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：翟晨昊

学　　号：1952216

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 分析

1.1 背景分析

排序问题也是我们在日常生活中经常会遇到的问题。有时候我们对人进行排序，有时候我们对物品进行排序，还有的时候我们仅仅是对数字进行排序。不管对什么进行排序，它总要有一个用来排序的依据，例如根据人的高低进行排序，此时排序的依据就是身高，按人的身高将人依次排列。

在计算机中，也有排序的存在，计算机中的排序是指将杂乱无章的数据元素，通过一定的方法按关键字顺序排列的过程。排序算法就是如何使得记录按照要求排列的方法。计算机中的排序算法有很多种，每一种算法都有它们独特的实现方式与优化方法。在本案例中我们就来讨论其中使用频率比较高的几种排序算法。

1.2 功能分析

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和十万个随机数，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。

2 设计

2.1 生成随机数设计

由于题目中要求最多会有100000个随机数，但c++中的rand()范围很小，这样的话生成的随机数质量不高，且速度也较慢。因此经过网上查找，最后选择使用mt19937来生成随机数。mt19937可生成32位数字，具有随机数范围大，随机性好，占用内存较小，产生随机数的速度快，周期长等优点。

代码实现：

```
class MT19937
{
public:
    int extractNumber()
    {
        int y;
        if (index >= N)
        {
            twist();
            index = 0;
        }
        y = mt[index];
        y ^= (y >> U) & D;
        y ^= (y << S) & B;
        y ^= (y << T) & C;
        y ^= (y >> L);
        index = index + 1;
        return y;
    }
    void seedMt(const int seed)
    {
        mt[0] = seed;
        for (int i = 1; i < N; i++)
        {
            mt[i] = (F * (mt[i - 1] ^ (mt[i - 1] >> 30)) + i);
        }
        index = N;
    }
private:
    enum
    {
        W = 32,
        N = 624,
        M = 397,
        R = 31,
```

```
        A = 0x9908B0DF,
        U = 11,
        D = 0xFFFFFFFF,
        S = 7,
        B = 0x9D2C5680,
        T = 15,
        C = 0xEFC60000,
        L = 18,
        F = 1812433253,
        MASK_LOWER = (1ull << R) - 1,
        MASK_UPPER = (1ull << R)
    };
    void twist()
    {
        int i, x, xA;
        for (i = 0; i < N; i++)
        {
            x = (mt[i] & MASK_UPPER) + (mt[(i + 1) % N] & MASK_LOWER);
            xA = x >> 1;
            if (x & 0x1)
            {
                xA ^= A;
            }
            mt[i] = mt[(i + M) % N] ^ xA;
        }
    }
    int mt[N] = { 0 };
    int index = N + 1;
};
```

参考文献：

http://www.cplusplus.com/reference/random/mt19937/
https://blog.csdn.net/caimouse/article/details/55668071
http://www.voidcn.com/article/p-zvocpawr-gc.html
https://blog.csdn.net/lockey23/article/details/77435752/

2.2 辅助函数设计

系统中设计了 creatRandomNum 函数，来生成指定数量的随机数。swap 函数用来将两个数据进行位置上的交换，copy 函数用来进行随机数序列的复制，check 函数用来检查排序完后的序列是否满足从小到大排列的性质。

代码实现：

```
void creatRandomNum(int num)
{
    MT19937 M;
    M.seedMt(random_device{}());
    for (int i = 0; i < num; ++i)
    {
        randomArray[i] = M.extractNumber();
    }
}

void swap(int& x, int& y)
{
    exchangeNum++;
    int temp = x;
    x = y;
    y = temp;
}

void copy(int num)
{
    for (int i = 0; i < num; i++)
    {
        tempArray[i] = randomArray[i];
    }
}

bool check(int tempArray[], int length)
{
    int i;
    for (i = 0; i < length - 1; i++)
    {
        if (tempArray[i] > tempArray[i + 1])
        {
            cout << "排序出错！" << endl;
            return false;
        }
    }
    cout << "排序完成！" << endl;
    return true;
}
```

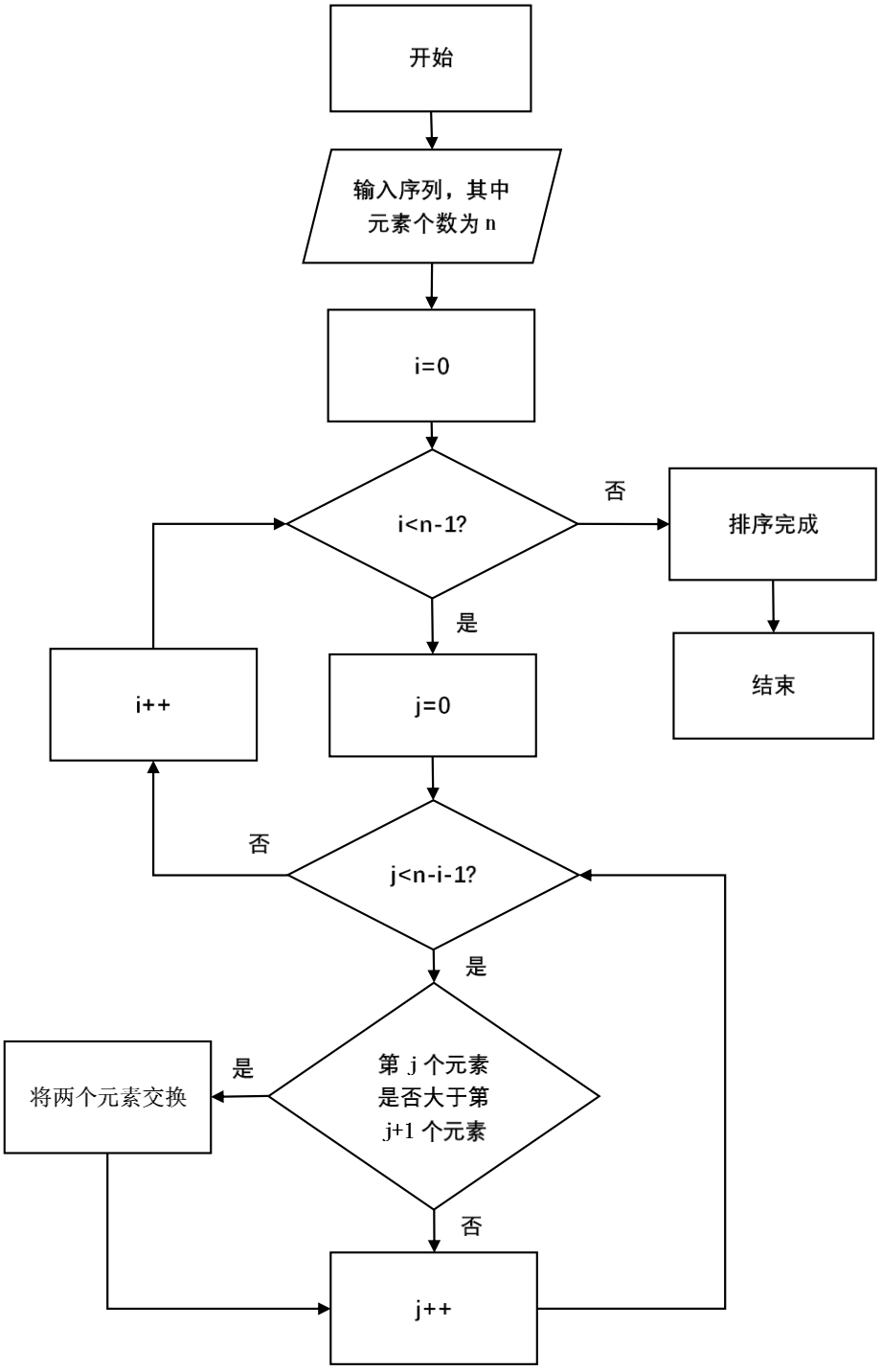
2.3 系统设计

系统在使用时，会生成菜单，用户根据菜单中的指示。通过输入操作码来完成对应的排序算法。同时系统也会记录本次算法所使用的时间，数据交换次数等信息并显示在屏幕上。

3 实现

3.1 冒泡排序

3.1.1 冒泡排序流程图（含优化）



3.1.2 冒泡排序核心代码

```
void bubbleSort(int tempArray[], int length)
{
    int i, j;
    bool flag = false;
    for (i = 0; i < length - 1; i++)
    {
        flag = false;
        for (j = 0; j < length - i - 1; j++)
        {
            compareNum++;
            if (tempArray[j] > tempArray[j + 1])
            {
                swap(tempArray[j], tempArray[j + 1]);
                flag = true;
            }
        }
        if (flag == false)
        {
            return;
        }
    }
}
```

