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

作 者 姓 名： 赵子昱

学 号： 1951459

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc60149489)

[1.1 项目简介 1](#_Toc60149490)

[1.2 功能分析 1](#_Toc60149491)

[2 设计 2](#_Toc60149492)

[2.1 类结构设计 2](#_Toc60149493)

[2.2 成员与操作设计 2](#_Toc60149494)

[2.3 系统设计 3](#_Toc60149495)

[3 实现 4](#_Toc60149496)

[3.1冒泡排序**BubbleSort()**的实现 4](#_Toc60149497)

[3.1.1 冒泡排序代码 4](#_Toc60149498)

[3.2 选择排序**SelectSort()**的实现 4](#_Toc60149499)

[3.2.1 选择排序代码 4](#_Toc60149500)

[3.3 直接插入排序**InsertionSort()**的实现 5](#_Toc60149501)

[3.3.1 直接插入排序代码 5](#_Toc60149502)

[3.4 希尔排序**ShellSort()**的实现 6](#_Toc60149503)

[3.4.1 希尔排序代码 6](#_Toc60149504)

[3.5 快速排序**QuickSort()**的实现 7](#_Toc60149505)

[3.5.1 快速排序代码 7](#_Toc60149506)

[3.6 堆排序**FliterDown()**的实现 8](#_Toc60149507)

[3.6.1 堆排序代码 8](#_Toc60149508)

[3.7归并排序**MergeSort ()**的实现 9](#_Toc60149509)

[3.7.1 堆排序代码 9](#_Toc60149510)

[3.8基数排序**RadixSort ()**的实现 11](#_Toc60149511)

[3.8.1 基数排序代码 11](#_Toc60149512)

[3.9 **Menu()**函数的实现 12](#_Toc60149513)

[3.9.1 Menu()代码 12](#_Toc60149514)

[3.10主函数的实现 13](#_Toc60149515)

[3.10.1 主函数代码 13](#_Toc60149516)

[4 数据记录和分析 15](#_Toc60149517)

[4.1 测试数据记录 15](#_Toc60149518)

[4.1.1 100个随机数 15](#_Toc60149519)

[4.1.2 1000个随机数 16](#_Toc60149520)

[4.1.3 10000个随机数 17](#_Toc60149521)

[4.1.4 100000个随机数 18](#_Toc60149522)

[4.2 不同排序方法的比较分析 19](#_Toc60149523)

# 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

首先，要在程序中实现八种排序算法，并在适当的位置增加比较次数与移动次数计数及计时。

最后，在主程序中，给用户提供界面，选择某种排序算法后，展示排序所需要的时间和比较次数。用户可以不断选择排序算法直到退出。用户也可以任意选择想排序的规模。

# 2 设计

## 2.1 类结构设计

项目中定义了一个**Sortor**类，封装了各个排序算法函数与要排序的数据表。

## 2.2 成员与操作设计

**Sortor类**

1. **template** <**class** T>
2. **class** Sortor {
3. **public**:
4. Sortor();
5. ~Sortor() { **delete**[]m\_DataList; }
7. **void** BubbleSort() **const**;
8. //冒泡排序
10. **void** SelectSort() **const**;
11. //选择排序
13. **void** InsertionSort() **const**;
14. //直接插入排序
16. **void** ShellSort() **const**;
17. //希尔排序
19. **void** QuickSort() **const**;
20. //快速排序
22. **void** HeapSort() **const**;
23. //堆排序
25. **void** MergeSort() **const**;
26. //归并排序
28. **void** RadixSort() **const**;
29. //基数排序
31. **private**:
32. T \*m\_DataList;
33. //数据表
35. **int** m\_Count;
36. //排序元素个数
38. T\* Copy() **const**;
39. //数据表复制函数（不带哨兵）
41. T\* Copy(**int** guard) **const**;
42. //数据表复制函数（带哨兵），用于直接插入排序
44. **void** Swap(T \*list, **const** **int** x, **const** **int** y,**int** &swaptime)**const**;
45. //交换
47. **void** QuickSort(T \*DataList,**int** left,**int** right,**int** &swaptime,**int** &ConTime) **const**;
48. //用于快速排序
50. **void** FliterDown(T\* Heap,**int** start,**int** end, **int** &swaptime, **int** &ConTime) **const**;
51. //堆的向下调整，用于堆排序
53. **void** Merge(T\* oldData, T\* newData, **int** Acurrent, **int** Bcurrent, **int** Bhigh, **int** &swaptime, **int** &ConTime)**const**;
54. //用于归并排序
56. };

