



同濟大學  
TONGJI UNIVERSITY

## 数据结构课程设计

### 项目说明文档

# 排序算法比较系统

姓 名： 马小龙

学 号： 2353814

学 院： 计算机科学与技术学院（软件学院）

专 业： 软件工程

指导教师： 张颖

二〇二四年十一月二十六日

# 目录

第 1 章 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 随机数生成功能 .....	2
1.3.2 排序算法实现 .....	2
1.3.2.1 冒泡排序.....	2
1.3.2.2 选择排序.....	2
1.3.2.3 插入排序.....	2
1.3.2.4 希尔排序.....	2
1.3.2.5 快速排序.....	2
1.3.2.6 堆排序.....	2
1.3.2.7 归并排序.....	3
1.3.2.8 基数排序.....	3
1.3.3 算法性能分析功能 .....	3
1.3.4 异常处理功能 .....	3
第 2 章 项目设计.....	4
2.1 数据结构设计.....	4
2.2 模板设计.....	4
2.3 项目框架设计.....	4
2.3.1 项目框架流程图 .....	4
2.3.2 项目框架流程 .....	5
第 3 章 项目功能实现.....	6
3.1 项目主体架构与性能分析功能.....	6
3.1.1 实现思路.....	6
3.1.2 流程图.....	6
3.1.3 核心代码.....	7
3.1.4 示例.....	9
3.2 排序算法实现.....	10
3.2.1 冒泡排序算法 .....	10
3.2.1.1 实现思路.....	10
3.2.1.2 核心代码.....	10
3.2.1.3 性能分析.....	10
3.2.2 选择排序算法 .....	11
3.2.2.1 实现思路.....	11

3.2.2.2 核心代码.....	11
3.2.2.3 性能分析.....	11
3.2.3 插入排序算法.....	12
3.2.3.1 实现思路.....	12
3.2.3.2 核心代码.....	12
3.2.3.4 性能分析.....	13
3.2.4 希尔排序算法.....	13
3.2.4.1 实现思路.....	13
3.2.4.2 核心代码.....	13
3.2.4.3 性能分析.....	14
3.2.5 快速排序算法.....	14
3.2.5.1 实现思路.....	14
3.2.5.2 核心代码.....	15
3.2.5.3 性能分析.....	15
3.2.6 堆排序算法.....	16
3.2.6.1 实现思路.....	16
3.2.6.2 核心代码.....	16
3.2.6.3 性能分析.....	17
3.2.7 归并排序算法.....	17
3.2.7.1 实现思路.....	17
3.2.7.2 核心代码.....	17
3.2.7.3 性能分析.....	18
3.2.8 基数排序算法.....	19
3.2.8.1 实现思路.....	19
3.2.8.2 核心代码.....	19
3.2.8.3 性能分析.....	20
3.3 异常处理功能.....	20
3.9.1 输入非法的异常处理.....	20
3.9.1.1 随机数个数输入非法的异常处理.....	20
3.9.1.2 操作类型输入非法的异常处理.....	21
3.9.2 动态内存申请失败的异常处理.....	21
第4章 项目测试.....	22
4.1 输入功能测试.....	22
4.1.1 随机数的个数输入功能测试.....	22
4.1.2 操作类型输入功能测试.....	22
4.2 排序算法正确性验证.....	23
第5章 相关说明.....	24
5.1 编程语言.....	24

5.2 Windows 环境 .....	24
5.3 Linux 环境.....	24

## 第 1 章 项目分析

### 1.1 项目背景分析

排序算法是计算机科学和数据处理的重要内容，其应用贯穿于多种场景，如搜索优化、数据库管理和统计分析。掌握不同排序算法的性能特征，有助于在不同需求场景中选择适宜的排序策略。

在计算机发展早期，计算资源有限，排序算法的效率尤为重要；而在现代，随着硬件性能的提升，算法的稳定性、扩展性以及在海量的数据中的表现成为新的关注点。

本项目旨在实现一个排序算法比较系统，允许用户输入待排序的数据，并根据选择的排序算法对其进行排序，最终输出排序结果及相关的性能指标，如排序所用时间和比较次数。用户可以选择多种常见的排序算法进行测试，包括冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序和基数排序。系统将通过比较不同算法的效率，帮助用户了解每种排序算法的性能差异。

### 1.2 项目需求分析

为了更好地理解和评估多种排序算法，本项目需要实现多种排序算法的实现，包括冒泡排序、插入排序、选择排序、希尔排序、快速排序、堆排序、归并排序和基数排序，以进行进一步分析；同时，本项目还需要实现随机生成数据集，为程序测试提供样本；此外，项目还需要测量并记录各种算法在相同规模下的运行时间、比较次数，以方便对其性能进行分析。最后，项目需要处理各种异常情况，确保系统稳定运行。