3.1.3 冒泡排序分析

排序过程：

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。

算法分析：

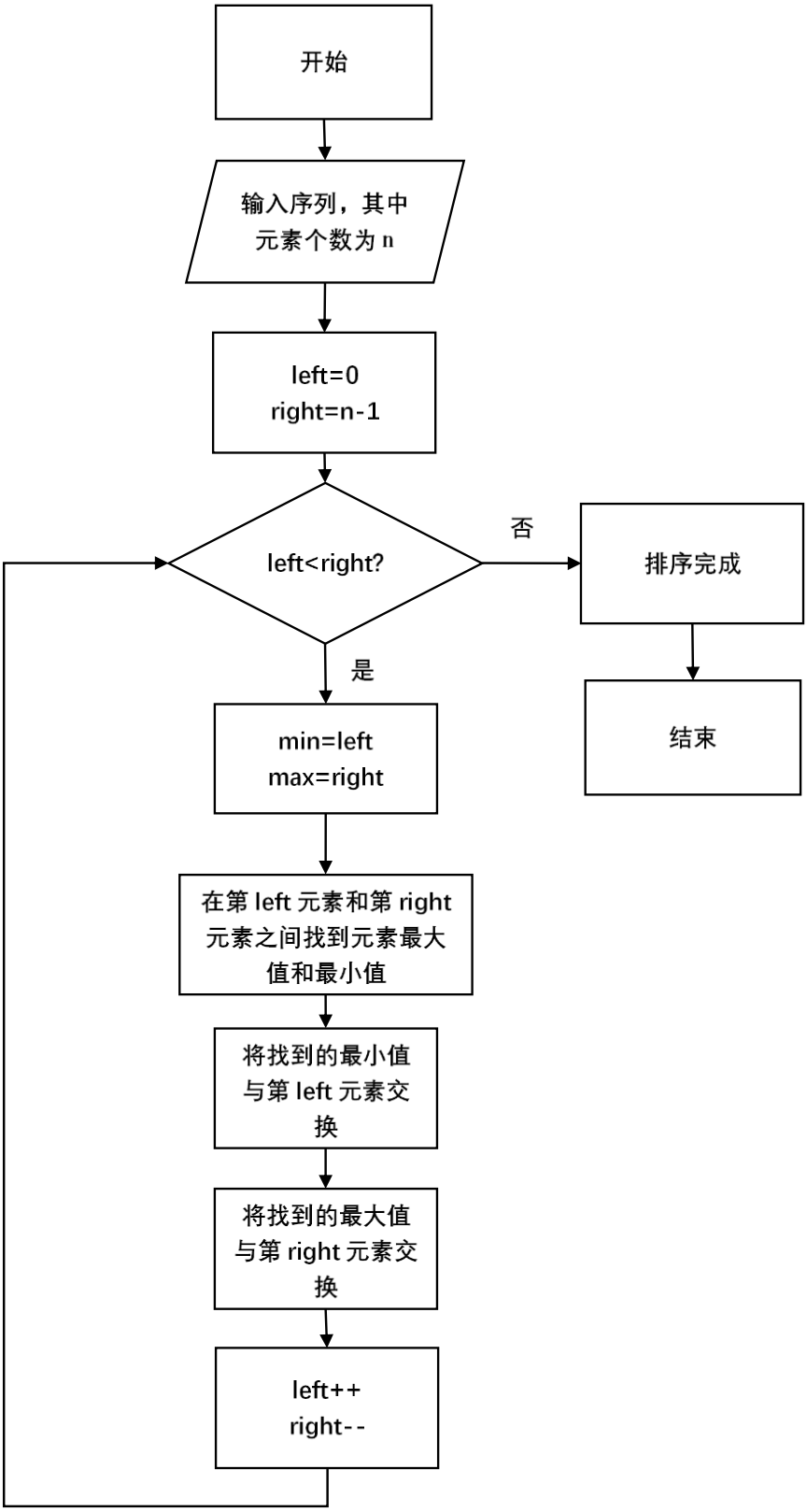
1. 冒泡排序的时间复杂度为$O(n^2)$
2. 冒泡排序的空间复杂度为O(1)
3. 冒泡排序是一种稳定的排序算法。

优化方法：

不一定每个序列都需要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flag，如果在一次排列中发生了元素交换，flag就为true。如果执行完一次排序后flag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。

3.2 选择排序

3.2.1 选择排序流程图（含优化）



3.2.2 选择排序核心代码

```
void selectionSort(int tempArray[], int length)
{
    int left = 0;
    int right = length - 1;
    int min = left;
    int max = right;
    while (left < right)
    {
        min = left;
        max = right;
        for (int i = left; i <= right; i++)
        {
            compareNum += 2;
            if (tempArray[min] > tempArray[i])
            {
                min = i;
            }
            if (tempArray[max] < tempArray[i])
            {
                max = i;
            }
        }
        if (min != left)
        {
            swap(tempArray[min], tempArray[left]);
        }
        if (left == max)
        {
            max = min;
        }
        if (max != right)
        {
            swap(tempArray[max], tempArray[right]);
        }
        left++;
        right--;
    }
}
```

3.2.3 选择排序分析

排序过程：

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。

算法分析：

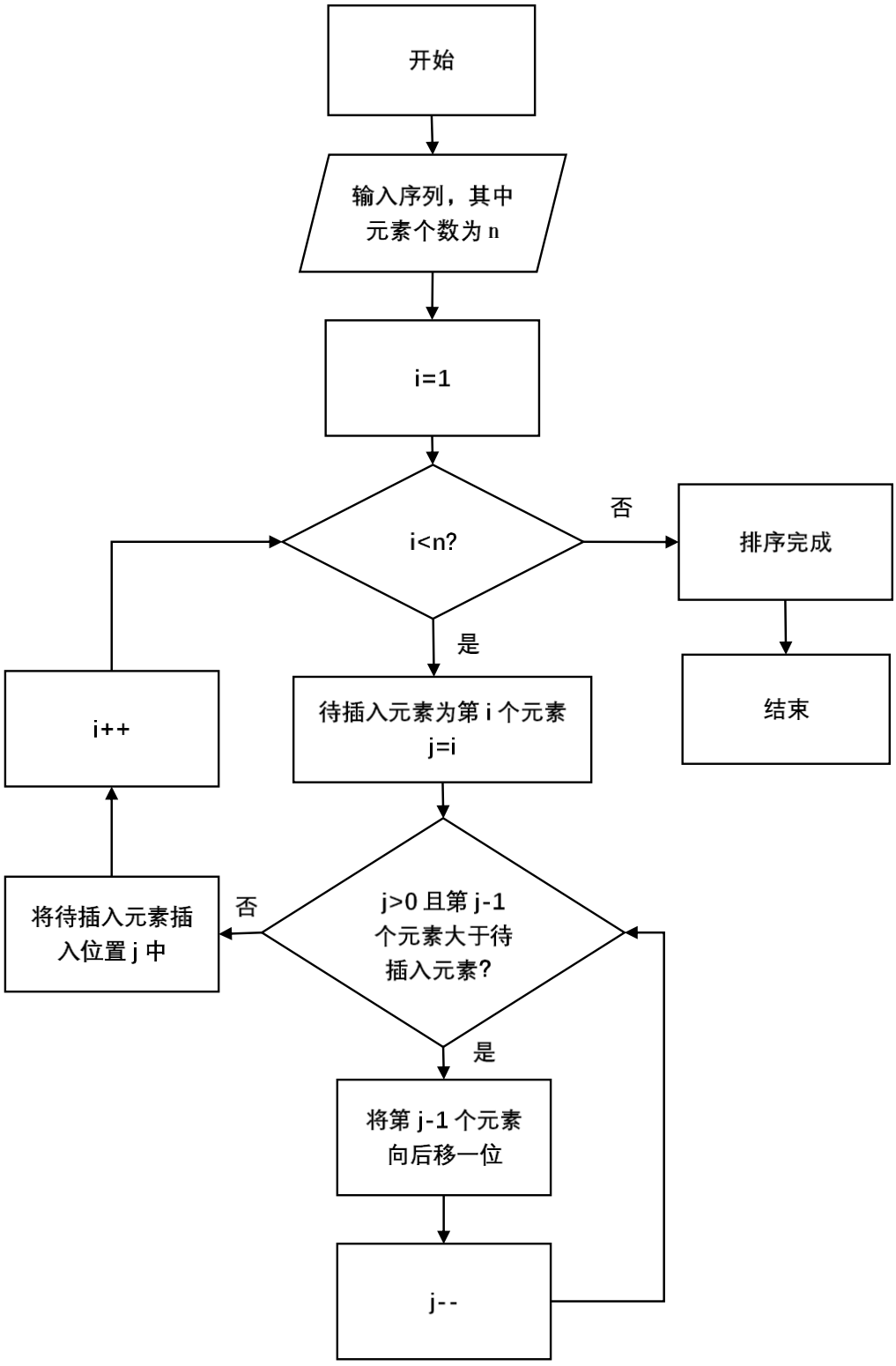
1. 选择排序的时间复杂度为 $O(n^2)$
2. 选择排序的空间复杂度为 O(1)
3. 选择排序是一种不稳定的排序算法。

