****

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**

**培养单位：软件学院**

**本 科 生：蓝 笙 聆**

**学 号：1951096**

**指导老师：张 颖**

二○二○年十二月

目录

[第1章 功能分析 1](#_Toc59791034)

[1.1 功能分析 1](#_Toc59791035)

[第2章 设计 2](#_Toc59791036)

[2.1 算法设计 2](#_Toc59791037)

[2.1.1 冒泡排序 2](#_Toc59791038)

[2.1.2 选择排序 2](#_Toc59791039)

[2.1.3 插入排序 2](#_Toc59791040)

[2.1.4 希尔排序 2](#_Toc59791041)

[2.1.5 快速排序 3](#_Toc59791042)

[2.1.6 归并排序 3](#_Toc59791043)

[2.1.7 堆排序 3](#_Toc59791044)

[2.1.8 基数排序 3](#_Toc59791045)

[2.2 系统设计 3](#_Toc59791046)

[第3章 实现 4](#_Toc59791047)

[3.1 冒泡排序功能的实现 4](#_Toc59791048)

[3.1.1 冒泡排序功能流程图 4](#_Toc59791049)

[3.1.2 冒泡排序功能核心代码 5](#_Toc59791050)

[3.1.3 冒泡排序功能截屏示例 5](#_Toc59791051)

[3.2 选择排序功能的实现 5](#_Toc59791052)

[3.2.1 选择排序功能流程图 5](#_Toc59791053)

[3.2.2 选择排序功能核心代码 6](#_Toc59791054)

[3.2.3 选择排序功能截屏示例 6](#_Toc59791055)

[3.3 插入排序功能的实现 6](#_Toc59791056)

[3.3.1 插入排序功能流程图 6](#_Toc59791057)

[3.3.2 插入排序功能核心代码 7](#_Toc59791058)

[3.3.3 插入排序功能截屏示例 7](#_Toc59791059)

[3.4 希尔排序功能的实现 7](#_Toc59791060)

[3.4.1 希尔排序功能流程图 7](#_Toc59791061)

[3.4.2 希尔排序功能核心代码 7](#_Toc59791062)

[3.4.3 希尔排序功能截屏示例 8](#_Toc59791063)

[3.5 快速排序功能的实现 8](#_Toc59791064)

[3.5.1 快速排序功能流程图 8](#_Toc59791065)

[3.5.2 快速排序功能核心代码 8](#_Toc59791066)

[3.5.3 快速排序功能截屏示例 9](#_Toc59791067)

[3.6 归并排序功能的实现 9](#_Toc59791068)

[3.6.1 归并排序功能流程图 9](#_Toc59791069)

[3.6.2 归并排序功能核心代码 9](#_Toc59791070)

[3.6.3 归并排序功能截屏示例 11](#_Toc59791071)

[3.7 堆排序功能的实现 11](#_Toc59791072)

[3.7.1 堆排序功能流程图 11](#_Toc59791073)

[3.7.2 堆排序功能核心代码 11](#_Toc59791074)

[3.7.3 堆排序功能截屏示例 12](#_Toc59791075)

[3.8 基数排序功能的实现 12](#_Toc59791076)

[3.8.1 基数排序功能流程图 12](#_Toc59791077)

[3.8.2 基数排序功能核心代码 12](#_Toc59791078)

[3.8.3 归并排序功能截屏示例 13](#_Toc59791079)

[3.9 总体系统的实现 14](#_Toc59791080)

[3.9.1 总体系统流程图 14](#_Toc59791081)

[3.9.2 总体系统代码实现 14](#_Toc59791082)

[3.9.3 总体功能截屏示例 18](#_Toc59791083)

[第4章 测试 19](#_Toc59791084)

[4.1 通常功能测试 19](#_Toc59791085)

# 第1章 功能分析

## 1.1 功能分析

随机函数产生一百，一千，一万和十万个随机数，分别用冒泡排序，选择排序，插入排序，希尔排序，快速排序，归并排序，堆排序，基数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。

# 第2章 设计

## 2.1 算法设计

如上所述，需要完成8种排序算法。

### 2.1.1 冒泡排序

比较相邻的两个数据，如果第二个数小，就交换位置。从后向前两两比较，一直到比较最前两个数据。最终最小数被交换到起始的位置，这样第一个最小数的位置就排好了。继续重复上述过程，依次将第2.3...n-1个最小数排好位置。

### 2.1.2 选择排序

在长度为N的无序数组中，第一次遍历n-1个数，找到最小的数值与第一个元素交换；第二次遍历n-2个数，找到最小的数值与第二个元素交换……第n-1次遍历，找到最小的数值与第n-1个元素交换，排序完成。