技术需求包括支持 Windows 和 Linux 等操作系统，采用 C++ 等主流编程语言，采用合适的数据结构。

### 1.3 项目功能分析

本项目旨在实现多种不同的排序算法、生成随机数，对不同排序进行性能分析，以比较不同排序算法优劣。

### 1.3.1 随机数生成功能

程序需要利用随机数生成器，创建不同规模和特点的数据集，为算法测试提供多样化场景。

### 1.3.2 排序算法实现

#### 1.3.2.1 冒泡排序

冒泡排序（Bubble Sort）是一种简单直观的排序算法，其核心思想是通过多次遍历数组，依次比较相邻的两个元素并交换顺序较大的元素，使其逐步“冒泡”到数组末尾。每次遍历都会将未排序部分中最大的元素放到正确位置。

#### 1.3.2.2 选择排序

选择排序（Selection Sort）是一种简单且直观的排序算法。其基本思想是将数组分为已排序部分和未排序部分，每次从未排序部分中选择最小（或最大）的元素，将其与未排序部分的第一个元素交换位置，从而逐步扩大已排序部分。

#### 1.3.2.3 插入排序

插入排序（Insertion Sort）是一种构造简单、适合小规模数据的排序算法。其基本思想是将数组分为已排序部分和未排序部分，每次从未排序部分取出一个元素，将其插入到已排序部分的适当位置，保持部分有序。

#### 1.3.2.4 希尔排序

希尔排序（Shell Sort）是一种基于插入排序的改进算法，也称为缩小增量排序。其核心思想是将待排序数组按一定间隔分组，对每组分别进行插入排序，逐步减少间隔，最终间隔为1时进行一次完整的插入排序。通过提前部分排序，减少了插入排序需要的元素移动次数。

#### 1.3.2.5 快速排序

快速排序（Quick Sort）是一种分治法的排序算法，通常被认为是最有效的排序算法之一。其基本思想是通过一个“分区”操作，将数组分为两个子数组：左侧子数组中的元素都比基准元素小，右侧子数组中的元素都比基准元素大。然后递归地对左右子数组进行排序，最终得到整个有序数组。

#### 1.3.2.6 堆排序

堆排序（Heap Sort）是一种基于完全二叉树的排序算法，它利用堆这一数据结构来实现排序。堆排序分为两个主要阶段：先将待排序的数组构建成一个最大堆，将根节点与数组的最后一个元素交换，然后减少堆的大小（即排除最后一个元素），对新的根节点进行堆化操作，恢复最大堆结构。重复这个过程，直到堆的大小为 1，排序完成。

#### 1.3.2.7 归并排序

归并排序（Merge Sort）是一种基于分治法的排序算法，它通过将一个大问题分解成多个小问题来解决。其基本思想是将待排序的数组分成两半，递归地对每一半进行排序，然后将已排序的两半合并成一个有序的整体。归并排序首先将数组不断地二分，直到每个子数组只有一个元素；然后将这些子数组两两合并，合并时按大小顺序排列，直到整个数组合并为一个有序数组。

#### 1.3.2.8 基数排序

基数排序（Radix Sort）是一种非比较型排序算法，主要用于处理整数或字符串等数据类型。它的基本思想是将数据按位（从低位到高位或高位到低位）逐位进行排序，通过多次排序最终得到有序序列。基数排序利用了整数的位数进行分组，而不是通过比较元素的大小来排序。排序从最低有效位（LSD, Least Significant Digit）开始，对所有元素按当前位的值进行排序。完成对最低有效位的排序后，进行下一位的排序，直到排序完所有位。每次排序需要确保使用稳定的排序算法（如计数排序）来处理每一位上的元素，以避免破坏先前排序的结果。

### 1.3.3 算法性能分析功能

实现对各种排序算法的性能分析，包括排序花费时间以及比较次数。

### 1.3.4 异常处理功能

程序对各种异常进行了基本处理，以提升系统稳定性。

## 第2章 项目设计

### 2.1 数据结构设计

本项目的主要数据结构是数组。

在排序过程中，所有的排序算法均操作整数数组。在排序前，生成一个大小为  $N$  的随机整数数组，后将该数组赋值给一个新数组（称临时数组），将临时数组作为操作数组以保持初始数组不变，使所有排序算法处理的均为同一个数组。每个排序算法都会对临时数组进行排序，并计算排序所需的时间和比较次数，排序结束输出相关结果后，释放临时数组内存空间。

