同濟大學
TONGJI UNIVERSITY

项目说明文档

8 种排序算法的比较案例

指导老师：张颖

1851009 沈益立

目录

目录
1.分析
- 1.1 背景分析
- 1.2 功能分析

2.设计
- 2.1 整体框架设计
- 2.2 类结构设计
- 2.3 成员和操作设计
- 2.4 排序算法初始化设计
- 2.5 系统设计

3.实现
- 3.1 冒泡排序的实现
  - 3.1.1 冒泡排序流程图
  - 3.1.2 冒泡排序核心代码实现
  - 3.1.3 冒泡排序性能分析
- 3.2 选择排序的实现
  - 3.2.1 选择排序流程图
  - 3.2.2 选择排序核心代码实现
  - 3.2.3 选择排序性能分析
- 3.3 插入排序的实现
  - 3.3.1 插入排序功能流程图
  - 3.3.2 插入排序核心代码实现
  - 3.3.3 插入排序性能分析
- 3.4 希尔排序的实现
  - 3.4.1 希尔排序流程图
  - 3.4.2 初始队列构造核心代码实现
  - 3.4.3 希尔排序性能分析
- 3.5 快速排序的实现
  - 3.5.1 快速排流程图
  - 3.5.2 快速排序核心代码实现
  - 3.5.3 快速排序的性能分析
- 3.6 堆排序的实现
  - 3.6.1 堆排序流程图
  - 3.6.2 堆排序核心代码实现
  - 3.6.3 堆排序的性能分析
- 3.7 归并排序的实现

3.7.1 归并排序流程图
3.7.2 归并排序核心代码实现
3.7.3 归并排序性能分析
3.8 基数排序的实现
3.8.1 基数排序流程图
3.8.2 基数排序核心代码实现
3.8.3 基数排序性能分析
4.测试
4.1 时间测试
4.1.1 小规模的随机数测试
对该结果的分析
4.1.2 中等规模的随机数测试
对该结果的分析
4.1.3 大规模的随机数测试
对该结果的分析
4.2 特殊测试
4.2.1 一位数的排序
对该结果的分析
4.2.2 三位数的排序
对该结果的分析
5.总结

1.分析

1.1 背景分析

排序问题在计算机数据处理中经常遇到，特别是在事务处理中，排序占了很大的比重。在日常的数据处理中，一般认为有四分之一的时间用在排序上，而对于安装程序，多达一半的时间花费在对表的排序上。 因此，对于计算机排序的研究从来都不是少数，因此也产生了大量的风格不同、效率不同、适用于 不同场合的排序方法。按照是否在内存排序，排序可以分成内排序和外排序两种，本次项目主要针对集中不同的内排序算法，通过实际测试比较出他们的特点和性能的优劣。

1.2 功能分析

要想观察出不同排序算法的性能差异，最直接的方法就是统计它们的运行时间，此外，还可以统计它们在一次算法内的交换次数。本次设计的排序算法有八种，对于待排序的数列，可以由随机数生成，通过实际对随机数列的排序的时间和效率，得出八种算法的性能。

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。本实验记录上述数据量下，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。

2.设计

2.1 整体框架设计

本次设计包含多种排序算法，而每种排序算法中又可能需要不止一种函数来作为辅助函数，又基于面向对象的思想，设计了一个类Sort。在公有操作部分设计了9个函数，它们分别作为初始化IO和8种排序算法的接口。而在私有操作部分设计了若干辅助函数。整体框架图如下。

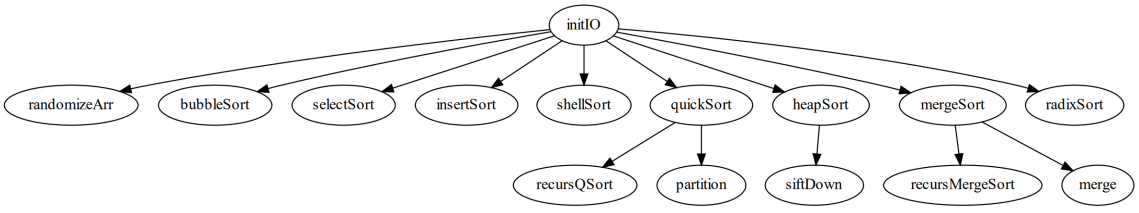
由于作业的规定，n个小区之间通过电网来连通，对小区的定义为带权值的无向图，设计了一个类Graph来存储这个无向图，并且使用邻接矩阵来作为图的存储形式。由于可能输入的顶点是char类型，因此设计了两个数组：`m_hash[256]`将字符(char)转化为顶点在邻接矩阵中的真实位置(int)、`m_charHash[101]`将顶点在邻接矩阵中的真实位置(int)转化为它的字符形式(char)。

定义了一个结构体：课程结构`Course`，用于存储每节课程的课程名、课号、课时、课程指定学期、先导课程等信息

定义了一个类：节点类`LNode`，作为链式队列的节点。

定义了一个类：链式队列`LinkedQueue`，可以将拓扑排序的序列存储下来。

定义了一个类：`Solution`，用于处理IO、调用算法和存放各种指针，避免使用全局变量。



2.2 类结构设计

由于本题目的相关要求，设计了一个主类`Sort`来实现IO和算法实现功能。

2.3 成员和操作设计

排序类`Sort`

公有操作：

```
void initIO();                  //初始化IO

void bubbleSort();              //冒泡排序
void selectSort();              //选择排序
void insertSort();              //插入排序
void shellSort();               //希尔排序
void quickSort();               //快速排序
void heapSort();                //堆排序
void mergeSort();               //归并排序
void radixSort();               //基数排序
```

私有操作：

```
/*辅助函数*/
void randomizeArr();            //每次操作前对数组随机化
void swap(int& a, int& b);      //基于引用的交换函数

void recursQSort(int left, int right, long long& count);
                                //递归地求解快速排序的辅助函数
void recursMergeSort(int left, int right, long long& cnt);
                                //递归地求解归并排序的辅助函数
void merge(int left, int mid, int right, long long& cnt);
                                //合并两个数组的辅助函数
int partition(int low, int high, long long& count);
                                //快排中分治的辅助函数
void siftDown(int start, int end, long long& cnt);
                                //堆排序中从上向下调整的辅助函数
```

私有成员：

```
int m_sortMethod;               //选择排序方式码
long long int m_sortNum;        //待排序数组的长度
int* m_array;                   //存放待排序数的数组
int* m_supArr;                  //归并排序中的辅助数组
```

2.4 排序算法初始化设计

在输入待排序数组的长度之后，执行一次`randomizeArr()`函数，将数组内的元素全部初始化：

```
void Sort::randomizeArr() {
    for (auto i = 0; i < m_sortNum; i++) {
        m_array[i] = rand();
    }
}
```

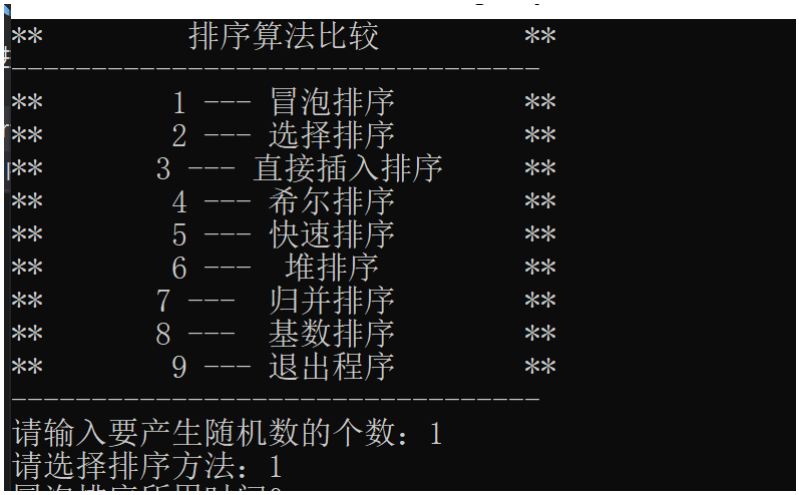
在开始每个算法的执行流程之前，创造一个时间戳`tStart`和计数器`cnt`，以记录比较或者值传递的次数和算法执行的总时长。

```
long long cnt = 0;                      //比较或者值传递的计数器
clock_t tStart = clock();               //排序开始的时间戳
```