优化方法：

可以一次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。

3.3 直接插入排序

3.3.1 直接插入排序流程图



3.3.2 直接插入排序核心代码

```
void insertionSort(int tempArray[], int length)
{
    int i, j;
    int temp = 0;
    for (i = 1; i < length; i++)
    {
        temp = tempArray[i];
        for (j = i; j > 0 && tempArray[j - 1] > temp; j--)
        {
            compareNum++;
            exchangeNum++;
            tempArray[j] = tempArray[j - 1];
        }
        compareNum++;
        tempArray[j] = temp;
    }
}
```

3.3.3 直接插入排序分析

排序过程：

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）。

算法分析：

1. 直接插入排序的时间复杂度为$O(n^2)$
2. 直接插入排序的空间复杂度为O(1)
3. 直接插入排序是一种稳定的排序算法。

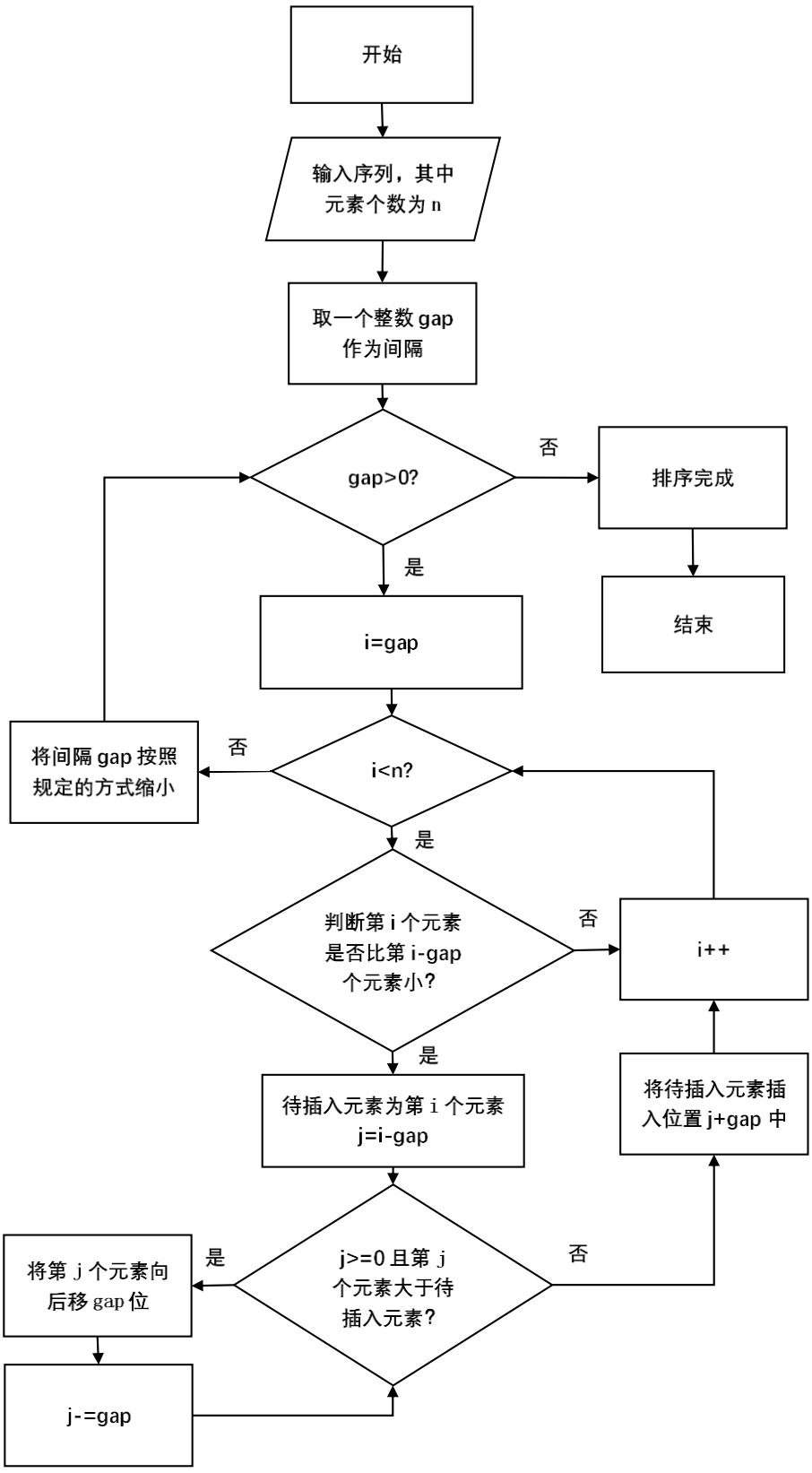
优化方法：

由于本排序方法题目中要求为直接插入排序，因此未加优化，如果要优化方法的话，可以使用折半插入的方法。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。

插入排序的另一种优化就是希尔排序，之后会提到。

3.4 希尔排序

3.4.1 希尔排序流程图



3.4.2 希尔排序核心代码

```
void shellSort(int tempArray[], int length)//使用了 sedgewick 排序
{
    int Sedgewick[29] =
    { 0, 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001,
    36289, 64769,146305, 260609, 587521, 1045505, 2354689, 4188161,
    9427969, 16764929,37730305, 67084289, 150958081, 268386305,
    603906049, 1073643521 };
    int i, j;
    int temp = 0;
    int gap = 0;
    int initGap = 0;
    while (Sedgewick[initGap] < length / 2)
    {
        initGap++;
    }
    gap = Sedgewick[initGap];
    while (gap > 0)
    {
        for (i = gap; i < length; i++)
        {
            compareNum++;
            if (tempArray[i] < tempArray[i - gap])
            {
                temp = tempArray[i];
                for (j = i - gap; j >= 0 && temp < tempArray[j]; j -
= gap)
                {
                    compareNum++;
                    exchangeNum++;
                    tempArray[j + gap] = tempArray[j];
                }
                compareNum++;
                tempArray[j + gap] = temp;
            }
        }
        initGap--;
        gap = Sedgewick[initGap];
    }
}
```

