8种排序算法的比较案例

作 者 姓 名： 王凌

学 号： 1951504

指 导 教 师： 张颖

学院、 专业： 软件工程

同济大学

Tongji University

目录

[1 分析 - 1 -](#_Toc7875)

[1.1 项目简介 - 1 -](#_Toc31071)

[1.2 功能分析 - 1 -](#_Toc12866)

[2 设计 - 2 -](#_Toc22119)

[2.1 排序算法设计 - 2 -](#_Toc17081)

[2.2 主程序设计 - 2 -](#_Toc25583)

[3 实现 - 4 -](#_Toc15481)

[3.1 八个排序方法的实现 - 4 -](#_Toc29256)

[3.1.1 冒泡排序 - 4 -](#_Toc30409)

[3.1.2 选择排序 - 4 -](#_Toc2796)

[3.1.3 直接插入排序 - 5 -](#_Toc28482)

[3.1.4 希尔排序 - 6 -](#_Toc3825)

[3.1.5 快速排序 - 7 -](#_Toc31995)

[3.1.6 堆排序 - 8 -](#_Toc29195)

[3.1.7 归并排序 - 9 -](#_Toc8712)

[3.1.8 基数排序 - 10 -](#_Toc15991)

[3.2 总体系统的实现 - 12 -](#_Toc5671)

[4 测试 - 14 -](#_Toc12691)

# 1 分析

## 1.1 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。

请在文档中记录上述数据量下，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。

## 1.2 功能分析

8种排序算法的比较案例的核心要求是做出八种排序并同济每种排序所花费的排序时间和交换次数。排序时间需要调用<ctime>库，交换次数用cnt整型变量来记录。

8种排序算法的函数依次为：

**void** BubbleSort(vector<**int**> &data,**int** &cnt)//冒泡排序

**void** SelectSort(vector<**int**> &data,**int** &cnt)//选择排序

**void** InsertSort(vector<**int**> &data,**int** &cnt)//插入排序

**void** ShellSort(vector<**int**> &data,**int** &cnt)//选择排序

**void** QuickSort(vector<**int**> &data,**int** left,**int** right,**int** &cnt)

//快速排序

**void** Siftup(vector<**int**> &array,**int** size,**int** pos,**int** &cnt)//堆化

**void** HeapSort(vector<**int**> &data,**int** &cnt)//堆排序

**void** Merge(vector<**int**> &data,vector<**int**> &output,**int** first,**int** mid,**int** last,**int** &cnt)

**void** Msort(vector<**int**> &data,vector<**int**> &output,**int** s,**int** t,**int** &cnt)

**void** MergeSort(vector<**int**> &data,**int** &cnt)//归并排序

**int** MaxBit(vector<**int**> &data)//求出数据中最大的位数

**void** RadixSort(vector<**int**> &data,**int** &cnt)

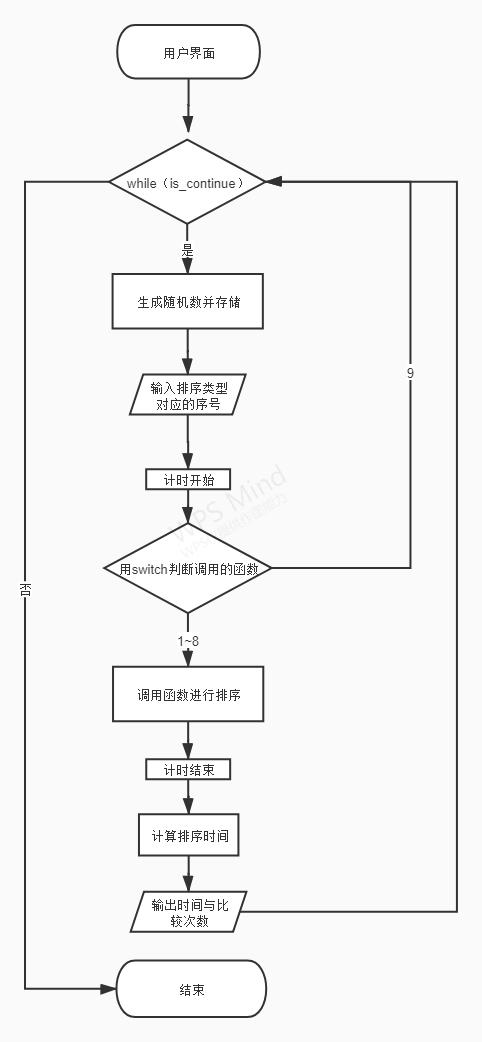
# 2 设计

## 2.1 排序算法设计

从体来说，每种排序算法都有vector<**int**> &data和**int** &cnt这两个引用，用来记录排序后的数列和交换的次数。其中，部分排序需要多个子函数和多个形参来实现。每一种排序的说明将在后面一一列举，此处不再赘述。

