
项目说明文档

数据结构课程设计

——8 种排序算法的比较案例

作者姓名：_____张翔_____

学 号：_____2352985_____

指导教师：_____张颖_____

学院、专业：_____计算机科学与技术学院 软件工程_____

同济大学

Tongji University

二〇二四年十二月六日

1 项目分析

1.1 项目背景分析

排序算法是计算机科学中至关重要的研究方向之一，其核心思想和实际应用广泛渗透于数据处理、信息检索和系统优化等领域。排序的效率直接影响到大规模数据处理的性能，特别是在数据库管理、搜索引擎优化以及电子商务推荐系统中，合理选择高效的排序算法能够显著提升系统响应速度和资源利用率。本项目旨在通过对多种经典排序算法的性能分析，包括运行时间、比较次数和交换次数等指标，探索不同算法在各种数据规模下的优缺点，揭示其时间复杂度、空间复杂度及稳定性等特性，从而为实际开发中的算法选择和优化提供科学依据，进一步加深对数据结构与算法设计核心思想的理解。

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）是一种简单直观的交换排序算法，其核心思想是通过相邻元素的两两比较和交换，将较大的元素逐步“冒泡”到数组的末尾。具体操作是，从数组的第一个元素开始，

依次比较相邻的两个元素，如果前一个元素比后一个大，就交换位置。每一轮遍历结束后，最大的元素会被固定在当前未排序部分的末尾。这个过程会重复进行，直到整个数组有序。

1.3.2 选择排序算法

选择排序（Selection Sort）是一种直接选择排序算法，其特点是每轮遍历从未排序部分中选出最小（或最大）的元素，并将其与未排序部分的第一个元素交换位置。具体操作是，从未排序部分的第一个元素开始，遍历整个未排序部分，找到其中的最小元素，与当前起始位置的元素交换。经过一轮操作后，未排序部分缩小，而已排序部分扩大，直至排序完成。

1.3.3 插入排序算法

插入排序（Insertion Sort）通过逐步构建有序序列，将未排序部分的第一个元素插入到有序部分的适当位置。具体操作是，从数组的第二个元素开始，将其与前面已排序部分的元素逐一比较，如果当前元素更小，就将前面的元素向后移动，为其腾出位置，直到找到合适位置插入。

1.3.4 希尔排序算法

希尔排序是插入排序的改进版本，通过分组的方式提高效率。算法将数组按照一定的间隔（gap）划分为若干组，对每组分别进行插入排序，然后逐步缩小间隔，最终在间隔为 1 时完成整体排序。通过这种方式，希尔排序减少了数据移动次数，提高了大规模数据排序的效率。

1.3.5 快速排序算法

快速排序是一种基于分治思想的高效排序算法。算法通过选择一个基准元素，将序列分为两部分：小于基准的部分和大于基准的部分。然后对这两部分分别递归进行快速排序。每次递归完成后，基准元素都会被放置到正确的位置，从而逐步形成有序序列。

1.3.6 堆排序算法

堆排序是一种基于堆结构的排序算法，常用大顶堆或小顶堆来实现。首先将数组构建成一个完全二叉堆结构，然后依次将堆顶元素（最大或最小值）与末尾元素交换，缩小堆的范围，并重新调整剩余部分为堆结构。这个过程重复进行，直到整个数组有序。但堆排序是不稳定的排序算法，适合对大规模数据进行高效排序。

1.3.7 归并排序算法

归并排序是一种稳定且高效的分治算法。它通过递归地将数组分割为两部分，分别排序后再合并为一个有序数组。具体过程是，将数组不断二分，直至每部分只包含一个元素，然后从最小子序

列开始，按大小顺序合并为更大的有序序列，直到最终形成完整的有序数组。但需要额外的存储空间，因此在内存资源有限时可能不适用。它因稳定性和效率高，广泛应用于需要稳定排序的场景。

