****

**《程序设计综合实践》设计文档**

# 第3章 第1次作业

**学号： 21052115**

**姓名： 姜曜楠**

完成时间 2022 年 4 月

## 1、题目

## 2、程序设计思路

1 依次编写各类排序函数：冒泡排序、简单选择排序、简单插入排序、归并排序、快速排序、其它排序（计数排序、桶排序、基数排序）；

2 数组产生函数：

1.完全随机数组：使用库函数srand和rand进行完全随机数组的初始化；

2.几乎有序数组：在进行过一次有效的排序后，生成排序后的数组的副本，并对副本进行随机打乱操作（因是小部分打乱，故将打乱的数据量控制在1-10）；

3 check函数所要检查的内容：

1.排序是否有效（升序）；

2.数据是否非法篡改；

4 排序时间：在排序前后分别调用库函数clock，计算差值即可得到，若超时（5分钟），则会提示“Error: Over time limited!!”并退出程序运行；

5 其他说明：

1.为保证各类排序函数所要排序的数组的数据一致性，故选择排序的数组为副本，而非直接操作随机产生的数组；

2．为保证不会发生内存溢出现象，每次排序后都因对副本进行清除操作；

3.为保证测量完整，若某个算法出现超时现象，则先使用另一个不超时的排序算法对数组进行排序后，再对该超时算法进行几乎有序数组的排序时间测量；

## 3、程序源码

/\* test.h -- 测试函数的接口 \*/

#ifndef TEST\_H

#define TEST\_H

#include <stdbool.h>

#include <time.h>

#include "sort.h"

#define NBUCKET 100

/\* 一般类型定义 \*/

typedef struct Array

{

    int size;

    int range;

    Elem\_T arr[0];

} Problem;

/\* 全局变量 \*/

extern clock\_t start;

/\* 函数原型 \*/

/\*void TestAllSort(Problem \*p); \*/

void Init(Problem \*p);

bool isTrue(const Problem \*origin, const Problem \*sorted);

void print(const Problem \*p);

void TestOneSort(Problem \*orgin, void (\*pfunction)(Problem \*p));

void TestMixedSort(Problem \*origin, void (\*pfunction1)(Problem \*p1), void (\*pfunction2)(Problem \*p2));

void BubbleSortTest(Problem \*p);

void SelectSortTest(Problem \*p);

void InsertionSortTest(Problem \*p);

void MergeSortTest(Problem \*p);

void QuickSortTest(Problem \*p);

void CountSortTest(Problem \*p);

void BucketSortTest(Problem \*p);

void RadixSortTest(Problem \*p);

#endif

/\* sort.h -- 排序函数的接口 \*/

#ifndef SORT\_H

#define SORT\_H

#include <stdbool.h>

#include <time.h>

#define LIMITSECONDS 300

#define LIMITBUFFER 2048

/\* 一般类型定义 \*/

typedef int Key\_T;

typedef struct

{

    Key\_T key;

} Elem\_T;

/\* 全局变量 \*/

clock\_t start, finish;

/\* 函数原型 \*/

/\*

    操作：冒泡排序

    后置条件：数组arr将被按照升序排列；

 \*/

void BubbleSort(Elem\_T arr[], int arrLength);

/\*

    操作：简单选择排序

    后置条件：数组arr将被按照升序排列；

 \*/

void SelectSort(Elem\_T arr[], int arrLength);

/\*

    操作：简单插入排序

    后置条件：数组arr将被按照升序排列；

 \*/

void InsertionSort(Elem\_T arr[], int arrLength);

/\*

    操作：归并排序

    前置条件：left为需要排序部分的最左端；

             right为需要排序部分的最右端；

    后置条件：数组arr将被按照升序排列；

 \*/

void MergeSort(Elem\_T arr[], int left, int right, Elem\_T temp[]);

/\*

    操作：快速排序

    前置条件：left为数组的下标；

             right为数组的下标；

    后置条件：数组arr将被按照升序排列；

 \*/

void QuickSort(Elem\_T arr[], int left, int right);

/\*

    操作：其它排序之计数排序

    前置条件：maxKey为数组元素上限；

             0 - maxKey范围不大；

             且Key\_T为int的别名；

    后置条件：数组des呈现数组src的升序排列结果；

 \*/

void CountSort(Elem\_T des[], const Elem\_T src[], int arrLength, int maxKey);

/\*

    操作：其它排序之桶排序

    前置条件：key均匀分布，且可缩放至(0,1)范围；

             bucketN代表桶的个数（零到bucketN-1）

    后置条件：数组arr将被按照升序排列；

 \*/

void BucketSort(Elem\_T arr[], int arrLength, int bucketN);

/\*

    操作：其它排序之基数排序

    前置条件：基于key的每一位分别进行桶排序；

    后置条件：数组arr将被按照升序排列；

 \*/

void RadixSort(Elem\_T arr[], int arrLength, int base, int maxDigit);

#endif

/\* main.c \*/

#include <stdlib.h>

#include "test.h"

#define LEN 100

#define RANGE 100

int main(void)

{

    Problem \*origin = (Problem \*)malloc(sizeof(Problem) + sizeof(Elem\_T) \* LEN);

    origin->size = LEN;

    origin->range = RANGE;

    Init(origin);

    TestOneSort(origin, BubbleSortTest);

    TestMixedSort(origin, QuickSortTest, BubbleSortTest);

    TestOneSort(origin, SelectSortTest);

    TestMixedSort(origin, QuickSortTest, SelectSortTest);

    TestOneSort(origin, InsertionSortTest);