## 2.2 主程序设计

首先制作可一个输出窗口作为用户界面进行提示。之后进入while循环，用户先输入数据规模n，然后随机生成n个数并存进不定长数组vector中，然后通过用户输入的数字用switch选择排序方式，调用相关函数完成排序后，输出交换的时间和次数。while循环到到用户输入数字9之后退出。



# 3 实现

## 3.1 八个排序方法的实现

### 3.1.1 冒泡排序

从第0个数开始，各位置依次与后一个数进行比较，如果后者比前者小，则交换二者位置；如果后者比前者大，则不做操作。完成一次全序列的比较后，序列最后的数是此序列的最大值。然后继续进行排序，此时参与冒泡的序列为上一轮序列去掉最大值的序列。重复这个过程n-1次。

每一次过程都是大的元素向上浮，小的元素沉下来 ，所以取名为冒泡排序。冒泡排序是稳定的排序，它的平均时间复杂度和最坏时间复杂度均为O(n2)。

代码如下：

1. **void** BubbleSort(vector<**int**> &data,**int** &cnt)//冒泡排序
2. {
3. **int** n=data.size();
4. **for**(**int** i=0;i<n-1;i++)
5. {
6. **for**(**int** j=0;j<n-i-1;j++)
7. {
8. **if**(data[j]>data[j+1])
9. {
10. //大的元素向上浮，小的元素沉下来
11. swap(data[j],data[j+1]);
12. cnt++;
13. }
14. }
15. }
16. }

### 3.1.2 选择排序

选择排序和冒泡排序类似，每一次循环，都能将最大或者最小的元素换到序列的最左边，然后进行n-1次循环，序列长度逐次减一，最终这个序列就变成有序的序列了。与冒泡排序相比，min\_pos记录的是当前最小值位置，而不是遇到一个相对小的就去和最左边的数进行交换，减少了交换次数。

选择排序是不稳定的排序。它的平均时间复杂度和最坏时间复杂度均为O(n2)。

1. **void** SelectSort(vector<**int**> &data,**int** &cnt)//选择排序
2. {
3. **int** min\_pos,n=data.size();//min\_pos记录的是当前最小值位置
4. **for**(**int** i=0;i<n-1;i++)
5. {
6. min\_pos=i;
7. **for**(**int** j=i+1;j<n;j++)
8. **if**(data[min\_pos]>data[j]) min\_pos=j;
9. //比较大小，更新min\_pos值
10. **if**(min\_pos!= i)
11. {
12. swap(data[min\_pos],data[i]);//将最小值交换到数组最左端
13. cnt++;
14. }
15. }
16. }

### 3.1.3 直接插入排序

直接插入排序算法的思想是：定义一个有序数列，之后遍历原本的无序数列，无序数列的每一个关键码再通过遍历有序数列进行插入，直到右边的无序数列为空为止，左边的有序数列即为最终所求的数列。

本题在实现的过程中，将data[1]~data[j-1]设为有序数列，data[j]到data[n]设为无序数列。每一次循环是将无序数列第一个元素data[j]插入到左边，从右往左依次交换，直到无法交换为止，此时插入过程完成。

直接插入排序是稳定的排序。它的平均时间复杂度和最坏时间复杂度均为O(n2)。

代码如下：

1. **void** InsertSort(vector<**int**> &data,**int** &cnt)//直接插入排序
2. {
3. **int** n=data.size();
4. **for**(**int** i=1;i<n;i++)
5. {
6. //data[1]~data[j-1]为有序数列，data[j]到data[n]为无序数列
7. //排序的时候是将无序数列第一个元素data[j]插入到左边
8. **for**(**int** j=i;j>=1 && data[j]<data[j-1];j--)
9. //从右往左依次交换，直到无法交换为止，此时插入过程完成
10. {
11. swap(data[j],data[j-1]);
12. cnt++;
13. }
14. }
15. }