由于每次排序都是基于成员变量数组的，故在排序后序列会变得有序，想要查看其他算法性能便变得不可能。在每个函数前都插入上述代码，可以在每次执行时都创建一组新的数列排序，虽然不能直接横向比较，但在大量数据时取平均值，排序的比较还是有意义的。

2.5 系统设计

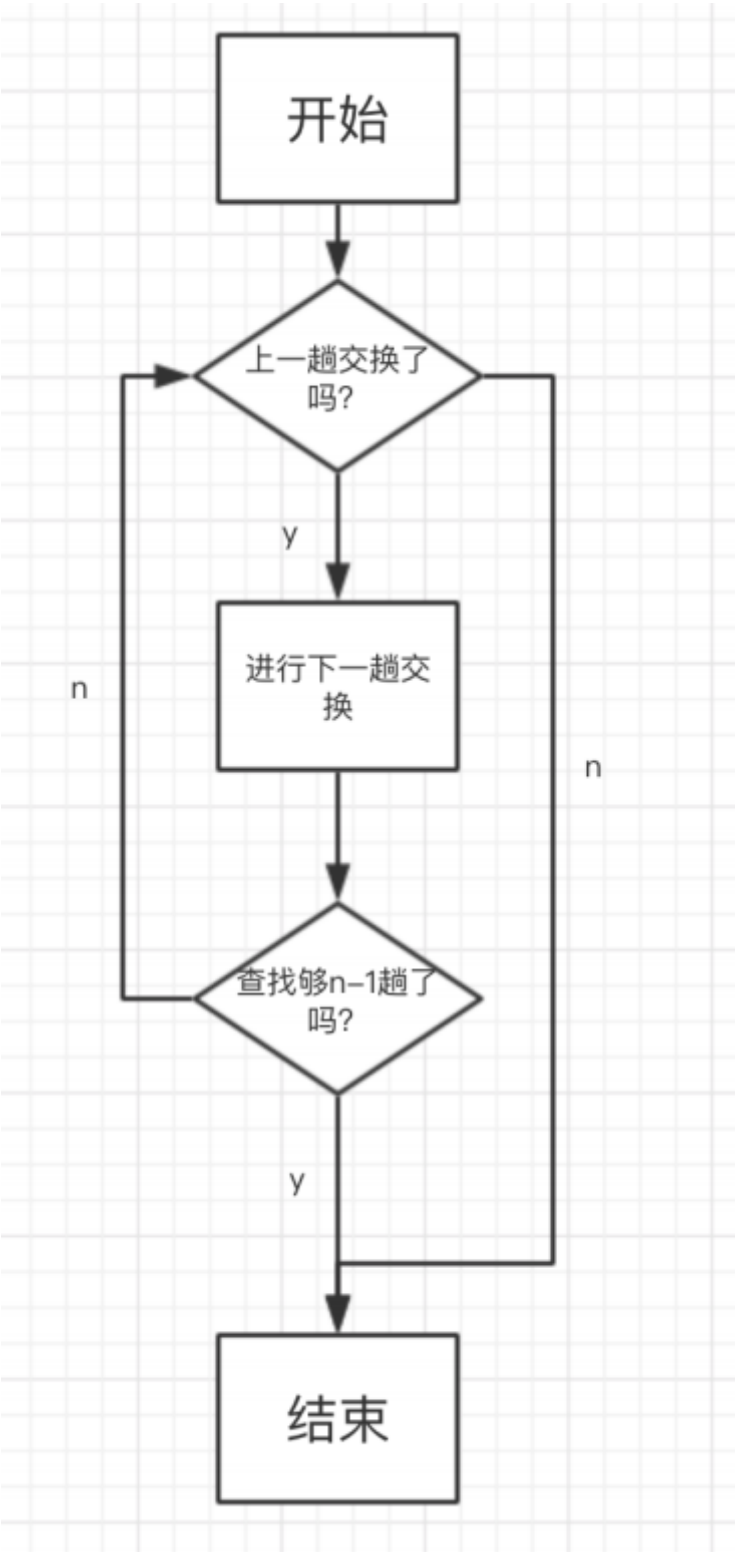
程序执行后，会自动实体化一个Sort类sol，随后调用`initIO()`，向用户询问要排序的个数和排序方法，随后调用上述`randomizeArr()`方法，再调用相应的排序函数进行排序。



3.实现

3.1 冒泡排序的实现

3.1.1 冒泡排序流程图



3.1.2 冒泡排序核心代码实现

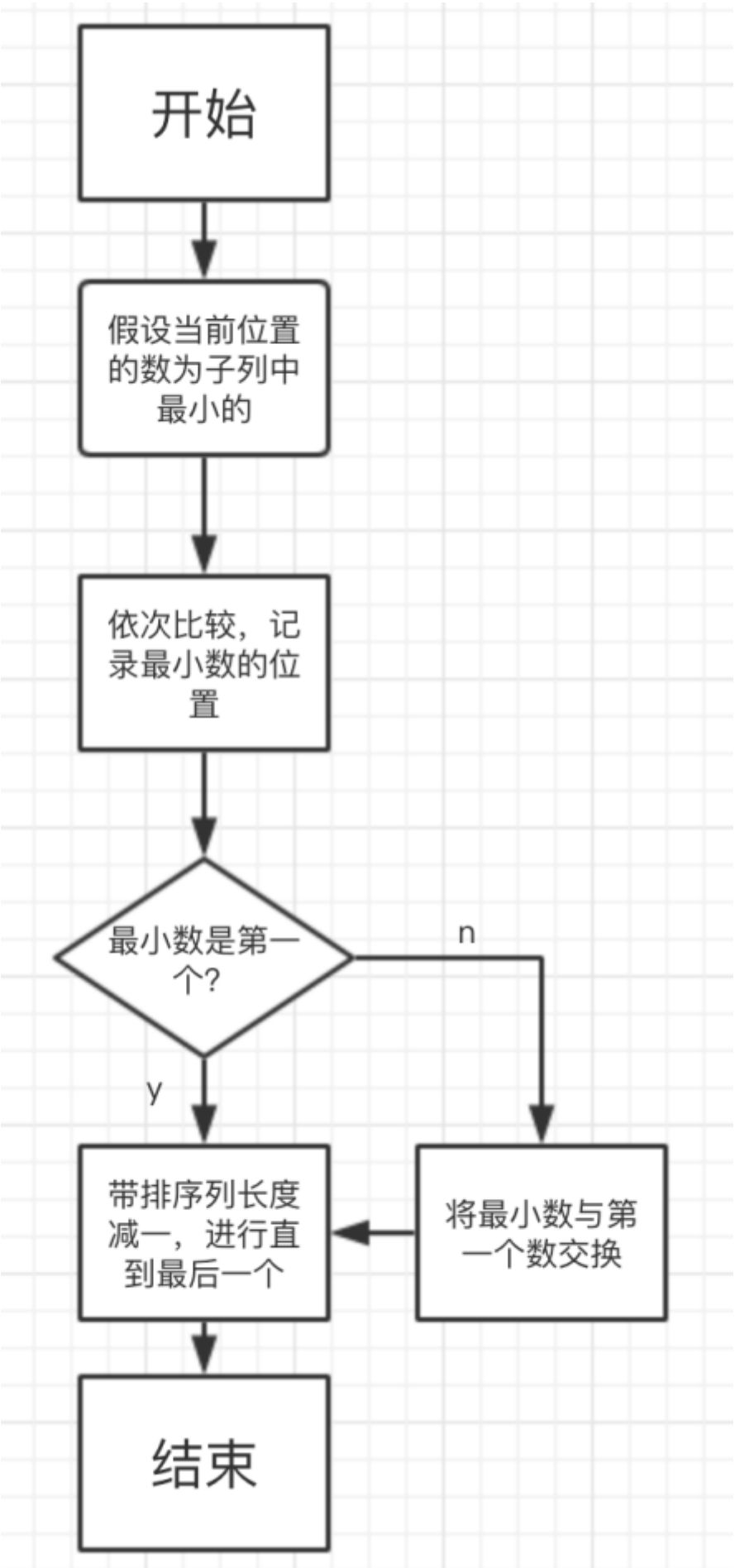
```
void Sort::bubbleSort() {
    long long cnt = 0;
    clock_t tStart = clock();
    for (auto i = 0; i < m_sortNum - 1; i++) {
        for (auto j = 0; j < m_sortNum - 1 - i; j++) {
            if (m_array[j] > m_array[j + 1]) {
                swap(m_array[j], m_array[j + 1]);
                cnt += 3;
            }
        }
    }
    cout << "冒泡排序所用时间" << clock() - tStart << endl;
    cout << "冒泡排序交换次数" << cnt << endl;
}
```

