**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： 张梓瀚

学 号： 2051943

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1项目分析 - 4 -](#_Toc121434726)

[1.1项目背景 - 4 -](#_Toc121434727)

[1.2 项目需求分析 - 4 -](#_Toc121434728)

[1.3 项目要求 - 4 -](#_Toc121434729)

[1.3.1 功能要求 - 4 -](#_Toc121434730)

[1.3.2 输入格式 - 4 -](#_Toc121434731)

[1.3.3 输出格式 - 4 -](#_Toc121434732)

[1.3.4 项目示例 - 5 -](#_Toc121434733)

[2 项目实现 - 5 -](#_Toc121434734)

[2.1 冒泡排序 - 6 -](#_Toc121434735)

[2.1.1 算法逻辑 - 6 -](#_Toc121434736)

[2.1.2 算法代码 - 6 -](#_Toc121434737)

[2.1.3 算法分析 - 6 -](#_Toc121434738)

[2.1.4 优化方法 - 7 -](#_Toc121434739)

[2.2 选择排序 - 7 -](#_Toc121434740)

[2.2.1 算法逻辑 - 7 -](#_Toc121434741)

[2.2.2 算法代码 - 7 -](#_Toc121434742)

[2.2.3 算法分析 - 8 -](#_Toc121434743)

[2.2.1 优化方法 - 8 -](#_Toc121434744)

[2.3 直接插入排序 - 8 -](#_Toc121434745)

[2.3.1 算法逻辑 - 8 -](#_Toc121434746)

[2.3.2 算法代码 - 8 -](#_Toc121434747)

[2.3.3 算法分析 - 9 -](#_Toc121434748)

[2.3.4 优化方法 - 9 -](#_Toc121434749)

[2.4 希尔排序 - 9 -](#_Toc121434750)

[2.4.1 算法逻辑 - 9 -](#_Toc121434751)

[2.4.2 算法代码 - 9 -](#_Toc121434752)

[2.4.3 算法分析 - 10 -](#_Toc121434753)

[2.4.4 优化方法 - 10 -](#_Toc121434754)

[2.5 快速排序 - 10 -](#_Toc121434755)

[2.5.1 算法逻辑 - 10 -](#_Toc121434756)

[2.5.2 算法代码 - 10 -](#_Toc121434757)

[2.5.3 算法分析 - 11 -](#_Toc121434758)

[2.5.4 优化方法 - 11 -](#_Toc121434759)

[2.6 堆排序 - 11 -](#_Toc121434760)

[2.6.1 算法逻辑 - 11 -](#_Toc121434761)

[2.6.2 算法代码 - 12 -](#_Toc121434762)

[2.6.3 算法分析 - 15 -](#_Toc121434763)

[2.7 归并排序 - 15 -](#_Toc121434764)

[2.7.1 算法逻辑 - 15 -](#_Toc121434765)

[2.7.2 算法代码 - 15 -](#_Toc121434766)

[2.7.3 算法分析 - 16 -](#_Toc121434767)

[2.7.4 优化方法 - 16 -](#_Toc121434768)

[2.8 基数排序 - 17 -](#_Toc121434769)

[2.8.1 算法逻辑 - 17 -](#_Toc121434770)

[2.8.2 算法代码 - 17 -](#_Toc121434771)

[2.8.3 算法分析 - 18 -](#_Toc121434772)

[3 总体程序流程图 - 19 -](#_Toc121434773)

[4 项目测试 - 20 -](#_Toc121434774)

[4.1 100个随机数测试 - 20 -](#_Toc121434775)

[4.2 1000个随机数测试 - 21 -](#_Toc121434776)

[4.3 10000个随机数测试 - 22 -](#_Toc121434777)

[4.4 100000个随机数测试 - 23 -](#_Toc121434778)

[4.5 错误输入测试 - 24 -](#_Toc121434779)

# 1项目分析

## 1.1项目背景

排序问题是研究使得若干个同类数据按照一定顺序排列的问题。排序算法就是使得数据按照要求排列的方法。排序算法在很多领域得到相当地重视，尤其是在大量数据的处理方面。一个优秀的排序算法可以节省大量的时间和空间。排序算法在各类数据处理和分析问题中有着广泛的应用，根据具体数据的分布、规模和其他实际要求选择一个合适的排序算法，可以大大提高程序的性能和实用性。

本项目通过对8种排序算法性能的探究，比较不同排序算法的优劣，为实际应用提供参考。

## 1.2 项目需求分析

在本案例中，我们选择了冒泡排序，选择排序，直接插入排序，希尔排序，快速排序，堆排序，归并排序，基数排序八种排序算法来进行分析。

随机函数产生一百，一千，一万……随机数，用各种排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的比较次数。同时，分析每一种排序算法是否具有稳定性，即在排序的过程中会不会改变元素彼此的位置的相对次序。在文档中记录每个数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

