项目说明文档
——8 种排序算法的比较

姓名：吴桐欣
学号：1652677
同济大学 软件学院 软件工程专业

目录

1.项目概述

1.1 项目简介

随机函数产生 10000 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。

1.1.1 功能分析

本项目要求实现功能

（1） 产生一定数量的随机数；
（2） 进行冒泡排序；
（3） 进行选择排序；
（4） 进行直接插入排序；
（5） 进行希尔排序；
（6） 进行快速排序；
（7） 进行堆排序；
（8） 进行归并排序；
（9） 进行基数排序。

1.2 文件目录

（1）P10_1652677_吴桐欣_说明文档.docx（本文档）
（2）P10_1652677_吴桐欣.exe（可执行文件）
（3）P10_1652677_吴桐欣.cpp（源文件）
（4）P10_1652677_吴桐欣.h（头文件）

1.3 操作指南

（1）运行程序后，将获得程序提示“请输入要产生的随机数的个数：”
用户输入一个数字表示考随机数的个数。
（2）随后程序提示“请选择排序算法：”
用户需要按照所提供的操作数表选择排序算法。
（3）操作数表如下：
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

1.4 注意事项

（1）用户不能输入除数字以外的字符
（2）用户输入的操作数不能超出所提供的范围

2.思路与设计

2.1 基本思路

每种排序方式用至少一个函数来实现。函数仅在函数内改变数组的值，函数调用结束后，数组顺序保持原来的乱序，以供多次利用、公平比较各个排序的效率。

2.2 设计

2.2.1 数据结构

用 vector 数组存放产生的随机数。

3.具体实现

3.1 冒泡排序

核心代码

```
void BubbleSort(vector<int> list){
    long n=list.size();//元素个数
    int count_comp=0;//记录比较次数
    int count_swap=0;//记录交换次数

    clock_t start=clock();

    for (int i=0; i<n-1; i++) {
        bool exchange=false;//是否产生交换
        for (long j=n-1; j>0; j--) {
            count_comp++;
            if (list[j-1]>list[j]) {
                //发现逆序，进行交换
                SWAP(list[j], list[j-1]);
                count_swap++;
                exchange = true;
            }
        }
        if (!exchange) {
            //本轮不产生交换，排序完毕
            break;
        }
    }
    clock_t end=clock();
    cout<<"冒泡排序所用时间："<<(float)(end-start)/CLOCKS_PER_SEC<<"
秒"<<endl;
    cout<<"冒泡排序比较次数："<<count_comp<<endl;
    cout<<"冒泡排序交换次数："<<count_swap<<endl<<endl;
}
```

说明

使用两个嵌套的 for 循环，内层表示遍历数组，发现逆序则前后交换数值；外层表示进行至多 n-1 次的遍历排序。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.2 选择排序

核心代码

```
void SelectSort(vector<int> list){
    long n=list.size();//元素个数
    int count_comp=0;//记录比较次数
    int count_swap=0;//记录交换次数

    clock_t start=clock();
    for (int i=0; i<n; i++) {
        int min_i=i;//最小元素的序号
        for (int j=i+1; j<n; j++) {
            count_comp++;
            if (list[min_i]>list[j]) {min_i = j;}
        }
        //交换
        SWAP(list[i], list[min_i]);
        count_swap++;
    }
    clock_t end=clock();
    cout<<"选择排序所用时间：  "<<(float)(end-start)/CLOCKS_PER_SEC<<" 秒
"<<endl;
    cout<<"选择排序比较次数：  "<<count_comp<<endl;
    cout<<"选择排序交换次数：  "<<count_swap<<endl<<endl;
}
```

说明

使用两个嵌套的 for 循环，内层表示在剩余的数中遍历寻找最小的数，外层表示从小到大依次放置元素。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.3 直接插入排序