    TestMixedSort(origin, QuickSortTest, InsertionSortTest);

    TestOneSort(origin, MergeSortTest);

    TestOneSort(origin, QuickSortTest);

TestOneSort(origin, CountSortTest);

    TestMixedSort(origin, QuickSortTest, CountSortTest);

    TestOneSort(origin, BucketSortTest);

    TestOneSort(origin, RadixSortTest);

    return 0;

}

/\* test.c -- 测试函数的接口实现 \*/

#include <stdio.h>

#include <stdlib.h>

#include "test.h"

/\* 局部函数原型 \*/

static bool isExist(Elem\_T e, int numOfValue[], const Problem \*origin);

static void copyProblem(Problem \*des, const Problem \*src);

static void result(Problem \*origin, Problem \*test);

static void shuffleArray(Problem \*test);

/\* 接口函数 \*/

void Init(Problem \*p)

{

    srand((unsigned)time(0));

    for (int i = 0; i < p->size; i++)

    {

        p->arr[i].key = rand() % p->range + 1;

    }

}

bool isTrue(const Problem \*origin, const Problem \*sorted)

{

    int \*numOfValue = (int \*)calloc(origin->range + 1, sizeof(int));

    for (int i = 0; i < origin->size; i++)

    {

        numOfValue[origin->arr[i].key]++;

    }

    for (int i = 0; i < sorted->size - 1; i++)

    {

        if (!isExist(sorted->arr[i], numOfValue, origin))

        {

            printf("\nPosition %d isn't exited", i + 1);

            free(numOfValue);

            return false;

        }

        if (sorted->arr[i].key > sorted->arr[i + 1].key)

        {

            printf("\nPosition %d is bigger than the one after", i + 1);

            free(numOfValue);

            return false;

        }

    }

    free(numOfValue);

    return true;

}

void print(const Problem \*p)

{

    printf("\nArray[%d]: {", p->size);

    for (int i = 0; i < p->size; i++)

    {

        if (i == p->size - 1)

        {

            printf(" %d};", p->arr[i].key);

        }

        else

        {

            printf(" %d,", p->arr[i].key);

        }

    }

}

void TestOneSort(Problem \*origin, void (\*pfunction)(Problem \*p))

{

    printf("\nFirst, a completely random array:");

    Problem \*test = (Problem \*)malloc(sizeof(Problem) + sizeof(Elem\_T) \* origin->size);

    copyProblem(test, origin);

    pfunction(test);

    result(origin, test);

    printf("\nThen, an almost ordered arrays:");

    Problem \*fake = (Problem \*)malloc(sizeof(Problem) + sizeof(Elem\_T) \* test->size);

    shuffleArray(test);

    copyProblem(fake, test);

    pfunction(test);

    result(fake, test);

    printf("\nThe end of this sorting method.");

    free(test);

    test = NULL;

    system("pause");

}

void TestMixedSort(Problem \*origin, void (\*pfunction1)(Problem \*p1), void (\*pfunction2)(Problem \*p2))

{

    printf("\nFirst, a completely random array:");

    Problem \*test = (Problem \*)malloc(sizeof(Problem) + sizeof(Elem\_T) \* origin->size);

    copyProblem(test, origin);

    pfunction1(test);

    result(origin, test);

    printf("\nThen, an almost ordered arrays:");

    Problem \*fake = (Problem \*)malloc(sizeof(Problem) + sizeof(Elem\_T) \* test->size);

    shuffleArray(test);

    copyProblem(fake, test);

    pfunction2(test);

    result(fake, test);

    printf("\nThe end of this sorting method.");

    free(test);

    test = NULL;

    system("pause");

}

void BubbleSortTest(Problem \*p)

{

    start = clock();

    BubbleSort(p->arr, p->size);

    finish = clock();

    printf("\n\tBubbleSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

}

void SelectSortTest(Problem \*p)

{

    start = clock();

    SelectSort(p->arr, p->size);

    finish = clock();

    printf("\n\tSelectSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

}

void InsertionSortTest(Problem \*p)

{

    start = clock();

    InsertionSort(p->arr, p->size);

    finish = clock();

    printf("\n\tInsertionSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

}

void MergeSortTest(Problem \*p)

{

    Elem\_T \*temp = NULL;

    temp = (Elem\_T \*)malloc(p->size \* sizeof(Elem\_T));

    start = clock();

    MergeSort(p->arr, 0, p->size - 1, temp);

    finish = clock();

    printf("\n\tMergeSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

    free(temp);

    temp = NULL;

}

void QuickSortTest(Problem \*p)

{

    start = clock();

    QuickSort(p->arr, 0, p->size - 1);

    finish = clock();

    printf("\n\tQuickSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

}

void CountSortTest(Problem \*p)

{

    Elem\_T \*temp;

    temp = (Elem\_T \*)malloc(p->size \* sizeof(Elem\_T));

    for (int i = 0; i < p->size; i++)

    {

        temp[i].key = p->arr[i].key;

    }

    start = clock();

    CountSort(p->arr, temp, p->size, p->range);

    finish = clock();

    printf("\n\tCountSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

    free(temp);

    temp = NULL;

}

void BucketSortTest(Problem \*p)

{

    start = clock();

    BucketSort(p->arr, p->size, NBUCKET);

    finish = clock();

    printf("\n\tBucketSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

}

void RadixSortTest(Problem \*p)

{

    int maxDigit = 0, base = 10;

    int temp = p->range;

    while (temp != 0)

    {

        maxDigit++;

        temp /= base;

    }

    start = clock();