## 1.3 项目要求

### 1.3.1 功能要求

随机函数产生一百，一千，一万和十万个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的比较次数。

请在文档中记录上述数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

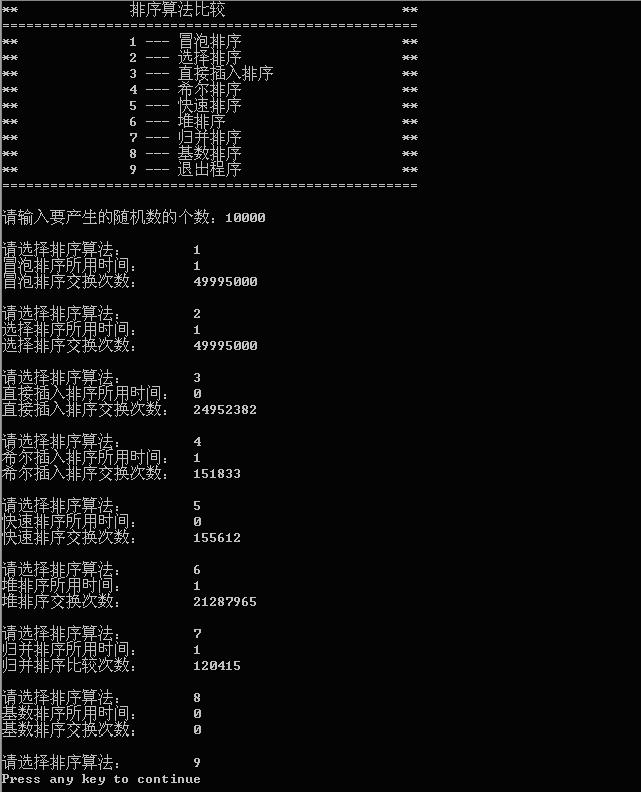
### 1.3.2 输入格式

生成随机数的个数。

### 1.3.3 输出格式

打印随机序列、升序序列、降序序列下各排序算法的排序时间和交换次数等。

### 1.3.4 项目示例



# 2 项目实现

本项目共实现了以下8种排序算法：

* 冒泡排序
* 选择排序
* 直接插入排序
* 希尔排序
* 堆排序
* 快速排序
* 归并排序
* MSD基数排序

下面将依次对本项目实现的这8种排序算法进行介绍。

# 2.1 冒泡排序

### 2.1.1 算法逻辑

冒泡排序是一种严格的稳定排序算法，它不改变序列中相同元素之间的相对位置关系。有一个待排序的元素序列中的元素个数为n，第一次的排序从第0个元素开始，首先比较第0个元素与第1个元素，如果发生逆序（即前一个大于后一个），则将这两个元素交换；然后再按此规律比较第1个元素与第2个元素，第2个元素与第3个元素……直到比较完第n-2个元素与第n-1个元素。一次排序会将序列中的最大的元素交换到序列的最后。然后再进行一次排序，仍然是从第0个元素开始，不过到n-2个元素就结束，这会使得序列中第二大的元素交换到序列的倒数第二个位置……进行n-1次排序就可以将所有元素排好序。

### 2.1.2 算法代码

//冒泡排序

template<typename T>

long long bubble\_sort(T arr[], int l, int r)

{

long long cnt = 0;

for (int i = l; i < r; i++)

{

bool exchange = false;

for (int j = r - 1; j >= i; j--)

{

if (arr[j - 1] > arr[j])

{

my\_swap(arr[j - 1], arr[j]);

cnt++;

exchange = true;

}

}

if (exchange == false)

return cnt;

}

return cnt;

}

### 2.1.3 算法分析

冒泡排序的时间复杂度为O(n^2)，空间复杂度为O(1)。

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再交换的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

### 2.1.4 优化方法

不一定要对每个序列都进行n-1次排序才能排好序。可以在冒泡排序中增加一个判断标志exchange，如果在一次排列中发生了元素交换，exchange就为true。如果执行完一次排序后exchange为false，就说明没有发生元素交换，此时的元素已经排好序，不需要再进行之后的排序。

## 2.2 选择排序

### 2.2.1 算法逻辑

选择排序算法的基本思路是为每一个位置选择当前最小的元素。传统的选择排序过程即先找到整个待排序序列中最小的一个元素，如果它不是序列中的第一个元素，就把它与第一个元素互换位置。然后在剩下的元素中再寻找最小的元素，将它放在第二个元素的位置……重新进行，直到整个序列排好序。

### 2.2.2 算法代码

//选择排序

template<typename T>

long long select\_sort(T arr[], int l, int r) {

static long long cnt = 0;

for (int i = l; i < r; ++i)

{

int min = i;

for (int j = i + 1; j < r; ++j)

{

if (arr[j] < arr[min])

min = j;

++cnt;

}

if (min != i)

{

my\_swap(arr[i], arr[min]);

++cnt;

}

}

return cnt;

}

### 2.2.3 算法分析

选择排序的时间复杂度为O(n^2)，空间复杂度为O(1)。

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n-1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了。因此，选择排序是一个不稳定的排序算法。

### 2.2.1 优化方法

排序开始之前，可以遍历序列，同时找出其中的最小元素和最大元素，将最小元素放在起始位置，最大元素放在末尾位置，这样可以减少整体排序次数，改进后对n个数据进行排序，最多只需进行n/2趟排序即可。在该优化方法中要特别注意如果最大值就在起始位置，那么将最小元素交换完后，最大元素的位置跑到了之前最小元素所在的位置，要将此处的元素与末尾位置进行交换。

## 2.3 直接插入排序

### 2.3.1 算法逻辑

直接插入排序通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入，如此重复，直至完成序列排序。从第1个元素开始，将第1个元素取出，找到前面该插入的位置（即插入位置的前一个元素比其小，后一个元素比其大）并插入，随后再取出第2个元素，按照此方式插入……以此类推，这样的话，当要插入第i个元素时，前面的第0，1，2……i-1个元素已经排好序，只需要将第i个元素插入，原来插入位置上及之后的元素向后顺延即可，保证插入后第0，1，2……i-1，i个元素仍保持有序。将所有的元素都插入一遍，排序就完成了。注意在排序过程中，如果之前的元素中有与待插入元素相同的元素，就要把待插入元素插入到该元素后面（保持稳定性）。