核心代码

```
void InsertSort(vector<int> list){
    long n=list.size();//元素个数
    int count_comp=0;//记录比较次数
    int count_swap=0;//记录交换次数
    int temp;//暂存要插入的元素

    clock_t start=clock();
    for (int i=1; i<n; i++) {
        //获取要插入的元素
        temp=list[i];
        //从后向前一个个比较
        for (int j=i-1; j>=0; j--) {
            count_comp++;
            if (list[j]>temp) {
                //将前一个元素后移
                list[j+1]=list[j];
            } else {
                //若前面数小于temp，则找到要插入的位置
                list[j+1]=temp;
                break;
            }
        }
    }
    count_swap=count_comp;//每次比较都会修改数据，因此交换次数与比较次数相等
    clock_t end=clock();
    cout<<"直接插入排序所用时间："<<(float)(end-start)/CLOCKS_PER_SEC<<"
秒"<<endl;
    cout<<"直接插入排序比较次数："<<count_comp<<endl;
    cout<<"直接插入排序交换次数："<<count_swap<<endl<<endl;
}
```

说明

使用两个嵌套的 for 循环，内层表示在已排好的序列片段中寻找某元素的插入位置，外层表示依次插入数组中的元素。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.4 希尔排序

核心代码

```
void ShellSort(vector<int> list){
    int count_comp=0;//记录比较次数
    int count_swap=0;//记录交换次数
    long n=list.size();//元素个数
    int gap=(int)n;//增量

    clock_t start=clock();
    do{
        gap = gap/3+1;//计算增量
        for (int i=gap; i<n; i+=gap) {
            count_comp++;
            //若在子序列中发现逆序
            if (list[i-gap]>list[i]) {
                int temp=list[i];
                //用直接插入法，从后向前比较
                for (int j=i-gap; ; j-=gap) {
                    count_comp++;
                    if (list[j]<=temp) {
                        //小于等于temp，找到temp插入位置
                        list[j+gap]=temp;
                        count_swap++;
                        break;
                    } else {
                        //若比temp大，则后移一位
                        list[j+gap]=list[j];
                        count_swap++;
                    }
                }
            }
        }
    } while (gap>1);
    clock_t end=clock();
    cout<<"希尔排序所用时间："<<(float)(end-start)/CLOCKS_PER_SEC<<" 秒
"<<endl;
    cout<<"希尔排序比较次数："<<count_comp<<endl;
    cout<<"希尔排序交换次数："<<count_swap<<endl<<endl;
}
```

说明

三层循环嵌套，最外层是每次计算出不同的 gap 增量数值；中层表示待排序序列被划分成的几个子序列依次分别进行排序；内层表示对一个子序列进行排序（直接插入排序）。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.5 快速排序

核心代码

```
//快速排序-三者取中
int& median3(vector<int>& list, const int left, const int right, int&
count_comp, int& count_swap){
    int mid=(left+right)/2;
    int k=left;
    if (list[mid]<list[k]) {k=mid;}
    if (list[right]<list[k]) {k=right;}//left,mid,right三者选最小
    if (k!=left) {
        SWAP(list[k], list[left]);

    }//把最小值移到left位置
    if (mid!=right && list[mid]<list[right]) {
        SWAP(list[right], list[mid]);
    }//把中间值，即基准，移到right位置
    return list[right];
}
```

```
//快速排序--一趟划分
int Partition(vector<int>& list, const int left, const int right, int&
count_comp, int& count_swap){
    int i=left;
    int j=right-1;
    if (left<right) {
        int pivot=median3(list, left, right, count_comp, count_swap);
        for (; ; ) {
            while (i<j && list[i]<pivot) {i++;count_comp++;}
            while (i<j && list[j]>pivot) {j--;count_comp++;}
            if (i<j) {
                SWAP(list[i], list[j]);//交换list[i],list[j]的值
                count_swap++;
                i++;
                j--;
            } else {
                break;
            }
        }
        if (list[i]>pivot) {
            list[right]=list[i];
            list[i]=pivot;
            count_swap++;
            count_comp++;
        }
    }
    return i;
}
```

```
//快速排序
void QuickSort(vector<int> list, const int left, const int right, int&
count_comp, int& count_swap){
    if (left==right) {}
    else if (left==right-1) {
        count_comp++;
        if (list[left]>list[right]) {
            SWAP(list[left], list[right]);
            count_swap++;
        }
    }
    else {
        int pivotpos=Partition(list, left, right, count_comp,
count_swap);//划分
        QuickSort(list, left, pivotpos-1, count_comp, count_swap);
        QuickSort(list, pivotpos+1, right, count_comp, count_swap);
    }
}
```

```
int count_comp=0;//记录比较次数
int count_swap=0;//记录交换次数
clock_t start=clock();
QuickSort(numbs, 0, (int)(numbs.size()-1), count_comp, count_swap);
clock_t end=clock();
cout<<"快速排序所用时间："<<(float)(end-start)/CLOCKS_PER_SEC<<" 秒
"<<endl;
cout<<"快速排序比较次数："<<count_comp<<endl;
cout<<"快速排序交换次数："<<count_swap<<endl<<endl;
```