3.4.3 希尔排序分析

排序过程：

待排序元素序列有n个元素，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。

算法分析：

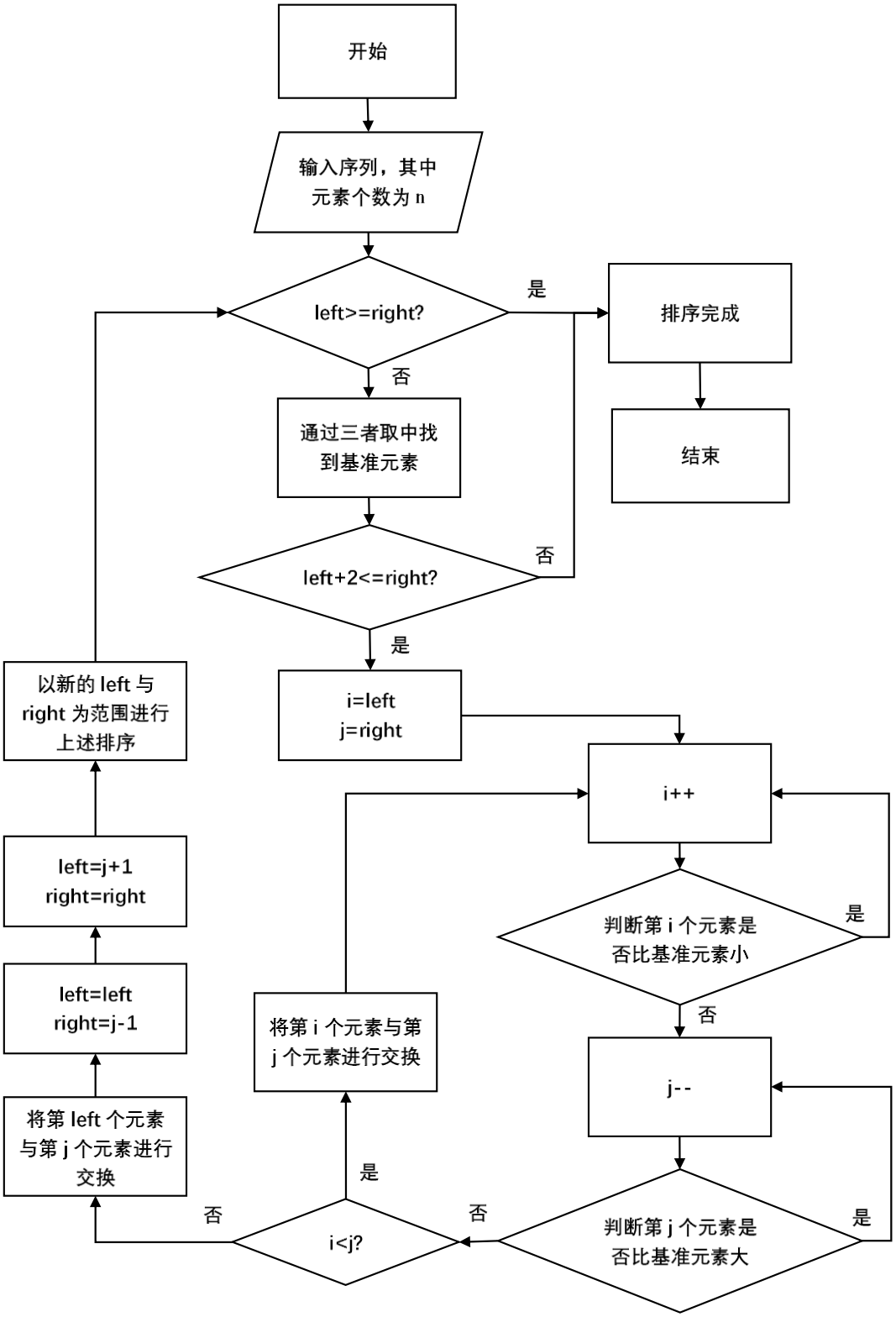
1. 应用不同的序列会使希尔排序的性能存在有很大差异，本系统中选择的是Sedgewick序列，其平均时间复杂度为$O(n^{\frac{7}{6}})$，最坏时间复杂度为$O(n^{\frac{4}{3}})$。
2. 希尔排序是一种不稳定的排序算法。

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$O(n^{\frac{5}{4}})$，最坏时间复杂度为$O(n^{\frac{3}{2}})$。本系统中选择的Sedgewick序列是已知比较好的增量序列，其平均时间复杂度为$O(n^{\frac{7}{6}})$，最坏时间复杂度为$O(n^{\frac{4}{3}})$。

3.5 快速排序

3.5.1 快速排序流程图（含优化）



3.5.2 快速排序核心代码

```
void partitionMedianOfThree(int tempArray[], int left, int right)
{
    compareNum += 3;
    int middle = (left + right) / 2;
    if (tempArray[right] < tempArray[middle])
    {
        swap(tempArray[right], tempArray[middle]);
    }
    if (tempArray[right] < tempArray[left])
    {
        swap(tempArray[right], tempArray[left]);
    }
    if (tempArray[left] < tempArray[middle])
    {
        swap(tempArray[left], tempArray[middle]);
    }
}

void qSort(int tempArray[], int left, int right)
{
    int i = 0, j = 0;
    int pivot = 0;
    if (left >= right)
    {
        return;
    }
    partitionMedianOfThree(tempArray, left, right);
    pivot = tempArray[left];
    if (left + 2 <= right)
    {
        i = left;
        j = right;
        while (true)
        {
            while (tempArray[++i] < pivot)
            {
                compareNum++;
            }
            compareNum++;
            while (tempArray[--j] > pivot)//[j]一定大
            {
                compareNum++;
```

```
            }
            compareNum++;
            if (i < j)
            {
                swap(tempArray[i], tempArray[j]);
            }
            else break;
        }
        swap(tempArray[j], tempArray[left]);
        qSort(tempArray, left, j - 1);
        qSort(tempArray, j + 1, right);
    }
}

void quickSort(int tempArray[], int length)
{
    qSort(tempArray, 0, length - 1);
}
```

3.5.3 快速排序分析

排序过程：

待排序元素序列有n个元素，首先找到当前第left个元素到第right个元素排序范围内的一个基准元素，对整个元素序列进行划分，把排序范围内比基准元素小的都放在左边，比基准元素大的都放在右边，基准元素放在中间。这样一次下来就得到左右两个子序列，再将子序列按照上述操作进行排列，直至所有元素排列完成为止。

算法分析：

1. 快速排序的平均时间复杂度为$O(n\log_2 n)$。
2. 快速排序的空间复杂度为$O(\log_2 n)$。
3. 快速排序是一种不稳定的排序算法。

优化方法：

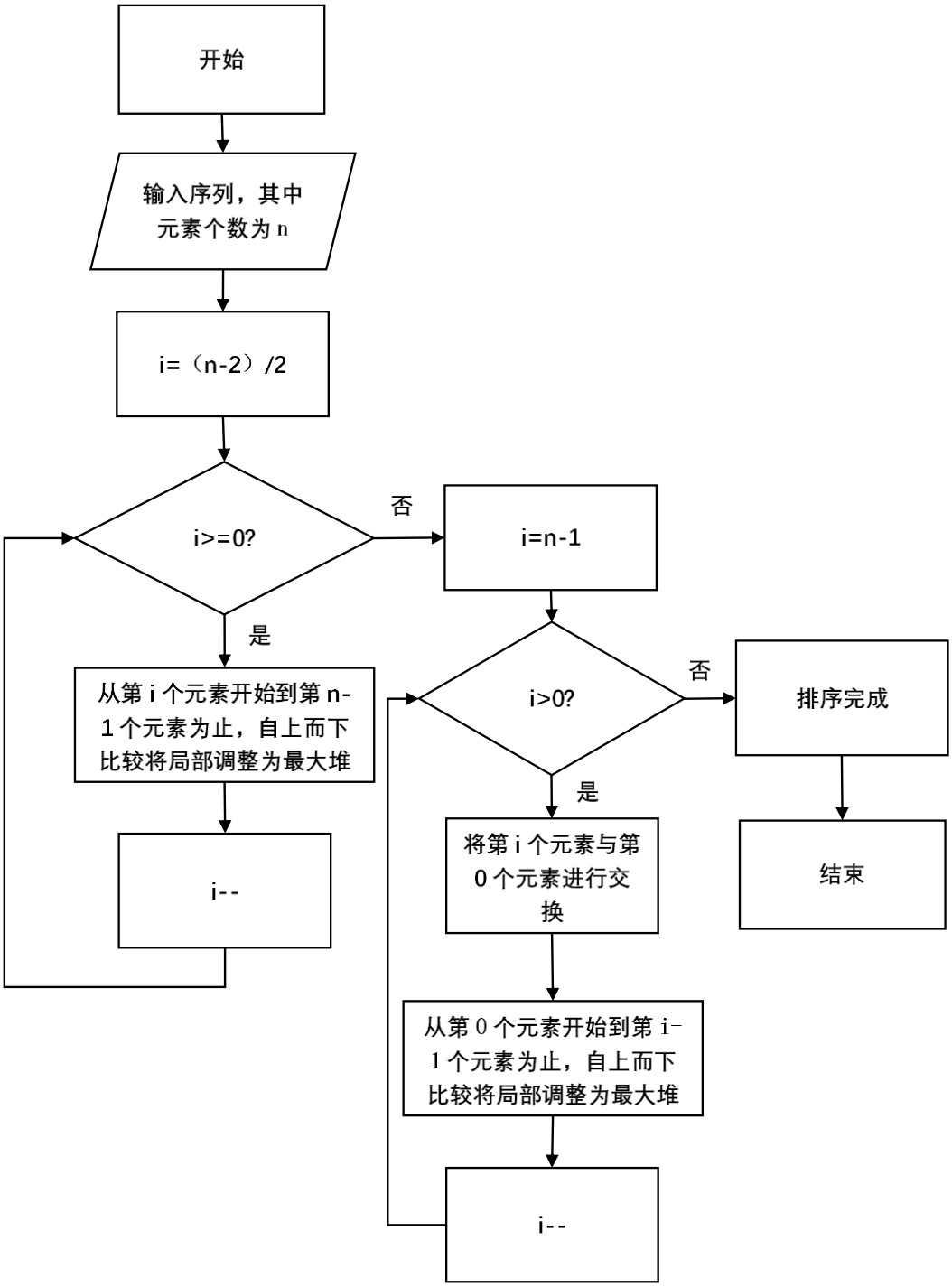
在快速排序算法中，每次划分时用于比较的基准元素的选择对于算法的性能有很大影响。如果基准元素选择不合适，会导致算法性能的大幅度退化。如果要可以合理选择基准元素，使得每次划分所得的两个子序列中的元素个数尽可能地接近，就可以加速排序速度。因此，我们可以使用三者取中的算法对快速排序进行优化。