### 3.1.4 希尔排序

希尔排序的算法思想是将数组按一定间隔gap进行分离成几个组，然后分别对每个组执行插入排序，执行完毕后，缩小gap，循环进行该过程，直到gap等于1，此时最后再进行一次直接插入排序，就完成了。

想要弄清希尔排序关键码的比较次数和对象移动次数和增量选择之间的依赖关系，并给出完整的数学分析是无法做到的。目前有大量的实验数据表明，关键码的平均比较次数和对象平均移动次数大约在n1.25到1.6n1.25之间。

由于希尔排序存在远距离交换，所以它是不稳定的排序。

1. **void** ShellSort(vector<**int**> &data,**int** &cnt)
2. {
3. **int** gap,n=data.size();
4. **bool** f=1;
5. **while**(f)
6. //当gap==1时，f变为0，做完这一次就结束
7. {
8. **if**(gap==1) f=0;
9. //直接插入排序，间隔为gap
10. **for**(**int** i=gap;i<n;i++)
11. {
12. //认为i左边的序列有序，将data[i]往左边的那一组序列插入
13. **int** j=i-gap;
14. **int** temp=data[i];
15. **for**(;j>=0;j-=gap)
16. {
17. **if**(data[j]>temp)
18. {
19. data[j+gap]=data[j];
20. cnt++;
21. }
22. **else** **break**;
23. }
24. data[j+gap]=temp;
25. cnt++;
26. }
27. gap=gap/3+1;//knuth提出gap取(gap/3+1)时最好
28. }
29. }

### 3.1.5 快速排序

快排采用分治的思想，选取一个数字作为基准值，将比它小的元素放在该元素位置之前，比它大的元素放在该元素之后。然后在隔开的两个部分再进行递归如上操作，将序列不断划分，直到排序完成。

最坏的情况下，快排的时间复杂度为O(n2)，一般情况下时间复杂度为O(nlogn)。由于快排为递归函数，所以需要消耗栈空间。在最坏情况下，快拍的空间复杂度为O(n)一般情况下，空间复杂度为O(logn)。存在远距离交换,是不稳定的排序。

1. **void** QuickSort(vector<**int**> &data,**int** left,**int** right,**int** &cnt)//快速排序
2. {
3. **if**(left>=right) **return**;
4. **int** i=left,j=right;
5. **int** base=data[left];//取最左边的数为基准数
6. //通过三个while循环来调整这个区间
7. //使得基准数左边的比它小，右边的比它大
8. **while**(i<j)
9. {
10. **while**(data[j]>=base && i<j) j--;
11. //从右往左找到下一个比基准数小的位置
12. **while**(data[i]<=base && i<j) i++;
13. //从左往右找到下一个比基准数大的位置
14. **if**(i<j)
15. {
16. swap(data[i],data[j]);
17. cnt++;
18. }
19. }
21. swap(data[left],data[i]);
22. cnt++;
23. //因为基准数一直在data[left]处
24. //所以这个交换为了让基准数归位第i位小于基准数，第i+1也就是第j位大于基准数
26. //以基准数的位置为标准左右分别递归，这也是分治思想的体现
27. QuickSort(data,left,i-1,cnt);//递归基准数左边
28. QuickSort(data,i+1,right,cnt);//递归基准数右边
29. }

### 3.1.6 堆排序

Siftdown函数主要是将元素下沉，即通过对比父节点pos与它的下一层左右子节点2pos与2pos+1中较大的一个，如果父节点的值小则交换。通过while循环不断比较交换，结束条件是到达堆末尾或者到达了关键码大于子节点的父节点处。这样每一个父节点都满足大于子节点，大根堆构造成功。

用大根堆的主要原因是可以重复利用空间结构进行存储，节约了空间。

首先初始化堆，此时堆顶的元素即为最大的值，此时，将堆顶元素与堆的最后一个元素交换，再将记录堆的数组长度减一，然后通过向下转移Siftdown函数重新排序，while循环n-1次即可得到排序排好的数组。