3.1.3 冒泡排序性能分析

冒泡排序共要进行n-1趟比较和交换操作，对于比较次数，冒泡排序是稳定的，因为要对所有的有序对进行遍历。对于交换次数则与序列的初始状态密切相关：最好的情况下可以一次都不需要交换；最差的情况每一次都要进行交 换。 将冒泡排序改进后，如果元素本身有序，那么只需要一趟排序就结束了。即使是这样，计算得到冒泡排序的时间复杂度仍然是O(n^2)。由于不存在远距离交换，冒泡排序是一种稳定的排序算法。

3.2 选择排序的实现

3.2.1 选择排序流程图


开始
假设当前位置的数为子列中最小的
依次比较，记录最小数的位置
最小数是第一个?
n
y
带排序列长度减一，进行直到最后一个
将最小数与第一个数交换
结束

3.2.2 选择排序核心代码实现

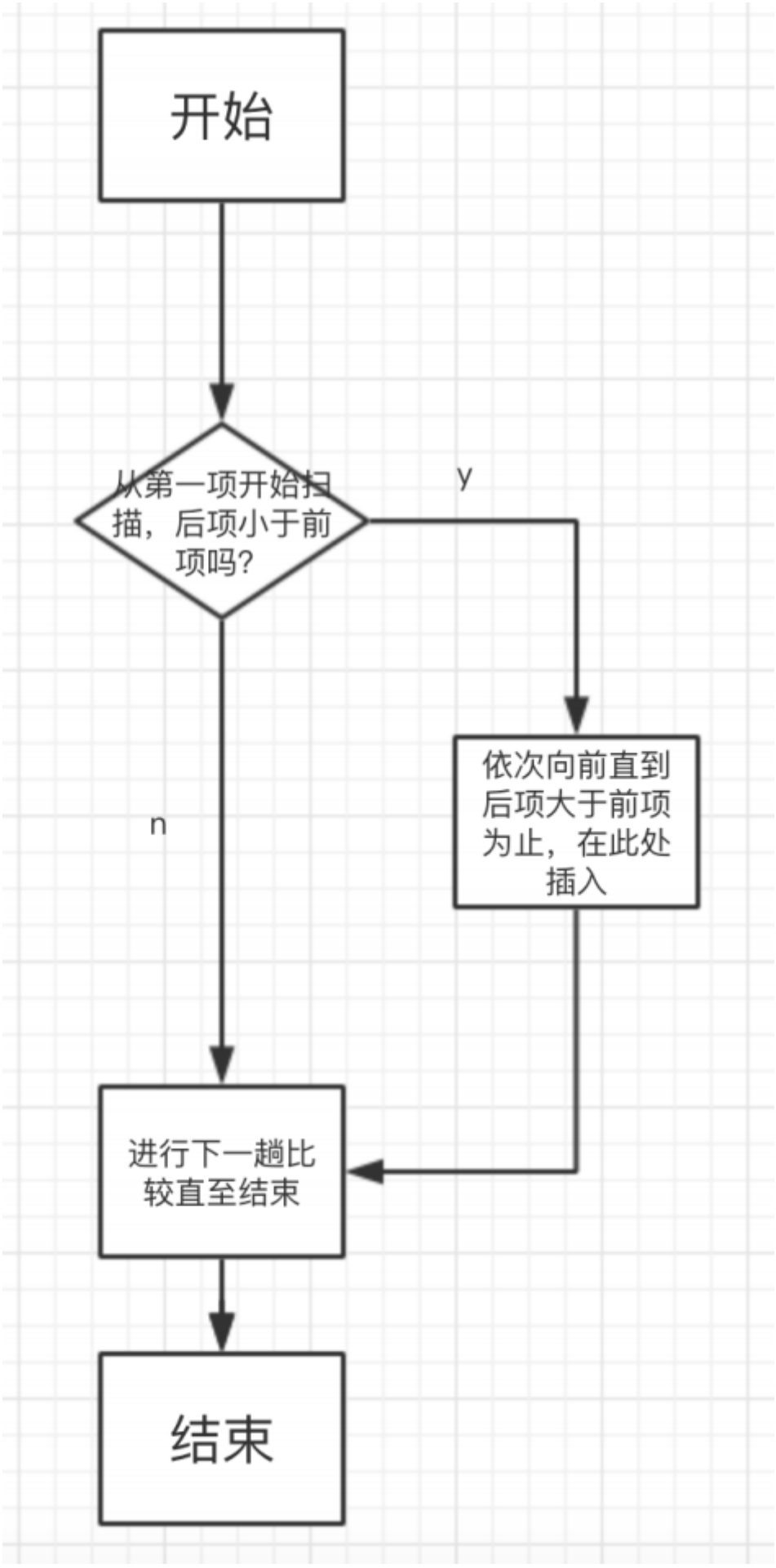
```
void Sort::selectSort() {
    long long cnt = 0;
    clock_t tStart = clock();
    int minIndex, tmp;
    for (auto i = 0; i < m_sortNum; i++) {
        minIndex = i;
        for (auto j = i; j < m_sortNum; j++) {
            if (m_array[j] < m_array[minIndex]) {
                minIndex = j;
            }
            if (i != minIndex) {
                swap(m_array[i], m_array[minIndex]);
                cnt += 3;
            }
        }
    }
    cout << "选择排序所用时间" << clock() - tStart << endl;
    cout << "选择排序交换次数" << cnt << endl;
}
```

3.2.3 选择排序性能分析

选择排序的比较次数与带排序列的初始状态无关，即每次都要比较相同的次数。对于交换次数，最好的情况下可以达到0，最坏的情况则为3(n-1)。对于选择排序，序列的初始状态对排序的影响不大，因此它的时间消耗较为稳定，渐进时间复杂度为O(n^2)。 选择排序在序列数字较小但元素规模很大的时候有较好的效果，因为在这种情况下交换的时间会远远大于比较的时间，而选择排序涉及的交换次数很少。 由于存在远距离交换，选择排序是一种不稳定的排序算法。