### 2.1.3 插入排序

在要排序的一组数中，假定前n-1个数已经排好序，现在将第n个数插到前面的有序数列中，使得这n个数也是排好顺序的。如此反复循环，直到全部排好顺序。

### 2.1.4 希尔排序

在要排序的一组数中，根据某一增量分为若干子序列，并对子序列分别进行插入排序。然后逐渐将增量减小,并重复上述过程。直至增量为1,此时数据序列基本有序,最后进行插入排序。

### 2.1.5 快速排序

先从数列中取出一个数作为key值；将比这个数小的数全部放在它的左边，大于或等于它的数全部放在它的右边；对左右两个小数列重复第二步，直至各区间只有1个数。

### 2.1.6 归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。

首先考虑下如何将2个有序数列合并。这个非常简单，只要从比较2个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

### 2.1.7 堆排序

利用堆这种数据结构所设计的一种排序算法。堆积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。堆排序可以说是一种利用堆的概念来排序的选择排序。

### 2.1.8 基数排序

基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。

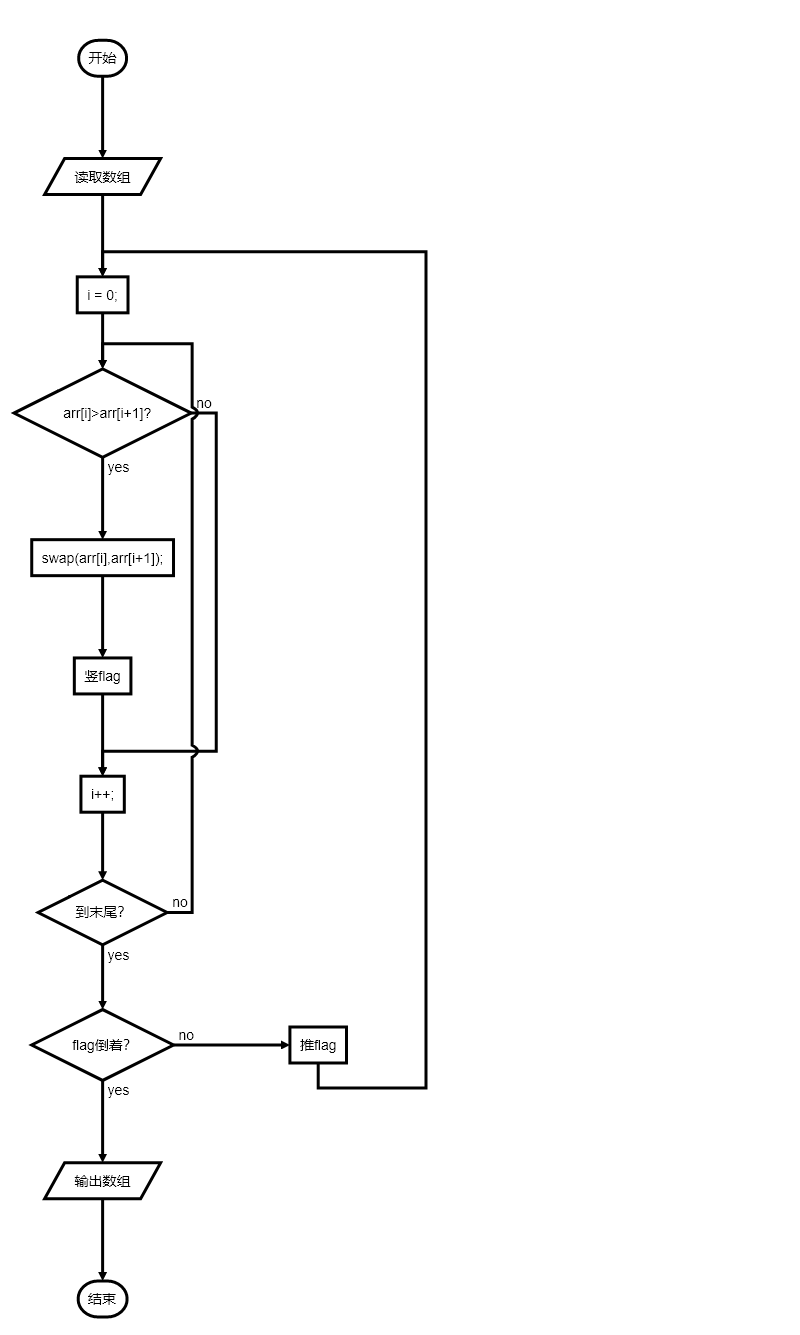
## 2.2 系统设计

系统首先调用opening ()函数实现对屏幕的初始化。然后输入数n来创建长度为n的随机数列，然后进入loop ()循环，根据用户所输入的操作码（operatorCode）执行对应的函数。