堆排序的最好、最坏和一般情况下的时间复杂度均为O(nlogn)。存在远距离交换，所以堆排序是不稳定的排序。

1. **void** Siftdown(vector<**int**> &array,**int** size,**int** pos,**int** &cnt)//堆化
2. {
3. **int** left\_child=pos\*2+1,right\_child=pos\*2+2;
4. **while**(right\_child<size)//当结点有孩子的时候一直做
5. {
6. **if**(array[pos]>=array[left\_child] && array[pos]>=array[right\_child]) **return**;
7. **if**(array[left\_child]>=array[right\_child])
8. {
9. swap(array[pos],array[left\_child]);
10. pos=left\_child;
11. cnt++;
12. }
13. **else**
14. {
15. swap(array[pos],array[right\_child]);
16. pos=right\_child;
17. cnt++;
18. }
19. left\_child=pos\*2+1,right\_child=pos\*2+2;
20. }
21. **if**(left\_child<size&&array[left\_child]>array[pos])
22. //左孩子还可能在长度内，判断是否需要调整
23. {
24. swap(array[pos],array[left\_child]);
25. cnt++;
26. }
27. **return**;
28. }
29. **void** HeapSort(vector<**int**> &data,**int** &cnt)//堆排序
30. {
31. **int** n=data.size();
32. **for**(**int** i=n-1;i>=0;i--) Siftdown(data,n,i,cnt);//初始堆
33. **while**(n>0)//重建最大堆
34. {
35. swap(data[n-1],data[0]);
36. n--;
37. Siftdown(data,n,0,cnt);
38. }
39. }

### 3.1.7 归并排序

与快速排序类似，归并排序也采用了分治的思想。首先将整个待排序列分为左子表和右子表。对这些子表分别进行递归地进行排序，然后再把排好序的两个子表进行归并。划分到最后的结果就是，子表只有一个元素或者没有元素，这种子表属于排完了序的，直接返回。然后返回上一层递归，不断合并，一直到整个数组合并为一个有序数组。

我们还定义了一个辅助函数Merge（混合）来将左右两个已经排完序的子表进行合并。方法是：用两个变量分别指向左右两个子表的开头，然后比较指向的两个元素，辅助数组output存较小的那个元素，然后指向该元素的后面一个位置，一直对比到其中一个子表被遍历完。之后，未遍历完的数组的剩余元素直接全部存储剩余进output辅助数组就行了。

归并排序将序列进行划分的时间复杂度为O(logn)，而每次进行合并的时间复杂度为O(n)。所以时间复杂度为O(nlogn)。

1. **void** Merge(vector<**int**> &data,vector<**int**> &output,**int** first,**int** mid,**int** last,**int** &cnt)
2. //归并排序
3. {
4. **int** left=first,right=mid+1;
5. **int** tail=first;//记录output末尾的位置
6. **while**(left<=mid && right<=last)
7. {
8. **if**(data[left]<=data[right])
9. {
10. output[tail++]=data[left++];
11. cnt++;
12. }
13. **else** **if**(data[right]<data[left])
14. {
15. output[tail++]=data[right++];
16. cnt++;
17. }
18. }
19. **while**(left<=mid) output[tail++]=data[left++];
20. //如果左序列未到头，而右序列已经结束
21. **while**(right<=last) output[tail++]=data[right++];
22. //如果右序列未到头，而左序列已经结束
23. }
25. //通过递归将待排序数列不断二分，进行排序
26. **void** Msort(vector<**int**> &data,vector<**int**> &output,**int** s,**int** t,**int** &cnt) {
27. vector<**int**> temp(data.size());
28. **if**(s==t) output[s]=data[s];
29. **else**
30. {
31. **int** m=(s+t)/2;
32. Msort(data,temp,s,m,cnt);//排左边的
33. Msort(data,temp,m+1,t,cnt);//排右边的
34. Merge(temp,output,s,m,t,cnt);//混合
35. }
36. }
37. **void** MergeSort(vector<**int**> &data,**int** &cnt)//归并排序
38. {
39. **int** n=data.size();
40. vector<**int**> output(n);
41. Msort(data,output,0,n-1,cnt);
42. data=output;
43. }