**数组类型：**使用 `int[]` 类型的数组用于存储待排序的整数序列。每个排序函数都通过引用或指针对该数组进行修改。

**动态内存分配：**为了支持任意大小的数组，所有的数组内存通过 `new` 操作符动态分配，排序后及时释放内存。

### 2.2 模板设计

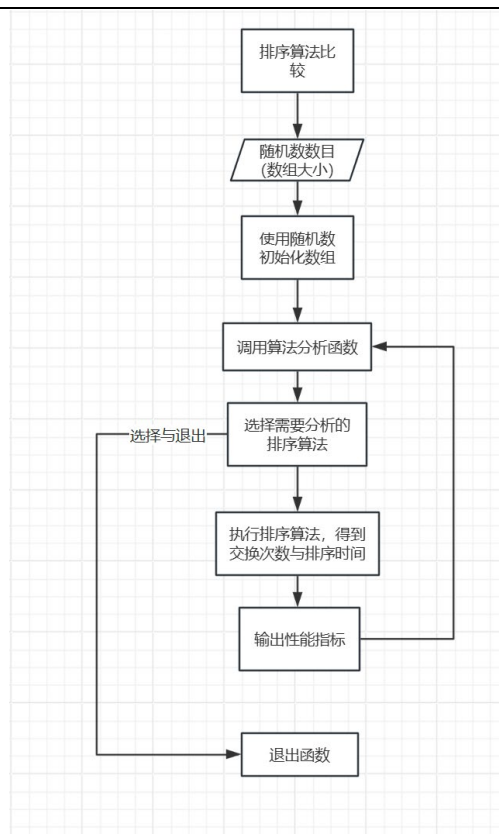
模板（Template）是 C++ 语言中的一项强大特性，允许函数或类在定义时不指定具体的数据类型，而是在使用时由编译器根据实际的参数类型自动推导或指定。模板使得代码的复用性大大提高，在本项目中，模板的使用主要体现在以下几个方面：排序算法的实现、数据交换、以及性能计数功能。

模板允许为不同数据类型编写一次排序算法代码，而不必重复为每种类型编写多份排序函数。通过模板的参数化，所有的排序算法都可以在同一函数体中处理不同类型的数据。同时，模板的设计使得我们可以轻松地添加新的排序算法，或者处理其他类型的数组，而不必添加新的函数实现。模板函数简化了代码的实现，避免了冗余的代码，并且减少了程序出错的可能性。

### 2.3 项目框架设计

#### 2.3.1 项目框架流程图





### 2.3.2 项目框架流程

1. 进入排序算法比较系统;
2. 输入生成随机数的个数, 根据随机数个数为数组分配内存空间;
3. 生成一个随机数生成器, 设定随机数生成范围, 依次为数组赋随机数;
4. 进入 while 循环, 调用分析函数, 选择需要分析的排序算法, 记录开始执行时间, 执行排序算法, 同时记录比较次数, 执行结束后, 记录结束时间;
5. 计算并输出排序算法执行时间、比较次数; 继续循环
5. 当用户选择退出后, 退出循环以及程序。

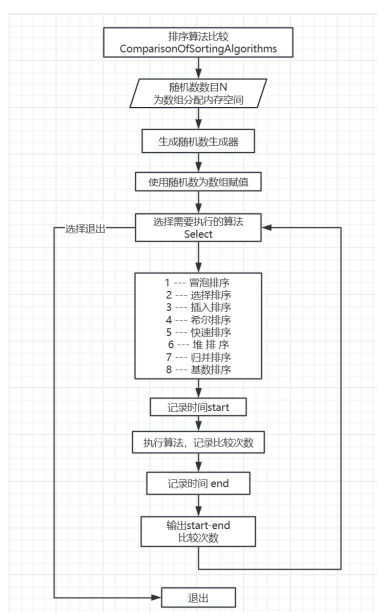
## 第3章 项目功能实现

### 3.1 项目主体架构与性能分析功能

#### 3.1.1 实现思路

1. 进入 ComparisonOfSortingAlgorithms 函数，输出用户交互信息。
2. 提示用户输入随机数个数，用户输入数据，根据用户输入的数据为数组 numbers 分配内存空间。
3. 使用 Mersenne Twister 生成伪随机数，并为数组 numbers 的元素赋值。
4. 输出选项提示信息，进入以 Evaluation(numbers, N, sort\_type\_name) 返回值作为循环条件的 while 循环，循环调用 Evaluation 函数进行条件判断。
5. 在 Evaluation 中，首先调用 selection 函数进行算法选择，若选择退出，则 Evaluation 返回 false，退出循环及程序；选择具体算法，初始化全局变量 compare\_count，为临时数组分配内存并赋值；
6. 根据算法选项，选择算法开始对临时数组排序；记录排序算法的开始时间与结束时间，同时在排序算法中计算比较次数；
7. 计算并输出排序算法执行时间及比较次数；返回 true，进行新一次循环。

#### 3.1.2 流程图



## 3.1.3 核心代码

```

void ComparisonOfSortingAlgorithms()
{
    std::cout << "+-----+\n";
    std::cout << "|          排序算法比较          |\n";
    std::cout << "| Comparison Of Sorting Algorithms  |\n";
    std::cout << "+-----+\n\n";
    double dN;// 用户输入的随机数个数 (double 用于初步验证合法性)
    while (true){
        std::cout << "请输入要产生随机数的个数: \n";
        std::cin>>dN;
        // 输入验证: 判断是否为有效的整数
        if (std::cin.fail() || dN<=0 || dN != static_cast<int>(dN)) {
            std::cout << "输入非法, 请重新输入! \n";
            Clear();
            continue;
        }
        Clear();
        break;
    }
    int N = static_cast<int>(dN);
    int* numbers=new(std::nothrow) int[N];
    assert(numbers!=nullptr);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(min, max);
    for(int i=0;i<N;i++)
        numbers[i]=dist(gen);
#ifdef MY_DEBUG
    Print(numbers,N,"生成的随机数列为: \n");
#endif
    std::cout << "+-----+\n";
    std::cout << "|    1 --- 冒泡排序    |\n";
    std::cout << "|    2 --- 选择排序    |\n";