3.3 插入排序的实现

3.3.1 插入排序功能流程图



3.3.2 插入排序核心代码实现

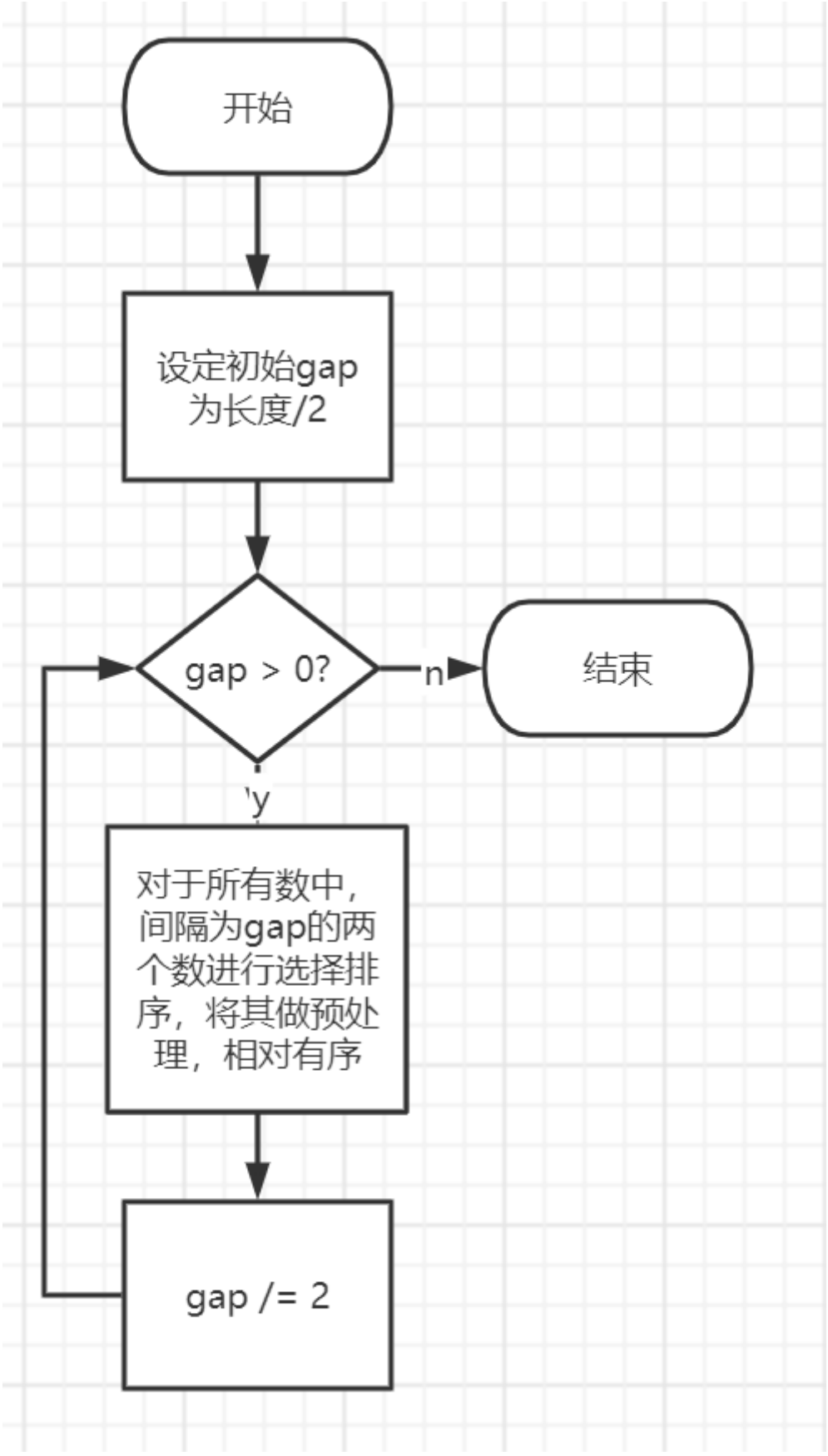
```
void Sort::insertSort() {
    long long cnt = 0;
    clock_t tStart = clock();
    int tmp;
    for (int i = 1; i < m_sortNum; i++) {
        if (m_array[i] < m_array[i - 1]) {
            tmp = m_array[i];
            int j = 1;
            while (tmp < m_array[i - j]) {
                m_array[i - j + 1] = m_array[i - j];
                cnt++;
                j++;
            }
            m_array[i - j + 1] = tmp;
            cnt++;
        }
    }
    cout << "直接插入排序所用时间" << clock() - tStart <<
endl;
    cout << "直接插入排序交换次数" << cnt << endl;

}
```

3.3.3 插入排序性能分析

插入排序的比较和交换次数与数组的初始状态相关。在最好的情况下，只需比较n-1次并且不需要任何交换就可以完成对序列的排序。在最坏的情况下，比较次数和交换次数均为平方级别， 故它的效率和序列的初始状态相关度较大，总时间复杂度也是平方级。 由于不存在远距离交换，插入排序是一种稳定的排序算法。

3.4 希尔排序的实现

3.4.1 希尔排序流程图



3.4.2 初始队列构造核心代码实现

```
void Sort::shellSort() {
    long long cnt = 0;
    clock_t tStart = clock();

    cout << endl;
    for (int gap = m_sortNum / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < m_sortNum; i++) {
            int j = i;
            while (j - gap >= 0 && m_array[j] < m_array[j
- gap]) {
                swap(m_array[j], m_array[j - gap]);
                j -= gap;
                cnt += 3;
            }
        }
    }

    cout << "希尔排序所用时间" << clock() - tStart << endl;
    cout << "希尔排序交换次数" << cnt << endl;
}
```

3.4.3 希尔排序性能分析

希尔排序实质上是选择排序的优化。由于选择排序性能会随着数列的有序程度变化而变化，将选择排序作预处理，将其变得部分有序，即可降低其时间复杂度。

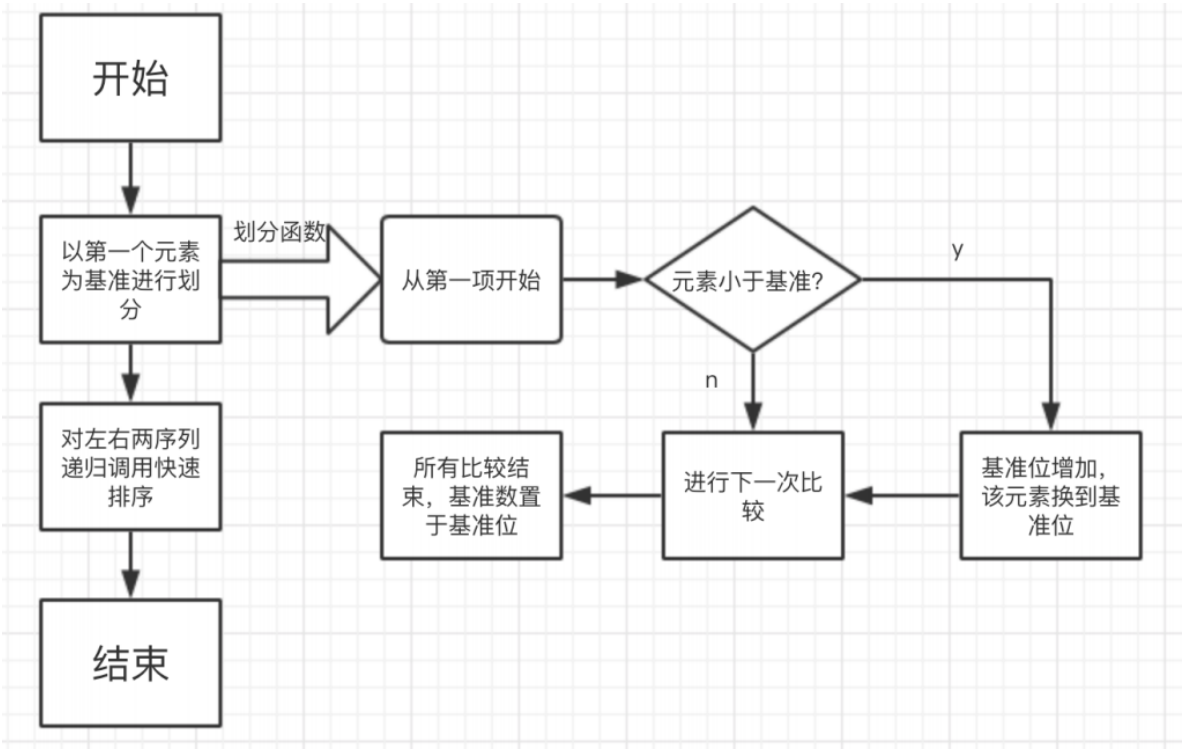
希尔排序是较为困难的算法，gap的取法没有统一的规定。本程序的gap取值为上次gap的1/2,。