# 第3章 实现

## 3.1 冒泡排序功能的实现

### 3.1.1 冒泡排序功能流程图



### 3.1.2 冒泡排序功能核心代码

template <class T>

int bubbleSort(vector<T>& arr) {

    unsigned swpnum = 0;

    bool flag = true;

    while (flag) {

        flag = false;

        for (int i = 1; i < arr.size(); i++)

            if (arr[i] < arr[i - 1]) {

                swpnum++;

                swap(arr[i], arr[i - 1]);

                flag = true;

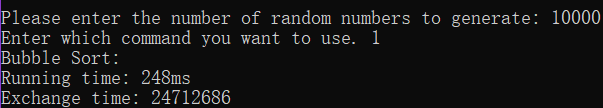
            }

    }

    return swpnum;

}

### 3.1.3 冒泡排序功能截屏示例



## 3.2 选择排序功能的实现

### 3.2.1 选择排序功能方法描述

第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。选择排序是不稳定的排序方法。

假设序列第一个位置为最小，然后依次和后面的元素进行比较，比第一个元素小的元素就设个标记再往后依次比较，直到找到最小值然后与第一个位置元素进行交换。

### 3.2.2 选择排序功能核心代码

template <class T>

int selctionSort(vector<T>& arr) {

    unsigned swpnum = 0;

    for (int i = 0; i < arr.size(); i++) {

        T\* min = &arr[i];

        for (int j = i + 1; j < arr.size(); j++)

            if (\*min > arr[j]) {

                min = &arr[j];

            }

        if (\*min != arr[i]) {

            swap(arr[i], \*min);

            swpnum++;

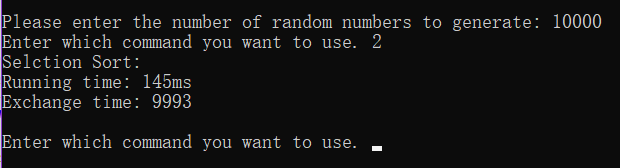
        }

    }

    return swpnum;

}

### 3.2.3 选择排序功能截屏示例



## 3.3 插入排序功能的实现

### 3.3.1 插入排序功能方法描述

插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，时间复杂度为O(n^2)。是稳定的排序方法。

 在一个数组中我们开始把第一个元素看为是一个排好序的队列，剩余的看为是乱序的序列，在剩余的队列中依次选取元素与前面队列进行比较再尔进行排序，至此我们就保证前面的元素永远都是有序的，从而达到插入排序的效果。

### 3.3.2 插入排序功能核心代码

template <class T>

int insertionSort(vector<T>& arr) {

    unsigned swpnum = 0;

    for (int i = 0; i < arr.size() - 1; i++) {

        for (int j = i + 1; j > 0; j--) {

            if (arr[j] < arr[j - 1]) {

                swap(arr[j], arr[j - 1]);

                swpnum++;

            } else

                break;

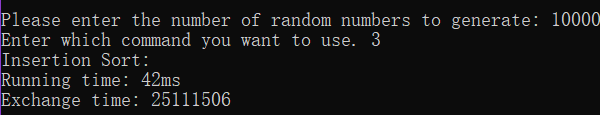
        }

    }

    return swpnum;

}

### 3.3.3 插入排序功能截屏示例



## 3.4 希尔排序功能的实现

### 3.4.1 希尔排序功能方法描述

希尔排序，也叫缩小增量排序。希尔排序就是按照下标一定增量进行分组，每组再按照直接插入算法排序，随着组的减少，每组的元素也越来越少，当组数减少至为1时，整个文件分成1组，算法便终止。

### 3.4.2 希尔排序功能核心代码

template <class T>

int shellSort(vector<T>& arr) {

    unsigned swpnum = 0;

    int incre = arr.size();

    while (true) {

        incre = incre / 2;

        for (int k = 0; k < incre; k++)

            for (int i = k + incre; i < arr.size(); i++)

                for (int j = i; j >= k; j -= incre) {

                    if (j - incre >= k && arr[j] < arr[j - incre]) {

                        swap(arr[j], arr[j - incre]);

                        swpnum++;

                    } else

                        break;

                }

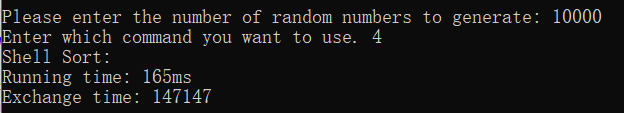
        if (incre == 1) break;

    }

    return swpnum;