    RadixSort(p->arr, p->size, base, maxDigit);

    finish = clock();

    printf("\n\tRadixSort cost: %lf seconds", (double)(finish - start) / CLOCKS\_PER\_SEC);

}

/\* 局部函数 \*/

static bool isExist(Elem\_T e, int numOfValue[], const Problem \*origin)

{

    if (e.key > origin->range)

    {

        return false;

    }

    for (int i = 0; i < origin->size; i++)

    {

        if (origin->arr[i].key == e.key && numOfValue[e.key] > 0)

        {

            numOfValue[e.key]--;

            return true;

        }

    }

}

static void copyProblem(Problem \*des, const Problem \*src)

{

    des->size = src->size;

    des->range = src->range;

    for (int i = 0; i < des->size; i++)

    {

        des->arr[i].key = src->arr[i].key;

    }

}

static void result(Problem \*origin, Problem \*test)

{

    if (isTrue(origin, test))

    {

        printf("\n\tSort result is true.");

    }

    else

    {

        printf("\n\tError: sort result is wrong!!");

    }

}

static void shuffleArray(Problem \*test)

{

    srand((unsigned)time(0));

    int n = 0;

    while (n == 0)

    {

        n = rand() % 10;

    }

    for (int i = 0; i < n; i++)

    {

        int j = rand() % test->size;

        int newValue = rand() % test->range + 1;

        test->arr[j].key = newValue;

    }

}

/\* sort.c -- 排序函数的接口实现 \*/

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "sort.h"

/\* 静态全局变量声明 \*/

static bool flag = true;

/\* 局部函数原型 \*/

static bool needToAdjust(Key\_T smaller, Key\_T bigger);

static void swap(Elem\_T \*e1, Elem\_T \*e2);

static void merge(Elem\_T arr[], int left, int middle, int right, Elem\_T temp[]);

static void copyArr(Elem\_T des[], Elem\_T src[], int start, int end);

static int QuickPass(Elem\_T arr[], int left, int right);

static void initialICount(int iCount[], int maxKey);

static void getICount(const Elem\_T src[], int n, int iCount[]);

static bool timeOut(void);

/\* 接口函数 \*/

void BubbleSort(Elem\_T arr[], int arrLength)

{

    for (int i = 0; i < arrLength - 1; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        for (int j = 0; j < arrLength - 1 - i; j++)

        {

            if (needToAdjust(arr[j].key, arr[j + 1].key))

            {

                swap(&arr[j], &arr[j + 1]);

            }

        }

    }

}

void SelectSort(Elem\_T arr[], int arrLength)

{

    for (int i = 0; i < arrLength; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        int min = i;

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

        {

            if (needToAdjust(arr[min].key, arr[j].key))

            {

                min = j;

            }

        }

        swap(&arr[i], &arr[min]);

    }

}

void InsertionSort(Elem\_T arr[], int arrLength)

{

    for (int i = 0; i < arrLength; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        Elem\_T temp = arr[i];

        int j = i - 1;

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

        {

            arr[j + 1] = arr[j];

            --j;

        }

        arr[j + 1] = temp;

    }

}

void MergeSort(Elem\_T arr[], int left, int right, Elem\_T temp[])

{

    if (flag == false)

    {

        return;

    }

    if (timeOut())

    {

        printf("Error: Over time limited!!");

        flag = false;

        system("pause");

        exit(1);

    }

    if (left >= right)

    {

        return;

    }

    int middle = (left + right) / 2;

    MergeSort(arr, left, middle, temp);

    MergeSort(arr, middle + 1, right, temp);

    merge(arr, left, middle, right, temp);

    copyArr(arr, temp, left, right);

}

void QuickSort(Elem\_T arr[], int left, int right)

{

    if (flag == false)

    {

        return;

    }

    if (timeOut())

    {

        printf("Error: Over time limited!!");

        flag = false;

        system("pause");

        exit(1);

    }

    if (left >= right)

    {

        return;

    }

    int pivot = QuickPass(arr, left, right);

    QuickSort(arr, left, pivot - 1);

    QuickSort(arr, pivot + 1, right);

}

void CountSort(Elem\_T des[], const Elem\_T src[], int arrLength, int maxKey)

{

    int iCount[maxKey + 1];

    initialICount(iCount, maxKey);

    getICount(src, arrLength, iCount);

    for (int i = 0, start = 0; i < maxKey + 1; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        int next = start + iCount[i];

        iCount[i] = start;

        start = next;

    }

    for (int i = 0; i < arrLength; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        des[iCount[src[i].key]++] = src[i];

    }

}

void BucketSort(Elem\_T arr[], int arrLength, int bucketN)

{

    /\* 创建bucketN个空桶，编号0 - bucketN-1 \*/

    struct node

    {

        Elem\_T data;

        struct node \*next;

    };

    struct node \*\*bucket = (struct node \*\*)malloc(sizeof(struct node \*) \* bucketN);

    for (int i = 0; i < bucketN; i++)

    {

        bucket[i] = NULL;

    }

    /\* 计算桶号,丢桶 \*/

    for (int i = 0; i < arrLength; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        /\* 根据arr[i].key计算出桶号k \*/

        int k = (arr[i].key - 1) / (arrLength / bucketN);

        /\* 根据arr[i]丢到k号桶 \*/

        struct node \*current = (struct node \*)malloc(sizeof(struct node));

        current->data = arr[i],

        current->next = bucket[k],

        bucket[k] = current;

    }

    /\* 排序 \*/

    for (int i = 0, now = 0; i < bucketN; i++)

    {

        if (timeOut())

        {

            printf("Error: Over time limited!!");

            system("pause");

            exit(1);