说明

三者取中函数通过比较三个数的大小，获得较为合理的基准，防止划分的子序列长度相差太多。
一趟划分函数根据基准将待排序序列划分为大于基准数、小于基准数两个子序列。
递归调用快速排序函数，终止条件是划分出的子序列只有 2 个元素。
因涉及递归，计算次数和记录时间不能在函数内进行。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.6 堆排序

核心代码

```
//堆排序-从s到m自顶向下调整
void siftDown(vector<int>& heap, int s, int m, int& count_comp, int&
count_swap){
    int i=s;
    int j=2*i+1;//j是i的左子女
    int temp=heap[i];
    while (j<=m) {
        count_comp++;
        if (j<m && heap[j]<heap[j+1]) {j++;}//j指向两子女中最大者
        count_comp++;
        if (temp>=heap[j]){
            break;
        } else {
            count_swap++;
            heap[i]=heap[j];
            i=j;
            j=2*i+1;//i降到子女的位置
        }

    }
    heap[i]=temp;
    count_swap++;
}
```

```
//堆排序
void HeapSort(vector<int> list){
    int count_comp=0;//记录比较次数
    int count_swap=0;//记录交换次数
    int n=(int)list.size();//元素个数

    clock_t start=clock();
    for (int i=(n-2)/2; i>=0; i--) {
        siftDown(list, i, n-1, count_comp, count_swap);
    }
    for (int i=n-1; i>=0; i--) {
        SWAP(list[0], list[i]);
        count_swap++;
        siftDown(list, 0, i-1, count_comp, count_swap);
    }
    clock_t end=clock();
    cout<<"堆排序所用时间："<<(float)(end-start)/CLOCKS_PER_SEC<<" 秒
"<<endl;
    cout<<"堆排序比较次数："<<count_comp<<endl;
    cout<<"堆排序交换次数："<<count_swap<<endl<<endl;
}
```

说明

siftDown 函数用 while 循环调整成为最大堆。
堆排序函数先 for 循环多次调用 siftDown 函数，使序列调整为堆。
再用 for 循环令每次循环都将得到的最大堆的堆顶放置在序列后面。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.7 归并排序

核心代码

```
//归并排序-合并
void Merge(vector<int>& list, const int left, const int mid, const int
right, int& count_comp, int& count_swap){
    vector<int> tempList(list);
    int s1=left,s2=mid+1,t=left;//s1，s2是检测指针，t是存放指针
    while(s1<=mid && s2<=right){
        count_comp++;
        count_swap++;
        if (tempList[s1]<=tempList[s2]) {
            list[t++]=tempList[s1++];
        } else {
            list[t++]=tempList[s2++];
        }
    }
    while (s1<=mid) {
        list[t++]=tempList[s1++];
        count_swap++;
    }
    while (s2<=right) {
        list[t++]=tempList[s2++];
        count_swap++;
    }
}
```

```
//归并排序函数
void MergeSort(vector<int>& list, int left, int right, int&
count_comp, int& count_swap){
    if (left>=right) {
        return;
    }
    int mid=(left+right)/2;
    MergeSort(list, left, mid, count_comp, count_swap);//左子序列排序
    MergeSort(list, mid+1, right, count_comp, count_swap);//右子序列排序
    Merge(list, left, mid, right, count_comp, count_swap);//合并左右子序
列
}
```

```
                vector<int> copyList(numbs);
                int count_comp=0;//记录比较次数
                int count_swap=0;//记录交换次数
                clock_t start=clock();
                MergeSort(copyList, 0, (int)(numbs.size()-1),
count_comp, count_swap);
                clock_t end=clock();
                cout<<"归并排序所用时间："<<(float)(end-
start)/CLOCKS_PER_SEC<<" 秒"<<endl;
                cout<<"归并排序比较次数："<<count_comp<<endl;
                cout<<"归并排序交换次数："<<count_swap<<endl<<endl;
```