1.3.8 基数排序算法

基数排序是一种非比较型排序算法，适用于整数或特定格式的数据排序。算法通过对数据的每一位（如个位、十位）进行排序，逐步完成整体排序。首先按照最低有效位（如个位）将数据分组排序，再依次对次低位、次高位等进行排序，直至最高有效位排序完成。它是一种稳定排序算法，效率高于常规比较型算法，但对数据的格式有特殊要求。

1.3.9 算法性能测评功能

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合法信息，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) 便于理解：通过使用传统的 C 语言数组，可以更直观地学习和掌握排序算法和底层数据结构的工作原理。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。

在使用 C 语言数组时，需要特别关注内存的正确分配与释放，并防止出现内存泄漏或数组越界等问题，以确保程序的稳定性和可靠性。

2.2 泛型化设计

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；

(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，甚至还能对重载了比较运算符的结构体等进行排序。

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；

(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。

2.3 结构体和类的设计

2.3.1 SortAlgorithm 类的设计

2.3.1.1 概述

SortAlgorithm 类是一个用于输入无序的数组并通过选择不同排序算法进行排序的类。该类的目的是方便用户直观地观察到不同排序算法在处理同一段无序数组时的性能差异，从而帮助用户选择进行排序的最佳选项。

2.3.1.2 类定义

```
// 排序算法类的设计
template<typename Type>
```

```
class SortAlgorithm {
private:
    // 要排序的数组
    Type* arr;
    int numCount;
    // 比较次数
    int compareCount;
public:
    // 构造函数和析构函数
    SortAlgorithm(int num);
    ~SortAlgorithm();
    // 输入需要排序的数组
    void inputArr(Type _arr[]);
    // 选择排序算法
    bool inputOptn();
    // 实现排序算法
    void bubbleSort();
    void selectionSort();
    void insertionSort();
    void shellSort();
    int partition(int low, int high);
    void quickSort(int low, int high);
    void heapify(int n, int i);
    void heapSort();
    void merge(int left, int mid, int right);
    void mergeSort(int left, int right);
    Type getMaxVal();
    void countSort(int exp);
    void radixSort();
};
```

2.4 项目主体架构设计

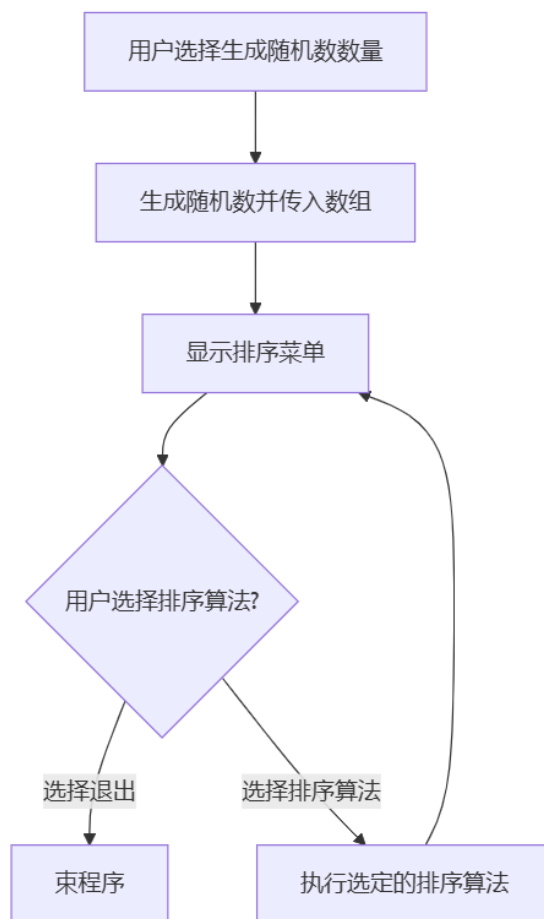


图 2.3.1 项目主体架构设计流程图

3 项目功能实现

3.1 项目主体架构的实现

3.1.1 项目主体架构的实现思路

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) 创建一个 `SortAlgorithm<int>` 类型的对象 `sort`，并传入随机数数量 `num`，调用 `sort` 对象的 `inputArr` 方法，将生成的随机数数组 `_arr` 传递给排序类处理；
- (3) 使用 `while` 循环，持续向用户展示排序算法选择菜单，并等待用户输入；

