存储空间。

### 3.8 归并排序算法的实现

#### 3.8.1 归并排序算法实现思路

归并排序算法的实现思路为：

- (1) 将数组分成两半，然后对每半分别进行归并排序；
- (2) 将排序后的两部分合并成一个完整的排序数组；
- (3) 合并时，从两个数组的起始位置开始比较，选择较小的元素放入到结果数组中，直到其中一个数组结束；
- (4) 将另一个数组中剩余的元素复制到结果数组中。

#### 3.8.2 归并排序算法核心代码

```
template <typename Type>
void merge(Type arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1, n2 = right - mid, i = 0, j = 0, k = left;
    Type* leftArr = new(std::nothrow) Type[n1];
    if (leftArr == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    Type* rightArr = new(std::nothrow) Type[n2];
    if (rightArr == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        rightArr[i] = arr[mid + 1 + i];
    while (i < n1 && j < n2) {
        compareCount++;
        if (leftArr[i] <= rightArr[j])
            arr[k++] = leftArr[i++];
        else
            arr[k++] = rightArr[j++];
    }
    while (i < n1) {
```

```

        compareCount++;
        arr[k++] = leftArr[i++];
    }
    while (j < n2) {
        compareCount++;
        arr[k++] = rightArr[j++];
    }
    delete[] leftArr;
    delete[] rightArr;
}

template <typename Type>
void mergeSort(Type arr[], int left, int right)
{
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

template <typename Type>
void mergeSort(Type arr[], int n)
{
    mergeSort(arr, 0, n - 1);
}

```

### 3.8.3 归并排序算法功能示例

```

>>> 排序算法: 归并排序 Merge Sort
>>> 排序时间: 0.000007s
>>> 比较次数: 34
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127

```

图 3.8.3.1 归并排序算法功能示例

### 3.8.4 归并排序算法性能分析

#### 3.8.4.1 归并排序算法时间复杂度分析

归并排序在所有情况下的时间复杂度均为  $O(n \log n)$ 。这是因为每次分割将问题规模减半，而每层分割需要  $O(n)$  时间合并。

#### 3.8.4.2 归并排序算法空间复杂度分析

空间复杂度为  $O(n)$ ，主要是因为归并排序需要与原数组相同数量的空间来

存储合并结果。

## 3.9 基数排序算法的实现

### 3.9.1 基数排序算法实现思路

基数排序算法的实现思路为：

- (1) 找到数组中最大数，并找出最大数的位数；
- (2) 从最低位开始，对数组中的每个元素按照当前位的数值进行排序；
- (3) 使用稳定的排序算法（如计数排序）来排序每一位；
- (4) 重复上述过程，直到最高位。

### 3.9.2 基数排序算法核心代码

```
template <typename Type>
Type getMaxVal(Type arr[], int n)
{
    Type maxVal = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > maxVal)
            maxVal = arr[i];
    return maxVal;
}

template <typename Type>
void countSort(Type arr[], int n, int exp)
{
    Type* output = new(std::nothrow) Type[n];
    if (output == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    int i, count[10] = { 0 };
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

```

        delete[] output;
    }

    template <typename Type>
    void radixSort(Type arr[], int n)
    {
        Type maxVal = getMaxVal(arr, n);
        for (int exp = 1; maxVal / exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }

```

### 3.9.3 基数排序算法功能示例