### 2.3.2 算法代码

//直接插入排序

template<typename T>

long long insert\_sort(T arr[], int l, int r) {

long long cnt = 0;

for (int i = l + 1; i < r; i++) {

T key = arr[i];

int j = i - 1;

while (j >= 0 && key < arr[j]) {

arr[j + 1] = arr[j];

j--;

cnt++;

}

arr[j + 1] = key;

cnt++;

}

return cnt;

}

### 2.3.3 算法分析

直接插入排序的时间复杂度为O(n^2)，空间复杂度为O(1)。

在排序过程中，如有与待插入元素大小相等的元素，则待插入元素与这些元素的前后位置关系不会改变，因此直接插入排序是稳定的算法。

### 2.3.4 优化方法

本题中要求为直接插入排序，可以改用折半插入的方法作优化。即在寻找一个元素的插入位置时，使用折半查找的方法，每次与当前范围的中间值作比较，如果大于的话将范围缩小到后一半，如果小于的话将范围缩小到前一半，直至找到插入位置。这样优化的话可以减少排序次数，但是元素的移动次数不会改变。

## 2.4 希尔排序

### 2.4.1 算法逻辑

希尔排序实质上是一种分组插入方法。它的基本思想是：对于n个待排序的数列，首先取一个整数gap<n作为间隔，将全部元素分为gap个子序列，所有距离为gap的元素放置在同一个子序列中，在每一个子序列中分别进行直接插入排序。然后缩小间隔gap，重复上述的子序列划分与排序工作。直到最后将所有元素放在同一个序列中排序为止。

### 2.4.2 算法代码

//希尔排序

template<typename T>

long long shell\_sort(T arr[], int l, int r) {

long long cnt = 0;

for (int gap = (r - l + 1) / 2; gap > 0; gap /= 2) {

for (int i = gap; i < r; i++) {

T tmp = arr[i];

int j;

for (j = i; j >= gap && tmp < arr[j - gap]; j -= gap)

arr[j] = arr[j - gap];

arr[j] = tmp, cnt++;

}

}

return cnt;

}

### 2.4.3 算法分析

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。应用不同的gap序列会使希尔排序的性能存在有很大差异，本系统中选择的是折半序列，在n处于一定范围时，其平均时间复杂度为O(n^1.3)，最坏时间复杂度为O(n^2）。由于存在跨位置之间的交换，所以希尔排序是不稳定的排序算法。

### 2.4.4 优化方法

希尔排序最主要的优化方法是使用不同的gap序列，最初Shell提出每次将gap除以2的序列，这样的话希尔排序的最坏时间复杂度可以达到O(n^2)。之后的人们都提出了不同的增量序列，例如Hibbard序列，其平均时间复杂度为O(n^(5/4))，最坏时间复杂度为O(n^(3/2)）。在不同序列中，Sedgewick序列是已知比较好的增量序列，其平均时间复杂度为O(n^(7/6))，最坏时间复杂度为O(n^(4/3)）。

## 2.5 快速排序

### 2.5.1 算法逻辑

快速排序的基本思想是：选择一个基准元素，根据元素与基准元素的大小关系，通过一趟排序算法把所需要排序的序列的元素分割成两大块，其中，一部分的元素都要小于或等于另外一部分的序列元素，然后仍根据该种方法对划分后的这两块序列的元素分别再次实行快速排序算法，直至所有元素排列完成为止。

### 2.5.2 算法代码

//快速排序

template<typename T>

long long quick\_sort(T arr[], int l, int r)

{

long long cnt = 0;

if (l >= r) return 0;

int i = l - 1, j = r + 1;

T x = arr[(l + r) >> 1];

while (i < j)

{

do

{

i++;

} while (arr[i] < x);

do

{

j--;

} while (arr[j] > x);

if (i < j) my\_swap(arr[i], arr[j]), cnt++;

}

return cnt + quick\_sort(arr, l, j) + quick\_sort(arr, j + 1, r);

}

### 2.5.3 算法分析

快速排序的平均时间复杂度是O(nlog2n)。因此，该排序方法被认为是目前最好的内部排序方法之一。

从空间性能上看，尽管快速排序只需要一个元素的辅助空间，但快速排序需要一个栈空间来实现递归。最好的情况下，即快速排序的每一趟排序都将元素序列均匀地分割成长度相近的两个子表，所需栈的最大深度为log2(n+1)；但最坏的情况下，栈的最大深度为n。这样，快速排序的空间复杂度为O(log2n)。

同时，快速排序是一个不稳定的排序算法。

### 2.5.4 优化方法

在快速排序算法中，每次划分时用于比较的基准元素的选择对于算法的性能有很大影响。如果基准元素选择不合适，会导致算法性能的大幅度退化。如果要可以合理选择基准元素，使得每次划分所得的两个子序列中的元素个数尽可能地接近，就可以加速排序速度。因此，我们可以使用三者取中的算法对快速排序进行优化。

所谓三者取中，就是在取基准元素时，将序列左端点left，序列右端点right与序列中间位置（left+right）/2相互进行比较，然后取中间值作为基准元素，对整个序列进行划分。

还有其他的方法，比如在划分过程中对小规模子序列不进行排序而跳过，这样在划分之后得到的是一个整体上几乎已经排好序的元素序列，随后在进行一遍直接插入排序将序列排好。有研究表明，三者取中和小规模序列的中止两种措施结合起来，可以将用递归实现的快速排序算法效率提高20%—25%。