```

```

std::cout << "|    3 --- 插入排序    |\n";
std::cout << "|    4 --- 希尔排序    |\n";
std::cout << "|    5 --- 快速排序    |\n";
std::cout << "|    6 --- 堆 排 序    |\n";
std::cout << "|    7 --- 归并排序    |\n";
std::cout << "|    8 --- 基数排序    |\n";
std::cout << "|    9 --- 退出程序    |\n";
std::cout << "+-----+\n";

char sort_type_name[8][15] = {"冒泡排序", "选择排序", "插入排序",
"希尔排序", "快速排序", "堆排序", "归并排序", "基数排序"};

while(Evaluation(numbers,N,sort_type_name))

    ;

delete[] numbers;

std::cout<<"成功推出算法比较系统! \n";

}

```

```

bool Evaluation(const int numbers[],const int N,const char sort_type_name[8][15])
{
    int selection = Select();
    if (selection == 9)
        return false;
    compare_count=0;
    int* nums= new(std::nothrow) int[N];
    assert(nums != nullptr);
    for(int i=0;i<N;i++)
        nums[i]=numbers[i];
    clock_t start_time = clock();
    if (selection == 1)
        BubbleSort(nums, N);
    else if (selection == 2)
        SelectionSort(nums, N);
    else if (selection == 3)
        InsertionSort(nums, N);
    else if (selection == 4)
        ShellSort(nums, N);
    else if (selection == 5)
        QuickSort(nums, N,0,N-1);
    else if (selection == 6)
        HeapSort(nums, N);
    else if (selection == 7)

```

```

MergeSort(nums, N, 0, N-1);
else
    RadixSort(nums, N);
clock_t end_time=clock();
std::cout<<"排序算法: "<<sort_type_name[selection-1]<<"\n";
std::cout<<"排序时间: "<<(end_time -
start_time)/static_cast<double>(CLOCKS_PER_SEC) <<"s\n";
std::cout<<"比较次数: " << compare_count << "\n";
#ifdef MY_DEBUG
    Print(nums, N, "排序后的数列为: ");
#endif
delete[] nums;
return true;
}

```

### 3.1.4 示例

```

          排序算法比较
        Comparison Of Sorting Algorithms

请输入要产生随机数的个数:
10000

1 --- 冒泡排序
2 --- 选择排序
3 --- 插入排序
4 --- 希尔排序
5 --- 快速排序
6 --- 堆排序
7 --- 归并排序
8 --- 基数排序
9 --- 退出程序

请输入您选择的算法选项: 1
排序算法: 冒泡排序
排序时间: 0.145s
比较次数: 49995000

请输入您选择的算法选项: 2
排序算法: 选择排序
排序时间: 0.086s
比较次数: 49995000

请输入您选择的算法选项: 3
排序算法: 插入排序
排序时间: 0.044s
比较次数: 25226111

请输入您选择的算法选项: 4
排序算法: 希尔排序
排序时间: 0.002s
比较次数: 257848

请输入您选择的算法选项: 5
排序算法: 快速排序
排序时间: 0.002s
比较次数: 161172

请输入您选择的算法选项: 6
排序算法: 堆排序
排序时间: 0.003s
比较次数: 387597

请输入您选择的算法选项: 7
排序算法: 归并排序
排序时间: 0.003s
比较次数: 133616

请输入您选择的算法选项: 8
排序算法: 基数排序
排序时间: 0.002s
比较次数: 0

请输入您选择的算法选项: 9
成功推出算法比较系统!

进程已结束, 退出代码为 0

```

## 3.2 排序算法实现

### 3.2.1 冒泡排序算法

#### 3.2.1.1 实现思路

1. 将数组分为已排序区和未排序区；开始时，所有元素在未排序区；
2. 从第一个元素开始，依次比较未排序区中某个元素及其后一个元素（没有后一个元素则不用比较），如果后一个元素大于前一个元素，则交换两个元素的值；
3. 重复以上步骤，直到到达未排序区末尾；实现将最大的数冒泡到最后；
4. 此时，未排序区最后一个元素是整个未排序区中元素最大的，将其划分给已排序区；
5. 重复以上步骤，直到未排序区元素为空；则此时的数组即为排好序的数组。

#### 3.2.1.2 核心代码

```
template<typename T>
void BubbleSort(T arr[],const int len)
{
    for(int i=0;i<len-1;i++)
        for(int j=0;j<len-i-1;j++) {
            compare_count++;
            if (arr[j] > arr[j + 1]) {
                Swap(arr[j],arr[j+1]);
            }
        }
}
```