```

>>> 排序算法: 基数排序 Radix Sort
>>> 排序时间: 0.000005s
>>> 比较次数: 0
>>> 排序数组: 1153 2044 4086 8965 15007 15379 22408 24416 29502 30127

```

图 3.9.3.1 基数排序算法功能示例

### 3.9.4 基数排序算法性能分析

#### 3.9.4.1 基数排序算法时间复杂度分析

基数排序的时间复杂度通常表示为  $O(nk)$ ，其中  $n$  是排序元素的个数， $k$  是数字的最大位数。这是因为基数排序对每个位数都进行了一次排序。

#### 3.9.4.2 基数排序算法空间复杂度分析

空间复杂度为  $O(n+k)$ ，主要是因为基数排序需要额外的空间来存放临时数组。

## 3.10 算法性能评测功能的实现

### 3.10.1 算法性能评测功能实现思路

算法性能评测功能的函数名为 `performSort`，算法性能评测功能实现的思路为：

- (1) 初始化比较次数计数器；
- (2) 输出排序算法名称；
- (3) 进行排序数组的内存分配和复制；
- (4) 使用 `QueryPerformanceFrequency` 和 `QueryPerformanceCounter` 测量排序函数的执行时间；
- (5) 调用排序函数；



- (6) 输出该排序算法的排序时间和比较次数；  
 (7) 释放动态分配的 `sortArr` 数组来避免内存泄漏。

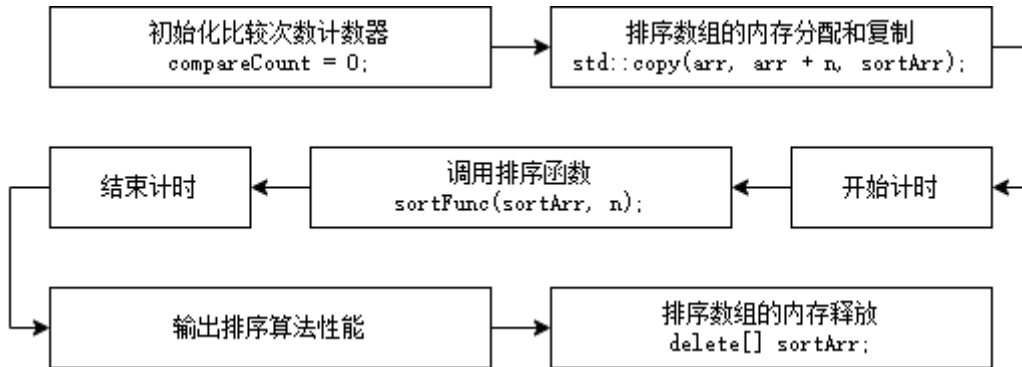


图 3.10.1.1 算法性能评测功能实现流程图

### 3.10.2 算法性能评测功能核心代码

```

template <typename Type>
void performSort(SortFunction sortFunc, Type arr[], int n, const char*
prompt)
{
    compareCount = 0;
    std::cout << std::endl << ">>> 排序算法: " << prompt << std::endl;
    int* sortArr = new(std::nothrow) int[n];
    if (sortArr == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    std::copy(arr, arr + n, sortArr);
    LARGE_INTEGER tick, begin, end;
    QueryPerformanceFrequency(&tick);
    QueryPerformanceCounter(&begin);
    sortFunc(sortArr, n);
    QueryPerformanceCounter(&end);
    std::cout << ">>> 排序时间: " << std::setiosflags(std::ios::fixed)
<< std::setprecision(6) << static_cast<double>(end.QuadPart -
begin.QuadPart) / tick.QuadPart << "s" << std::endl;
    std::cout << ">>> 比较次数: " << compareCount << std::endl;
    //std::cout << ">>> 排序数组: ";
    //for (int i = 0; i < n; i++)
    //    std::cout << sortArr[i] << " ";
    //std::cout << std::endl;
    delete[] sortArr;
}
  
```

### 3. 10. 3 算法性能评测功能示例

```
请输入要生成随机数的个数 [整数范围: 1~2147483647]: 10000
>>> 随机数生成成功 (随机数数量: 10000)
请选择排序算法: [1]
>>> 排序算法: 冒泡排序 Bubble Sort
>>> 排序时间: 0.181776s
>>> 比较次数: 49995000
请选择排序算法: [2]
>>> 排序算法: 选择排序 Selection Sort
>>> 排序时间: 0.068570s
>>> 比较次数: 49995000
请选择排序算法: [3]
>>> 排序算法: 插入排序 Insertion Sort
>>> 排序时间: 0.037815s
>>> 比较次数: 24912315
请选择排序算法: [4]
>>> 排序算法: 希尔排序 Shell Sort
>>> 排序时间: 0.001281s
>>> 比较次数: 162326
请选择排序算法: [5]
>>> 排序算法: 快速排序 Quick Sort
>>> 排序时间: 0.000886s
>>> 比较次数: 154732
请选择排序算法: [6]
>>> 排序算法: 堆排序 Heap Sort
>>> 排序时间: 0.001610s
>>> 比较次数: 387831
请选择排序算法: [7]
>>> 排序算法: 归并排序 Merge Sort
>>> 排序时间: 0.004009s
>>> 比较次数: 133616
请选择排序算法: [8]
>>> 排序算法: 基数排序 Radix Sort
>>> 排序时间: 0.000790s
>>> 比较次数: 0
```

图 3. 10. 3. 1 算法性能评测功能示例

## 3.11 异常处理功能的实现

### 3.11.1 动态内存申请失败的异常处理

在进行动态内存申请时，程序使用 `new(std::nothrow)` 来尝试分配内存。`new(std::nothrow)` 在分配内存失败时不会引发异常，而是返回一个空指针（`NULL` 或 `nullptr`），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

(1) 向标准错误流 `std::cerr` 输出一条错误消息 "Error: Memory allocation failed."，指出内存分配失败；

(2) 调用 `exit` 函数，返回错误码 `MEMORY_ALLOCATION_ERROR`（通过宏定义方式定义为-1），用于指示内存分配错误，并导致程序退出。

下面是动态内存申请的异常处理的一个代码示例：