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%。

3.6 堆排序

3.6.1 堆排序流程图



3.6.2 堆排序核心代码

```
void siftDown(int tempArray[], int start, int max)
{
    int current = start;
    int lessChild = 2 * current + 1;
    int temp = tempArray[current];
    while (lessChild <= max)
    {
        if (lessChild < max)
        {
            compareNum++;
            if (tempArray[lessChild] < tempArray[lessChild + 1])
            {
                lessChild++;
            }
        }
        compareNum++;
        if (temp >= tempArray[lessChild])
        {
            break;
        }
        else
        {
            exchangeNum++;
            tempArray[current] = tempArray[lessChild];
            current = lessChild;
            lessChild = 2 * lessChild + 1;
        }
    }
    tempArray[current] = temp;
}

void heapSort(int tempArray[], int length)
{
    int i;
    for (i = (length -2)/ 2; i >= 0; i--)
    {
        siftDown(tempArray, i, length - 1);
    }
    for (i = length - 1; i > 0; i--)
    {
        swap(tempArray[0], tempArray[i]);
        siftDown(tempArray, 0, i - 1);
```

```
    }
}
```

3.6.3 堆排序分析

排序过程：

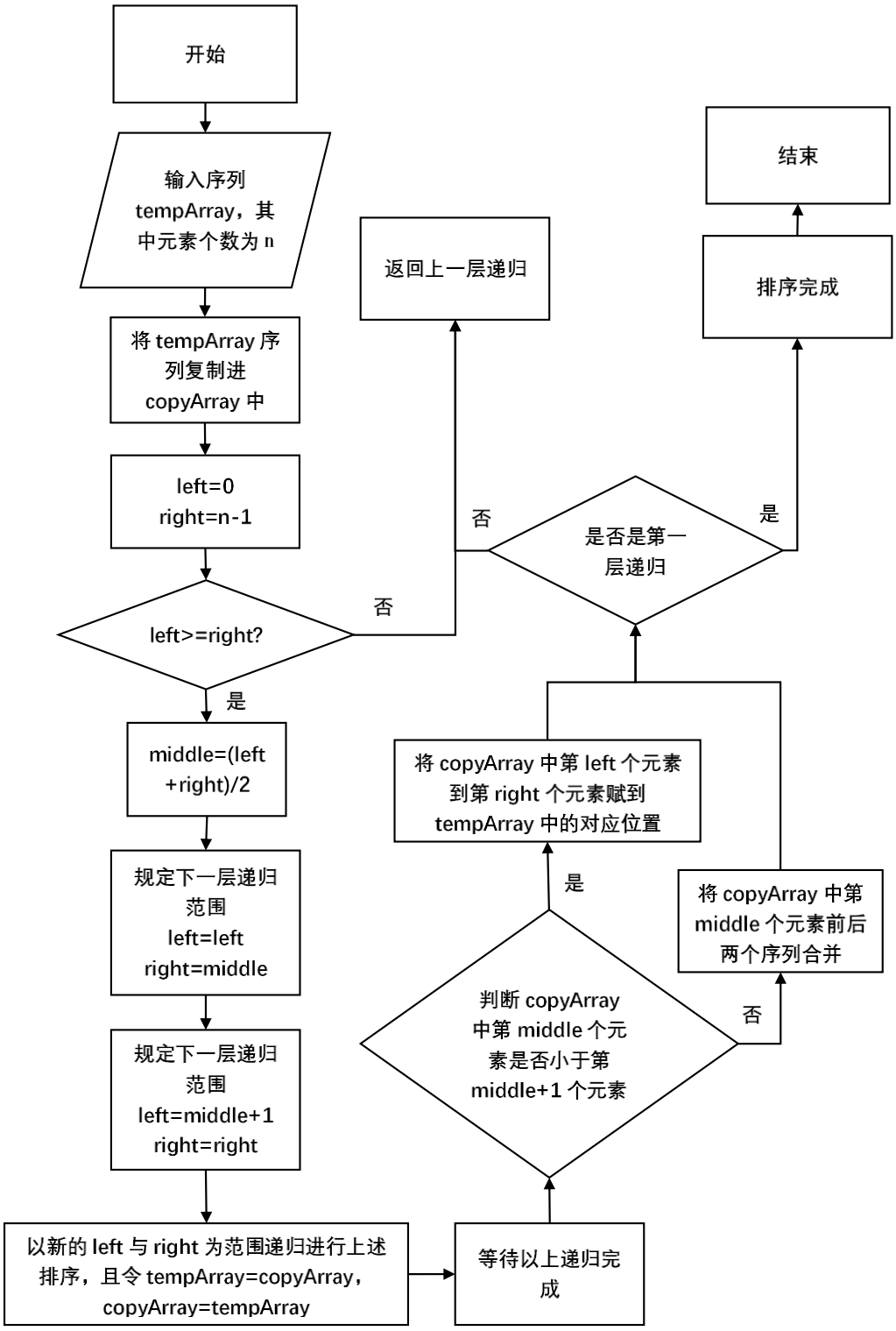
待排序元素序列有n个元素，首先先用调整算法siftDown()将堆初始化为最大堆，siftDown()的作用是从结点i开始到n-1为止，自上而下比较，如果子女的值小于双亲结点的值，就把它们相互交换，这样就可以将一个局部调整为最大堆，当i取遍从0到（n-2）/2的所有整数后，整个堆就被初始化为最大堆。随后将堆中最大元素，即第0个元素与第n-1个元素进行交换，之后使用算法siftDown()将从第0个元素到第n-2个元素重新调整为最大堆，此时最大元素就被成功安排在了序列的最后一位；再交换第0个元素与第n-2个元素，之后使用算法siftDown()将从第0个元素到第n-3个元素重新调整为最大堆……反复执行，最后就会得到排序完成的元素序列。