## 2.3 系统设计

系统首先调用Menu()函数实现对屏幕的初始化，然后创建一个Sortor对象，应用switch结构，根据用户输入的操作码来调用相应的排序函数。

# 3 实现

## 3.1冒泡排序**BubbleSort()**的实现

将数据表从后向前扫描至多n-1次，如发现逆序则交换两个元素的顺序，直到扫描够n-1或某次扫描无逆序。第i次循环后第i个位置有序。

### 3.1.1 冒泡排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::BubbleSort() **const** {
3. T\* DataList = Copy();
4. **long** **long** SwapTime = 0, ConTime = 0;
5. **clock\_t** start, end;
6. start = clock();
7. **bool** exchangeTag = **false**;
8. **for** (**int** i = m\_Count - 1; i > 0; i--) {
9. **int** end = m\_Count - 1 - i;
10. **for** (**int** j = m\_Count - 1; j > end; j--) {
11. **if** (ConTime++,DataList[j - 1] > DataList[j]) {
12. Swap(DataList, j - 1, j, SwapTime);
13. exchangeTag = **true**;
14. }//if
15. }//for
16. **if** (!exchangeTag) { **break**; }
17. exchangeTag = **false**;
18. }//for
19. end = clock();
20. **delete**[]DataList;
21. cout << "比较次数：" << ConTime << endl;
22. cout << "交换次数：" << SwapTime << endl;
23. cout << "运行时间：" << end - start << endl;
24. }

## 3.2 选择排序**SelectSort()**的实现

执行n-1次循环，第i次循环从数据表的第i个数据扫描到表尾，找出最小的数据并与第i为交换。每次循环后第i个位置有序。

### 3.2.1 选择排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::SelectSort() **const** {
3. T\* DataList = Copy();
4. **long** **long** SwapTime = 0, ConTime = 0;
5. **clock\_t** start, end;
6. start = clock();
7. **for** (**int** i = 1; i < m\_Count; i++) {
8. T min = DataList[i - 1];
9. **int** minNum = i - 1;
10. **for** (**int** j = i; j < m\_Count; j++) {
11. **if** (ConTime++, DataList[i] < min) {
12. min = DataList[i];
13. minNum = i;
14. }//if
15. }//for
16. Swap(DataList, i - 1, minNum,SwapTime);
17. }
18. end = clock();
19. **delete**[]DataList;
20. cout << "比较次数：" << ConTime << endl;
21. cout << "交换次数：" << SwapTime << endl;
22. cout << "运行时间：" << end - start << endl;
23. }

## 3.3 直接插入排序**InsertionSort()**的实现

做n-1次循环，第i次循环将数据表的第i+1个元素插入到前位组成的有序序列的正确位置。

### 3.3.1 直接插入排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::InsertionSort() **const** {
3. T\* DataList = Copy(1);
4. **long** **long** SwapTime = 0, ConTime = 0;
5. **clock\_t** start, end;
6. start = clock();
7. **for** (**int** i = 2; i <= m\_Count; i++) {
8. DataList[0] = DataList[i];
9. **int** icpy = i;
10. **while**(ConTime++,DataList[--icpy]> DataList[0]){}
11. **for** (**int** j = i - 1; j > icpy; j--) {
12. DataList[j + 1] = DataList[j];
13. SwapTime++;
14. }
15. DataList[icpy + 1] = DataList[0];
16. SwapTime++;
17. }//for
18. end = clock();
19. **delete**[]DataList;
20. cout << "比较次数：" << ConTime << endl;
21. cout << "交换次数：" << SwapTime << endl;
22. cout << "运行时间：" << end - start << endl;
23. }

