数据结构课程设计

项目说明文档

8种排序方法的比较案例

软件工程

张靖凯

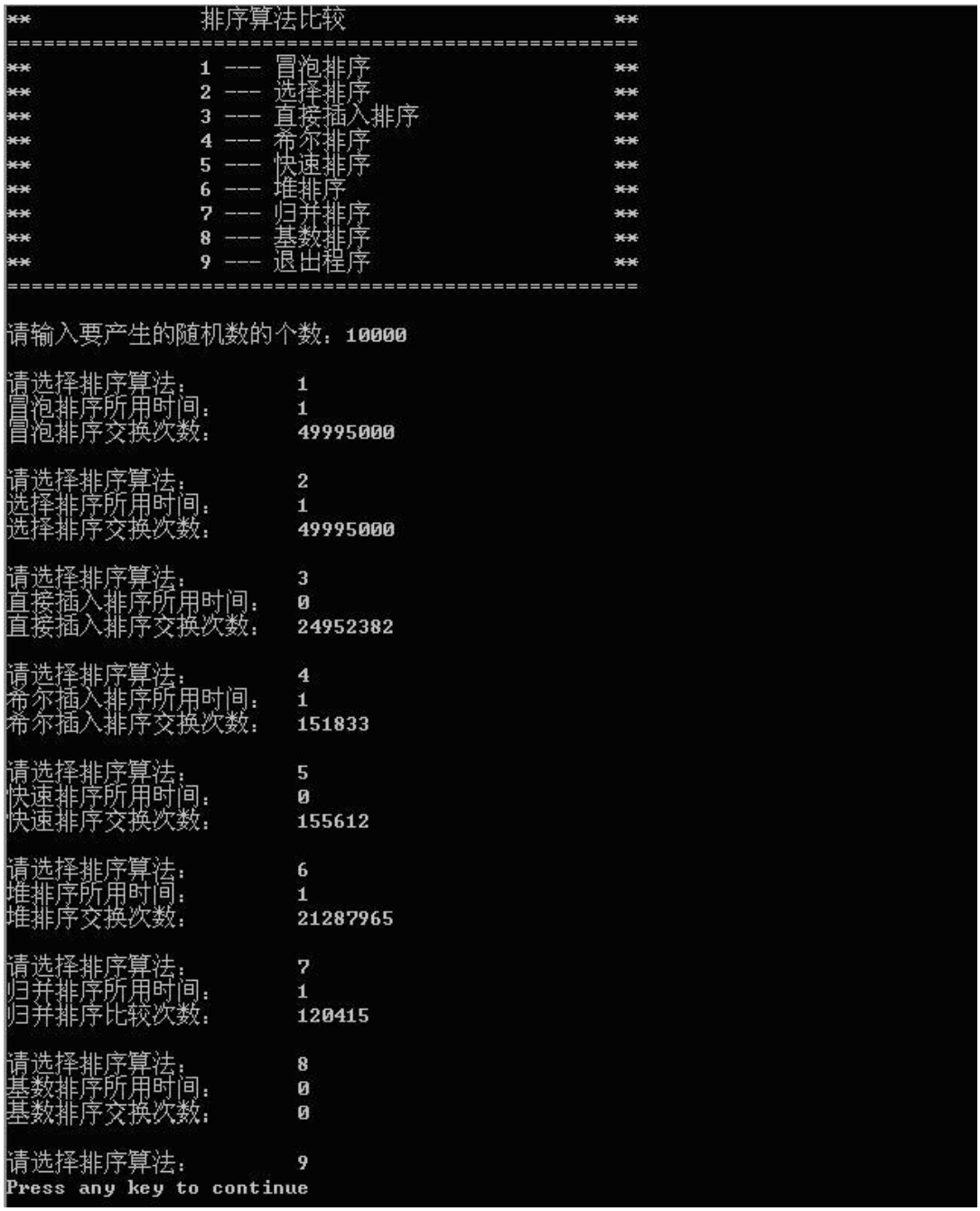
2151396

目录

项目简介

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。

项目示例



项目设计

数据结构与类设计

整体设计了 Sort 类，内涵 8 种排序方法以及其他计时工具等。

自行实现的 Vector 储存产生的随机序列；模板参数 T 代表随机值的类型，默认为 int 型；mySwap 为转移函数。

Public Member Functions

	Sort ()
void	**mySwap** (T &a, T &b)
void	**generateVec** (const int &n) 产生的随机数为n个，其中类型为T，范围从0-1e5 More...
void	**bubbleSort** ()
void	**selectionSort** ()
void	**insertSort** ()
void	**shellSort** ()
void	**quickSort** ()
void	**heapSort** ()
void	**mergeSort** ()
void	**baseSort** ()