(4)调用 sort 对象的 inputOptn 方法获取用户选择的排序算法选项, inputOptn 方法的职责是读取用户输入, 执行相应的排序算法, 并返回是否继续循环;

(5)用户选择退出选项后, 结束循环, 输出提示信息, 表明程序运行结束;

(6)每次执行排序操作后, 调用 sort.inputArr 方法重新加载原始的随机数数组, 以确保不同排序方法基于相同的输入数据。

3.1.2 项目主体架构的核心代码

```
int main()
{
    srand((unsigned int)time(0));

    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::cout << "|          [1] --- 冒泡排序          |" << std::endl;
    std::cout << "|          [2] --- 选择排序          |" << std::endl;
    std::cout << "|          [3] --- 插入排序          |" << std::endl;
    std::cout << "|          [4] --- 希尔排序          |" << std::endl;
    std::cout << "|          [5] --- 快速排序          |" << std::endl;
    std::cout << "|          [6] --- 堆 排 序          |" << std::endl;
    std::cout << "|          [7] --- 归并排序          |" << std::endl;
    std::cout << "|          [8] --- 基数排序          |" << std::endl;
    std::cout << "|          [9] --- 退出程序          |" << std::endl;
    std::cout << "+-----+" << std::endl;

    int num = inputInteger(1, INT_MAX, "要生成随机数的个数");
    int* _arr = new(std::nothrow) int[num];
    if (_arr == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }
    for (int i = 0; i < num; i++) {
        _arr[i] = rand();
    }
    SortAlgorithm<int> sort(num);
```

```

    sort.inputArr(_arr);
    std::cout << std::endl << ">>> 随机数生成成功 (生成数量:" << num << ")" <<
std::endl;

    while (sort.inputOptn()) { sort.inputArr(_arr); }

    std::cout << std::endl << ">>> 排序算法比较结束" << std::endl;
    delete[] _arr;
    return 0;
}

