项目说明文档

数据结构课程设计

——八种排序算法的比较

作 者 姓 名： 郑柯凡

学 号： 1950072

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 3](#_Toc24482)

[1.1 背景分析 3](#_Toc6747)

[1.2 功能分析 3](#_Toc24563)

[2 设计 4](#_Toc10508)

[2.2 类的设计 4](#_Toc20111)

[3 功能实现 4](#_Toc12368)

[3.1 初始化循环模块的实现 4](#_Toc22052)

[3.1.1初始化循环模块流程图 4](#_Toc32141)

[3.1.2初始化循环模块核心代码 5](#_Toc3603)

[3.1.3输入未知操作码错误提示 6](#_Toc5108)

[3.1.4初始化循环模块截屏示例 6](#_Toc14570)

[3.2 冒泡排序的实现 6](#_Toc20334)

[3.2.1冒泡排序流程图 6](#_Toc16414)

[3.2.2冒泡排序核心代码 7](#_Toc17414)

[3.2.3冒泡排序截屏示例 7](#_Toc7762)

[3.3 选择排序的实现 8](#_Toc26002)

[3.3.1选择排序流程图 8](#_Toc13528)

[3.3.2选择排序核心代码 8](#_Toc8611)

[3.3.3选择排序截屏示例 9](#_Toc11870)

[3.4 直接插入排序的实现 9](#_Toc9738)

[3.4.1直接插入排序流程图 9](#_Toc24546)

[3.4.2直接插入排序核心代码 9](#_Toc8306)

[3.4.3直接插入排序截屏示例 10](#_Toc25186)

[3.5 希尔排序的实现 10](#_Toc26595)

[3.5.1希尔排序流程图 10](#_Toc13023)

[3.5.2希尔排序核心代码 11](#_Toc18208)

[3.5.3希尔排序截屏示例 12](#_Toc21423)

[3.6 快速排序的实现 12](#_Toc5954)

[3.6.1快速排序流程图 12](#_Toc308)

[3.6.2快速排序核心代码 12](#_Toc5132)

[3.6.3快速排序截屏示例 13](#_Toc23919)

[3.7 堆排序的实现 13](#_Toc24500)

[3.7.1堆排序流程图 13](#_Toc20747)

[3.7.2堆排序核心代码 13](#_Toc6723)

[3.7.3堆排序截屏示例 14](#_Toc5584)

[3.8 归并排序的实现 14](#_Toc20671)

[3.8.1归并排序流程图 14](#_Toc15774)

[3.8.2归并排序核心代码 15](#_Toc27310)

[3.8.3归并排序截屏示例 16](#_Toc25189)

[3.9 基数排序的实现 16](#_Toc12477)

[3.9.1基数排序流程图 16](#_Toc20146)

[3.9.2基数排序核心代码 17](#_Toc7562)

[3.9.3基数排序截屏示例 19](#_Toc11209)

[4 测试（release模式下运行） 19](#_Toc29162)

[4.1 100个随机数排序测试 19](#_Toc26977)

[4.1.1 冒泡排序 19](#_Toc4121)

[4.1.2 选择排序 20](#_Toc29829)

[4.1.3 直接插入排序 20](#_Toc32187)

[4.1.4 希尔排序 20](#_Toc15901)

[4.1.5 快速排序 20](#_Toc10903)

[4.1.6 堆排序 20](#_Toc22323)

[4.1.7 归并排序 20](#_Toc19057)

[4.1.8 基数排序 21](#_Toc22514)

[4.1.9实验结论 21](#_Toc17008)

[4.2 1000个随机数排序测试 21](#_Toc12260)

[4.2.1 冒泡排序 21](#_Toc28399)

[4.2.2 选择排序 21](#_Toc12555)

[4.2.3 直接插入排序 22](#_Toc27824)

[4.2.4 希尔排序 22](#_Toc29888)

[4.2.5 快速排序 22](#_Toc10440)

[4.2.6 堆排序功 22](#_Toc28792)

[4.2.7 归并排序 22](#_Toc21691)

[4.2.8 基数排序 22](#_Toc492)

[4.2.9实验结论 23](#_Toc2205)

[4.3 10000个随机数排序测试 23](#_Toc13615)

[4.3.1 冒泡排序 23](#_Toc27454)

[4.3.2 选择排序 23](#_Toc23729)

[4.3.3 直接插入排序 24](#_Toc14502)

[4.3.4 希尔排序 24](#_Toc4688)

[4.3.5 快速排序 24](#_Toc357)

[4.3.6 堆排序功 24](#_Toc12762)

[4.3.7 归并排序 24](#_Toc4494)