}

### 3.4.3 希尔排序功能截屏示例



## 3.5 快速排序功能的实现

### 3.5.1 快速排序功能方法描述

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。大致意思就是在一个数组中取中间元素比它小的方左边比它大的则放右边 两边元素再按照快排要求，最终变成有序序列。

### 3.5.2 快速排序功能核心代码

template <class T>

int quickSort(vector<T>& arr, int left, int right) {

    unsigned swpnum = 0;

    int mid = (left + right) / 2;

    int temp, j, i, middle = arr[mid];

    i = left;

    j = right;

    while (j >= i) {

        while (middle > arr[i]) (i++);

        while (middle < arr[j]) (j--);

        if (i <= j) {

            swap(arr[i], arr[j]);

            swpnum++;

            i++;

            j--;

        }

    }

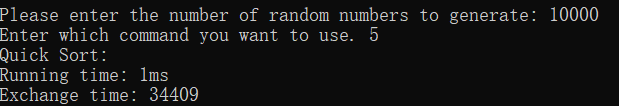
    if (i < right) (swpnum += quickSort(arr, i, right));

    if (j > left) (swpnum += quickSort(arr, left, j));

    return swpnum;

}

### 3.5.3 快速排序功能截屏示例



## 3.6 归并排序功能的实现

### 3.6.1 归并排序功能方法描述

归并排序是建立在归并操作上的一种有效的排序算法,该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

### 3.6.2 归并排序功能核心代码

template <class T>

int merge(vector<T>& nums, int l1, int r1, int l2, int r2) {

    unsigned swpnum = 0;

    int i = l1;                             //左半部分起始位置

    int j = l2;                             //右半部分起始位置

    int n = (r1 - l1 + 1) + (r2 - l2 + 1);  //要合并的元素个数

    vector<T> temp(n);                      //辅助数组

    int k = 0;                              //辅助数组其起始位置

    while (i <= r1 && j <= r2) {  //挑选两部分中最小的元素放入辅助数组中

        swpnum++;

        if (nums[i] < nums[j])

            temp[k++] = nums[i++];

        else

            temp[k++] = nums[j++];

    }

    //如果还有剩余，直接放入到辅助数组中

    while (i <= r1) temp[k++] = nums[i++];

    while (j <= r2) temp[k++] = nums[j++];

    //更新原始数组元素

    for (int i = 0; i < n; i++) {

        nums[l1 + i] = temp[i];

    }

    return swpnum;

}

//二路归并排序

template <class T>

int mergeSort(vector<T>& nums, int start, int end) {

    unsigned swpnum = 0;

    if (start < end) {

        int mid = (start + end) >> 1;           //分割序列

        swpnum += mergeSort(nums, start, mid);  //对序列左半部分进行规并排序

        swpnum += mergeSort(nums, mid + 1, end);  //对序列右半部分进行规并排序

        swpnum +=

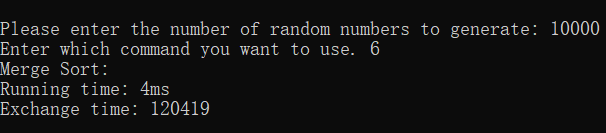
            merge(nums, start, mid, mid + 1, end);  //合并已经有序的两个序列

    }

    return swpnum;

}

### 3.6.3 归并排序功能截屏示例



## 3.7 堆排序功能的实现

### 3.7.1 堆排序功能方法描述

对简单选择排序的优化。将序列构建成大顶堆，将根节点与最后一个节点交换，然后断开最后一个节点。重复第一、二步，直到所有节点断开。

### 3.7.2 堆排序功能核心代码

template <class T>

int MaxSort(vector<T>& arr, int i, int n) {

    int swpnum = 0;

    int j = 2 \* i + 1;  //找到当前结点的左孩子

    int temp = arr[i];  //把当前结点的数赋给temp变量，后面会发现这个变量很有用

    while (j < n)  //判断必须条件，大于或等于该结点都属于数组越界

    {

        if (j + 1 < n && arr[j] < arr[j + 1])

            //判断条件，第一个条件是判断是不是最后一个结点。第二个判断条件是找出孩子结点最大的数方便与结点交换。

            ++j;  //如果后面的大，那么交换的变成后面的孩子。

        if (temp > arr[j])

            break;  //因为我们是MaxSort所以如果父结点本身就大不用判断直接跳出循环。

        else {  //判断过程，把最大的孩子结点的数赋给父结点。并利用递归思想找出子节点的子节点。

            arr[i] = arr[j];

            i = j;

            j = 2 \* i + 1;

            swpnum++;

        }

    }  // i已经成为了孩子结点的下标，赋值temp，也就是原本父结点的值，达成交换。

    arr[i] = temp;

    return swpnum;