算法分析：

1. 堆排序的平均时间复杂度为$O(n\log_2 n)$。
2. 堆排序的空间复杂度为O(1)。
3. 堆排序是一种不稳定的排序算法。

3.7 归并排序

3.7.1 归并排序流程图（含优化）



3.7.2 归并排序核心代码

```
void merge(int tempArray[], int copyArray[], int left, int middle, int
right)
{
    int temp1 = left;
    int temp2 = middle + 1;
    int current = left;
    while (temp1 <= middle && temp2 <= right)
    {
        exchangeNum++;
        compareNum++;
        if (copyArray[temp1] <= copyArray[temp2])
        {
            tempArray[current++] = copyArray[temp1++];
        }
        else
        {
            tempArray[current++] = copyArray[temp2++];
        }
    }
    while (temp1 <= middle)
    {
        tempArray[current++] = copyArray[temp1++];
        exchangeNum++;
    }
    while (temp2 <= right)
    {
        tempArray[current++] = copyArray[temp2++];
        exchangeNum++;
    }
}

void mSort(int tempArray[], int copyArray[], int left, int right)
{
    if (left >= right)
    {
        return;
    }
    int middle = (left + right) / 2;
    mSort(copyArray, tempArray, left, middle);
    mSort(copyArray, tempArray, middle + 1, right);
    compareNum++;
    if (copyArray[middle] <= copyArray[middle + 1])
```

```
    {
        for (int i = left; i <= right; i++)
        {
            tempArray[i] = copyArray[i];
            exchangeNum++;
        }
        return;
    }
    merge(tempArray, copyArray, left, middle, right);
}

void mergeSort(int tempArray[], int length)
{
    int* copyArray = new int[length];
    for (int i = 0; i < length; i++)
    {
        copyArray[i] = tempArray[i];
        exchangeNum++;
    }
    mSort(tempArray, copyArray, 0, length - 1);
    delete[] copyArray;
    copyArray = nullptr;
}
```

3.7.3 归并排序分析

排序过程：

待排序元素序列有n个元素，首先把长度为n的输入序列分成两个长度为n/2的子序列，再将n/2的子序列分为两个长度为n/4的子序列……不断进行递归，直到子序列的长度为1，然后递归开始回升进行排序，每层递归将两个子序列合并排好序后返回上一层递归继续此操作，这样就使得每一层在进行子序列合并时，这两个子序列都是已经排好序的序列。当递归结束，序列就已经排好序。

算法分析：

1. 归并排序的平均时间复杂度为$O(n\log_2 n)$。
2. 归并排序的空间复杂度为$O(n)$。
3. 归并排序是一种稳定的排序算法。

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。

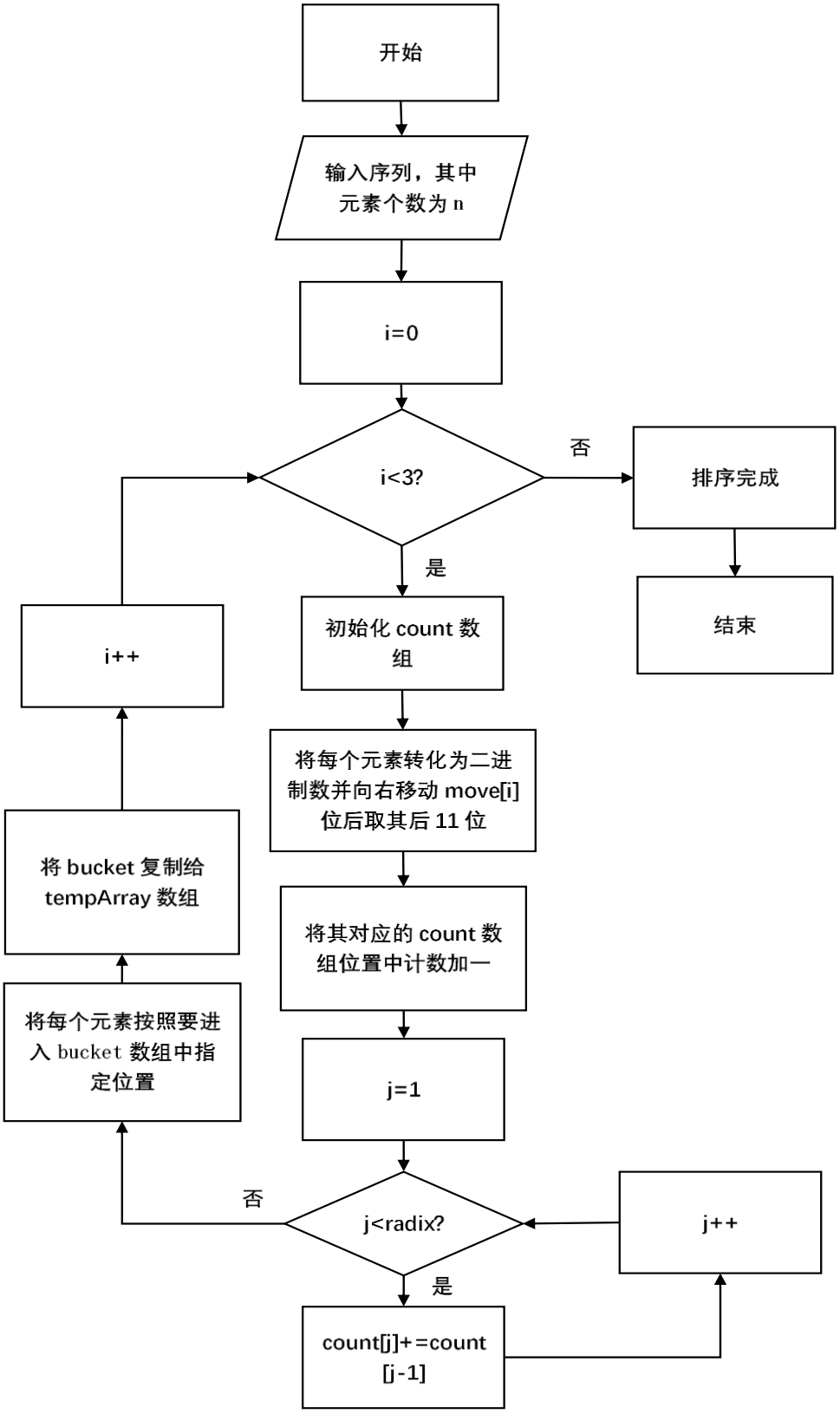
例如：5 6 4 7 1 3 8 2

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。

3.8 基数排序

3.8.1 基数排序流程图（含优化）



3.8.2 基数排序核心代码

```
int getPart(int curData, int move)
{
    return (curData >> move) & 2047;
}

void radixSort(int tempArray[], int length)//基数 2048
{
    int radix = 2048;
    int move[4] = { 0,11,22 };
    int* bucket = new int[length];
    int* count = new int[radix];
    int i, j, k;
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < radix; j++)
        {
            count[j] = 0;
        }
        for (j = 0; j < length; j++)
        {
            count[getPart(tempArray[j], move[i])]++;
        }
        for (j = 1; j < radix; j++)
        {
            count[j] += count[j - 1];
        }
        for (j = length - 1; j >= 0; j--)
        {
            int k = getPart(tempArray[j], move[i]);
            bucket[count[k] - 1] = tempArray[j];
            exchangeNum++;
            count[k]--;
        }
        for (j = 0; j < length; j++)
        {
            tempArray[j] = bucket[j];
            exchangeNum++;
        }
    }
    delete[] bucket;
    delete[] count;
}
```