```

3.1.3 项目主体架构示例

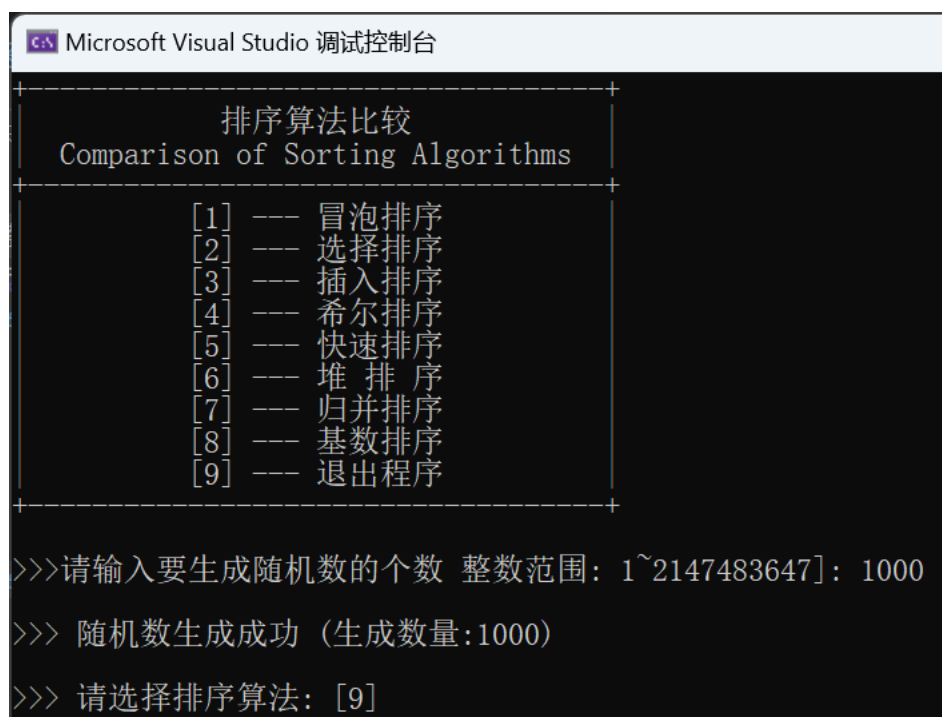


图 3.1.3.1 项目主体架构示例

3.2 冒泡排序算法的实现

3.2.1 冒泡排序算法的实现思路

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 冒泡排序算法的核心代码

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

3.2.3 冒泡排序算法示例

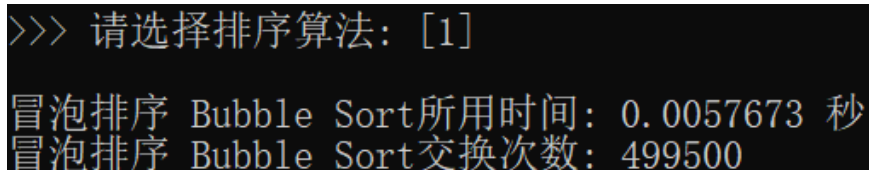
A terminal window with a black background and yellow text. The first line shows a prompt '>>>>' followed by the input '请选择排序算法: [1]'. The second line shows the output '冒泡排序 Bubble Sort所用时间: 0.0057673 秒'. The third line shows the output '冒泡排序 Bubble Sort交换次数: 499500'.

图 3.2.3.1 冒泡排序算法功能示例

3.3 选择排序算法的实现

3.3.1 选择排序算法的实现思路

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 选择排序算法的核心代码

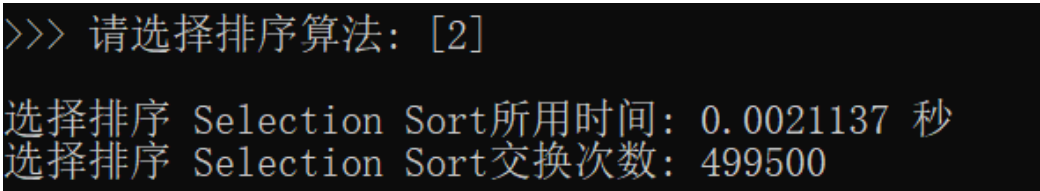
```
template<typename Type>
```

```

void SortAlgorithm<Type>::selectionSort() {
    for (int i = 0; i < numCount - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < numCount; j++) {
            compareCount++;
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }
        mySwap(arr[i], arr[minIdx]);
    }
}

```

3.3.3 选择排序算法构示例



```

>>> 请选择排序算法: [2]
选择排序 Selection Sort所用时间: 0.0021137 秒
选择排序 Selection Sort交换次数: 499500

```

图 3.3.3.1 选择排序算法功能示例

3.4 插入排序算法的实现

3.4.1 插入排序算法的实现思路

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 插入排序算法的核心代码

```

template<typename Type>
void SortAlgorithm<Type>::insertionSort() {

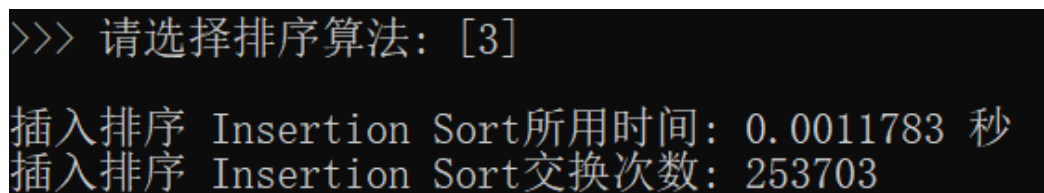
```

```

    for (int i = 1; i < numCount; 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 插入排序算法示例



```

>>> 请选择排序算法: [3]
插入排序 Insertion Sort所用时间: 0.0011783 秒
插入排序 Insertion Sort交换次数: 253703

```

图 3.4.3.1 插入排序算法功能示例

3.5 希尔排序算法的实现

3.5.1 希尔排序算法的实现思路

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 希尔排序算法的核心代码