## 2.6 堆排序

### 2.6.1 算法逻辑

堆排序是指利用堆（Heap）这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆是一类特殊数据结构的简称，通常是一个可以被看做一棵树的的数组对象。堆满足下述两个性质：

1. 堆中某个节点的值总是不大于或者不小于其父节点的值；
2. 堆总是一棵完全二叉树。

将根节点最大的堆叫做最大堆或者大根堆，根节点最小的堆叫做最小堆小根堆。常见的堆有二叉堆、斐波拉契堆等。

### 2.6.2 算法代码

小顶堆

//优先队列（小顶堆）类

template<typename T>

class priQue

{

private:

//数据

T\* pArray;

//堆大小

int m\_length;

//结点下沉

void swim(int k, long long& cnt);

//结点上浮

void sink(int k, long long& cnt);

public:

//默认构造函数

priQue();

//指定堆大小的构造函数

priQue(int N);

//取堆顶

T top();

//弹出堆顶

void pop(long long& cnt);

//将元素入堆

void push(const T& v, long long& cnt);

//判断堆是否为空

bool empty();

//获取堆大小

int size();

};

template <typename T>

void priQue<T>::swim(int k, long long& cnt)

{

//若k结点数据比它的父结点小，则一直上浮

while (k > 1 && pArray[k] < pArray[k / 2]) {

swap(pArray[k / 2], pArray[k]);

cnt++;

k /= 2;

}

}

template <typename T>

void priQue<T>::sink(int k, long long&cnt)

{

while (2 \* k <= m\_length)

{

int j = 2 \* k;

//比较k的左右儿子哪一个较小

if (j < m\_length && (pArray[j] > pArray[j + 1])) j++;

//如果已经下沉到正确位置就退出循环

if (pArray[k] < pArray[j]) break;

//k结点下沉到j位置

my\_swap(pArray[k], pArray[j]);

k = j;

cnt++;

}

}

template <typename T>

priQue<T>::priQue()

{

pArray = new T[DEFAULT\_SIZE + 1];

m\_length = 0;

}

template <typename T>

priQue<T>::priQue(int N)

{

pArray = new T[N + 1];

m\_length = 0;

}

template <typename T>

T priQue<T>::top()

{

return pArray[1];

}

template <typename T>

void priQue<T>::pop(long long& cnt)

{

//将堆顶与末尾元素交换，再减小堆大小，达到弹顶的效果

swap(pArray[1], pArray[m\_length--]);

//堆顶下沉，维护小顶堆性质

sink(1, cnt);

}

template <typename T>

void priQue<T>::push(const T& v, long long& cnt)

{

//在末尾加入新元素

pArray[++m\_length] = v;

//该叶子结点上浮，维护小顶堆性质

swim(m\_length, cnt);

}

template <typename T>

bool priQue<T>::empty()

{

return m\_length == 0;

}

template <typename T>

int priQue<T>::size()

{

return m\_length;

}

堆排序

//堆排序

template<typename T>

long long heap\_sort(T arr[], int l, int r)

{

long long cnt = 0;

priQue<T>heap(r - l + 1);

for (int i = l; i <= r; i++)

heap.push(arr[i], cnt);

for (int i = l; i <= r; i++)

heap.pop(cnt);

return cnt;

}

### 2.6.3 算法分析

堆排序是一种选择排序，整体主要由构建初始堆+交换堆顶元素和末尾元素并重建堆两部分组成。其中构建初始堆经推导复杂度为O(n)，在交换并重建堆的过程中，需交换n-1次，而重建堆的过程中，根据完全二叉树的性质，[log2(n-1),log2(n-2)...1]逐步递减，近似为nlogn。所以堆排序时间复杂度一般认为就是O(nlogn)级。

堆排序是一种不稳定的排序算法。

## 2.7 归并排序

### 2.7.1 算法逻辑

归并排序算法就是把序列递归划分成为一个个短序列，以其中只有1个元素的直接序列或者只有2个元素的序列作为短序列的递归出口，再将全部有序的短序列按照一定的规则进行排序为长序列。归并排序融合了分治策略，即将含有n个记录的初始序列中的每个记录均视为长度为1的子序列，再将这n个子序列两两合并得到n/2个长度为2(当凡为奇数时会出现长度为l的情况)的有序子序列；将上述步骤重复操作，直至得到1个长度为n的有序长序列。需要注意的是，在进行元素比较和交换时，若两个元素大小相等则不必刻意交换位置，因此该算法不会破坏序列的稳定性，即归并排序也是稳定的排序算法。

### 2.7.2 算法代码

//部分归并，arr为原数组，tmp为临时数组

long long merge(int\* arr, int\* tmp, int l, int mid, int r)

{

for (int k = l; k <= r; k++)

tmp[k] = arr[k];

int i = l, j = mid;

for (int k = l; k <= r; k++)

{

if (i >= mid)

arr[k] = tmp[j++];

else if (j >= r)

arr[k] = tmp[i++];

else

{

if (tmp[j] < tmp[i])

arr[k] = tmp[j++];

else

arr[k] = tmp[i++];

}

}

return r - l + 1;

}

//归并排序

template<typename T>

long long merge\_sort(T arr[], int l, int r)

{

if (r - l <= 1)

return 0;

int mid = (l + r) / 2;

int\* tmp = new int[r + 1];

long long cnt = merge\_sort(arr, l, mid) + merge\_sort(arr, mid, r) + merge(arr, tmp, l, mid, r);

delete tmp;

tmp = nullptr;

return cnt;