3.8.3 基数排序分析

排序过程：

待排序元素序列有n个元素，首先选择基数，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。

算法分析：

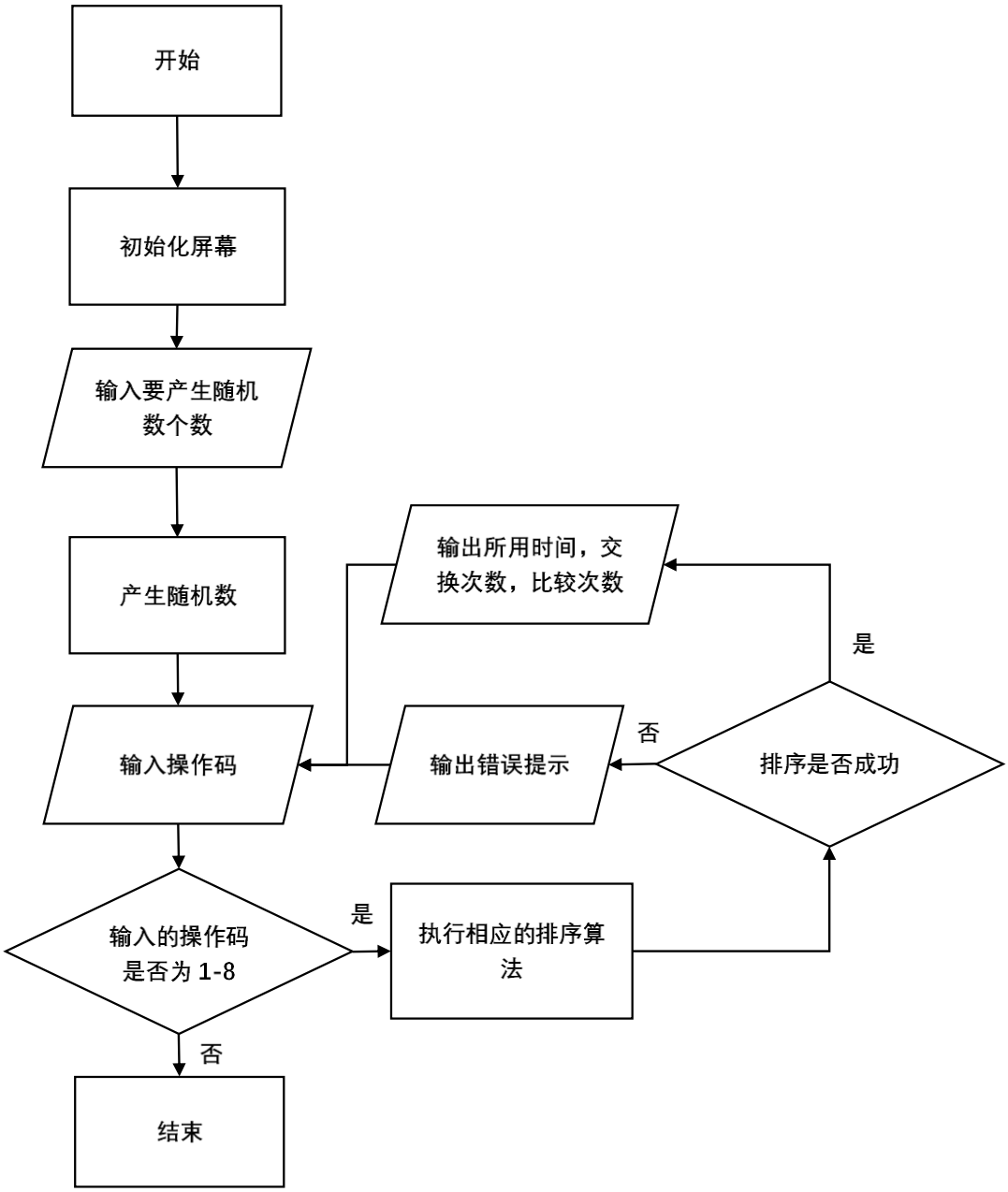
1. 基数排序的平均时间复杂度为$O((n+radix)d)$，d为进行“分配”与“收集”的趟数。
2. 基数排序的空间复杂度为$O(n+radix)$
3. 基数排序是一种稳定的排序算法。

优化方法：

在选择基数时我们总会习惯性的以10的幂作为基数，这与我们平时多用10进制运算相符合。但是，计算机是以二进制存储数据的，如果选择2的幂作为基数，可以通过位运算来简化代码。比如在本项目中，由于int类型的范围为-2^31——2^31-1，因此采用基数为2048，这样的话10进制中的除法操作与取模操作就会变成右移操作和与操作。由于2047的二进制表示为11111111111共11位，因此只需要进行三趟就可以覆盖所有的int类型数据。三次分别取元素的低11位，中间11位与高11位进行排序操作，实现方法为将元素进行右移操作后与2047做与运算，这样的话输出的结果即为需要的位数。

3.8 总体功能的实现

3.8.1 总体功能流程图



3.8.2 总体功能核心代码

```
int main()
{
    menu();
    int num = 0;
    int choice = 0;
    string sortName = "";
    clock_t start = 0, end = 0;
    cout << "请输入要产生的随机数个数：";
    cin >> num;
    creatRandomNum(num);
    while (true)
    {
        cout << endl;
        cout << "请选择排序算法：" << "\t";
        cin >> choice;
        copy(num);
        sortName = "";
        exchangeNum = 0;
        compareNum = 0;
        start = clock();
        switch (choice)
        {
        case 1:
            bubbleSort(tempArray, num);
            sortName = "冒泡排序";
            break;
        case 2:
            selectionSort(tempArray, num);
            sortName = "选择排序";
            break;
        case 3:
            insertionSort(tempArray, num);
            sortName = "直接插入排序";
            break;
        case 4:
            shellSort(tempArray, num);
            sortName = "希尔排序";
            break;
        case 5:
            quickSort(tempArray, num);
            sortName = "快速排序";
            break;
```

```
        case 6:
            heapSort(tempArray, num);
            sortName = "堆排序";
            break;
        case 7:
            mergeSort(tempArray, num);
            sortName = "归并排序";
            break;
        case 8:
            radixSort(tempArray, num);
            sortName = "基数排序";
            break;
        default:
            return 0;
        }
        end = clock();
        cout << endl;
        if (check(tempArray, num))
        {
            cout << sortName << "所用时间是" << "\t"
                 << end - start << "毫秒" << endl;
            cout << sortName << "交换次数是" << "\t"
                 << exchangeNum << "次" << endl;
            cout << sortName << "比较次数是: " << "\t"
                 << compareNum << "次" << endl;
        }
    }
    return 0;
}
```

3.8.3 总体功能截屏示例

```
请输入要产生的随机数个数：20

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    0毫秒
冒泡排序交换次数是：    135次
冒泡排序比较次数是：    190次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    0毫秒
选择排序交换次数是：    15次
选择排序比较次数是：    220次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：       0毫秒
直接插入排序交换次数是：       135次
直接插入排序比较次数是：       154次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    0毫秒
希尔排序交换次数是：    45次
希尔排序比较次数是：    102次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    0毫秒
快速排序交换次数是：    29次
快速排序比较次数是：    96次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      0毫秒
堆排序交换次数是：      65次
堆排序比较次数是：      105次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    0毫秒
归并排序交换次数是：    108次
归并排序比较次数是：    72次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    0毫秒
基数排序交换次数是：    120次
基数排序比较次数是：    0次

请选择排序算法：        9
```

4 测试

4.1 小范围测试

4.1.1 100个随机数

```
请输入要产生的随机数个数：100

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    0毫秒
冒泡排序交换次数是：    2562次
冒泡排序比较次数是：    4884次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    0毫秒
选择排序交换次数是：    96次
选择排序比较次数是：    5100次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：       0毫秒
直接插入排序交换次数是：       2562次
直接插入排序比较次数是：       2661次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    0毫秒
希尔排序交换次数是：    480次
希尔排序比较次数是：    1017次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    0毫秒
快速排序交换次数是：    202次
快速排序比较次数是：    711次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      0毫秒
堆排序交换次数是：      582次
堆排序比较次数是：      1024次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    0毫秒
归并排序交换次数是：    772次
归并排序比较次数是：    616次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    0毫秒
基数排序交换次数是：    600次
基数排序比较次数是：    0次
```

4.1.2 1000个随机数

```
请输入要产生的随机数个数：1000

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    1毫秒
冒泡排序交换次数是：    254525次
冒泡排序比较次数是：    499434次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    0毫秒
选择排序交换次数是：    993次
选择排序比较次数是：    501000次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：        1毫秒
直接插入排序交换次数是：        254525次
直接插入排序比较次数是：        255524次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    0毫秒
希尔排序交换次数是：    8055次
希尔排序比较次数是：    17602次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    0毫秒
快速排序交换次数是：    2663次
快速排序比较次数是：    11435次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      0毫秒
堆排序交换次数是：      9101次
堆排序比较次数是：      16929次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    0毫秒
归并排序交换次数是：    10976次
归并排序比较次数是：    9341次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    0毫秒
基数排序交换次数是：    6000次
基数排序比较次数是：    0次
```

4.1.3 10000个随机数

```
请输入要产生的随机数个数：10000

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    120毫秒
冒泡排序交换次数是：    24952904次
冒泡排序比较次数是：    49975299次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    49毫秒
选择排序交换次数是：    9990次
选择排序比较次数是：    50010000次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：        30毫秒
直接插入排序交换次数是：        24952904次
直接插入排序比较次数是：        24962903次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    1毫秒
希尔排序交换次数是：    108118次
希尔排序比较次数是：    251261次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    1毫秒
快速排序交换次数是：    34488次
快速排序比较次数是：    152633次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      0毫秒
堆排序交换次数是：      124272次
堆排序比较次数是：      235401次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    1毫秒
归并排序交换次数是：    143616次
归并排序比较次数是：    126610次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    0毫秒
基数排序交换次数是：    60000次
基数排序比较次数是：    0次
```