### 3.1.8 基数排序

基数排序是采用“分配”和“收集”的办法，用对多关键码进行排序的思想实现对单关键码进行排序的方法。

首先确定基数radix，因为是对数进行排序，故radix从1开始取，之后依次乘10。然后遍历所有的数，将当前位相同的数分配到同一块连续的存储区域，记录存储区。

基数排序的时间复杂度为O(d(n+radix))，其中d为最大元素的位数，radix为基数的个数。基数排序是稳定的排序方法。

1. **int** MaxBit(vector<**int**> &data)//求出数据中最大的位数
2. {
3. **int** maxx=1,flag=10;
4. **for**(**int** i=0;i<data.size();i++)
5. {
6. **while**(data[i]>flag)
7. {
8. flag\*=10;
9. maxx++;
10. }
11. }
12. **return** maxx;
13. }
15. **void** RadixSort(vector<**int**> &data,**int** &cnt)
16. {
17. **int** n=data.size(),maxx=MaxBit(data),radix=1;
18. **int** i,j,k;
19. vector<**int**> temp(n),count(10);
20. **for**(i=1;i<=maxx;i++)//根据最大位数决定排序次数
21. {
22. **for**(j=0;j<10;j++) count[j]=0;//每次分配前清空计数器
23. **for**(j=0;j<n;j++)//统计这一位上数字0~9的个数
24. {
25. k=(data[j]/radix)%10;
26. count[k]++;
27. }
28. **for**(j=1;j<10;j++) count[j]=count[j-1]+count[j];
29. //将temp中的位置依次分配
30. **for**(j=n-1;j>=0;j--)//将所有记录依次收集到temp中
31. {
32. k=(data[j]/radix)%10;
33. temp[count[k]-1]=data[j];
34. cnt++;
35. count[k]--;
36. }
37. data=temp;//将临时数组的内容复制到data中
38. radix\*=10;
39. }
40. }