```
int* sortArr = new(std::nothrow) int[n];
if (sortArr == NULL) {
    std::cerr << "Error: Memory allocation failed." << std::endl;
    exit(MEMORY_ALLOCATION_ERROR);
}
std::copy(arr, arr + n, sortArr);
```

### 3.11.2 输入非法的异常处理

#### 3.11.2.1 要生成随机数的个数输入非法的异常处理

程序通过调用 `inputInteger` 函数输入要生成随机数的个数。`inputInteger` 函数用于获取用户输入的整数，同时限制输入必须在指定的范围内，函数的代码如下：

```
int inputInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    while (true) {
        std::cout << "请输入" << prompt << " [整数范围: " << lowerLimit
        << "~" << upperLimit << "]: ";
        double tempInput;
        std::cin >> tempInput;
        if (std::cin.good() && tempInput == static_cast<int>(tempInput)
        && tempInput >= lowerLimit && tempInput <= upperLimit) {
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
            return static_cast<int>(tempInput);
        }
        else {
            std::cerr << std::endl << ">>> " << prompt << "输入不合法，
            请重新输入" << prompt << "! " << std::endl << std::endl;
```

```

        std::cin.clear();
        std::cin.ignore(INT_MAX, '\n');
    }
}
}

```

**inputInteger** 函数对输入非法的情况进行了处理，代码具体执行逻辑如下：

- (1) 进入一个无限循环，它会一直运行直到用户提供有效的输入；
- (2) 用户的输入被读取到 **tempInput** 变量中，这里采用 **double** 类型来接收输入以便后续检查；
- (3) 进行输入验证：**std::cin.good()**检查输入流的状态是否正常，确保没有发生数据类型输入错误，**tempInput==static\_cast<int>(tempInput)**检查用户输入是否为整数，通过将其转换为整数再比较，**tempInput>=lowerLimit**和 **tempInput<=upperLimit** 确保输入在指定的范围内；
- (4) 合法输入处理：如果用户提供了合法的输入，函数会清除输入流的错误状态，丢弃输入缓冲区中的任何剩余内容，然后返回转换后的整数值；
- (5) 非法输入处理：如果用户提供的输入不合法，函数会输出错误消息，清除输入流的错误状态，丢弃输入缓冲区中的内容，并继续循环以等待用户提供合法的输入。

### 3. 11. 2. 2 排序算法类型输入非法的异常处理

排序算法类型输入非法的异常处理通过如下代码实现：