}

//堆排序

template <class T>

int heapSort(vector<T>& arr, int n) {

    int swpnum = 0;

    for (int i = n / 2 - 1; i >= 0; i--)

        swpnum += MaxSort(arr, i, n);  //从最后一个结点的父结点开始“向前遍历”。

    for (int i = n - 1; i >= 1; i--) {

        swap(arr[i], arr[0]);

        swpnum++;

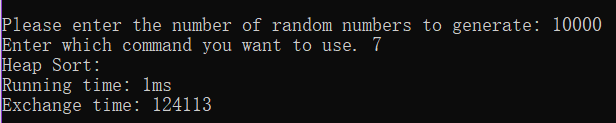
        swpnum += MaxSort(arr, 0, i);

    }  //逆序

    return swpnum;

}

### 3.7.3 堆排序功能截屏示例



## 3.8 基数排序功能的实现

### 3.8.1 基数排序功能方法描述

基数排序会分配是个桶 标号分别是0-9，在第一次排序时会将每个元素的个位取出，放到相应编号的桶中，然后按照桶的顺序依次放回原来的数组；进行第二次排序时，会将每个元素的十位取出放到相应编号的桶中，然后按照桶的顺序依次放回原来的数组；以此类推直到最高位排完，排序也就完成。

### 3.8.2 基数排序功能核心代码

/\*获得数组中数值的最大位数\*/

int maxBit(const vector<int> arr) {

    int len = 0;

    for (auto&& i : arr) {

        int c = 0;

        int tmp = i;

        while (tmp > 0) {

            tmp /= 10;

            c++;

        }

        if (len < c) len = c;

    }

    return len;

}

/\*基于int型数组的基数排序简单实现\*/

int radixSort(vector<int>& arr) {

    const int BUCKETS = 10;                //设置桶的个数

    vector<vector<int>> buckets(BUCKETS);  //设置基数桶

    int len = maxBit(arr);  //获得数组中数值的最大位数，依次决定排序次数

    int r = 1;

    for (int i = 0; i < len; ++i) {  //从低位到高位进行基数排序，依次方法桶中

        for (int& s : arr) {  //此循环按相应基数排序

            int k = s / r;

            int q = k % BUCKETS;

            buckets[q].push\_back(std::move(s));

        }

        int idx = 0;

        for (auto& thisBucket : buckets) {  //循环每个桶中的元素

            for (int& s : thisBucket)  //将每个桶中的元素放入原始数组中

                arr[idx++] = std::move(s);

            thisBucket.clear();

        }

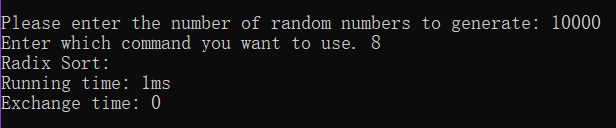
        r = r \* 10;

    }

    return 0;