## 3.4 希尔排序**ShellSort()**的实现

首先取一个整数gap = n / 2 作为间隔，将全部对象分为gap 个子序列，所有距离为gap 的对象放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔gap = gap == 2 ? 1 : (int)gap / 2.2，重复上述的子序列划分和排序工作。直到最后取gap = 1，将所有对象放在同一个序列中排序为止。

### 3.4.1 希尔排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::ShellSort() **const** {
3. T\* DataList = Copy();
4. **long** **long** SwapTime = 0, ConTime = 0;
5. **clock\_t** start, end;
6. start = clock();
7. **int** gap = m\_Count / 2;
8. **while** (gap) {
9. **for** (**int** i = gap; i < m\_Count; i++) {
10. T tmp = DataList[i];
11. **int** icpy = i - gap;
12. **while** (ConTime++, DataList[icpy] > tmp&&icpy >= 0) {
13. DataList[icpy + gap] = DataList[icpy];
14. SwapTime++;
15. icpy -= gap;
16. }//while
17. DataList[icpy + gap] = tmp;
18. }//for
19. gap = gap == 2 ? 1 : (**int**)gap / 2.2;
20. }//while
21. end = clock();
22. **delete**[]DataList;
23. cout << "比较次数：" << ConTime << endl;
24. cout << "交换次数：" << SwapTime << endl;
25. cout << "运行时间：" << end - start << endl;
26. }

## 3.5 快速排序**QuickSort()**的实现

项目中取数据表的第一个元素作为划分的基准元素，采用左右两个指针，从右边选一个小的移动到左边再从左边选一个大的移动到右边的方法进行划分，直到两个指针相遇，然后递归对左右序列进行划分。

### 3.5.1 快速排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::QuickSort(T \*DataList, **int** left, **int** right, **long** **long** &swaptime, **long** **long** &ConTime) **const** {
3. T tmp = DataList[left];
4. **int** small=left, big=right+1;
5. **bool** tag = **true**;
6. **while** (big > small) {
7. **switch** (tag){
8. **case** **true**:
9. **while**(ConTime++,DataList[--big]>=tmp){}
10. **if** (big > small) {
11. DataList[small] = DataList[big];
12. swaptime++;
13. tag = **false**;
14. }
15. **else** {
16. DataList[small] = tmp;
17. swaptime++;
18. **if** (small - left > 1) { QuickSort(DataList, left, small - 1, swaptime, ConTime); }
19. **if** (right - small > 1) { QuickSort(DataList, small + 1, right, swaptime, ConTime); }
20. }
21. **break**;
22. **default**:
23. **while** (ConTime++, DataList[++small] <= tmp) {}
24. **if** (big > small) {
25. DataList[big] = DataList[small];
26. swaptime++;
27. tag = **true**;
28. }
29. **else** {
30. DataList[big] = tmp;
31. swaptime++;
32. **if** (big - left > 1) { QuickSort(DataList, left, big - 1, swaptime, ConTime); }
33. **if** (right - big > 1) { QuickSort(DataList, big + 1, right, swaptime, ConTime); }
34. }
35. **break**;
36. }
37. }
38. }
40. **template** <**class** T>
41. **void** Sortor<T>::QuickSort() **const** {
42. T\* DataList = Copy();
43. **long** **long** SwapTime = 0, ConTime = 0;
44. **clock\_t** start, end;
45. start = clock();
46. QuickSort(DataList, 0, m\_Count - 1, SwapTime, ConTime);
47. end = clock();
48. **delete**[]DataList;
49. cout << "比较次数：" << ConTime << endl;
50. cout << "交换次数：" << SwapTime << endl;
51. cout << "运行时间：" << end - start << endl;
52. }

## 3.6 堆排序**FliterDown()**的实现

首先从队尾开始向下调整，将数据表调成最大堆，然后进行n-1次循环，第i次将第一个元素与第n+1-i个交换，这时第n+1-i个元素有序，然后从第一个元素到第n-1个元素进行向下调整重新调成最大堆，进入下一循环。