        }

        /\* 把i号桶内元素排序 \*/

        int last = now;

        while (bucket[i] != NULL)

        {

            arr[now++] = bucket[i]->data;

            bucket[i] = bucket[i]->next;

        }

        Elem\_T \*temp = NULL;

        temp = (Elem\_T \*)malloc((now - last) \* sizeof(Elem\_T));

        MergeSort(arr, last, now - 1, temp);

        free(temp);

        temp = NULL;

    }

    free(bucket);

    bucket = NULL;

}

void RadixSort(Elem\_T arr[], int arrLength, int base, int maxDigit)

{

    for (int i = 1; i <= maxDigit; i++)

    {

        /\* 创建base个空桶，编号0 - base-1 \*/

        struct node

        {

            Elem\_T data;

            struct node \*next;

        };

        struct node \*\*bucket = (struct node \*\*)malloc(sizeof(struct node \*) \* base);

        for (int i = 0; i < base; i++)

        {

            bucket[i] = NULL;

        }

        for (int j = arrLength - 1; j >= 0; j--)

        {

            if (timeOut())

            {

                printf("Error: Over time limited!!");

                system("pause");

                exit(1);

            }

            /\* 根据arr[j].key的第i位，计算出桶号k \*/

            int k = (arr[j].key / (int)pow(base, i - 1)) % base;

            /\* 把arr[j]丢到k号桶中 \*/

            struct node \*current = (struct node \*)malloc(sizeof(struct node));

            current->data = arr[j],

            current->next = bucket[k],

            bucket[k] = current;

        }

        for (int i = 0, k = 0; i < base; i++)

        {

            if (timeOut())

            {

                printf("Error: Over time limited!!");

                system("pause");

                exit(1);

            }

            /\* 顺序收集k号桶内所有元素，修改原数组arr\*/

            while (bucket[i] != NULL)

            {

                arr[k++] = bucket[i]->data;

                bucket[i] = bucket[i]->next;

            }

        }

        free(bucket);

        bucket = NULL;

    }

}

/\* 局部函数 \*/

static bool needToAdjust(Key\_T smaller, Key\_T bigger)

{

    return (smaller > bigger);

}

static void swap(Elem\_T \*e1, Elem\_T \*e2)

{

    Elem\_T temp;

    temp = \*e1;

    \*e1 = \*e2;

    \*e2 = temp;

}

static void merge(Elem\_T arr[], int left, int middle, int right, Elem\_T temp[])

{

    int k = 0;

    int i = left;

    int j = middle + 1;

    while (i <= middle && j <= right)

    {

        if (arr[i].key <= arr[j].key)

        {

            temp[k++] = arr[i++];

        }

        else

        {

            temp[k++] = arr[j++];

        }

    }

    if (i == middle + 1)

    {

        while (j <= right)

        {

            temp[k++] = arr[j++];

        }

    }

    if (j == right + 1)

    {

        while (i <= middle)

        {

            temp[k++] = arr[i++];

        }

    }

}

static void copyArr(Elem\_T des[], Elem\_T src[], int start, int end)

{

    for (int i = start, j = 0; j <= end - start; i++, j++)

    {

        des[i] = src[j];

    }

}

static int QuickPass(Elem\_T arr[], int left, int right)

{

    Elem\_T temp = arr[left];

    while (left < right)

    {

        while (left < right && arr[right].key >= temp.key)

        {

            right--;

        }

        if (left == right)

        {

            break;

        }

        arr[left++] = arr[right];

        while (left < right && arr[left].key <= temp.key)

        {

            left++;

        }

        if (left == right)

        {

            break;

        }

        arr[right--] = arr[left];

    }

    arr[left] = temp;

    return left;

}

static void initialICount(int iCount[], int maxKey)

{

    for (int key = 0; key < maxKey + 1; key++)

    {

        iCount[key] = 0;

    }

}

static void getICount(const Elem\_T src[], int arrLength, int iCount[])

{

    for (int i = 0; i < arrLength; i++)

    {

        iCount[src[i].key]++;

    }

}

static bool timeOut(void)

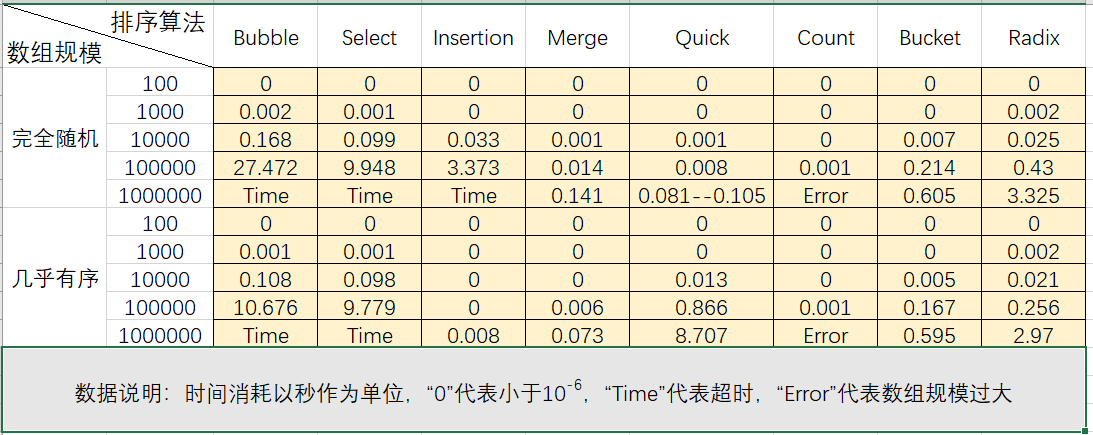
{

    finish = clock();

    return (int)(finish - start) / CLOCKS\_PER\_SEC > LIMITSECONDS;