根据Knuth所著的《计算机程序设计技巧》，大量统计显示：希尔排序的时间复杂度在O(n^1.25)到O((1.6n)^1.25)之间。

由于存在远距离交换，因此希尔排序是一种不稳定的排序算法

3.5 快速排序的实现

3.5.1 快速排流程图



3.5.2 快速排序核心代码实现

```
/*快排的框架函数*/
void Sort::recursQSort(int left, int right, long long&
cnt) {
    if (left < right) {
        int pivotPos = partition(left, right, cnt);
        recursQSort(left, pivotPos - 1, cnt);
        recursQSort(pivotPos + 1, right, cnt);
    }
}

/*快排的分治函数*/
int Sort::partition(int low, int high, long long& cnt) {
    int pivotPos = low;
    int pivot = m_array[low];
    for (int i = low + 1; i <= high; i++) {
        if (m_array[i] < pivot) {
            pivotPos++;
```

```
            if (pivotPos != i) {
                swap(m_array[pivotPos], m_array[i]);
                cnt += 3;
            }
        }
    }
    m_array[low] = m_array[pivotPos];
    m_array[pivotPos] = pivot;
    cnt += 2;
    return pivotPos;
}
```

3.5.3 快速排序的性能分析

快速排序是计算机科学领域应用最广的排序算法。一个良好优化的快速排序算法在大多数的计算机上运行得比其他排序算法都快，并且快速排序算法在空间上只需要使用一个辅助栈实现。

快排的速度取决于递归树的深度，如果左右两侧的序列数据量相似，快排能够达到最优性能。

然而对于最坏情况，每次划分有一侧只有一个元素，那么快排的效果将会退化为普通排序，甚至比普通排序更慢。

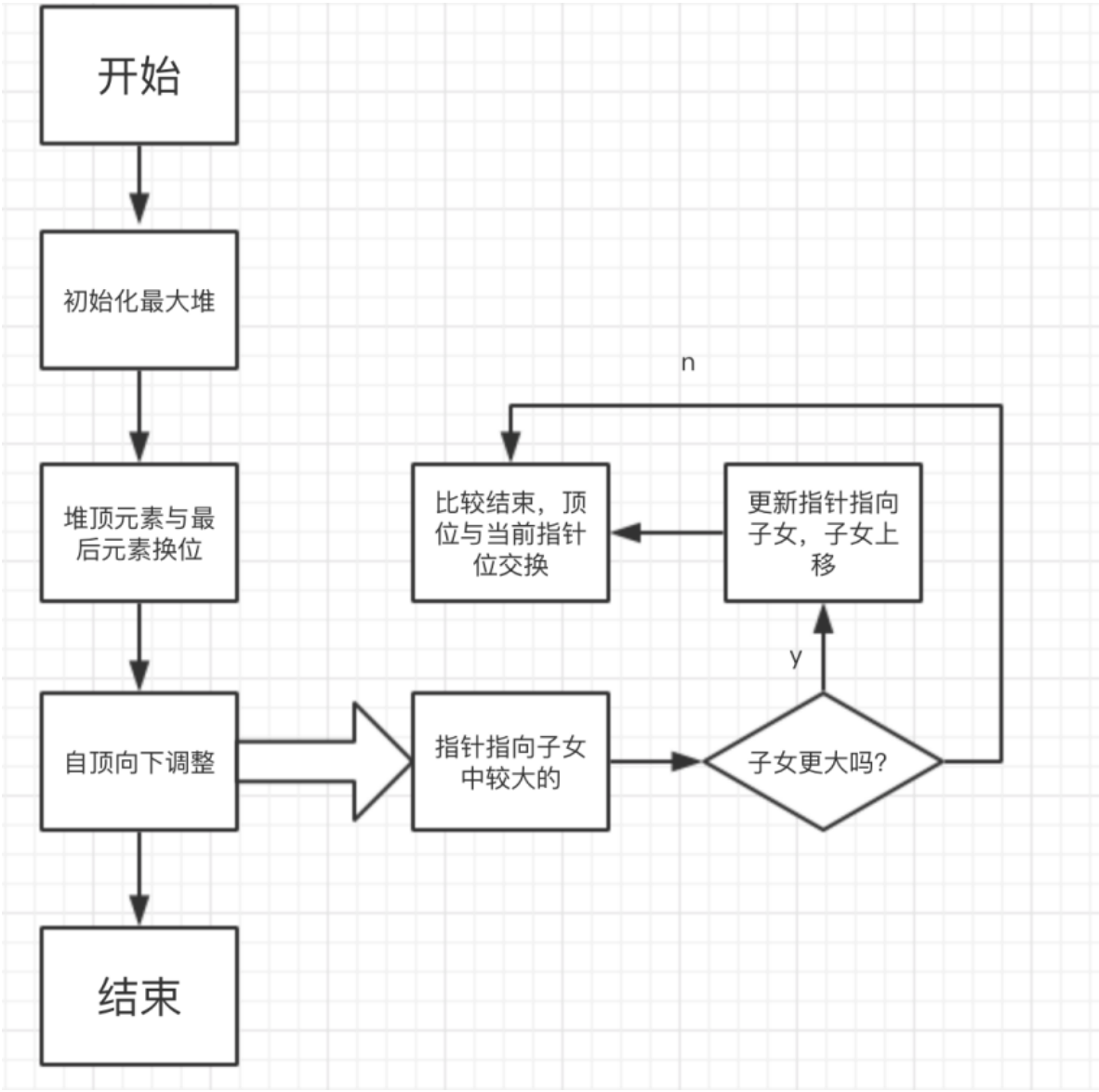
对此的优化措施是：当某侧序列长度较小时，直接采用稳定的插入排序。

然而在随机产生的一般情况下，快排的时间复杂度在O(nlogn)级别。

由于在分治时，存在远距离交换，因此快排是一种不稳定的算法。

3.6 堆排序的实现

3.6.1 堆排序流程图



3.6.2 堆排序核心代码实现

```
void Sort::heapSort() {
    long long cnt = 0;
    clock_t tStart = clock();
    for (int i = (m_sortNum - 2) / 2; i >= 0; i--) {
        siftDown(i, m_sortNum - 1, cnt);
    }
    for (int i = m_sortNum - 1; i >= 0; i--) {
        swap(m_array[0], m_array[i]);
        cnt += 3;
        siftDown(0, i - 1, cnt);
    }
    cout << "堆排序所用时间" << clock() - tStart << endl;
    cout << "堆排序交换次数" << cnt << endl;
```

```
}
/*堆排序向下调整函数*/
void Sort::siftDown(int start, int end, long long& cnt) {
    int i = start, j = 2 * i + 1; // j是i的左子女
    int tmp = m_array[i];
    cnt++;
    while (j <= end) {
        if (j < end && m_array[j] < m_array[j + 1]) {
            j++;
        }
        if (tmp >= m_array[j]) {
            break;
        }
        else {
            m_array[i] = m_array[j];
            cnt++;
            i = j;
            j = 2 * j + 1;
        }
    }
    m_array[i] = tmp;
    cnt++;
}
```