4.1.4 100000个随机数

```
请输入要产生的随机数个数：100000

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    12736毫秒
冒泡排序交换次数是：    2500856345次
冒泡排序比较次数是：    4999814540次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    4847毫秒
选择排序交换次数是：    99990次
选择排序比较次数是：    5000100000次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：        3151毫秒
直接插入排序交换次数是：        2500856345次
直接插入排序比较次数是：        2500956344次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    13毫秒
希尔排序交换次数是：    1410203次
希尔排序比较次数是：    3342626次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    7毫秒
快速排序交换次数是：    426761次
快速排序比较次数是：    1916513次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      9毫秒
堆排序交换次数是：      1574849次
堆排序比较次数是：      3019500次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    12毫秒
归并排序交换次数是：    1768928次
归并排序比较次数是：    1595172次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    2毫秒
基数排序交换次数是：    600000次
基数排序比较次数是：    0次
```

4.2 大范围测试

不包含冒泡排序，选择排序，直接插入排序

4.2.1 1000000 个随机数

```
请输入要产生的随机数个数：1000000

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    165毫秒
希尔排序交换次数是：    17447549次
希尔排序比较次数是：    41696328次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    76毫秒
快速排序交换次数是：    4984719次
快速排序比较次数是：    24391554次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      115毫秒
堆排序交换次数是：      19046647次
堆排序比较次数是：      36792253次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    137毫秒
归并排序交换次数是：    20951424次
归并排序比较次数是：    19321615次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    12毫秒
基数排序交换次数是：    6000000次
基数排序比较次数是：    0次
```

4.2.2 10000000 个随机数

```
请输入要产生的随机数个数：10000000

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    2046毫秒
希尔排序交换次数是：    209740562次
希尔排序比较次数是：    502950950次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    894毫秒
快速排序交换次数是：    58381304次
快速排序比较次数是：    269399844次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      1820毫秒
堆排序交换次数是：      223835894次
堆排序比较次数是：      434640523次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    1609毫秒
归并排序交换次数是：    243222784次
归并排序比较次数是：    226252617次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    142毫秒
基数排序交换次数是：    60000000次
基数排序比较次数是：    0次
```

4.2.3 100000000 个随机数

```
请输入要产生的随机数个数：100000000

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    24897毫秒
希尔排序交换次数是：    2614570268次
希尔排序比较次数是：    6088637199次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    10104毫秒
快速排序交换次数是：    663441577次
快速排序比较次数是：    3074320066次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      28159毫秒
堆排序交换次数是：      2571581487次
堆排序比较次数是：      5012899030次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    18150毫秒
归并排序交换次数是：    2765782272次
归并排序比较次数是：    2591549257次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    1542毫秒
基数排序交换次数是：    600000000次
基数排序比较次数是：    0次
```

4.3 重复元素序列测试

4.3.1 100000 个随机数

```
请输入要产生的随机数个数：100000

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    0毫秒
冒泡排序交换次数是：    0次
冒泡排序比较次数是：    99999次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    5808毫秒
选择排序交换次数是：    0次
选择排序比较次数是：    5000100000次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：        0毫秒
直接插入排序交换次数是：        0次
直接插入排序比较次数是：        99999次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    2毫秒
希尔排序交换次数是：    0次
希尔排序比较次数是：    1266128次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    16毫秒
快速排序交换次数是：    722845次
快速排序比较次数是：    1642295次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      5毫秒
堆排序交换次数是：      99999次
堆排序比较次数是：      299994次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    6毫秒
归并排序交换次数是：    1768928次
归并排序比较次数是：    99999次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    9毫秒
基数排序交换次数是：    600000次
基数排序比较次数是：    0次
```

4.4 升序序列测试

4.4.1 100000 个随机数

```
请输入要产生的随机数个数：100000

请选择排序算法：        1

排序完成！
冒泡排序所用时间是：    1毫秒
冒泡排序交换次数是：    0次
冒泡排序比较次数是：    99999次

请选择排序算法：        2

排序完成！
选择排序所用时间是：    5793毫秒
选择排序交换次数是：    0次
选择排序比较次数是：    5000100000次

请选择排序算法：        3

排序完成！
直接插入排序所用时间是：        1毫秒
直接插入排序交换次数是：        0次
直接插入排序比较次数是：        99999次

请选择排序算法：        4

排序完成！
希尔排序所用时间是：    3毫秒
希尔排序交换次数是：    0次
希尔排序比较次数是：    1266128次

请选择排序算法：        5

排序完成！
快速排序所用时间是：    5毫秒
快速排序交换次数是：    68928次
快速排序比较次数是：    1668944次

请选择排序算法：        6

排序完成！
堆排序所用时间是：      14毫秒
堆排序交换次数是：      1650854次
堆排序比较次数是：      3112517次

请选择排序算法：        7

排序完成！
归并排序所用时间是：    7毫秒
归并排序交换次数是：    1768928次
归并排序比较次数是：    99999次

请选择排序算法：        8

排序完成！
基数排序所用时间是：    10毫秒
基数排序交换次数是：    600000次
基数排序比较次数是：    0次
```

5 案例总结

1. 直接插入排序，冒泡排序，选择排序是最基本的排序方法。它们平均情况下的时间复杂度都是 $O(n^2)$，这三种基本的排序方法除了一个辅助元素外，都不需要其他额外内存。从稳定性来看，直接插入排序与冒泡排序都是稳定的，选择排序不是。它们适用于元素个数不是很多的情况。
2. 冒泡排序数据比较次数和输入序列中个排序元素的初始排列无关，数据的移动次数和各排序元素的初始序列有关。在最优情况下只需要一次冒泡过程就能实现排序。
3. 直接插入排序的时间复杂度与待排序序列的初始排列有关，在最好情况下，直接插入排序只需要 n-1 次比较操作就可以完成，不需要交换操作。
4. 选择排序的数据比较次数和输入序列中个排序元素的初始排列无关，数据的移动次数和各排序元素的初始序列有关。最优情况下，例如升序和重复序列时，一次也不用移动。
5. 希尔排序的时间复杂度介于基本排序算法与高效算法之间，这主要取决于 gap 序列的选择。希尔排序是一种不稳定的排序算法，数据比较次数和移动次数与输入序列中个排序元素的初始排列有关。一般在元素个数在几千时，希尔排序是很好的选择。
6. 快速排序，堆排序，归并排序都是高效算法，适合于元素个数很大的情况。
7. 快速排序是最通用的排序算法，它的时间复杂度为 $O(n\log_2 n)$，所需额外内存为 $O(\log_2 n)$，它的数据移动次数相比于其他高效算法来说较少，是一种不稳定的排序算法。快速排序的效率在序列越乱的时候，效率越高。在某些情况下，例如在数据有序时，会退化成冒泡排序，时间复杂度增加至 $O(n^2)$。
   可以通过选择更好的基准元素来优化此排序。
8. 堆排序的时间复杂度是 $O(n\log_2 n)$，是一种不稳定的排序算法。它的效率相对稳定，不会因为某些原因使时间复杂度明显增加，是对数据的有序性不敏感的一种算法。并且堆排序也不需要额外的空间，相比于另外两种高效算法，不需要担心可能会发生堆栈溢出错误。但由于需要时刻进行堆的维护，因此实际应用中不如快速排序广泛。
9. 归并排序的时间复杂度是 $O(n\log_2 n)$，它的优点在于它是一种稳定的高效算法，但它需要 $O(n)$的附加空间。对于元素较多，且要求稳定性时，可以使用归并排序。
10. 基数排序是一种相对特殊的排序算法，它是将排序码的不同部分进行处理和比较。基数排序基于的排序码抽取算法受到系统和排序元素的影响，其适应性远不如普通的比较与排序，因此实际工作中使用不多。
11. 不同的排序算法有时可以集成起来，例如在归并排序中对小规模子数组使用插入排序可以使得运行时间缩短。