Public Attributes

Vector< T >	**vec**

Private Member Functions

int	**maxbit** (**Vector**< int > data, int n)
void	**__quickSort** (int begin, int **end**)
void	**__mergeSort** (**Vector**< int > &tmp, int l, int r)

Private Attributes

system_clock::time_point	**start**
system_clock::time_point	**end**
double	**durationTime**
int	**count**

排序算法分析

排序法	平均时间	最差情形	稳定度	额外空间	备注
冒泡	$O(n^2)$	$O(n^2)$	稳定	O(1)	n 小时较好
选择	$O(n^2)$	$O(n^2)$	不稳定	O(1)	n 小时较好
插入	$O(n^2)$	$O(n^2)$	稳定	O(1)	大部分已排序时较好
基数	$O(\log_R B)$	$O(\log_R B)$	稳定	O(n)	B 是真数(0-9)， R 是基数(个十百)
Shell	O(nlogn)	$O(n^s)$ 1<s<2	不稳定	O(1)	s 是所选分组
快速	O(nlogn)	$O(n^2)$	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

冒泡排序

```
        for (int i = 0; i < vec.size() - 1; i++)
            for (int j = 0; j < vec.size() - i - 1; j++)
                if (vec[j] > vec[j + 1])
                    mySwap(vec[j], vec[j + 1]);
```

选择排序

```
        for (int i = 0; i < vec.size() - 1; i++)
        {
            int itemp = i;
            for (int j = i + 1; j < vec.size(); j++)
            {
                if (vec[j] < vec[itemp])
                    itemp = j;
            }
            if (i != itemp)
            {
                mySwap(vec[i], vec[itemp]);
            }
        }
```

插入排序

```
        for (int i = 0; i < vec.size() - 1; i++)
        {
            int end = i;
            T temp = vec[end + 1];
            while (end >= 0)
            {
                if (vec[end] > temp)
                {
                    vec[end + 1] = vec[end];
                    end--;
                    count++;
                }
                else
                    break;
```

```
            }
            vec[end + 1] = temp;
        }
```

希尔排序

```
        while (gap > 1)
        {
            gap /= 2;
            for (int i = 0; i < vec.size() - gap; i++)
            {
                int end = i;
                T temp = vec[end + gap];
                while (end >= 0)
                {
                    if (temp < vec[end])
                    {
                        vec[end + gap] = vec[end];
                        end -= gap;
                        count++;
                    }
                    else
                        break;
                }
                vec[end + gap] = temp;
            }
        }
```

快速排序

```
        if (begin > end)
            return;
        int tmp = vec[begin];
        int i = begin;
        int j = end;
        while (i != j) {
            count++;
            while (vec[j] >= tmp && j > i)
                j--;
            while (vec[i] <= tmp && j > i)
                i++;
            if (j > i) {
                mySwap(vec[i], vec[j]);
            }
```

```
        }
        vec[begin] = vec[i];
        vec[i] = tmp;
        __quickSort(begin, i - 1);
        __quickSort(i + 1, end);
```

堆排序

```
        auto f = [this](Vector<T>& rhsVec, int start, int end)
        {
            int dad = start, son = dad * 2 + 1;// 左孩子
            while (son <= end)
            {
                if (son + 1 <= end && rhsVec[son + 1] > rhsVec[son])
                    son++;
                if (rhsVec[dad] > rhsVec[son])
                    return;
                else
                {
                    mySwap(rhsVec[dad], rhsVec[son]);
                    dad = son;
                    son = dad * 2 + 1;
                }
            }
        };
        for (int i = vec.size() / 2 - 1; i >= 0; i--)
        {
            f(vec, i, vec.size() - 1);
        }
        // 将大根堆排序成从小到大的 vector
        for (int i = vec.size() - 1; i > 0; i--)
        {
            mySwap(vec[0], vec[i]);
            f(vec, 0, i - 1);
        }
```