说明

合并函数将两个分别有序的子序列合并为一个有序序列。
递归调用归并排序函数，每次先划分出子序列，分别用归并函数处理两个子序列，再用 merge 函数将两个子序列合并。
因涉及函数递归调用，所以计数和记录时间差不能在函数内进行。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

3.8 基数排序

核心代码

```
void RadixSort(vector<int> list){
    int n=(int)list.size();//元素个数
    int count_comp=0;//记录比较次数
    int count_swap=0;//记录交换次数
    vector<int> tempList(list);

    clock_t start=clock();
    for (long place=1; ; place*=10) {
        int k=0;//存放指针
        bool change=false;
        for (int radix=0;; radix++) {
            for (int i=0; i<tempList.size();i++) {
                count_comp++;
                if ((tempList[i]%(10*place))/place == radix) {
                    if (list[k]!=tempList[i]) {
                        list[k]=tempList[i];//放入结果列中
                        count_swap++;
                        change=true;
                    }
                    k++;//存放指针移至下一个位置
                    if (i!=tempList.size()-1) {
                        //将此数移到列尾
                        int temp_i=i;
                        while(temp_i!=tempList.size()-1){
                            tempList[temp_i]=tempList[temp_i+1];
                            temp_i++;
                        }
                        i--;//要使下一步从i位置开始
                    }
                    tempList.pop_back();
                }
            }
            if (radix>9 || k>=n) {
                break;
            }
        }
        if (change==false) {
            break;
        }
        tempList=list;
    }
    clock_t end=clock();
    cout<<"基数排序所用时间: "<<(float)(end-start)/CLOCKS_PER_SEC<<" 秒
"<<endl;
    cout<<"基数排序比较次数: "<<count_comp<<endl;
    cout<<"基数排序交换次数: "<<count_swap<<endl<<endl;
}
```

说明

首先选出每次比较的数值位，每个位有 0～9 共 10 个基数，根据基数对序列进行排序。
分别记录数据比较次数和数据交换次数。
排序开始前后分别用 clock()函数获得数据，后者减去前者，得到时间差。

4.测试

4.1 功能测试

```
[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

请输入要产生的随机数的个数：500
请选择排序算法：1
冒泡排序所用时间：0 秒
冒泡排序比较次数：236027
冒泡排序交换次数：62420

请选择排序算法：2
选择排序所用时间：0 秒
选择排序比较次数：124750
选择排序交换次数：500

请选择排序算法：3
直接插入排序所用时间：0 秒
直接插入排序比较次数：62919
直接插入排序交换次数：62919

请选择排序算法：4
希尔排序所用时间：0 秒
微软拼音 半 :次数：40136
希尔排序交换次数：39115

请选择排序算法：5
快速排序所用时间：0 秒
快速排序比较次数：2487
快速排序交换次数：941

请选择排序算法：6
堆排序所用时间：0 秒
堆排序比较次数：7446
堆排序交换次数：4812

请选择排序算法：7
归并排序所用时间：0 秒
归并排序比较次数：3851
归并排序交换次数：4488
```

```
请选择排序算法：8
基数排序所用时间：0 秒
基数排序比较次数：12406
基数排序交换次数：2492

请选择排序算法：9
退出程序

--------------------------------

Process exited after 17.52 seconds with return value 0
请按任意键继续. . .
```

4.2 出错测试

4.2.1 输入非法字符

```
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

请输入要产生的随机数的个数：F
输入了非法字符!
请输入要产生的随机数的个数：50
请选择排序算法：X
输入了非法字符!
请选择排序算法：
```

4.2.2 输入数字不符合要求