}

## 4、数据分析



由上图可得：

1. 在排序数据量不超过一千时，上述八种排序没有太大差异；
2. 在排序数据量不超过十万时：
   1. 如果是完全随机产生的数组，各排序算法的速度如下：

计数排序>快速排序>归并排序>桶排序>基数排序>简单插入排序>简单选择排序>冒泡排序

* 1. 如果是几乎有序的数组，各排序算法的速度如下：

简单插入函数>计数排序>归并排序>桶排序>基数排序>快速排序>简单选择排序>冒泡排序

1. 在排序数据量大于十万甚至在1000000（百万级别）及以上时：
   1. 如果是完全随机产生的数组，各排序算法的速度如下：

快速排序>归并排序>桶排序>基数排序>简单插入排序（超时）、简单选择排序（超时）、冒泡排序（超时）；计数排序出错；

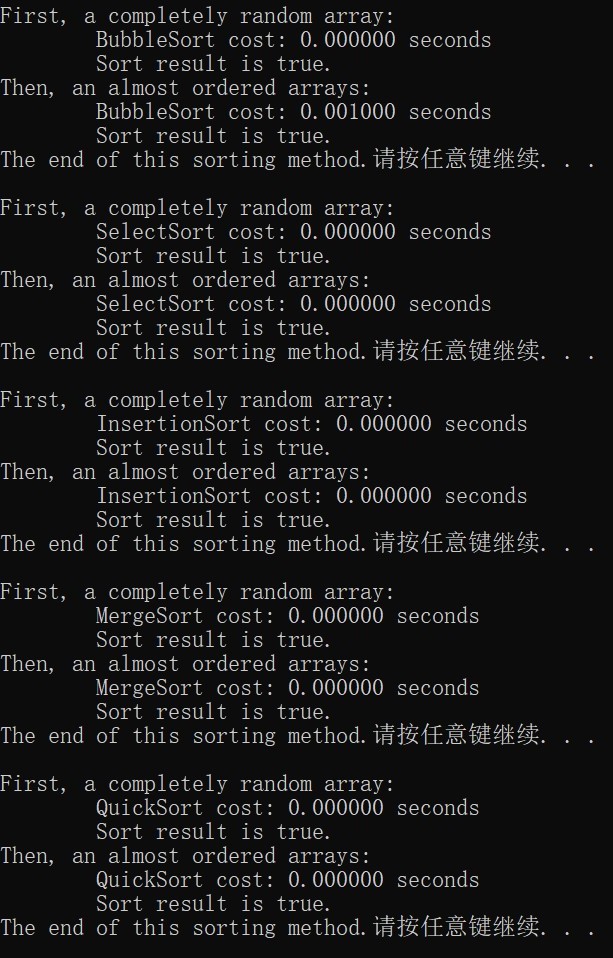
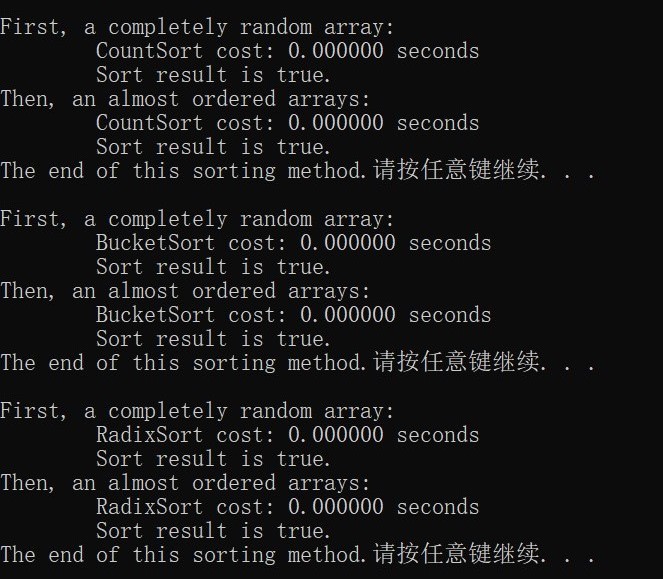
* 1. 如果是几乎有序的数组，各排序算法的速度如下：