}

### 2.7.3 算法分析

无论元素的大小关系如何，归并排序的操作都是固定的，所以花销的时间是不变的，所以该算法的最优时间复杂度、最差时间复杂度及平均时间复杂度都为O(nlogn)。

归并的空间复杂度就是临时数组和递归时压入栈的数据占用的空间：n + logn，所以空间复杂度为: O(n)。

归并排序算法中，归并最后到底都是相邻元素之间的比较交换，相同元素的相对位置并不会发生变化，故是稳定算法。

### 2.7.4 优化方法

归并排序有数种优化方法。

首先可以改进两路归并算法，把元素序列复制给辅助数组时，把第二个有序表的元素顺序翻转，这样的话两个待归并的表就会从两端开始处理，向中间归并。这样的话两个待归并的表的尾端就可以互成“监视哨”，就可以避免检查两个表是否已经复制结束的判断，提高程序效率。

其次，我们可以在合并的时候检查一下左边子序列的尾端是否小于右边子序列的首端，如果小于的话就说明此时的序列左右半边已经是有序序列，不需要再进行合并操作，直接复制即可。从而省去一定的时间。

最后，我们在每次执行归并的时候，都需要将待归并序列复制到辅助数组上，再从辅助数组归并到序列里。但我们可以把这个复制的过程给去除掉。方法就是要在递归调用的每个层次交换输入数组和输出数组的角色，从而不断地把输入数组排序到辅助数组，再将数据从辅助数组排序到输入数组。我们在进入递归前就将初始序列复制到一个辅助数组中，然后将它们两个一起传入递归，在递归调用的每个层次交换输入数组和输出数组的角色。

例如：5 6 4 7 1 3 8 2

我们在最后一层递归中利用辅助数组进行合并，结果进入原数组，为5 6 4 7 1 3 2 8，随后进入上一层，此时刚刚的辅助数组是这一层的原数组，刚刚的原数组是这一层的辅助数组，所以在这一层中是利用的辅助数组进行合并，原数组中结果为4 5 6 7 1 2 3 8……不断向前回升，最后就可以得到有序序列。

除此之外，对小规模子数组使用插入排序等方法也可以使归并排序的运行时间缩短。

## 2.8 基数排序

### 2.8.1 算法逻辑

待排序元素序列有n个元素，首先将所有待比较数值（这里以正整数为例）统一为同样的数位长度，数位较短的数前面补零。（本题中基数由低到高，即将元素先按低位排序，再按高位排序，因此为LSD基数排序）接着，选择基数，将序列中的所有元素分配到对应的“桶”中，并对每个桶里的元素个数进行计数，随后依次收集桶中元素并将其放在对应位置，重复执行“分配”与“收集”操作，直到所有元素已经排好序。

### 2.8.2 算法代码

//求数组的最大数位数

inline int max\_digit(int arr[],int l,int r)

{

int max\_data = arr[l];

for (int i = l + 1; i <= r; i++)

if (max\_data < arr[i])

max\_data = arr[i];

int upper\_bound = 10, num\_of\_digits = 1;

while (upper\_bound < max\_data)

{

upper\_bound \*= 10;

num\_of\_digits++;

}

return num\_of\_digits;

}

//LSD基数排序

long long radix\_sort(int arr[], int l, int r)

{

long long cnt = 0;

int num\_of\_digits = max\_digit(arr, l, r);

int len = r - l + 1;

int\* tmp = new int[r + 1]; //临时存放排过序的数据

int\* count = new int[10]; //0~9每个数字开一个桶记录

int radix = 1; //基数 从小到大

for (int i = 1; i <= num\_of\_digits; i++) //从低到高每一位分别排序

{

for (int j = 0; j < 10; j++)

count[j] = 0; //数组清零

cnt += 10;

for (int j = l; j <= r; j++)

{

int k = (arr[j] / radix) % 10; //提取对应数位数值

count[k]++;

}

for (int j = 1; j < 10; j++)

count[j] = count[j - 1] + count[j];

cnt += 9;

for (int j = r; j >= l; j--)

{

int k = (arr[j] / radix) % 10;

tmp[count[k] - 1] = arr[j];

count[k]--;

}

for (int j = 0; j < len; j++)

arr[j] = tmp[j];

radix \*= 10;

}

delete[] tmp;

delete[] count;

return cnt;