[4.3.8 基数排序 24](#_Toc8365)

[4.3.9实验结论 25](#_Toc9295)

[4.4 100000个随机数排序测试 25](#_Toc4957)

[4.4.1 冒泡排序 25](#_Toc29060)

[4.4.2 选择排序 25](#_Toc18825)

[4.4.3 直接插入排序 26](#_Toc19647)

[4.4.4 希尔排序 26](#_Toc14654)

[4.4.5 快速排序 26](#_Toc2447)

[4.4.6 堆排序功 26](#_Toc11936)

[4.4.7 归并排序 26](#_Toc14100)

[4.4.8 基数排序 26](#_Toc18865)

[4.4.9实验结论 27](#_Toc5409)

[5 综合实验结论 27](#_Toc21352)

**1 分析**

**1.1 背景分析**

排序是计算机内经常进行的一种操作，其目的是将一组“无序”的记录序列调整为“有序”的记录序列。分[内部排序](https://baike.baidu.com/item/%E5%86%85%E9%83%A8%E6%8E%92%E5%BA%8F" \t "https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F/_blank)和[外部排序](https://baike.baidu.com/item/%E5%A4%96%E9%83%A8%E6%8E%92%E5%BA%8F" \t "https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F/_blank)，若整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序。反之，若参加排序的记录数量很大，整个序列的排序过程不可能在[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98/103614" \t "https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F/_blank)中完成，则称此类排序问题为外部排序。

**1.2 功能分析**

利用随机函数产生随机数，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。

**2 设计**

**2.2 类的设计**

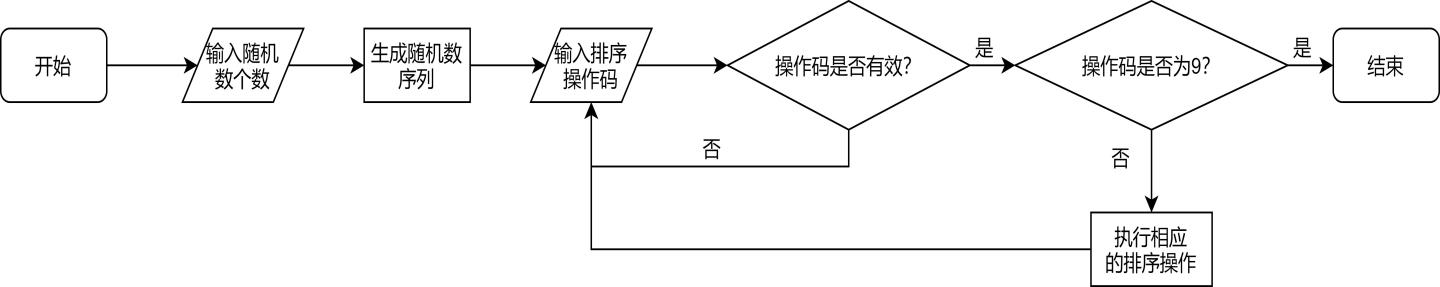
排序系统类：其私有成员包括一个动态数组的首元地址，以及数组的长度。其公有函数包括初始化和循环函数，以及冒泡排序，选择排序，直接插入排序，希尔排序，快速排序，堆排序，归并排序，基数排序八种排序函数。

1. //全部按升序排列
2. **class** SortSystem {
3. **private**:
4. **int**\* array;
5. **int** num;
6. **public**:
7. //构造函数
8. SortSystem() { array = **new** **int**[1]; num = 0; };
9. //析构函数
10. ~SortSystem() {};
11. //初始化
12. **void** Initialize();
13. //循环模块
14. **bool** Run();
15. //冒泡排序
16. **void** BubbleSort();
17. //选择排序
18. **void** SelectSort();
19. //直接插入排序
20. **void** InsertSort();
21. //希尔排序
22. **void** ShellSort();
23. //快速排序
24. **void** QuickSort();
25. //堆排序
26. **void** HeapSort();
27. //归并排序
28. **void** MergeSort();
29. //基数排序
30. **void** RadixSort();
31. };

**3 功能实现**

**3.1 初始化循环模块的实现**

**3.1.1初始化循环模块流程图**



**3.1.2初始化循环模块核心代码**

先打印操作表，提示用户可供选择的排序方式。然后令用户输入随机数个数，通过输入的随机数个数初始化动态数组以及数组大小。最后进入操作循环模块，用户可选择不同的排序算法观察其比较次数及交换次数。