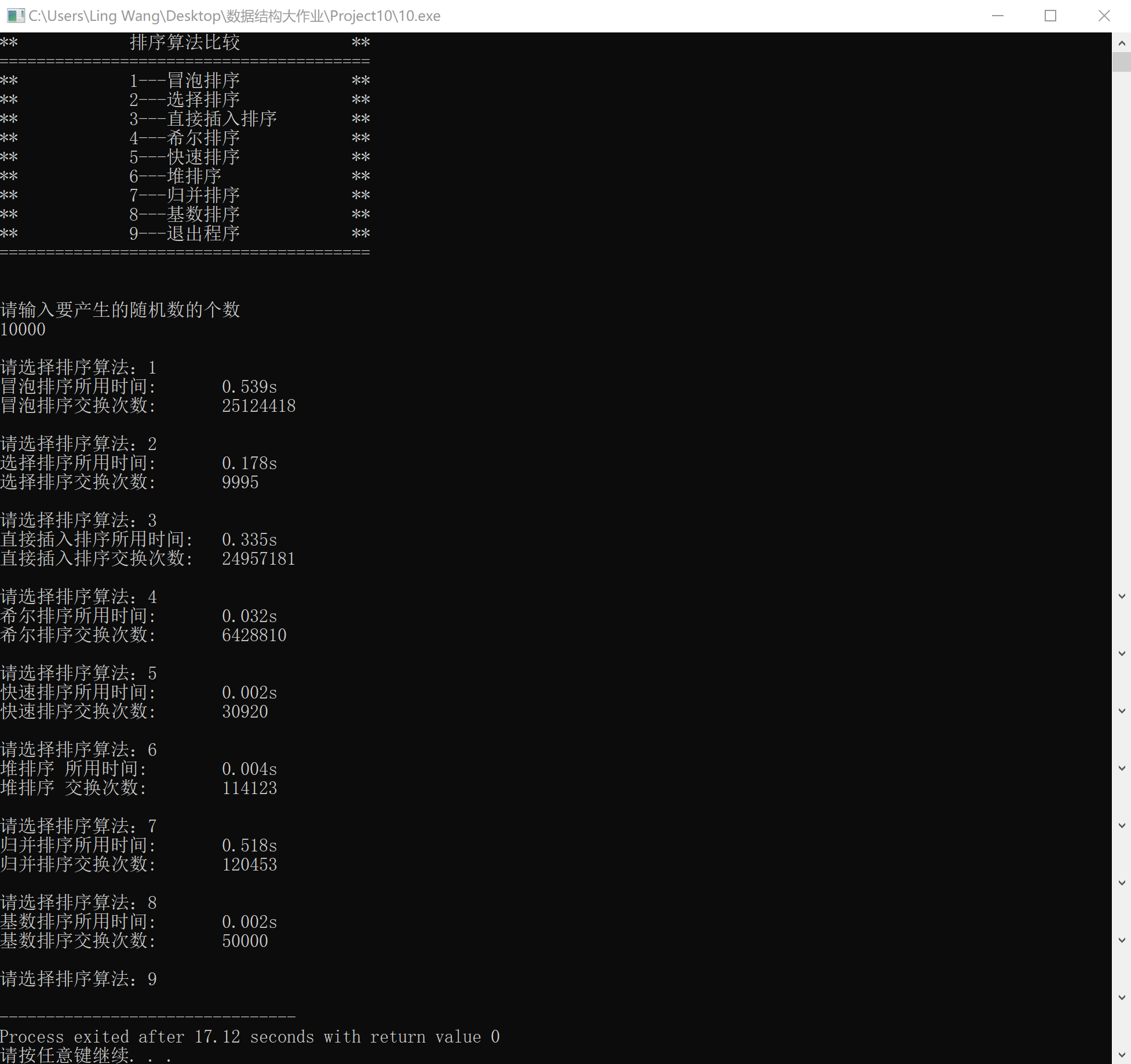
## 3.2 总体系统的实现

首先制作可一个输出窗口作为用户界面进行提示。之后进入while循环，用户先输入数据规模n，然后随机生成n个数并存进不定长数组vector中，然后通过用户输入的数字用switch选择排序方式，调用相关函数完成排序后，输出交换的时间和次数。while循环到到用户输入数字9之后退出。

1. **int** main()
2. {
3. vector<string> SortIndex=
4. {"","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"};
5. cout<<"\*\*            排序算法比较            \*\*"<<endl;
6. cout<<"========================================"<<endl;
7. cout<<"\*\*            1---冒泡排序            \*\*"<<endl;
8. cout<<"\*\*            2---选择排序            \*\*"<<endl;
9. cout<<"\*\*            3---直接插入排序        \*\*"<<endl;
10. cout<<"\*\*            4---希尔排序            \*\*"<<endl;
11. cout<<"\*\*            5---快速排序            \*\*"<<endl;
12. cout<<"\*\*            6---堆排序              \*\*"<<endl;
13. cout<<"\*\*            7---归并排序            \*\*"<<endl;
14. cout<<"\*\*            8---基数排序            \*\*"<<endl;
15. cout<<"\*\*            9---退出程序            \*\*"<<endl;
16. cout<<"========================================"<<endl;
17. cout<<endl<<endl<<"请输入要产生的随机数的个数"<<endl;
18. **int** n,choice=0;
19. **bool** is\_continue=**true**;
20. cin>>n;
21. **while**(is\_continue)
22. {
23. vector<**int**> data(n);
24. srand((unsigned)time(NULL));
25. data=\***new** vector<**int**>();
26. **for**(**int** i=0;i<n;i++) data.push\_back(rand());
27. cout<<endl<<"请选择排序算法：";
28. cin>>choice;
29. **int** cnt=0;
30. **clock\_t** start=clock();
31. **switch**(choice)
32. {
33. **case** 1:
34. BubbleSort(data,cnt);
35. **break**;
36. **case** 2:
37. SelectSort(data,cnt);
38. **break**;
39. **case** 3:
40. InsertSort(data,cnt);
41. **break**;
42. **case** 4:
43. ShellSort(data,cnt);
44. **break**;
45. **case** 5:
46. QuickSort(data,0,data.size()-1,cnt);
47. **break**;
48. **case** 6:
49. HeapSort(data,cnt);
50. **break**;
51. **case** 7:
52. MergeSort(data,cnt);
53. **break**;
54. **case** 8:
55. RadixSort(data,cnt);
56. **break**;
57. **case** 9:
58. is\_continue=**false**;
59. **break**;
60. **Default:**
61. cout<<"没有这个选项，请重新输入"<<endl;
62. **break**;
63. }
64. **if**(choice>=1 && choice<=8)
65. {
66. **clock\_t** end=clock();
67. cout<<SortIndex[choice]<<"所用时间:\t"<<(**double**)(end-start)/CLOCKS\_PER\_SEC<<"s"<<endl;
68. cout<<SortIndex[choice]<<"交换次数:\t"<<cnt<<endl;
69. }
70. }
71. **return** 0;
72. }

# 4 测试

## 4.1 答案检测



## 4.2 边界检测