```

template<typename Type>
void SortAlgorithm<Type>::shellSort() {
    for (int gap = numCount / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < numCount; i++) {

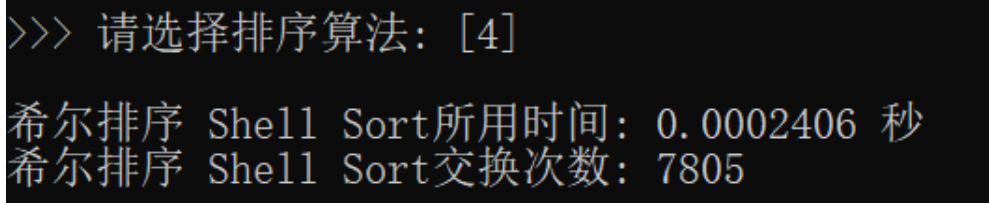
```

```

        Type temp = arr[i], j = i;
        while (j >= gap && arr[j - gap] > temp) {
            compareCount++;
            arr[j] = arr[j - gap];
            j -= gap;
        }
        arr[j] = temp;
    }
}

```

3.5.3 希尔排序算法示例



```

>>> 请选择排序算法: [4]
希尔排序 Shell Sort所用时间: 0.0002406 秒
希尔排序 Shell Sort交换次数: 7805

```

图 3.5.3.1 希尔排序算法功能示例

3.6 快速排序算法的实现

3.6.1 快速排序算法的实现思路

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 快速排序算法的核心代码

```

template<typename Type>
int SortAlgorithm<Type>::partition(int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

```

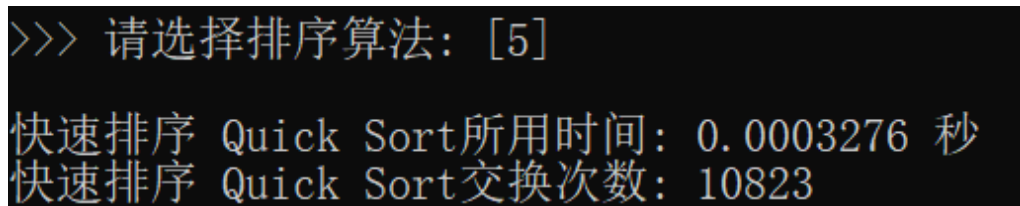
```

        for (int j = low; j < high; j++) {
            compareCount++;
            if (arr[j] < pivot)
                mySwap(arr[++i], arr[j]);
        }
        mySwap(arr[i + 1], arr[high]);
        return i + 1;
    }

template<typename Type>
void SortAlgorithm<Type>::quickSort(int low, int high) {
    if (low < high) {
        int pi = partition(low, high);
        quickSort(low, pi - 1);
        quickSort(pi + 1, high);
    }
}

```

3.6.3 快速排序算法示例



```

>>> 请选择排序算法: [5]
快速排序 Quick Sort所用时间: 0.0003276 秒
快速排序 Quick Sort交换次数: 10823

```

图 3.6.3.1 快速排序算法功能示例

3.7 堆排序算法的实现

3.7.1 堆排序算法的实现思路

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；

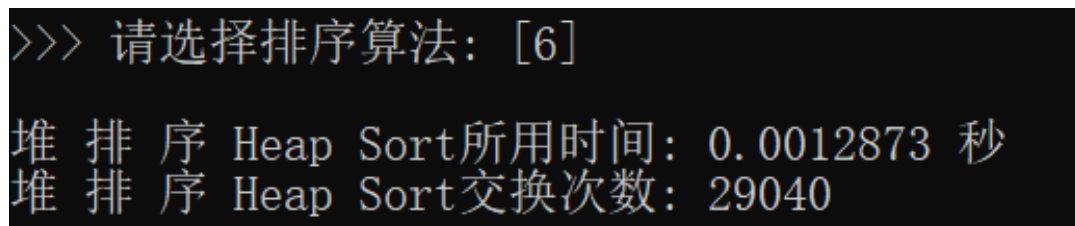
(3)重复这个过程，每次从堆中移除最大（或最小）元素，并减小堆的大小，直到堆的大小为1。

3.7.2 堆排序算法的核心代码

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

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

3.7.3 堆排序算法示例