简单插入排序>归并排序>桶排序>基数排序>快速排序>简单选择排序（超时）、冒泡排序（超时）；计数排序出错；

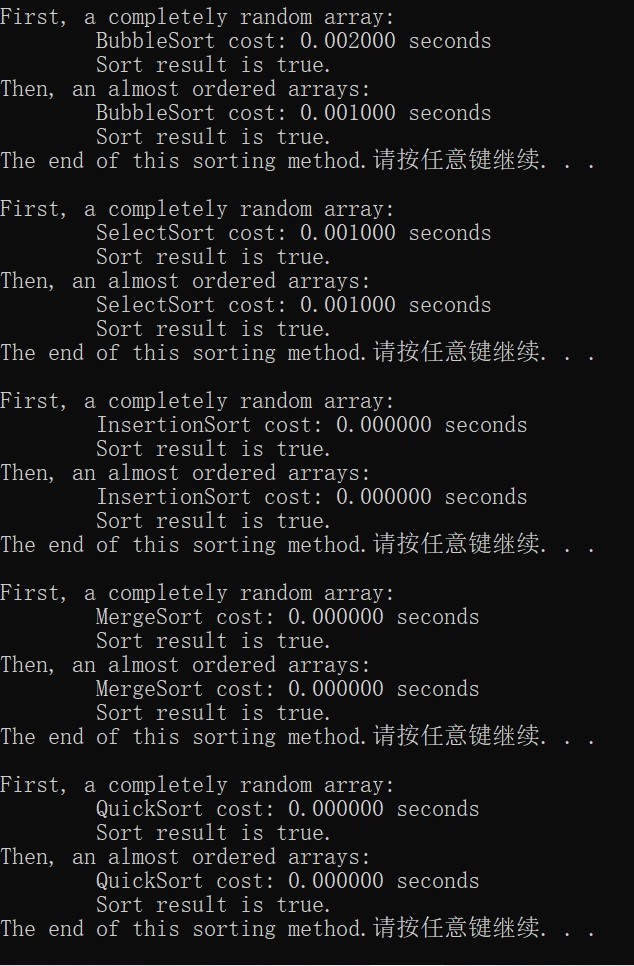
1. 特殊点分析：
   1. 快速排序用于几乎有序数组的效率远不如用于完全随机数组：这是因为当需要排序的数组几乎有序时，快排每次找到的排轴几乎只能把区域划分为1和n-1的状态，即分治效果并不明显，导致消耗的时间大大增加；
   2. 计数排序几乎是一般排序算法中速度最快的，但由于其空间复杂度为O(n)，导致数据量较大时需要创建较多个桶，当数据量到达十万级别时，缓存空间不足以创建十万级别个桶，从而导致程序报错；
   3. 桶排序内部会借用其他排序算法，消耗的时间会随借用的排序算法不同而有所不同，此处借用的是归并排序算法。

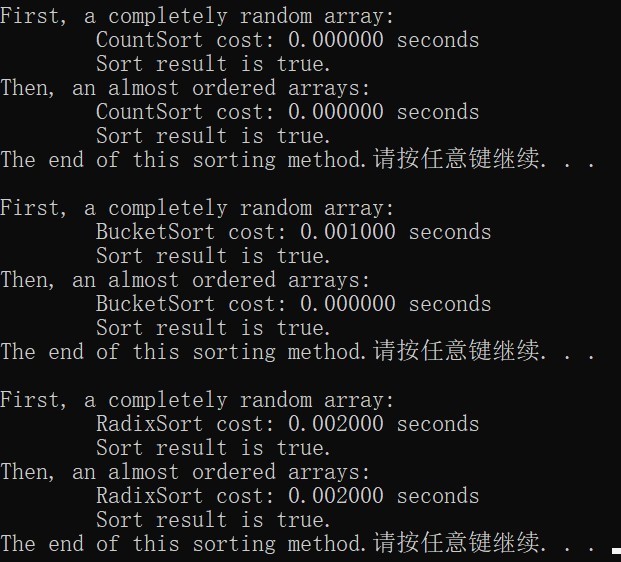
## 5、运行截图

（数据量100）

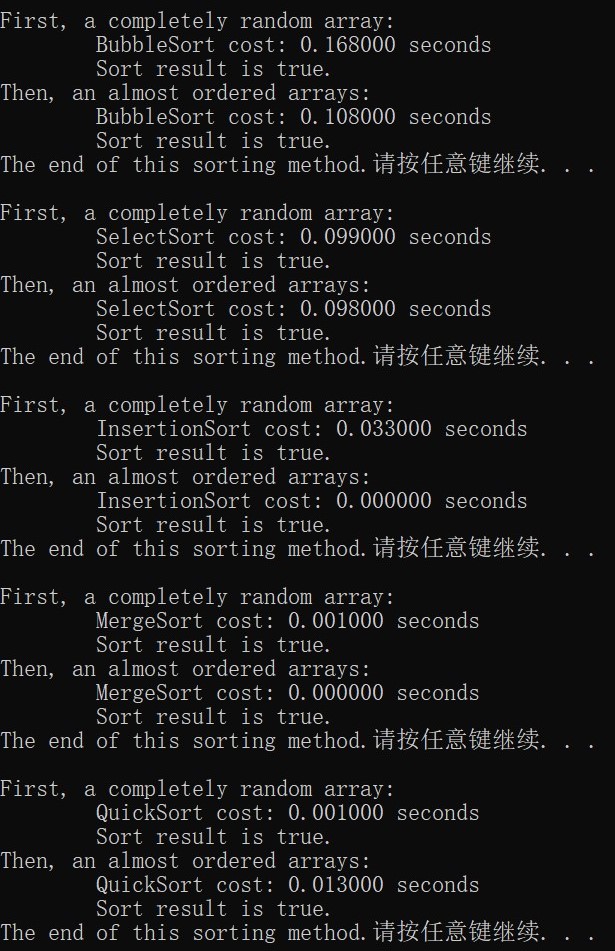
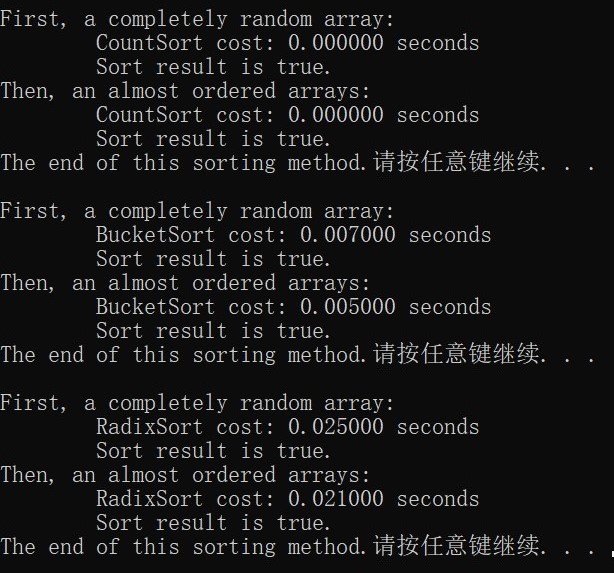