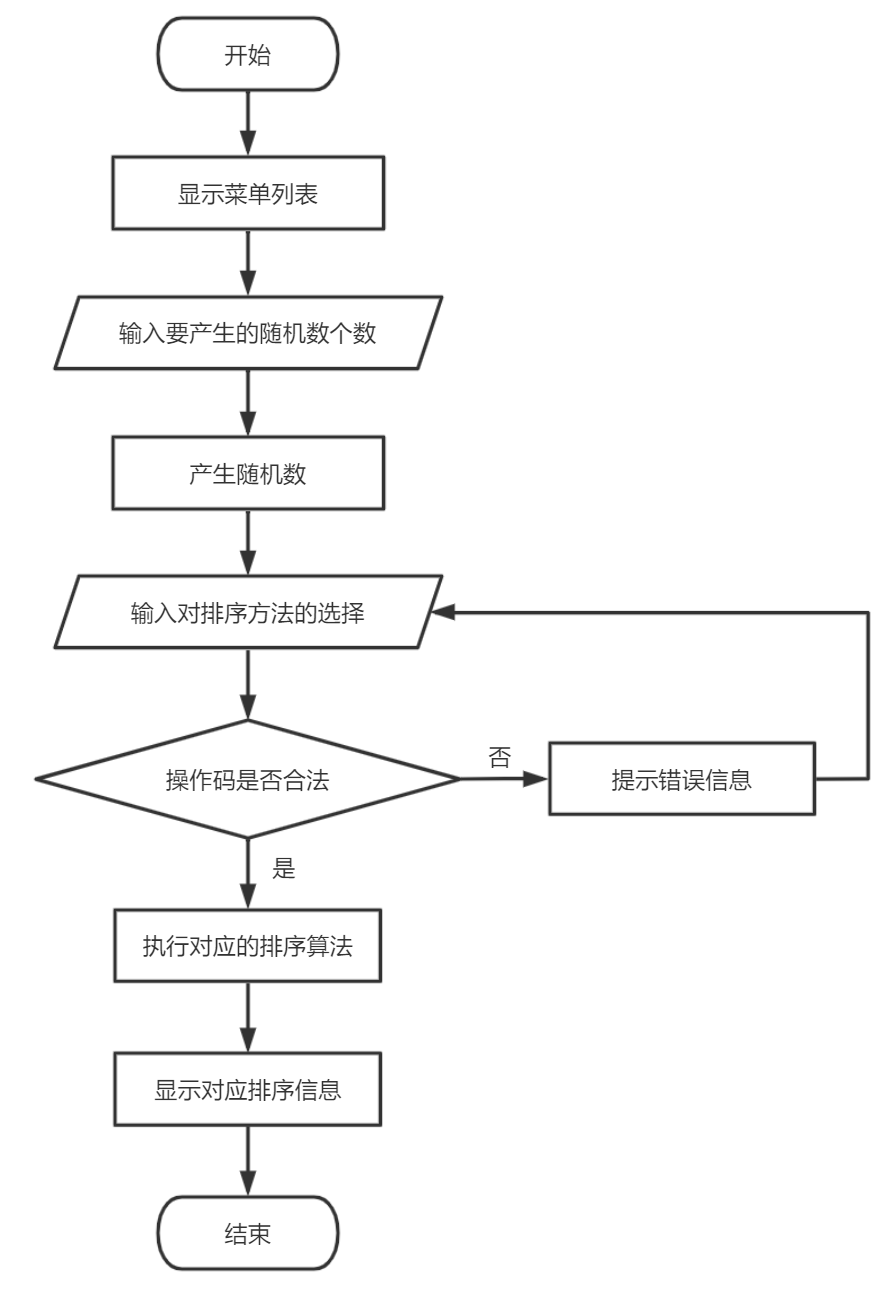
}

### 2.8.3 算法分析

基数排序的平均时间复杂度为O((n+radix)d)，d为进行“分配”与“收集”的趟数。基数排序的空间复杂度为O(n+radix)。