#### 3.2.1.3 性能分析

##### 1. 时间复杂度：

最坏情况：如果数组是逆序的，冒泡排序需要进行  $n \times (n-1)/2$  次比较和交换，因此时间复杂度为  $O(n^2)$ 。

最好情况：如果数组已经是有序的，算法只需要进行一遍比较，时间复杂度为  $O(n)$ ，但是为了优化实现，通常需要检查每一轮是否进行了交换。

平均情况：时间复杂度也为  $O(n^2)$ ，因为交换次数通常与元素的数量平方成正比。

## 2.空间复杂度：

冒泡排序是一种原地排序算法，只需要常数级别的额外空间，因此空间复杂度为  $O(1)$ 。

## 3.2.2 选择排序算法

### 3.2.2.1 实现思路

1. 将第一个元素假设为最小元素，记为 `minIndex`；
2. 从第 `minIndex + 1` 位置开始，遍历剩下的所有元素，找到当前未排序部分的最小元素；
3. 如果找到的最小元素与当前假设的最小元素不同，就交换它们的位置。这样就保证了最小元素（或最大元素）在每次遍历结束后被放到了已排序部分的末尾；
4. 对数组中剩余的元素重复相同的操作，逐渐增加已排序部分的大小，直到所有元素排序完毕。

### 3.2.2.2 核心代码

```
template<typename T>
void SelectionSort(T arr[], const int len)
{
    for(int i=0;i<len-1;i++) {
        int minIndex = i;
        for(int j=i+1;j<len;j++) {
            compare_count++;
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        if(minIndex != i) {
            Swap(arr[minIndex],arr[i]);
        }
    }
}
```

### 3.2.2.3 性能分析

### 1.时间复杂度:

最坏情况：选择排序在第一次遍历中，需要对数组中的  $n-1$  个元素进行比较；在第二次遍历中，需要对  $n-2$  个元素进行比较；以此类推，最后一轮只需要比较 1 个元素，所以时间复杂度为  $O(n^2)$ 。

最优情况：当数组已经排好序时，选择排序仍然需要进行  $n-1$  次遍历来确认每个位置上的元素是否已经是最小的，因此最优时间复杂度依然是  $O(n^2)$ 。

平均情况：无论输入数组是如何排序的，选择排序的时间复杂度始终是  $O(n^2)$ 。

### 2.空间复杂度:

选择排序是一种原地排序算法，即排序过程中不需要额外的存储空间。它仅使用了一个临时变量。因此，选择排序的空间复杂度为  $O(1)$ 。

## 3.2.3 插入排序算法

### 3.2.3.1 实现思路

1. 假设第一个元素已排序，从第二个元素开始逐一插入到已排序部分；
2. 取出当前元素（称为插入元素），并与已排序部分的元素从后向前逐个比较；
3. 如果已排序部分的元素大于插入元素，则将已排序部分元素右移一位，直到找到插入位置，将插入元素放入插入位置；
4. 对数组中的每一个元素重复上述操作，直到整个数组排序完成。

### 3.2.3.2 核心代码

```
template<typename T>
void InsertionSort(T arr[], const int len)
{
    for(int i=1;i<len;i++) {
        T temp = arr[i];
        int j=i-1;
        while(j>=0&&arr[j]>temp) {
            compare_count++;
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}
```



```

    }
}

```

### 3.2.3.4 性能分析

#### 1. 时间复杂度:

最坏情况：在最坏的情况下（如数组是逆序的），每个元素都需要与之前所有的元素进行比较和交换。因此，对于每个元素，最多需要进行  $n-1$  次比较和交换，所以，最坏情况下的时间复杂度是  $O(n^2)$ 。

最优情况：如果数组已经是有序的，插入排序只需要进行一次比较而不需要移动元素。每次插入都可以直接插入到正确的位置。在这种情况下，时间复杂度为  $O(n)$ 。

平均情况：平均情况下，每个元素需要比较  $n/2$  次。总体的时间复杂度仍然是  $O(n^2)$ 。

#### 2. 空间复杂度:

插入排序是原地排序算法，它仅使用常量级的额外空间来存储插入元素和进行交换，因此它的空间复杂度是  $O(1)$ 。

## 3.2.4 希尔排序算法

### 3.2.4.1 实现思路

1. 选择一个增量序列，初始时将增量设为一个较大的数。常见的增量序列有： $n/2$ ,  $n/4$ , ..., 1。增量逐渐减少，直到增量为 1 时，完成整个排序过程。

2. 根据当前增量，将数组划分为若干个子序列。例如，当间隔为  $gap$  时，将数组的元素按索引  $i$  和  $i + gap$  分到同一个子序列。

3. 对每个子序列独立进行插入排序。

4. 减小增量，重新进行分组和插入排序，直到增量为 1。

5. 当增量为 1，所有元素都已经按正确顺序排列，排序完成。

### 3.2.4.2 核心代码

```

template<typename T>
void ShellSort(T arr[], const int len)
{
    for(int gap=len/2;gap>0;gap/=2) {
        for(int i=gap;i<len;i++) {
            T temp = arr[i];