（数据量1000）

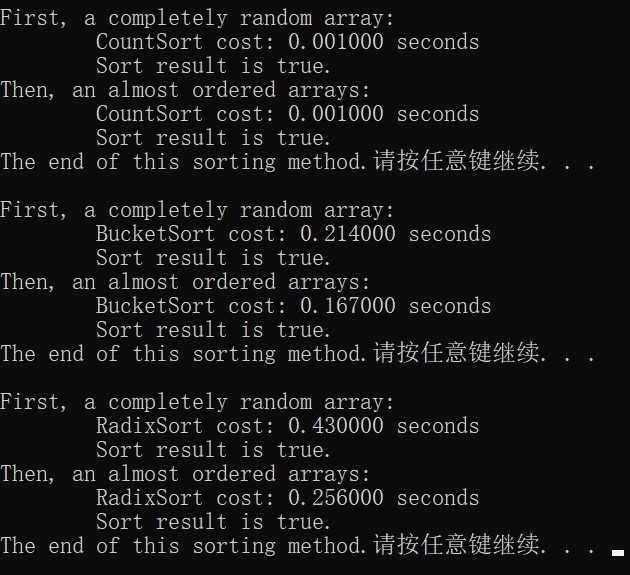
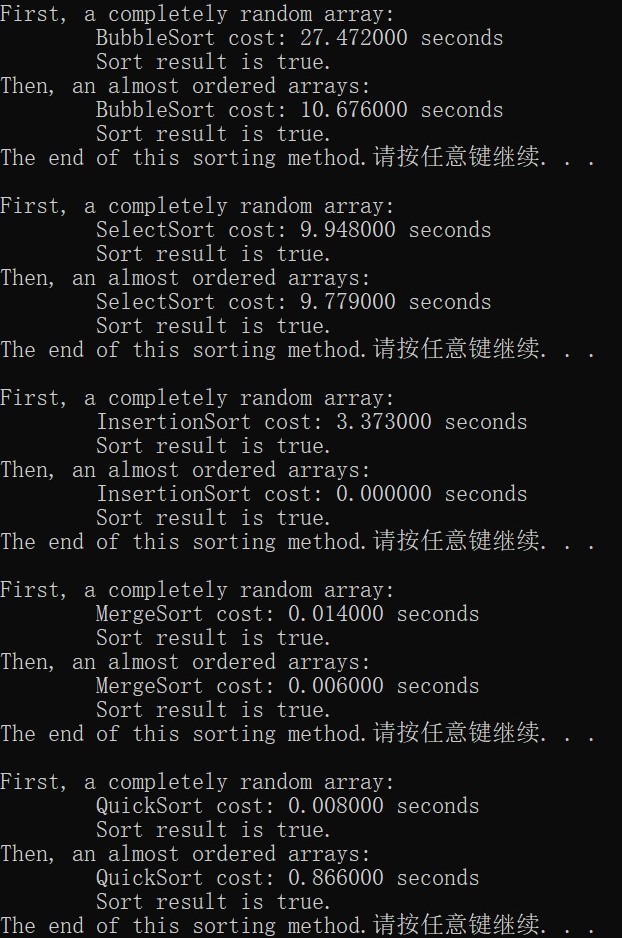




（数据量10000）

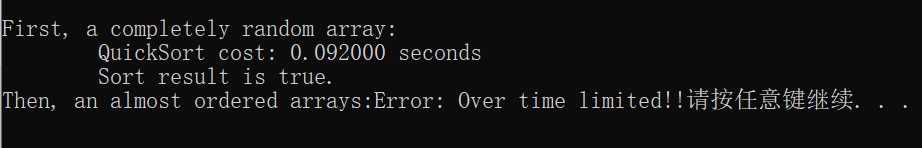
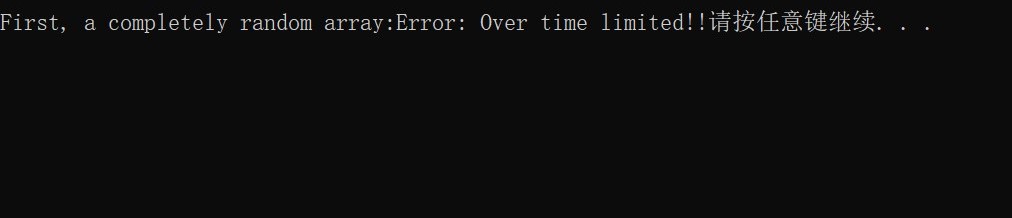


（数据量100000）

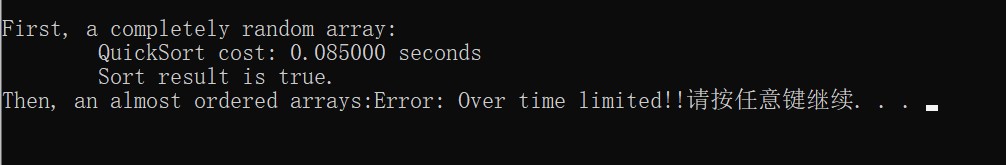
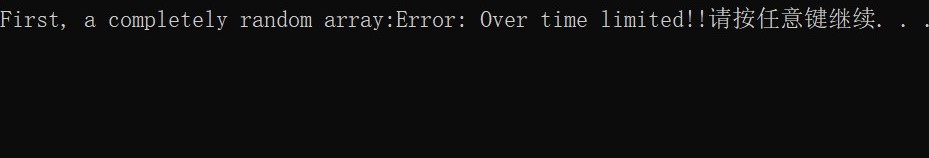