```
>>> 请选择排序算法: [6]
堆 排 序 Heap Sort所用时间: 0.0012873 秒
堆 排 序 Heap Sort交换次数: 29040
```

图 3.7.3.1 堆排序算法功能示例

3.8 归并排序算法的实现

3.8.1 归并排序算法的实现思路

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 归并排序算法的核心代码

```
template<typename Type>
void SortAlgorithm<Type>::merge(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 == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }
    Type* rightArr = new(std::nothrow) Type[n2];
    if (rightArr == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }
    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 SortAlgorithm<Type>::mergeSort(int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(left, mid);
        mergeSort(mid + 1, right);
        merge(left, mid, right);
    }
}

```

3.8.3 归并排序算法示例

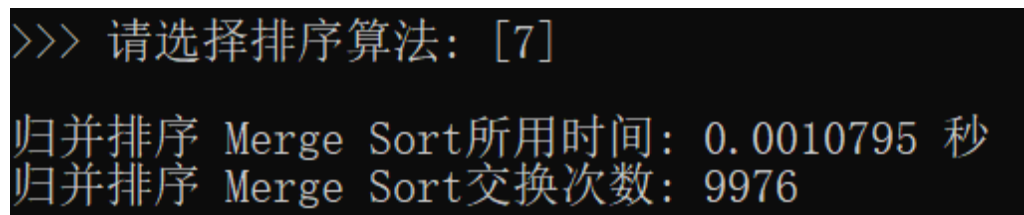
A terminal window with a black background and orange text. The first line shows a prompt '>>>' followed by the text '请选择排序算法: [7]'. The second line shows '归并排序 Merge Sort所用时间: 0.0010795 秒'. The third line shows '归并排序 Merge Sort交换次数: 9976'.

图 3.8.3.1 归并排序算法功能示例

3.9 基数排序算法的实现

3.9.1 基数排序算法的实现思路

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 基数排序算法的核心代码

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

template<typename Type>
void SortAlgorithm<Type>::countSort(int exp) {
    Type* output = new(std::nothrow) Type[numCount];
    if (output == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }
}
```

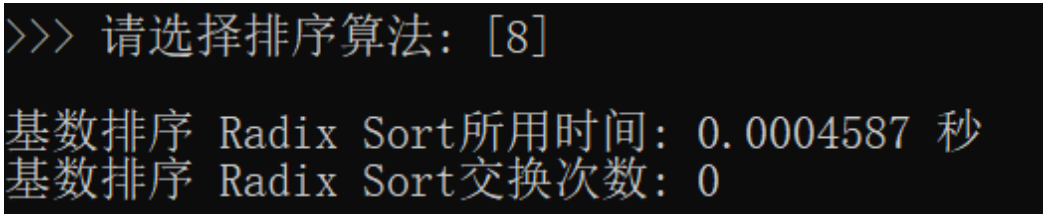
```

    }
    int i, count[10] = { 0 };
    for (i = 0; i < numCount; i++)
        count[(arr[i] / exp) % 10]++;
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (i = numCount - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (i = 0; i < numCount; i++)
        arr[i] = output[i];
    delete[] output;
}

template<typename Type>
void SortAlgorithm<Type>::radixSort() {
    Type maxVal = getMaxVal();
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countSort(exp);
    }
}

```

3.9.3 基数排序算法示例



```

>>> 请选择排序算法: [8]
基数排序 Radix Sort所用时间: 0.0004587 秒
基数排序 Radix Sort交换次数: 0