```

```

        int j = i - gap;
        for(;j>=0;j-=gap) {
            compare_count++;
            if(arr[j]>temp) {
                arr[j + gap] = arr[j];
            }
            else
                break;
        }
        arr[j + gap] = temp;
    }
}

```

### 3.2.4.3 性能分析

#### 1.时间复杂度:

希尔排序的时间复杂度依赖于增量序列的选择,不同的增量序列会导致不同的性能表现。

最坏情况:对于最常见的增量序列  $gap = n/2, n/4, \dots, 1$ ,最坏情况下时间复杂度为  $O(n^2)$

最优情况:如果初始数组已经基本有序,增量序列使得每次插入排序只需要少量比较,最佳情况下时间复杂度为  $O(n \log n)$

平均情况:平均情况下,随着增量逐步减少,插入排序的效率逐渐提高。对于常见的增量序列,平均时间复杂度为  $O(n^3/2)$

#### 2.空间复杂度:属于原地排序,空间复杂度为 $O(1)$

## 3.2.5 快速排序算法

### 3.2.5.1 实现思路

1. 从待排序数组中选择一个元素作为基准(通常选择第一个、最后一个、中间或随机选取),用来将数组划分成两部分;
2. 对数组进行遍历,确保所有小于基准元素的元素都位于基准元素的左侧;所有大于基准元素的元素都位于基准元素的右侧;
3. 分区操作完成后,基准元素会被放置在最终位置上,即它的位置已经确定;

3. 对基准元素左侧和右侧的子数组递归地应用快速排序，直到数组变得有序。

### 3.2.5.2 核心代码

```
template<typename T>
int Partition(T arr[], int low, int high)
{
    T pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        compare_count++;
        if (arr[j] < pivot) {
            Swap(arr[j], arr[++i]);
        }
    }
    Swap(arr[i + 1], arr[high]);
    return i + 1;
}

template<typename T>
void QuickSort(T arr[], const int len, int low, int high)
{
    if (low < high) {
        int pi = Partition(arr, low, high);
        QuickSort(arr, len, low, pi - 1);
        QuickSort(arr, len, pi + 1, high);
    }
}
```

### 3.2.5.3 性能分析

#### 1. 时间复杂度:

最坏情况：如果每次选择的基准都极端（例如每次都选择数组的最大或最小元素），则数组在每次递归时仅分成一个元素和剩余部分，导致递归深度为  $n$ ，时间复杂度为  $O(n^2)$ 。

最优情况：当每次选择的基准都能将数组平分成两半（即数组已经或接近有序时），时间复杂度为  $O(n \log n)$ 。

平均情况：在平均情况下，每次分区将数组大致分为两个相等的部分，因此时间复杂度是  $O(n \log n)$ 。

2.空间复杂度：快速排序是一种原地排序算法，主要消耗的空间是递归调用栈的空间。每次分区操作都在原数组上进行，因此空间复杂度是  $O(\log n)$ ，即递归的深度。

### 3.2.6 堆排序算法

#### 3.2.6.1 实现思路

1. 对于堆中的每一个元素，依次将它与左右子节点比较，如果它比子节点小，就交换它们的位置，直到满足最大堆性质；
2. 将堆顶的最大元素交换到数组的末尾，然后对剩余的部分重新进行堆化，重新调整为最大堆；
3. 重复步骤 2，直到所有元素排序完成；

#### 3.2.6.2 核心代码

```
template<typename T>
void Heapify(T arr[], const int len, int i)
{
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;
    compare_count++;
    if (left < len && arr[left] > arr[largest])
        largest = left;
    compare_count++;
    if (right < len && arr[right] > arr[largest])
        largest = right;
    compare_count++;
    if (largest != i) {
        Swap(arr[i], arr[largest]);
        Heapify(arr, len, largest);
    }
}

template<typename T>
void HeapSort(T arr[], const int len)
{

```

```

    for (int i = len / 2 - 1; i >= 0; i--)
        Heapify(arr, len, i);
    for (int i = len - 1; i > 0; i--) {
        Swap(arr[0], arr[i]);
        Heapify(arr, i, 0);
    }
}

```

### 3.2.6.3 性能分析

#### 1. 时间复杂度:

构建最大堆的时间复杂度是  $O(n)$ 、堆化操作的时间复杂度为  $O(\log n)$ ，总时间复杂度为  $O(n \log n)$ 。

#### 2. 空间复杂度:

堆排序是原地排序算法，它的空间复杂度是  $O(1)$ ，只需要常数级的额外空间来存储临时变量。

## 3.2.7 归并排序算法

### 3.2.7.1 实现思路

1. 将原数组递归地分成两部分，直到每部分只有一个元素；
2. 从两个有序数组的首部开始，逐一比较元素，将较小的元素加入到新的数组中，直到所有元素都被处理完。

### 3.2.7.2 核心代码

```

template<typename T>
void Merge(T arr[], int left, int mid, int right)
{
    int len1 = mid - left + 1, len2 = right - mid, i = 0, j = 0, k = left;
    T *left_arr = new(std::nothrow) T[len1];
    assert(left_arr != nullptr);
    T *right_arr = new(std::nothrow) T[len2];
    assert(right_arr != nullptr);
    for (int i = 0; i < len1; i++) {
        left_arr[i] = arr[left + i];
    }
    for (int i = 0; i < len2; i++) {