（数据量1000000）

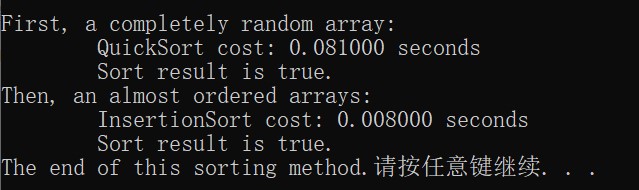
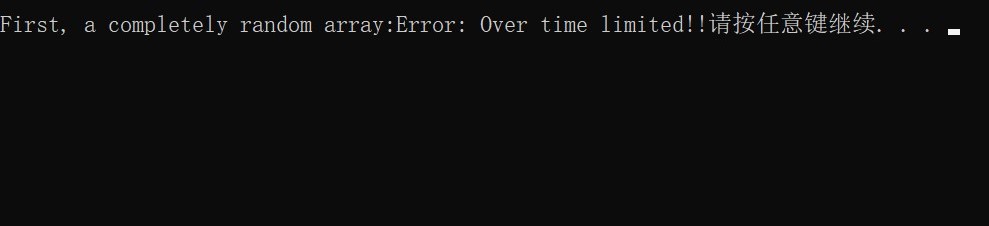
冒泡排序：



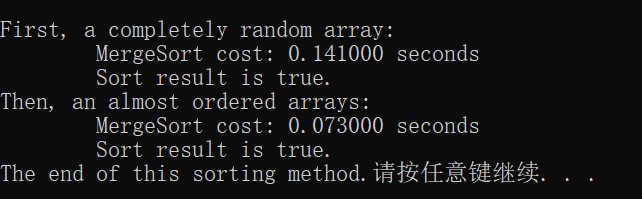
简单选择排序：



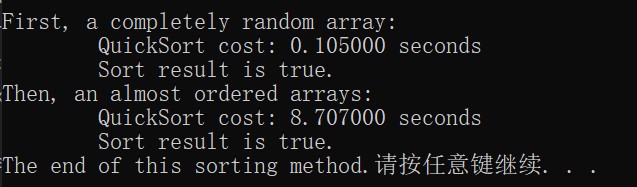
简单插入排序：



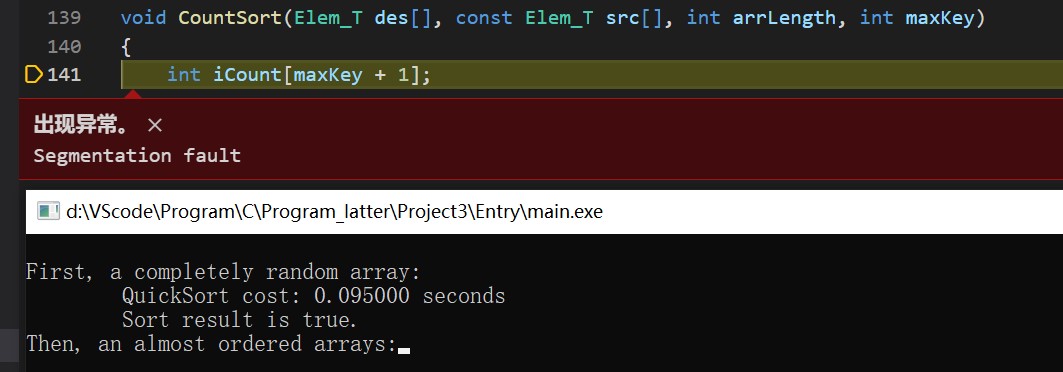
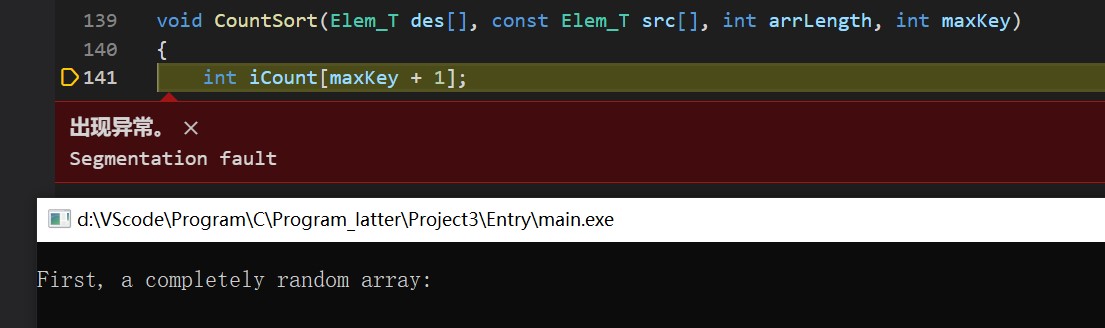
归并排序：



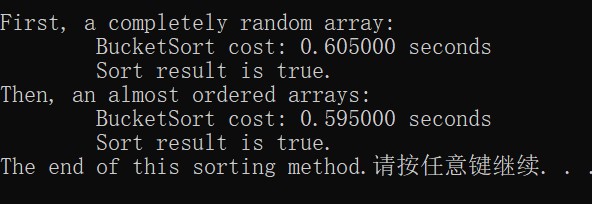
快速排序：



计数排序：



桶排序：



基数排序：