```

int selectOptn(void)
{
    std::cout << std::endl << "请选择排序算法：";
    char optn;
    while (true) {
        optn = _getch();
        if (optn == 0 || optn == -32)
            optn = _getch();
        else if (optn >= '0' && optn <= '8') {
            std::cout << "[" << optn << "]" << std::endl;
            return optn - '0';
        }
    }
}

```

这段代码会一直等待用户输入，只有当用户输入有效的数字字符（'0'到'8'）时，才会返回该数字的整数值，表示用户选择的操作选项。如果用户输入无效字符，循环会继续等待用户提供有效的输入。这段代码的具体执行逻辑如下：

- (1) 显示提示信息，其中包含选项[1]到[8]，以及[0]退出程序的选项；

- (2) 进入一个无限循环，以等待用户输入；
- (3) 用户输入的字符会被 `_getch()` 函数获取。这个函数通常用于在控制台应用程序中获取单个字符而不显示在屏幕上。用户的输入存储在变量 `optn` 中；
- (4) 代码检查用户输入是否是数字字符（'0'到'8'）或特殊的控制字符。如果用户按下非数字字符或特殊控制字符，它将不执行下面的操作；
- (5) 如果用户输入是数字字符（'0'到'8'），它会在屏幕上显示用户输入的数字字符，并返回对应的整数值；
- (6) 如果用户输入的不是数字字符（'0'到'8'），代码将保持在循环中，继续等待有效的输入，这确保了只有在用户输入正确的选项时才会退出循环。

## 4 项目测试

### 4.1 输入要生成随机数的个数功能测试

分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理。

当输入合法时，程序继续运行，并给出随机数生成成功的提示信息。

```
请输入要生成随机数的个数 [整数范围: 1~2147483647]: 0
>>> 要生成随机数的个数输入不合法, 请重新输入要生成随机数的个数!
请输入要生成随机数的个数 [整数范围: 1~2147483647]: 2147483648
>>> 要生成随机数的个数输入不合法, 请重新输入要生成随机数的个数!
请输入要生成随机数的个数 [整数范围: 1~2147483647]: 5.5
>>> 要生成随机数的个数输入不合法, 请重新输入要生成随机数的个数!
请输入要生成随机数的个数 [整数范围: 1~2147483647]: a
>>> 要生成随机数的个数输入不合法, 请重新输入要生成随机数的个数!
请输入要生成随机数的个数 [整数范围: 1~2147483647]: abc
>>> 要生成随机数的个数输入不合法, 请重新输入要生成随机数的个数!
请输入要生成随机数的个数 [整数范围: 1~2147483647]: 10000
>>> 随机数生成成功 (随机数数量: 10000)
```

图 4.1.1 输入要生成随机数的个数功能测试

## 4.2 排序算法功能正确性测试

修改代码使执行算法性能评测后输出排序后数组。输入要生成随机数的个数，可以验证各个排序算法的正确性，均正确实现了数据的升序排列。

```
请输入要生成随机数的个数 [整数范围: 1~2147483647]: 15
>>> 随机数生成成功 (随机数数量: 15)
请选择排序算法: [1]
>>> 排序算法: 冒泡排序 Bubble Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 105
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [2]
>>> 排序算法: 选择排序 Selection Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 105
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [3]
>>> 排序算法: 插入排序 Insertion Sort
>>> 排序时间: 0.000000s
>>> 比较次数: 63
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [4]
>>> 排序算法: 希尔排序 Shell Sort
>>> 排序时间: 0.000000s
>>> 比较次数: 21
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [5]
>>> 排序算法: 快速排序 Quick Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 41
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [6]
>>> 排序算法: 堆排序 Heap Sort
>>> 排序时间: 0.000001s
>>> 比较次数: 162
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [7]
>>> 排序算法: 归并排序 Merge Sort
>>> 排序时间: 0.000009s
>>> 比较次数: 59
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
请选择排序算法: [8]
>>> 排序算法: 基数排序 Radix Sort
>>> 排序时间: 0.000005s
>>> 比较次数: 0
>>> 排序数组: 1755 5822 7927 9209 9585 10036 11239 11314 12669 14160 14672 16148 16873 27121 29768
```

图 4.2.1 排序算法功能正确性测试

## 5 集成开发环境与编译运行环境

Windows 系统: Windows 11 x64

Windows 集成开发环境: Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境：本项目适用于 x86 架构和 x64 架构

Linux 系统：CentOS 7 x64

Linux 编译命令：

```
g++ -c '/root/桌面/Share_Folder/comparison_of_sorting_algorithms.cpp' -o '/root/桌面/Share_Folder/comparison_of_sorting_algorithms' -lnurses
```

Linux 运行命令：

```
'/root/桌面/Share_Folder/comparison_of_sorting_algorithms'
```



图 5.1 Linux 环境程序运行示例

本项目使用条件编译解决 Windows 系统和 Linux 系统编译环境的差异，示例代码如下。

```
#ifdef _WIN32
#include <conio.h>
#elif __linux__
#include <nurses.h>
#endif
```