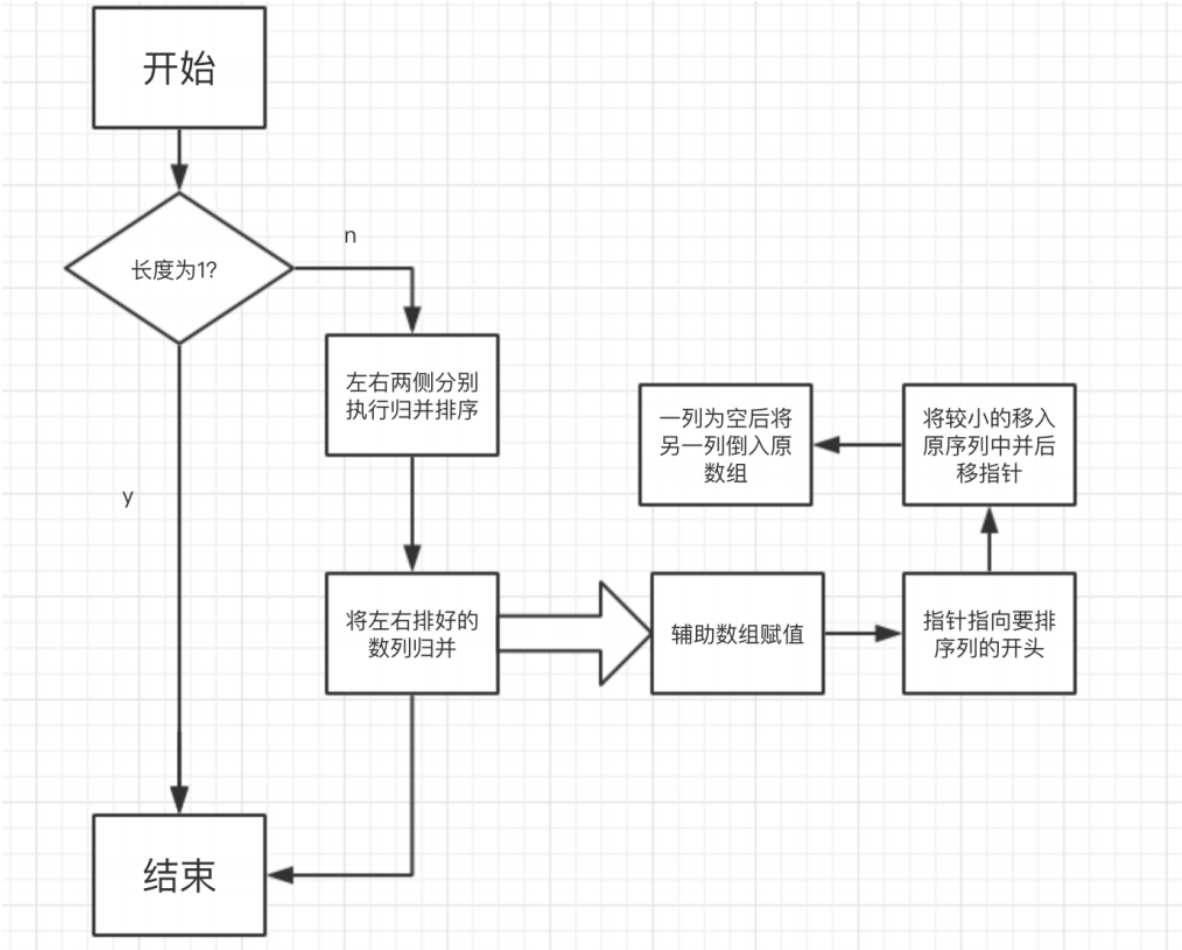
3.6.3 堆排序的性能分析

堆排序需要建立一个最小堆，同时需要最小堆的调整算法。其时间复杂度在O(nlogn)，性能很好，并且不需要额外的存储空间。

由于存在远距离交换，堆排序是一种不稳定的排序算法。

3.7 归并排序的实现

3.7.1 归并排序流程图



3.7.2 归并排序核心代码实现

```
/*归并排序主框架*/
void Sort::mergeSort() {
    m_supArr = new int[m_sortNum];
    long long cnt = 0;
    clock_t tStart = clock();
    recursMergeSort(0, m_sortNum - 1, cnt);
    cout << "归并排序所用时间" << clock() - tStart << endl;
    cout << "归并排序交换次数" << cnt << endl;
    delete[] m_supArr;
}

/*递归地处理归并排序*/
void Sort::recursMergeSort(int left, int right, long long&
cnt) {
    if (left >= right) {
        return;
    }
```

```
    int mid = (left + right) / 2;
    recursMergeSort(left, mid, cnt);
    recursMergeSort(mid + 1, right, cnt);
    merge(left, mid, right, cnt);
}
/*两数列的合并*/
void Sort::merge(int left, int mid, int right, long long&
cnt) {
    for (int i = left; i <= right; i++) {
        m_supArr[i] = m_array[i];
        cnt++;
    }
    int s1 = left, s2 = mid + 1, tmpPos = left;

    while (s1 <= mid && s2 <= right) {
        if (m_supArr[s1] <= m_supArr[s2]) {
            m_array[tmpPos++] = m_supArr[s1++];
        }
        else {
            m_array[tmpPos++] = m_supArr[s2++];
            cnt++;

        }
    }

    while (s1 <= mid) {
        m_array[tmpPos++] = m_supArr[s1++];
        cnt++;
    }

    while (s2 <= right) {
        m_array[tmpPos++] = m_supArr[s2++];
        cnt++;
    }
}
```