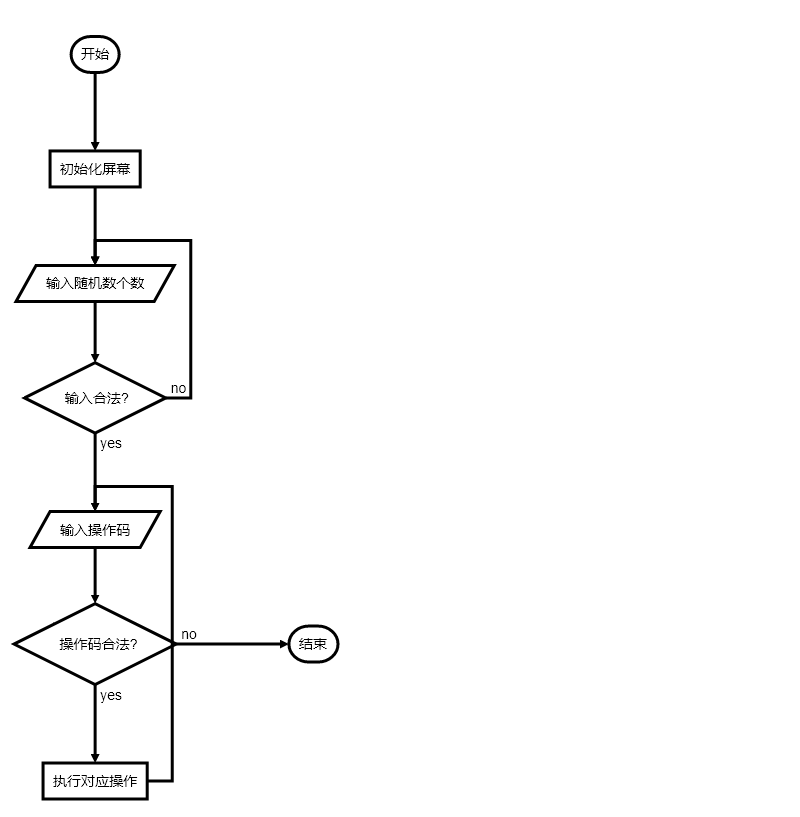
}

### 3.8.3 归并排序功能截屏示例



## 3.9 总体系统的实现

### 3.9.1 总体系统流程图



### 3.9.2 总体系统代码实现

bool loop(long temp) {

    srand((unsigned int)(time(NULL)));

    vector<int> arr;

    for (int i = 0; i < temp; i++) {

        arr.push\_back((int)random(0, 10000));

    }

    // for (auto&& i : arr) cout << i << " ";

    char ctemp = '\0';

    cout << "Enter which command you want to use. ";

    cin >> ctemp;

    bool flag = false;

    if (ctemp >= '1' && ctemp <= '8')

        flag = true;

    else if (ctemp == '9')

        flag = false;

    else

        return false;

    for (int i = 0; i < 8; i++) {

        if (flag) i = ctemp - '0' - 1;

        vector<int> arrc = arr;

        unsigned a = 0;

        double t = 0.0;

        clock\_t begin;

        clock\_t end;

        switch (i) {

            case 0:

                cout << "Bubble Sort: " << endl;

                begin = clock();

                a = bubbleSort(arrc);

                end = clock();

                break;

            case 1:

                cout << "Selction Sort: " << endl;

                begin = clock();

                a = selctionSort(arrc);

                end = clock();

                break;

            case 2:

                cout << "Insertion Sort: " << endl;

                begin = clock();

                a = insertionSort(arrc);

                end = clock();

                break;

            case 3:

                cout << "Shell Sort: " << endl;

                begin = clock();

                a = shellSort(arrc);

                end = clock();

                break;

            case 4:

                cout << "Quick Sort: " << endl;

                begin = clock();

                a = quickSort(arrc, 0, arrc.size() - 1);

                end = clock();

                break;

            case 5:

                cout << "Merge Sort: " << endl;

                begin = clock();

                a = mergeSort(arrc, 0, arrc.size() - 1);

                end = clock();

                break;

            case 6:

                cout << "Heap Sort: " << endl;

                begin = clock();

                a = heapSort(arrc, arrc.size());

                end = clock();

                break;

            case 7:

                cout << "Radix Sort: " << endl;

                begin = clock();

                a = radixSort(arrc);

                end = clock();

                break;

            default:

                return false;

        }

        t = (double)(end - begin) / CLOCKS\_PER\_SEC \* 1000;

        cout << "Running time: " << t << "ms" << endl;

        cout << "Exchange time: " << a << endl << endl;

        if (flag) return true;

    }

    // for (auto&& i : arr) cout << i << " ";

    return true;

}

void opening() {

    cout << "\*\*                Sorting algorithm comparison                \*\*"

         << endl

         << "==============================================================="

         << endl

         << endl

         << "Enter the number of random numbers, and enter which command you "

            "want to use. "

         << endl

         << "1. Bublle Sort" << endl

         << "2. Selction Sort" << endl

         << "3. Insertion Sort" << endl

         << "4. Shell Sort" << endl

         << "5. Quick Sort" << endl

         << "6. Merge Sort" << endl

         << "7. Heap Sort" << endl

         << "8. Radix Sort" << endl

         << "9. All methods above execute once in sequence." << endl;

}

int main() {

    opening();

    long temp = 0;

    string temps;

    while (true) {

        cout << endl

             << "Please enter the number of random numbers to generate: ";

        cin >> temps;

        if (isNum(temps)) {

            temp = str2num(temps);

            if (temp <= 0) {

                cout << "Error: Invalid input detected. Please try again.";

            } else

                break;

        } else

            cout << "Error: Invalid input detected. Please try again.";