### 3.6.1 堆排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::FliterDown(T\* Heap, **int** start, **int** end, **long** **long** &swaptime, **long** **long** &ConTime)**const** {
3. **if** (2 \* start + 1 > end) { **return**; }
4. T temp = Heap[start];
5. **int** Parent = start, LeftChild = 2\*start+1, BigChild;
6. **while** (LeftChild <= end) {
7. //选出较大的孩子
8. **if** (LeftChild + 1 <= end && (ConTime++,Heap[LeftChild] < Heap[LeftChild + 1])) {
9. BigChild = LeftChild + 1;
10. }
11. **else** { BigChild = LeftChild; }
12. //如果较小的孩子小于要调整的元素则调换位置
13. **if** (ConTime++, Heap[BigChild] > temp) {
14. Heap[Parent] = Heap[BigChild];
15. swaptime++;
16. }
17. **else** { **break**; }
18. LeftChild = 2 \* BigChild + 1;
19. Parent = BigChild;
20. }
21. Heap[Parent] = temp;
22. }
24. **template** <**class** T>
25. **void** Sortor<T>::HeapSort() **const** {
26. T\* DataList = Copy();
27. **long** **long** SwapTime = 0, ConTime = 0;
28. **clock\_t** start, end;
29. start = clock();
30. **for** (**int** i = 0; i < m\_Count; i++) {
31. FliterDown(DataList, m\_Count - 1 - i, m\_Count - 1, SwapTime, ConTime);
32. }
33. **for** (**int** i = m\_Count - 1; i > 0; i--) {
34. Swap(DataList, 0, i,SwapTime);
35. FliterDown(DataList, 0, i - 1, SwapTime, ConTime);
36. }
37. end = clock();
38. **delete**[]DataList;
39. cout << "比较次数：" << ConTime << endl;
40. cout << "交换次数：" << SwapTime << endl;
41. cout << "运行时间：" << end - start << endl;

## 3.7归并排序**MergeSort ()**的实现

首先将数据表的元素进行两两归并，得到若干个长度为2的有序表，然后不断将有序表两两归并，直到有序表的长度大于n。

### 3.7.1 堆排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::Merge(T\* oldData, T\* newData, **int** Acurrent, **int** Bcurrent, **int** Bhigh,**long** **long** &swaptime, **long** **long** &ConTime)**const** {
3. **int** Ahigh = Bcurrent - 1, newCurrent = Acurrent;
4. **while** (Acurrent <= Ahigh && Bcurrent <= Bhigh) {
5. **if** (ConTime++, oldData[Acurrent] <= oldData[Bcurrent]) {
6. newData[newCurrent++] = oldData[Acurrent++];
7. swaptime++;
8. }
9. **else** {
10. newData[newCurrent++] = oldData[Bcurrent++];
11. swaptime++;
12. }
13. }//while
14. **if** (Acurrent > Ahigh) {
15. **for** (**int** i = Bhigh - Bcurrent + 1; i > 0; i--) {
16. newData[newCurrent++] = oldData[Bcurrent++];
17. swaptime++;
18. }
19. }//if
20. **else** {
21. **for** (**int** i = Ahigh - Acurrent + 1; i > 0; i--) {
22. newData[newCurrent++] = oldData[Acurrent++];
23. swaptime++;
24. }
25. }//else
26. }
28. **template** <**class** T>
29. **void** Sortor<T>::MergeSort() **const** {
30. T\* DataList1 = Copy();
31. T\* DataList2 = **new** T[m\_Count];
32. T \*Old, \*New;
33. **long** **long** SwapTime = 0, ConTime = 0, sortedNum;
34. **clock\_t** start, end;
35. start = clock();
36. **bool** List1Emp = **false**;
37. **for** (**int** len = 1; len < m\_Count; len \*= 2) {
38. sortedNum = 0;
39. **if** (List1Emp) {
40. Old = DataList2;
41. New = DataList1;
42. List1Emp = **false**;
43. }
44. **else** {
45. New = DataList2;
46. Old = DataList1;
47. List1Emp = **true**;
48. }
49. **while** (sortedNum + 2 \* len <= m\_Count) {
50. Merge(Old, New, sortedNum, sortedNum + len, sortedNum + 2 \* len - 1, SwapTime, ConTime);
51. sortedNum += 2 \* len;
52. }//while
53. **if** (m\_Count - sortedNum > len) {
54. Merge(Old, New, sortedNum, sortedNum + len, m\_Count - 1, SwapTime, ConTime);
55. }//if
56. }
57. end = clock();
58. cout << "比较次数：" << ConTime << endl;
59. cout << "交换次数：" << SwapTime << endl;
60. cout << "运行时间：" << end - start << endl;
61. **delete**[]DataList1;
62. **delete**[]DataList2;
63. }