```

```

        right_arr[i] = arr[mid + 1 + i];
    }
    while (i < len1 && j < len2) {
        compare_count++;
        if (left_arr[i] <= right_arr[j]) {
            arr[k++] = left_arr[i++];
        }
        else {
            arr[k++] = right_arr[j++];
        }
    }
    while (i < len1) {
        compare_count++;
        arr[k++] = left_arr[i++];
    }
    while (j < len2) {
        compare_count++;
        arr[k++] = right_arr[j++];
    }
    delete[] left_arr;
    delete[] right_arr;
}

template<typename T>
void MergeSort(T arr[], const int len, int left, int right)
{
    if(left < right) {
        int mid = left + (right - left) / 2;
        MergeSort(arr, len, left, mid);
        MergeSort(arr, len, mid + 1, right);
        Merge(arr, left, mid, right);
    }
}

```

### 3.2.7.3 性能分析

#### 1.时间复杂度:

分割过程的递归深度为  $\log n$ ，每一层递归的合并操作总共涉及  $n$  个元素，合并的时间复杂度是  $O(n)$ ，总时间复杂度为  $O(n \log n)$ 。

## 2. 空间复杂度：

归并排序需要额外的空间来存储合并过程中的临时数组。对于一个包含  $n$  个元素的数组，合并时需要额外的  $O(n)$  空间来存储合并后的结果。因此，归并排序的空间复杂度为  $O(n)$ 。

## 3.2.8 基数排序算法

### 3.2.8.1 实现思路

1. 找到数组中的最大值，确定排序的位数，即最大值的位数；
2. 从最低位开始，对每一位使用稳定的排序算法（如计数排序）进行排序；
3. 重复步骤 2 直到所有位都排序完成。

### 3.2.8.2 核心代码

```
template<typename T>
void SortByDigit(T arr[], const int len, const int digit_place)
{
    int frequency[10] = {0};
    for(int i=0; i<len; i++)
        ++frequency[(arr[i] / digit_place) % 10];
    for(int i=1; i<10; i++)
        frequency[i] += frequency[i-1];
    T* output = new(std::nothrow) T[len];
    assert(output != nullptr);
    for(int i=len-1; i>=0; i--) {
        output[frequency[(arr[i] / digit_place) % 10] - 1] = arr[i];
        --frequency[(arr[i] / digit_place) % 10];
    }
    for(int i=0; i<len; i++)
        arr[i] = output[i];
    delete[] output;
}
```

```

template<typename T>
void RadixSort(T arr[], const int len)
{
    T max_element=arr[0];
    for(int i = 0; i < len; i++) {
        max_element = max_element > arr[i] ? max_element : arr[i];
    }
    for(int digit_place = 1; max_element/digit_place>0 ; digit_place*=10) {
        SortByDigit(arr, len,digit_place);
    }
}

```

### 3.2.8.3 性能分析

#### 1.时间复杂度:

由于每一位的排序都使用稳定的排序算法（如计数排序），每次排序的时间复杂度是  $O(n)$ ，重复  $d$  次（ $d$  为最大元素的位数），总时间复杂度为  $O(n * d)$ 。

#### 2.空间复杂度:

$O(n + k)$ ，其中  $n$  是元素个数， $k$  是计数排序中的计数数组的大小。

## 3.3 异常处理功能

### 3.9.1 输入非法的异常处理

#### 3.9.1.1 随机数个数输入非法的异常处理

在建立进行算法分析之前，首先需要输入生成随机数的个数，通过以下逻辑判断输入是否正确：

1. 定义一个浮点型变量，进入一个 while 循环；
2. 输出提示信息，用户输入数据；
3. 判断以下错误是否存在：cin.fail() 为 true，即用户输入了字符或字符串或数据超过了 double 上下限；输入的数据小于等于 0；将数据转换为 int 后与之前不相等（原数为小数或超 int 上限）；
4. 若以上错误均不存在，说明输入数据无误，退出循环；反之，输出错误信息，清除当前输入状态和缓冲区，重新输入。



### 3.9.1.2 操作类型输入非法的异常处理

操作类型输入非法的异常处理通过如下代码实现

```
int Select()
{
    double selection;
    while (true) {
        std::cout<<"\n 请输入您选择的算法选项： ";
        std::cin >> selection;
        if (std::cin.fail() || selection > 9 || selection <= 0 || selection !=
static_cast<int>(selection)) {
            std::cout << "输入非法，请重新输入！ \n";
            Clear();
            continue;
        }
        Clear();
        break;
    }
    return static_cast<int>(selection);
}
```