3.7.3 归并排序性能分析

归并排序是一种依赖递归算法的排序，类似于上述的快速排序，其复杂度也是O(nlogn)。

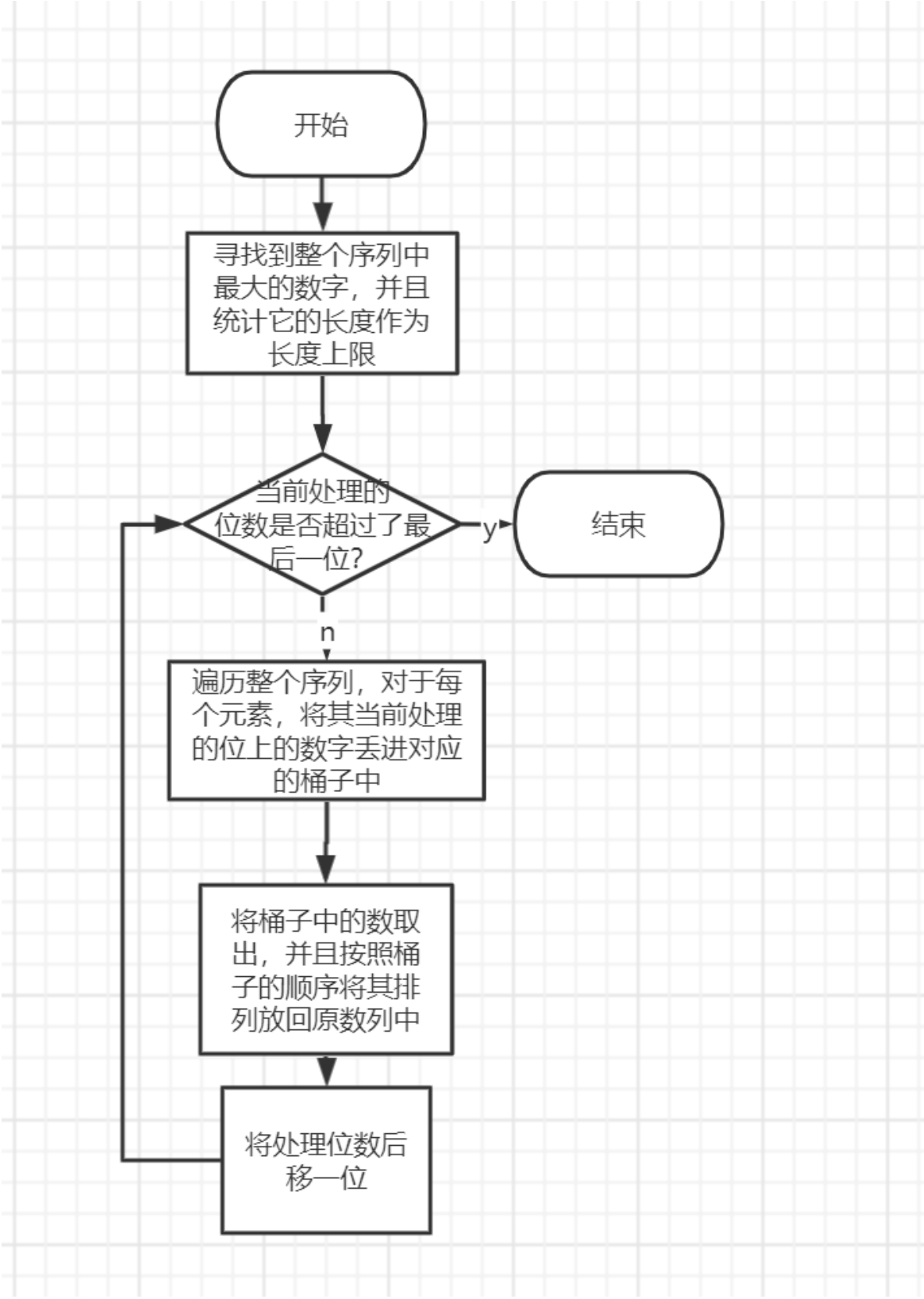
归并排序并不依赖初始输入序列，它始终将序列分成长度相等的两个部分。

其缺点在于归并的过程中，需要开辟额外的辅助数组m_supArr。

由于辅助空间的指针移动是连续的，归并排序不存在远距离交换，是一种稳定的排序算法。

3.8 基数排序的实现

3.8.1 基数排序流程图



3.8.2 基数排序核心代码实现

```
void Sort::radixSort() {
    long long cnt = 0;
    clock_t tStart = clock();
```

```
    m_supArr = new int[m_sortNum];
    int maxNum = -1;
    for (int i = 0; i < m_sortNum; i++) {
        maxNum = max(maxNum, m_array[i]);
    }//find maximum number.
    int d = 0;
    while (maxNum) {
        d++;
        maxNum /= 10;
    }
    //find maximum bits.
    int radix = 1;
    int bucket[10] = { 0 };
    int* tmp = new int[m_sortNum];
    for (int i = 1; i <= d; i++) {
        for (int j = 0; j < 10; j++) {
            bucket[j] = 0;
        }

        for (int j = 0; j < m_sortNum; j++) {
            int k = (m_array[j] / radix) % 10;
            bucket[k]++;
        }
        for (int j = 1; j < 10; j++) {
            bucket[j] += bucket[j - 1];
        }

        for (int j = m_sortNum - 1; j >= m_sortNum; j--) {
            int k = (m_array[j] / radix) % 10;
            tmp[bucket[k] - 1] = m_array[j];
            cnt++;
            bucket[k]--;
        }

        for (int j = 0; j < m_sortNum; j++) {
            m_array[j] = tmp[j];
            cnt++;
        }
        radix *= 10;
    }
```

```
    //radSort(0, m_sortNum - 1, i, cnt);
    cout << "基数排序所用时间" << clock() - tStart << endl;
    cout << "基数排序值传递次数" << cnt << endl;
}
```

3.8.3 基数排序性能分析

基数排序事实上是桶排序的优化，对于元素的每一位都进行桶排序。基数排序需要大量的存储空间来实现。它的效率计算很特别，与待排序的数字的最大位数有关。因为基数排序是一种稳定的排序，对于待排序元素较多，而其关键码的位数都很小的情况下，使用基数排序是非常理想的选择。

4.测试

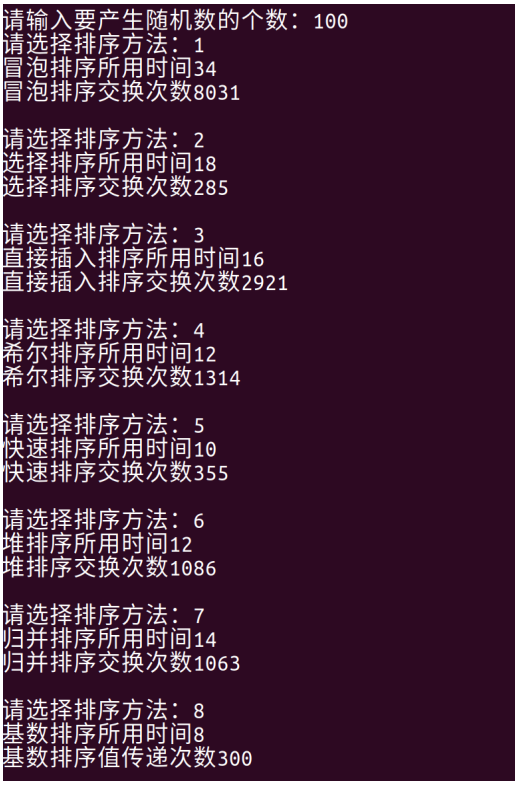
4.1 时间测试

4.1.1 小规模的随机数测试

测试用例：

> 随机数数量：100

实验结果：



对该结果的分析

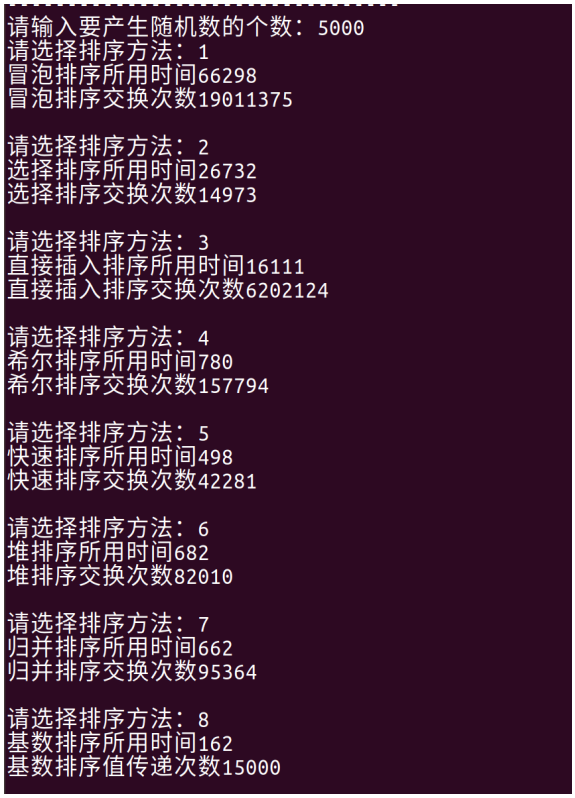
1. 小规模测试时，快排并不具备明显优势，和大多数算法性能相近。
2. 由于rand()产生了10位随机数，基数排序效果相比于基于比较的算法效率并无巨大优势。
3. 即使是非常小批量的测试，冒泡排序效果仍然不理想，是最差的。
4. 在O(n^2)算法中，选择排序所需要的的交换次数是最少的。

4.1.2 中等规模的随机数测试

测试用例：

> 随机数数量：5000

实验结果：



对该结果的分析

1. 在中等排序下，O(nlogn)以及更低的算法表现出了非常明显的优势，使用时间相对于O(n^2)算法低了数个量级。
2. 基数排序由于数据批量不够大，仍然没有体现出其绝对优势。
3. 快速排序的在交换次数和时间上的优势体现出来了。
4. 选择排序仍然是所需交换次数非常占优势的排序方式。