## 3.8基数排序**RadixSort ()**的实现

为了减少数据移动次数，采用静态链表实现基数排序，首先需要定义节点，每个节点包含数据域和指针域，首先利用原数据表完成数据链表的初始化（此部分不计入排序时间），然后以10为基数，循环按位进行分配与收集，得到有序序列。

### 3.8.1 基数排序代码

1. **template** <**class** T>
2. **void** Sortor<T>::RadixSort() **const** {
3. **class** Node {
4. **public**:
5. T data;
6. **int** link;
7. };
8. Node \*DataList = **new** Node[m\_Count];
9. **for** (**int** i = 0; i < m\_Count; i++) {
10. DataList[i].data = m\_DataList[i];
11. DataList[i].link = i + 1;
12. }
13. DataList[m\_Count - 1].link = 0;
14. **long** **long** SwapTime = 0, ConTime = 0;
15. **clock\_t** start, end;
16. start = clock();
17. **const** **int** Radix = 10;
18. **int** rear[Radix], front[Radix], n = 1, Max = 32767, allFront = 0, current = 0;
19. **while** ((Max /= 10) > 0) {
20. n++;
21. }
22. **int** tmp = 1, currentUnit;
23. **for** (**int** i = 0; i < n; i++) {
24. **for** (**int** i = 0; i < Radix; i++) {
25. rear[i] = front[i] = -1;
26. }
27. **for** (**int** j = 0; j < m\_Count; j++) {
28. currentUnit = DataList[current].data / tmp % 10;
29. **if** (front[currentUnit] == -1) { front[currentUnit] = rear[currentUnit] = current; }
30. **else** {
31. DataList[rear[currentUnit]].link = current;
32. rear[currentUnit] = current;
33. }
34. current = DataList[current].link;
35. }
36. **int** lastFull = 0;
37. **bool** first = **true**;
38. **for** (**int** j = 0; j < Radix; j++) {
39. **if** (front[j] != -1) {
40. **if** (first) {
41. allFront = front[j];
42. lastFull = j;
43. first = **false**;
44. }
45. DataList[rear[lastFull]].link = front[j];
46. lastFull = j;
47. }
48. }
49. tmp \*= 10;
50. }
51. end = clock();
52. **delete**[]DataList;
53. cout << "比较次数：" << ConTime << endl;
54. cout << "交换次数：" << SwapTime << endl;
55. cout << "运行时间：" << end - start << endl;
56. }

## 3.9 **Menu()**函数的实现

### 3.9.1 Menu()代码

1. **void** Menu() {
2. cout << "\*\*      排序算法比较      \*\*" << endl;
3. cout << "============================" << endl;
4. cout << "\*\*      1-冒泡排序        \*\*" << endl;
5. cout << "\*\*      2-选择排序        \*\*" << endl;
6. cout << "\*\*      3-直接插入排序    \*\*" << endl;
7. cout << "\*\*      4-希尔排序        \*\*" << endl;
8. cout << "\*\*      5-快速排序        \*\*" << endl;
9. cout << "\*\*      6-堆排序          \*\*" << endl;
10. cout << "\*\*      7-归并排序        \*\*" << endl;
11. cout << "\*\*      8-基数排序        \*\*" << endl;
12. cout << "\*\*      9-退出程序        \*\*" << endl;
13. cout << "============================" << endl;
14. cout << endl;
15. }