这段代码的具体执行逻辑如下：

1. 定义一个浮点型变量，进入一个 while 循环；
2. 输出提示信息，用户输入数据；
3. 判断以下错误是否存在：cin.fail() 为 true，即用户输入了字符或字符串或数据超过了 double 上下限；输入的数据小于等于 0 或大于 9；将数据转换为 int 后与之前不相等（原数为小数）；
4. 若以上错误均不存在，说明输入数据无误，退出循环，返回整数操作类型；反之，输出错误信息，清除当前输入状态和缓冲区，重新输入。

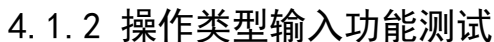
### 3.9.2 动态内存申请失败的异常处理

在为数组分配动态内存时，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），使用 assert 断言判断指针是否为空指针，如果为空指针，意味着内存分配失败，抛出异常信息。

## 4.1 输入功能测试

分别输入超过 int 上下限的整数、非正整数、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



22

```
请输入您选择的算法选项: 999999999999999999
输入非法, 请重新输入!

请输入您选择的算法选项: -999999999999999999
输入非法, 请重新输入!

请输入您选择的算法选项: 0
输入非法, 请重新输入!

请输入您选择的算法选项: 10
输入非法, 请重新输入!

请输入您选择的算法选项: a
输入非法, 请重新输入!

请输入您选择的算法选项: asad
输入非法, 请重新输入!

请输入您选择的算法选项: 1
排序算法: 冒泡排序
排序时间: 0.174s
比较次数: 49995000
```

4.2 排序算法正确性验证

使用预处理指令处理数组打印函数，当宏定义 MY\_DEBUGU 时，启用条件编译。

输入要生成的随机数的个数，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。

```
排序算法比较
Comparison Of Sorting Algorithms

请输入要产生随机数的个数:
10
生成的随机数列为:
375062074 922395229 1950015376 696577787 209157708 86099814 1327686072 867383687 516242104 810341627

1 --- 冒泡排序
2 --- 选择排序
3 --- 插入排序
4 --- 快速排序
5 --- 堆排序
6 --- 归并排序
7 --- 希尔排序
8 --- 计数排序
9 --- 桶排序

请输入您选择的算法选项: 1
排序算法: 冒泡排序
排序时间: 0s
比较次数: 45
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 2
排序算法: 选择排序
排序时间: 0s
比较次数: 45
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 3
排序算法: 插入排序
排序时间: 0s
比较次数: 24
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 4
排序算法: 快速排序
排序时间: 0s
比较次数: 29
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 5
排序算法: 堆排序
排序时间: 0s
比较次数: 20
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 6
排序算法: 归并排序
排序时间: 0s
比较次数: 99
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 7
排序算法: 希尔排序
排序时间: 0s
比较次数: 34
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项: 8
排序算法: 计数排序
排序时间: 0s
比较次数: 0
排序后的数列为:
86099814 209157708 375062074 516242104 696577787 810341627 867383687 922395229 1327686072 1950015376

请输入您选择的算法选项:
```

## 第 5 章 相关说明

### 5.1 编程语言

本项目全部 .cpp 文件以及 .h 文件均使用 C++ 编译完成, 使用 UTF-8 编码。

### 5.2 Windows 环境

Windows 系统: Windows 11 x64

Windows 集成开发环境: CLion 2024

工具集: MinGW 11.0 w64

### 5.3 Linux 环境

基于 Linux 内核的操作系统发行版: Ubuntu 24.04.1

Linux 命令编译过程为:

1. 定位包含项目所在文件夹, 包括 .pp 与 .h 文件; 具体命令为: `cd /home/bruce/programe/comparison_of_sorting_algorithms`

2. 编译项目, 生成可执行文件; 具体命令为: `g++ -static -o comparison_of_sorting_algorithms_linux comparison_of_sorting_algorithms.cpp`

其中指令含义分别为:

`g++`: 调用 GNU 的 C++ 编译器

`-static`: 使用静态链接而非动态链接, 将所有依赖库直接嵌入到可执行文件, 文件存储空间变大, 但可以单独运行

`-o comparison_of_sorting_algorithms_linux`: `-o` 表示输出文件选项, `comparison_of_sorting_algorithms_linux` 为可执行文件名

`comparison_of_sorting_algorithms.cpp`: 编译所需要的文件

3. 运行可执行文件; 具体命令为 `./comparison_of_sorting_algorithms_linux`

```
bruce@Jarvis: ~/progame/comparison_of_sorting_algorithms
bruce@Jarvis:~/progame/comparison_of_sorting_algorithms$ g++ -static -o comparison_of_sorting_algorithms_linux comparison_of_sorting_algorithms.cpp
bruce@Jarvis:~/progame/comparison_of_sorting_algorithms$ ./comparison_of_sorting_algorithms_linux
+-----+
|           排序算法比较           |
|   Comparison Of Sorting Algorithms   |
+-----+

请输入要产生随机数的个数:

```