```

图 3.9.3.1 基数排序算法功能示例

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

算法性能评测功能实现的思路为：通过用户选择不同排序算法，执行排序并记录时间和比较次数，以测试算法性能。

(1) 用户输入对应数字选择排序算法，`_getch()` 用于捕获输入。

(2) 执行排序前重置 `compareCount`，使用 `std::chrono` 记录开始时间。

(3) 完成排序后记录结束时间，计算运行时长并输出，同时打印比较次数以反映算法复杂性。

它支持多种经典排序算法（冒泡、快速、堆排序等），逻辑清晰，方便扩展新算法。

退出选项由 '9' 控制返回 `false`。通过运行时间和比较次数，直观评估算法效率。

3.10.2 算法性能评测功能核心代码

```
// 进行排序算法的选择
template<typename Type>
bool SortAlgorithm<Type>::inputOptn() {
    // 输入提示
    std::cout << std::endl << ">>> 请选择排序算法: ";
    char option;
    while (true) {
        option = _getch();
        if (option == 0 || option == -32) {
            option = _getch();
        }
        else if (option >= '1' || option <= '9') {
            std::cout << "[" << option << "]" << std::endl << std::endl;
            break;
        }
    }

    // 清零比较次数
    compareCount = 0;
```

```
// 开始记录时间
auto start = std::chrono::high_resolution_clock::now();

// 根据选项执行相应的排序算法
if (option == '1')
    bubbleSort();
else if (option == '2')
    selectionSort();
else if (option == '3')
    insertionSort();
else if (option == '4')
    shellSort();
else if (option == '5')
    quickSort(0, numCount - 1);
else if (option == '6')
    heapSort();
else if (option == '7')
    mergeSort(0, numCount - 1);
else if (option == '8')
    radixSort();
else if (option == '9')
    return false;

// 结束计时
auto end = std::chrono::high_resolution_clock::now();
// 计算耗时
std::chrono::duration<double> elapsed = end - start;

int alg = option - '0' - 1;
if (alg >= 0 || alg <= 8) {
```

```

        std::cout << Algorithm[alg] << "所用时间: " << elapsed.count() << " 秒" <<
std::endl;

        std::cout << Algorithm[alg] << "交换次数: " << compareCount << std::endl;
    }

    return true;
}

```

3.10.3 算法性能评测功能示例

见于上面 8 种排序算法的功能示例，这里不重复展示。

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。

(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 -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* _arr = new(std::nothrow) int[num];
if (_arr == nullptr) {
    std::cerr << "Error: Memory allocation failed." << std::endl;
    exit(-1);
}

```

3.11.2 生成随机数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。函数代码如下：

```

int inputInteger(int lowerLimit, int upperLimit, const char* prompt) {
    std::cout << std::endl;
    std::cout << ">>>" << "请输入" << prompt << " 整数范围: " << lowerLimit << "~" <<
upperLimit << "]: ";
}

```

```

int input;
while (true) {
    std::cin >> input;
    if (std::cin.good() && input >= lowerLimit && input <= upperLimit) {
        std::cin.clear();
        std::cin.ignore(INT_MAX, '\n');
        return input;
    }
    else {
        std::cerr << ">>> " << prompt << "输入不合法，请重新输入!" << std::endl;
        std::cin.clear();
        std::cin.ignore(INT_MAX, '\n');
    }
}
}

```

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) `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，再通过与最大值最小值比较看输入数据范围是否正确；
- (3) 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。

3.11.3 排序算法选择非法的异常处理

该部分功能代码如下：

```

std::cout << std::endl << ">>> 请选择排序算法: ";
char option;
while (true) {
    option = _getch();
    if (option == 0 || option == -32) {
        option = _getch();
    }
}

```

```
    else if (option >= '1' && option <= '9') {  
        std::cout << "[" << option << "]" << std::endl << std::endl;  
        break;  
    }  
}
```

该段代码的具体执行逻辑如下：通过 `_getch()` 等待用户输入，只有当用户输入有效的数字字符（'1' 到 '9'）时，才会返回该数字的整数值，否则会循环会继续等待用户提供有效的输入，只有在用户输入正确的选项时才会退出循环。

4 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)