## 3.10主函数的实现

主函数首先调用Menu()函数实现对屏幕的初始化，然后创建一个Sortor对象，应用switch结构，根据用户输入的操作码来调用相应的排序函数。

### 3.10.1 主函数代码

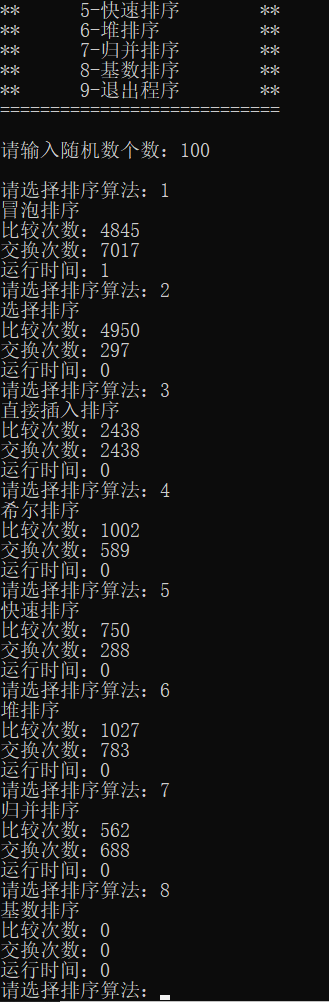
1. **int** main(){
2. Menu();
3. Sortor<**int**> mySortor;
4. **int** operation=0;
5. **while** (operation!=9) {
6. cout << "请选择排序算法：";
7. cin >> operation;
8. **switch** (operation) {
9. **case** 1:
10. cout << "冒泡排序" << endl;
11. mySortor.BubbleSort();
12. **break**;
13. **case** 2:
14. cout << "选择排序" << endl;
15. mySortor.SelectSort();
16. **break**;
17. **case** 3:
18. cout << "直接插入排序" << endl;
19. mySortor.InsertionSort();
20. **break**;
21. **case** 4:
22. cout << "希尔排序" << endl;
23. mySortor.ShellSort();
24. **break**;
25. **case** 5:
26. cout << "快速排序" << endl;
27. mySortor.QuickSort();
28. **break**;
29. **case** 6:
30. cout << "堆排序" << endl;
31. mySortor.HeapSort();
32. **break**;
33. **case** 7:
34. cout << "归并排序" << endl;
35. mySortor.MergeSort();
36. **break**;
37. **case** 8:
38. cout << "基数排序" << endl;
39. mySortor.RadixSort();
40. **break**;
41. **default**:
42. operation = 9;
43. **break**;
44. }
45. }//while
46. system("pause");
47. **return** 0;
48. }