归并排序

```
        if (l >= r) return;
        int mid = l + r >> 1;
        __mergeSort(tmp, l, mid), __mergeSort(tmp, mid + 1, r);
        int k = 0, i = l, j = mid + 1;
        while (i <= mid && j <= r)
        {
```

```
            count++;
            if (vec[i] <= vec[j]) tmp[k++] = vec[i++];
            else tmp[k++] = vec[j++];
        }
        while (i <= mid) tmp[k++] = vec[i++], count++;
        while (j <= r) tmp[k++] = vec[j++], count++;
        for (i = l, j = 0; i <= r; i++, j++) vec[i] = tmp[j];
```

基数排序

```
        for (i = 1; i <= d; i++) //进行 d 次排序
        {
            for (j = 0; j < 10; j++)
                cnt[j] = 0; //每次分配前清空计数器
            for (j = 0; j < n; j++)
            {
                k = (vec[j] / radix) % 10; //统计每个桶中的记录数
                cnt[k]++;
            }
            for (j = 1; j < 10; j++)
            {
                count++;
                cnt[j] = cnt[j - 1] + cnt[j]; //将 tmp 中的依次分配给每个桶
            }
            for (j = n - 1; j >= 0; j--) //将所有桶中记录依次收集到 tmp 中
            {
                k = (vec[j] / radix) % 10;
                tmp[cnt[k] - 1] = vec[j];
                cnt[k]--;
            }
            for (j = 0; j < n; j++) //将临时数组的内容复制到 data 中
                vec[j] = tmp[j];
            radix = radix * 10;
```

设计亮点

代码注释规范

采用了 doxygen 注释规范，对类、函数等有简要说明，命名采用驼峰命名法，类内的各个声明规范，方便本人回顾之前写过的代码和 code reviewer 查看，使 API 规范，帮助这个开发流程高效、规范地进行。

```
/// @brief 8种方法的排序类
/// @notion 本题中每次选择一定要重新创建新的Sort对象!
/// @attention 模板参数代表比较数的类型
```

项目测试（包含 linux 下 Ubuntu 运行）

```
**          2 --- 选择排序          **
**          3 --- 直接插入排序      **
**          4 --- 希尔排序          **
**          5 --- 快速排序          **
**          6 --- 堆排序            **
**          7 --- 归并排序          **
**          8 --- 基数排序          **
**          9 --- 退出程序          **
==================================
请输入要产生随机数的个数: 10000

请选择排序算法: 1
冒泡排序所用时间: 790511 微秒
冒泡排序交换次数: 24738980 次

请选择排序算法: 2
选择排序所用时间: 430282 微秒
选择排序交换次数: 9993 次

请选择排序算法: 3
插入排序所用时间: 228137 微秒
插入排序交换次数: 24836065 次

请选择排序算法: 4
希尔排序所用时间: 3760 微秒
希尔排序交换次数: 153533 次

请选择排序算法: 5
快速排序所用时间: 1651 微秒
快速排序交换次数: 56992 次

请选择排序算法: 6
堆排序所用时间: 3188 微秒
堆排序交换次数: 124359 次

请选择排序算法: 7
归并排序所用时间: 3148 微秒
归并排序交换次数: 133616 次

请选择排序算法: 8
基数排序所用时间: 1514 微秒
基数排序交换次数: 63 次
```

```
kk@LAPTOP-UJDPHKT8:~/dataStruct/homework10$ ./run
**           排序算法比较           **
==================================
**          1 --- 冒泡排序          **
**          2 --- 选择排序          **
**          3 --- 直接插入排序      **
**          4 --- 希尔排序          **
**          5 --- 快速排序          **
**          6 --- 堆排序            **
**          7 --- 归并排序          **
**          8 --- 基数排序          **
**          9 --- 退出程序          **
==================================
请输入要产生随机数的个数: 10000

请选择排序算法: 1
冒泡排序所用时间: 493839 微秒
冒泡排序交换次数: 24937574 次

请选择排序算法:
```