1. //初始化
2. **void** SortSystem::Initialize()
3. {
4. cout << "\*\*          排序算法比较          \*\*" << endl;
5. cout << "====================================" << endl;
6. cout << "\*\*          1---冒泡排序          \*\*" << endl;
7. cout << "\*\*          2---选择排序          \*\*" << endl;
8. cout << "\*\*          3---直接插入排序      \*\*" << endl;
9. cout << "\*\*          4---希尔排序          \*\*" << endl;
10. cout << "\*\*          5---快速排序          \*\*" << endl;
11. cout << "\*\*          6---堆排序            \*\*" << endl;
12. cout << "\*\*          7---归并排序          \*\*" << endl;
13. cout << "\*\*          8---基数排序          \*\*" << endl;
14. cout << "\*\*          9---退出程序          \*\*" << endl;
15. cout << "====================================" << endl;
16. cout << endl;
17. cout << "请输入要产生的随机数的个数：";
18. **int** arraysize = 0; cin >> arraysize;
19. array = **new** **int**[arraysize];
20. num = arraysize;
21. cout << endl;
22. srand((unsigned)time(NULL));
23. **for** (**int** i = 0; i < arraysize; i++)
24. {
25. array[i] = rand();
26. }
27. **while** (Run())
28. {
29. cout << endl;
30. }
31. }

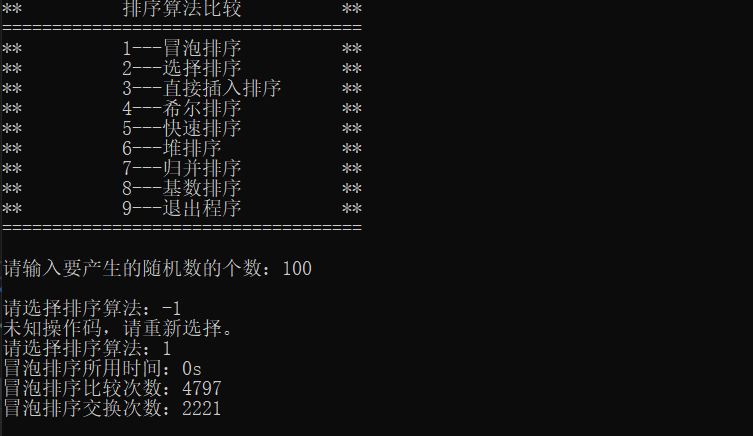
运行模块。函数类型为bool类型，用户可根据提供的操作码选择不同的排序方式，若用户输入的是结束操作码，则函数返回false，从主程序中退出，否则程序根据用户的输入调用不同的函数执行相应的操作。

1. //运行
2. **bool** SortSystem::Run()
3. {
4. cout << "请选择排序算法：";
5. **int** sortnum = 0;
6. cin >> sortnum;
7. **switch** (sortnum)
8. {
9. **case** 1:BubbleSort(); **break**;
10. **case** 2:SelectSort(); **break**;
11. **case** 3:InsertSort(); **break**;
12. **case** 4:ShellSort(); **break**;
13. **case** 5:QuickSort(); **break**;
14. **case** 6:HeapSort(); **break**;
15. **case** 7:MergeSort(); **break**;
16. **case** 8:RadixSort(); **break**;
17. **case** 9:**return** **false**;
18. **default**:cout << "未知操作码，请重新选择。"; **break**;
19. }
20. **return** **true**;
21. }

**3.1.3输入未知操作码错误提示**

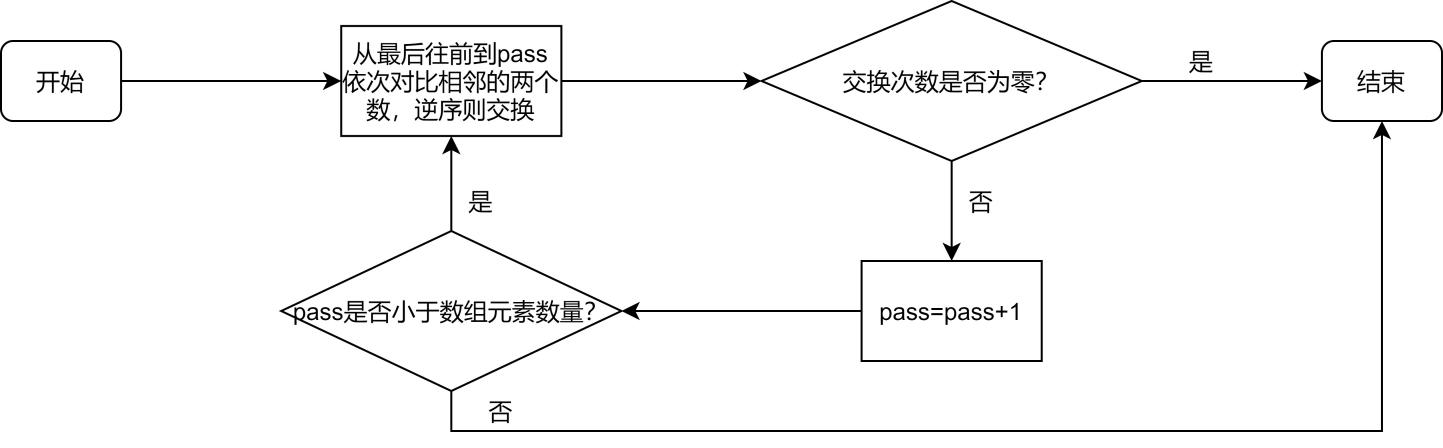
若用户输入了未知的操作码，程序会给出警告“未知的操作码，请重新选择”，并进入下一轮选择。