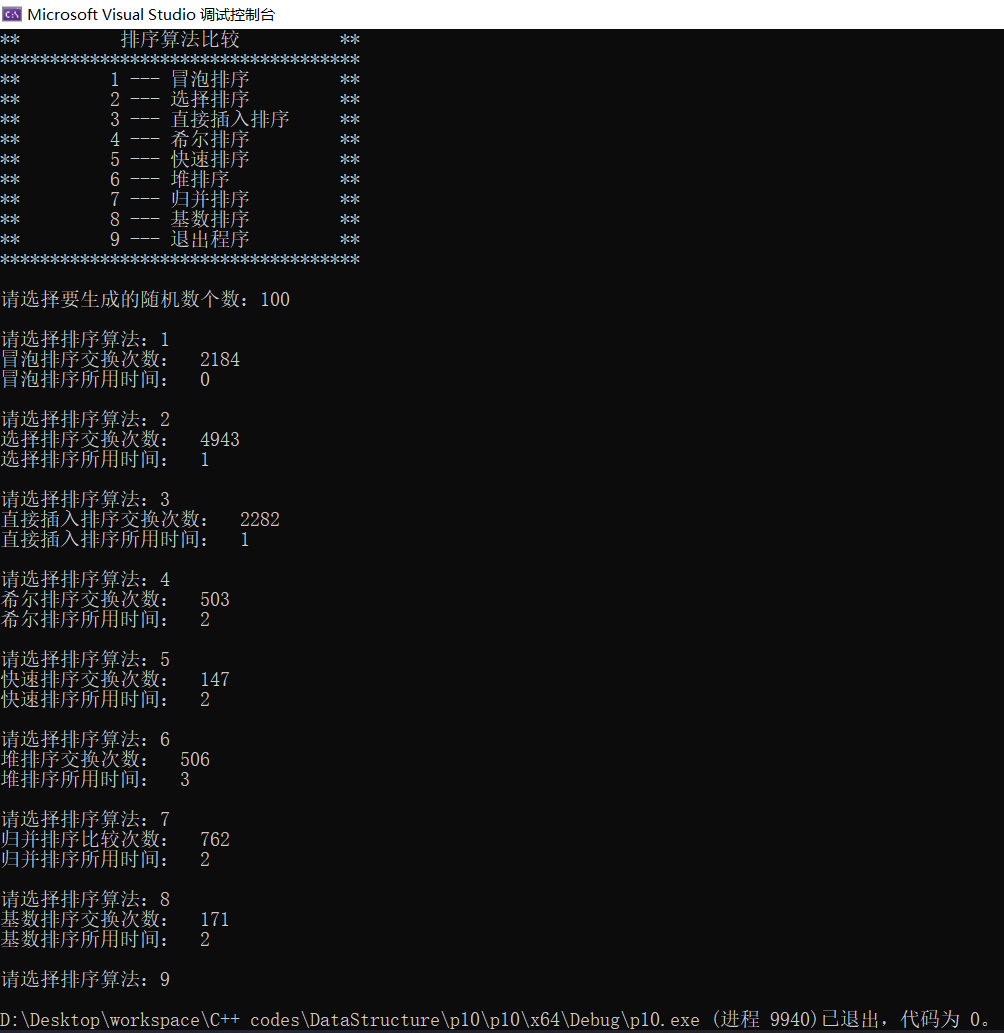
基数排序是一种稳定的排序算法。

# 3 总体程序流程图

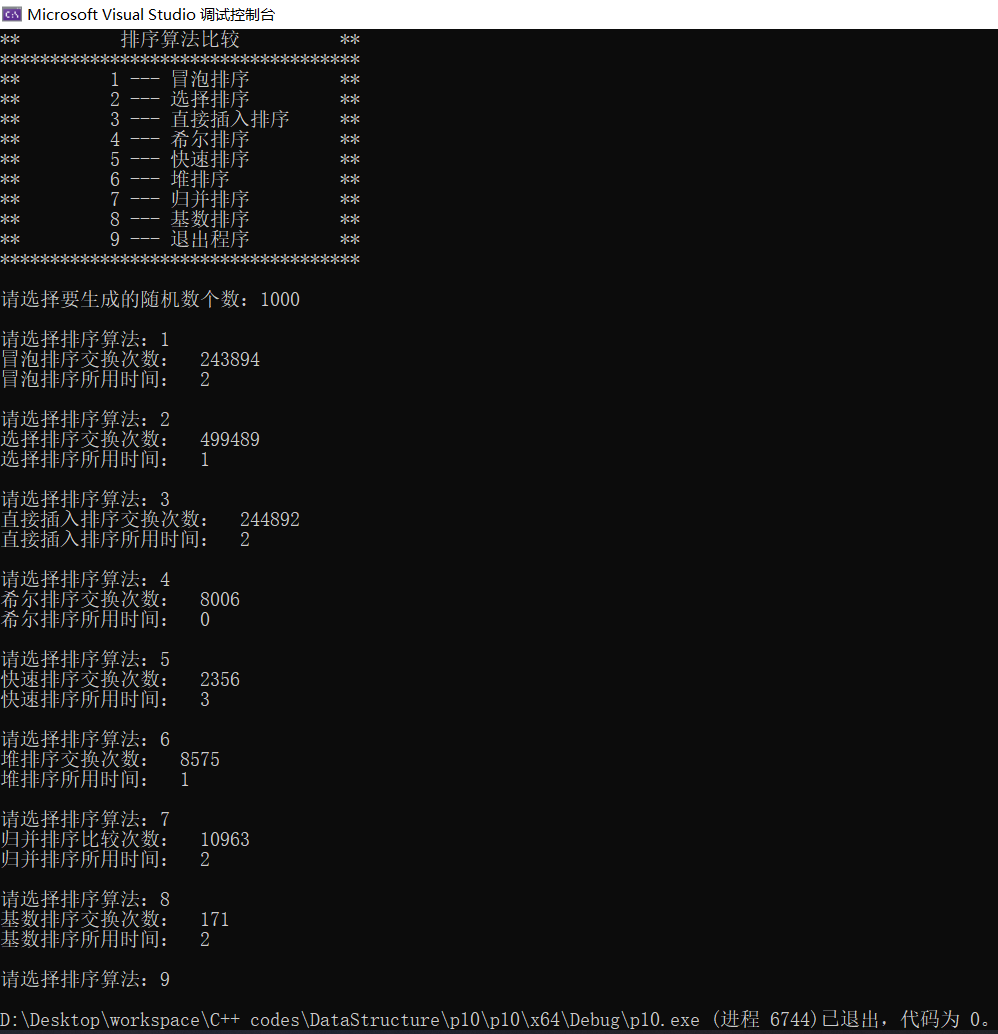


# 4 项目测试

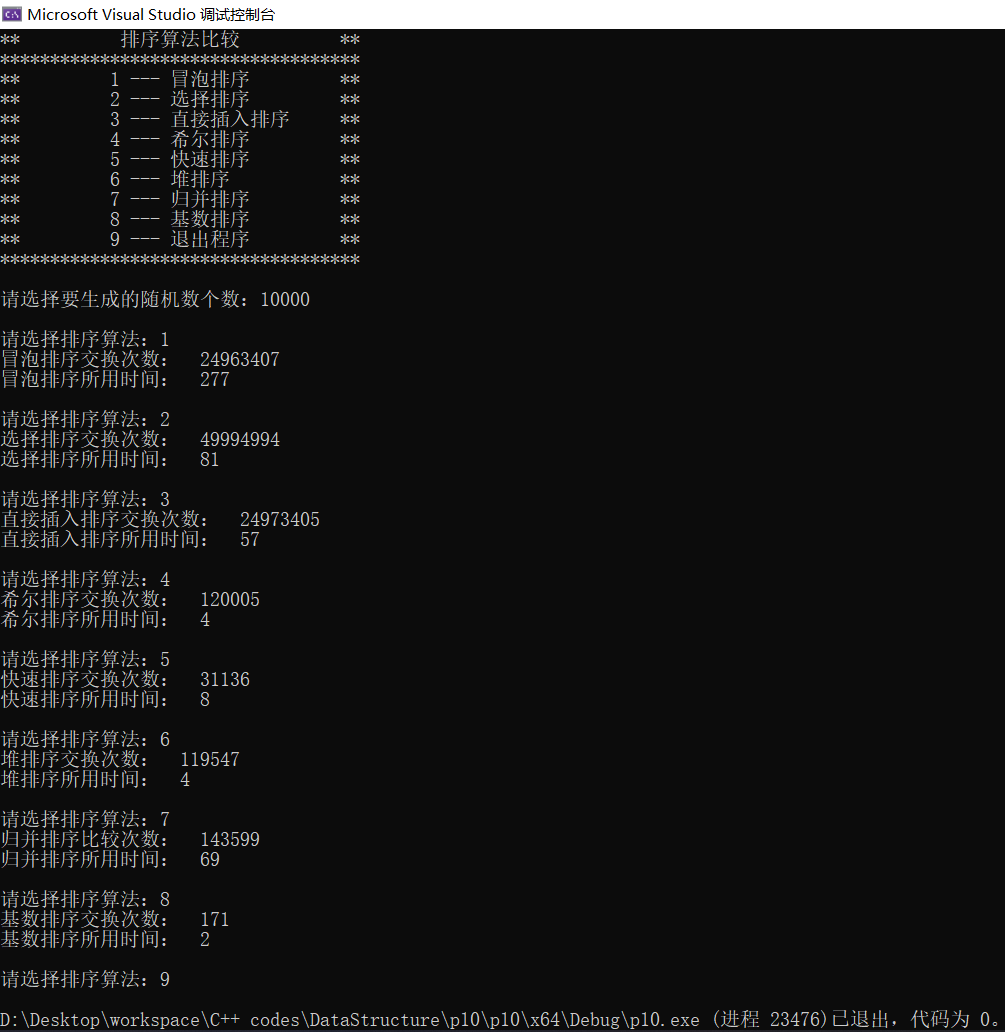