# 4 数据记录和分析

## 4.1 测试数据记录

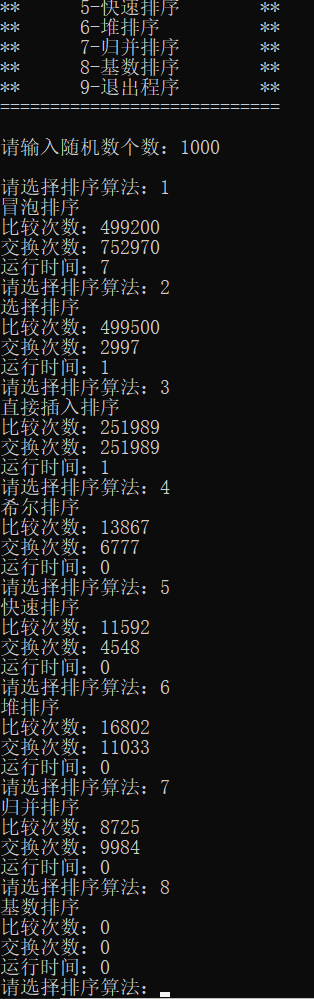
### 4.1.1 100个随机数

**实验结果**：



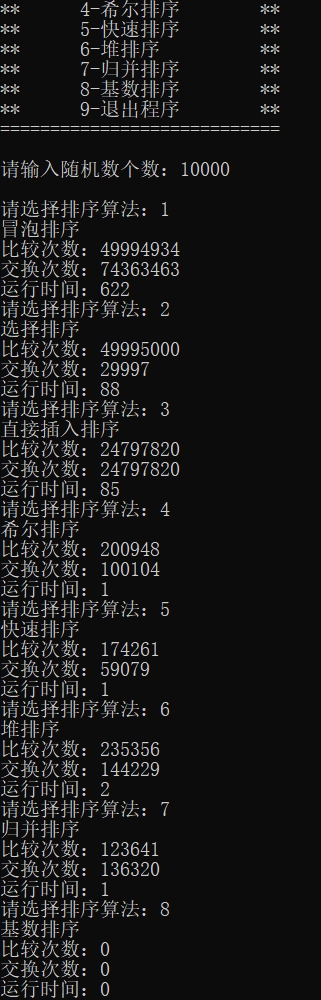
### 4.1.2 1000个随机数

**实验结果：**



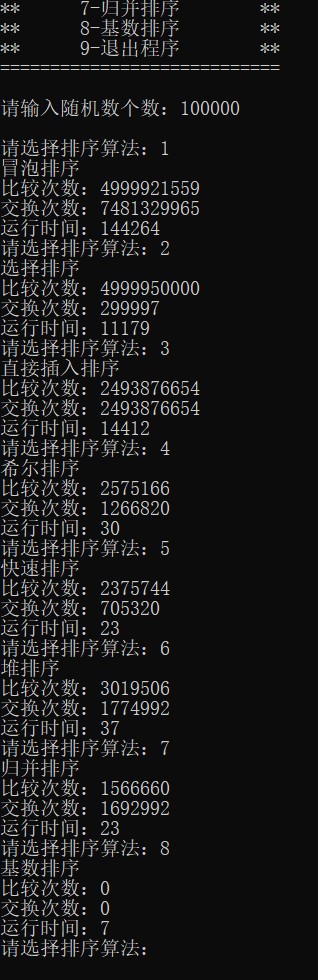
### 4.1.3 10000个随机数

**实验结果：**



### 4.1.4 100000个随机数

**实验结果：**



## 4.2 不同排序方法的比较分析

从时间上来看，在数据量较小时（100、1000）不同算法的排序时间都很短，但能明显看出冒泡排序需要的时间是最长的。在数据量较大时，各算法的排序时间有了明显区别，其中冒泡排序最慢，和比选择排序与直接插入排序多了一个数量级；希尔排序、快排、归并排序、堆排序用时相近，比选择排序与直接插入排序少了三个数量级，基数排序是最快的。

从上述数据不难看出，排序的时间与数据移动次数、数据比较次数成正比，以冒泡排序和基数排序为例，冒泡排序由于每次扫描数据表都需要大量的比较与交换，最后呈现的结果是交换次数与比较次数远大于其他几种算法，而基数排序是静态排序，只需更改数据节点的指针而不需要移动数据，并且没有涉及数据间的比较，所以在几种算法中是最快的，速度与冒泡相比简直一个天上一个地下。

从空间开销来看，冒泡排序，选择排序，直接插入排序、希尔排序、快速排序、堆排序的空间复杂度都是O(1)级别，归并排序需要需要另外一个与原待排序对象数组同样大小的辅助数组，空间复杂度为O(n)级别，基数排序需要增加 n+2radix个附加链接指针，空间复杂度为O(n)级别，因此基数排序与归并排序对设备要需求较高，不适用于超大量的数据排序。

理论分析：

8种排序算法的平均比较次数、平均移动次数，时间、空间复杂度如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 排序算法 | 平均比较次数 | 平均移动次数 | 时间复杂度 | 空间复杂度 |
| 冒泡排序 | O(*n*2) | O(*n*2) | O(*n*2) | O(1) |
| 选择排序 | O(*n*2) | O(*n*) | O(*n*2) | O(1) |
| 直接插入排序 | O(*n*2) | O(*n*2) | O(*n*2) | O(1) |
| 希尔排序 | O(*n*1.25) | O(*n*1.25) | O(*n*1.25) | O(1) |
| 快速排序 | O(*n*log2*n*) | O(*n*log2*n*) | O(*n*log2*n*) | O(log2*n*) |
| 堆排序 | O(*n*log2*n*) | O(*n*log2*n*) | O(*n*log2*n*) | O(1) |
| 归并排序 | O(*n*log2*n*) | O(*n*log2*n*) | O(*n*log2*n*) | O(*n*) |
| 基数排序 | 0 | 0 | O(*n*) | O(*n+r*) |