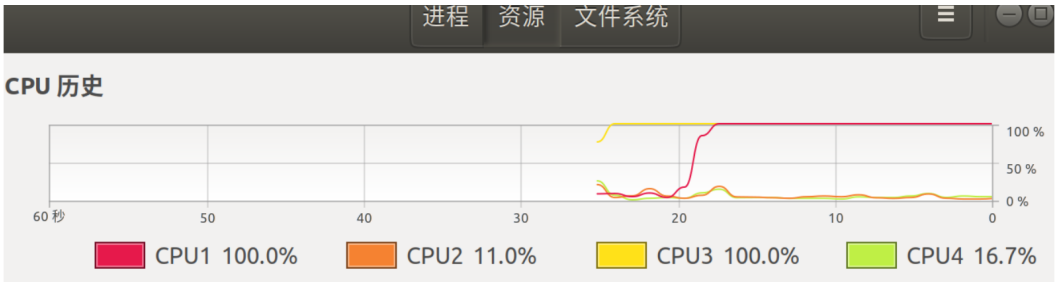
4.1.3 大规模的随机数测试

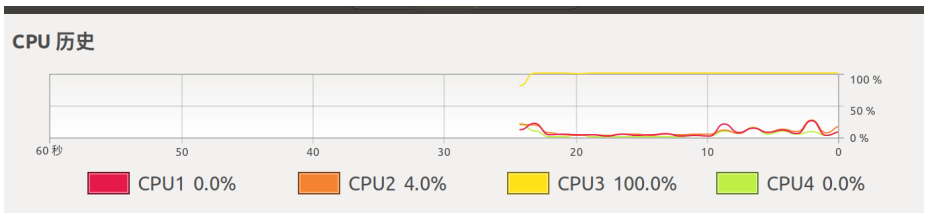
测试用例：

> 随机数数量：100000

实验结果：

```
请输入要产生随机数的个数：100000
请选择排序方法：1
冒泡排序所用时间33337059
冒泡排序交换次数7482182067

请选择排序方法：2
选择排序所用时间9457206
选择排序交换次数299658

请选择排序方法：3
直接插入排序所用时间6310528
直接插入排序交换次数2490244233

请选择排序方法：4
希尔排序所用时间24402
希尔排序交换次数5965551

请选择排序方法：5
快速排序所用时间18295
快速排序交换次数762555

请选择排序方法：6
堆排序所用时间16763
堆排序交换次数2073789

请选择排序方法：7
归并排序所用时间16152
归并排序交换次数2560846

请选择排序方法：8
基数排序所用时间2733
基数排序值传递次数300000
```





上图分别为在执行1、2、3排序和4、5、6、7、8排序时，CPU的占用率图。可以看出，在执行O(n^2)算法时，CPU几乎在满载运行。而在执行O(nlogn)算法时，没有使用太多算力。

对该结果的分析

1. 当随机数个数达到100000时，时间复杂度非常重要。冒泡排序时间极长，交换次数较多；直接插入排序的时间消耗较少，但是仍然不可以被接受；选择排序的交换次数很占优势，但是其速度太慢。而高级算法的运行时间已经与其产生了量级上的差异，完全可以被接受。
2. 基数排序体现出了一定的优势，它的运行时间已经是最短，然而还是没有和基于比较的算法产生质的差距。
3. 快速排序在达到100000个的时候速度优势没有那么明显了，可能是由于Linux系统的随机数机制的原因。在此情况下仍然产生了部分重复数字，因此快排效果受损。

4.2 特殊测试

4.2.1 一位数的排序

修改`randomizeArr()`函数，改为

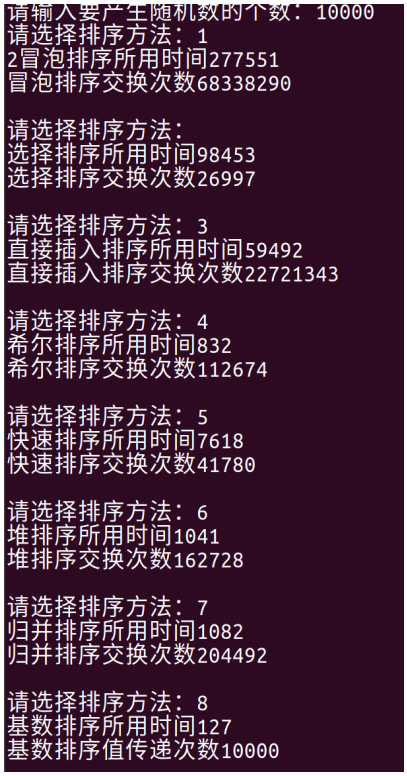
```
void Sort::randomizeArr() {
    for (auto i = 0; i < m_sortNum; i++) {
        m_array[i] = rand() % 10;
    }
}
```

使其产生0-9的数。

测试用例：

```
随机数个数：10000
```

测试结果：



对该结果的分析

1. 快速排序的效率明显下降，是因为在该数列中有大量相同的元素，因此划分效率降低，多出了很多重复移动的操作。
2. 基数排序体现出极为明显的优势，与O(nlogn)的排序方法产生了一个量级的优势。这是由于现在只需要用一个桶就可实现基数排序。
3. 直接插入排序略好于冒泡和选择排序。

4.2.2 三位数的排序

修改`randomizeArr()`函数，改为

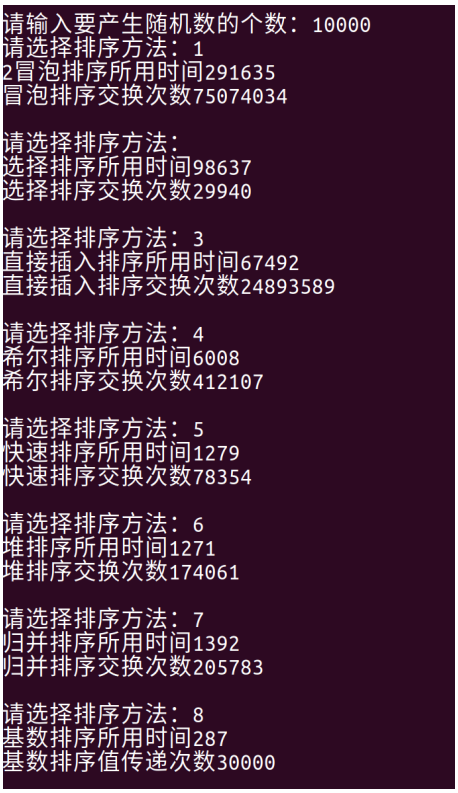
```
void Sort::randomizeArr() {
    for (auto i = 0; i < m_sortNum; i++) {
        m_array[i] = rand() % (1000 - 100) + 100;
    }
}
```

使其产生100-999的数。

测试用例：

```
随机数个数：10000
```

测试结果：



对该结果的分析

1. 快速排序的效率相较于一位数的情况，效率明显上升了一些。这是由于对于三位数而言，重复的元素大大减少，其划分的效率也变高了很多。
2. 基数排序仍然有一定优势，但是由于现在要使用三个桶来排序，差距没有一位数那么明显。
3. 直接插入排序仍然略好于冒泡和选择排序。
4. 希尔排序被正规O(n^2)排序拉开差距，可能是因为gap取2的原因。

5.总结

对于上述测试结果，可总结如下：

1. 冒泡排序：始终是交换次数较多并且时间复杂度较高的。
2. 选择排序：比较次数太多，交换次数较少，不一定是最优算法，适用于待排序个数较多的情况
3. 插入排序：稳定的O(n^2)算法，时间复杂度受序列影响较小。
4. 希尔排序：性能接近高级算法，但是对gap的取值有较高要求。
5. 快速排序：重复元素较多时，效率非常差，但是如果有优化，对于重复元素进行插入排序，几乎是应用效果最好的内排序算法。
6. 堆排序：速度次于快排，是性能较高的算法，并且不需要额外内存即可实现。
7. 归并排序：不受序列影响的高性能算法，但是内存开销较大。
8. 基数排序：在待排序的最大数字位数少，数列长度较长时，性能非常好，并且优于几乎所有高级算法。但是在一般情况下，性能没有理想中那么好，并且基数排序的内存开销太大了。