## 4.1 100个随机数测试



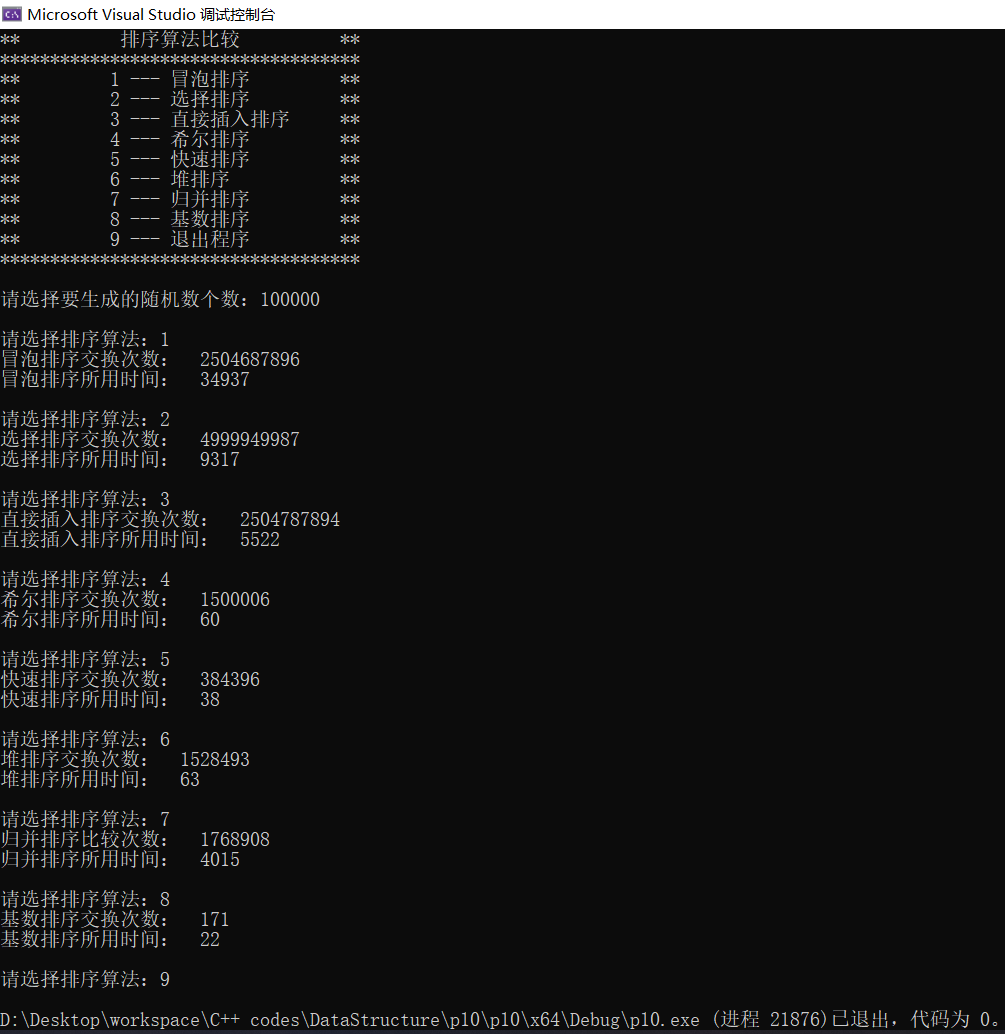
## 4.2 1000个随机数测试



## 4.3 10000个随机数测试



## 4.4 100000个随机数测试



## 4.5 错误输入测试