    }

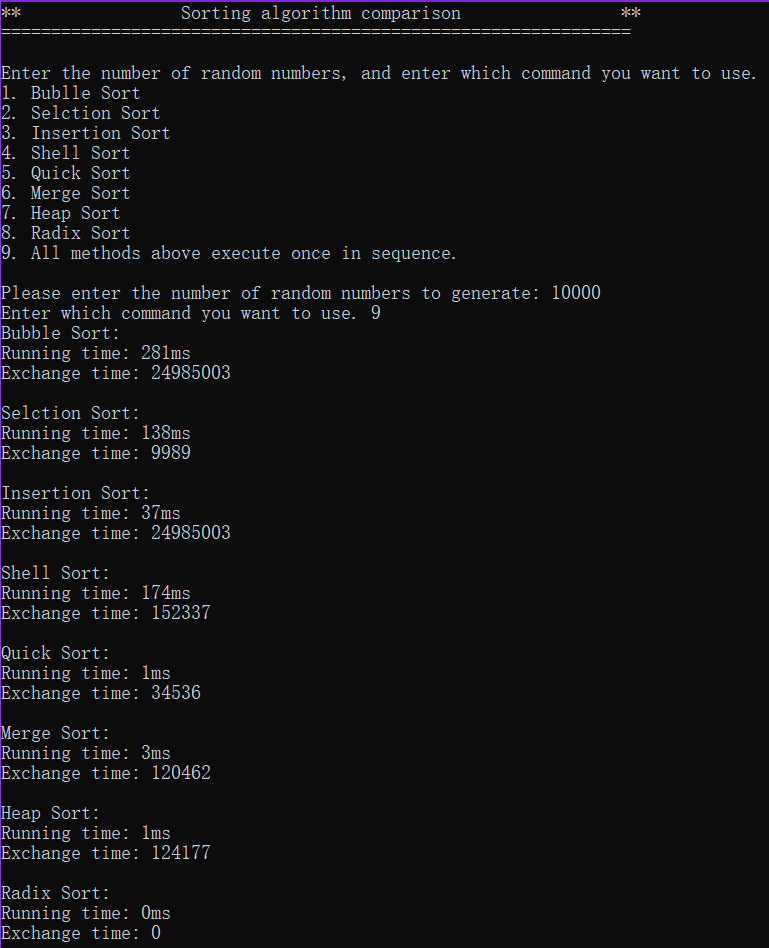
    while (loop(temp))

        ;

    return 0;

}

### 3.9.3 总体功能截屏示例



# 第4章 测试

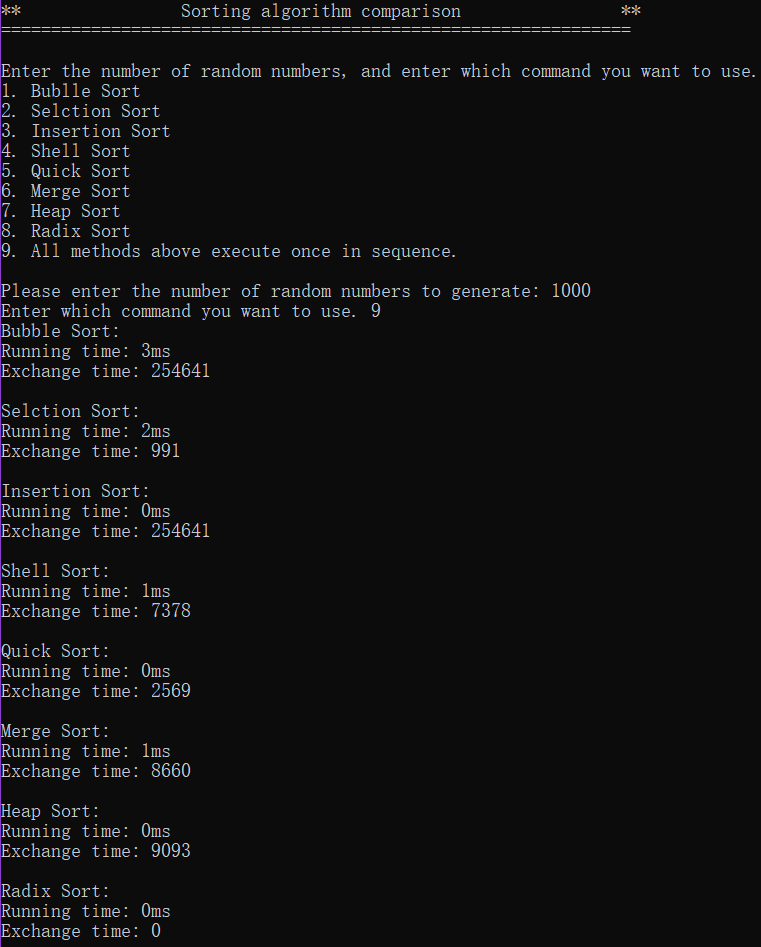
## 4.1 通常功能测试

**测试用例**：

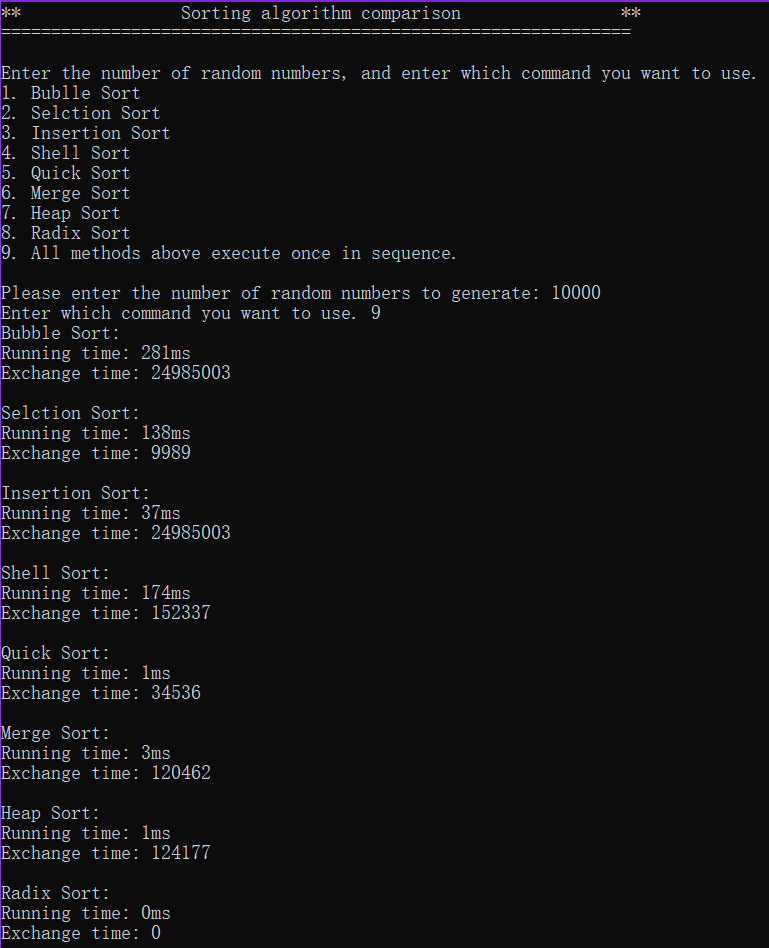
先后使用1000，10000，100000个数据进行测试

**实验结果**

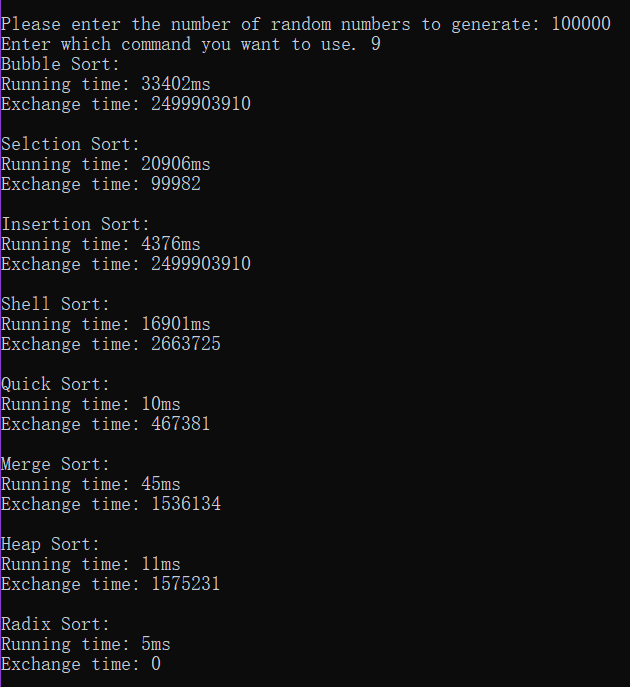
1000：



10000：



100000：



## 4.2 八种排序算法总结

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 排序法 | 平均时间 | 最差情形 | 稳定度 | 额外空间 | 备注 |
| 冒泡 | O(n2) | O(n2) | 稳定 | O(1) | n小时较好 |
| 选择 | O(n2) | O(n2) | 不稳定 | O(1) | n小时较好 |
| 插入 | O(n2) | O(n2) | 稳定 | O(1) | 大部分已排序时较好 |
| Shell | O(nlogn) | O(n) | 不稳定 | O(1) | s是所选分组 |
| 快速 | O(nlogn) | O(n2) | 不稳定 | O(nlogn) | n大时较好 |
| 归并 | O(nlogn) | O(nlogn) | 稳定 | O(1) | n大时较好 |
| 堆 | O(nlogn) | O(nlogn) | 不稳定 | O(1) | n大时较好 |
| 基数 | O(D(R+n)) | O(D(n+RD)) | 稳定 | O(RD+n) | D是长度，  R是基数(个十百) |