**3.1.4初始化循环模块截屏示例**



**3.2 冒泡排序的实现**

**3.2.1冒泡排序流程图**

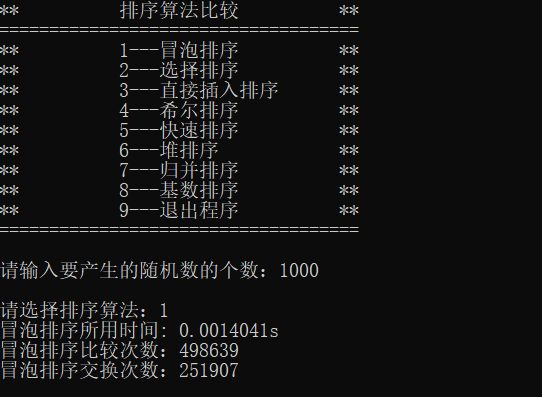
****

**3.2.2冒泡排序核心代码**

从后往前逐次对比相邻的两个数，若逆序则交换。若一趟比较中没有进行交换，说明数列已经是有序的了，结束排序。若一趟比较中进行了交换，则缩小比较的规模（比较范围减1）重复以上步骤。

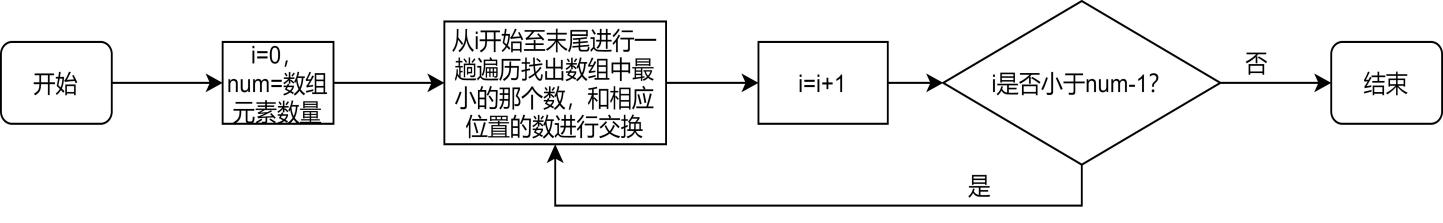
1. //冒泡排序
2. **void** SortSystem::BubbleSort()
3. {
4. //初始化拷贝数组
5. **int**\* copy = **new** **int**[num];
6. **for** (**int** i = 0; i < num; i++)
7. {
8. copy[i] = array[i];
9. }
10. //定义时间
11. steady\_clock::time\_point starttime, endtime;
12. steady\_clock::duration durationtime;
13. **long** **long** compare = 0;//记录比较次数
14. **long** **long** cnt = 0;//记录交换次数
15. starttime = steady\_clock::now();
16. **int** pass = 1; **int** exchange = 1;
17. **while** (pass < num && exchange)
18. {
19. exchange = 0;
20. **for** (**int** i = num - 1; i >= pass; i--)
21. {
22. **if** (copy[i - 1] > copy[i])
23. {
24. Swap(copy[i - 1], copy[i]);
25. cnt++;
26. exchange = 1;
27. }
28. compare++;
29. }
30. pass++;
31. }
32. endtime = steady\_clock::now();
33. durationtime = endtime - starttime;
34. cout << "冒泡排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den/1000 << "s"<<endl;
35. cout << "冒泡排序比较次数：" << compare << endl;
36. cout << "冒泡排序交换次数：" << cnt << endl;
37. }

**3.2.3冒泡排序截屏示例**



**3.3 选择排序的实现**

**3.3.1选择排序流程图**

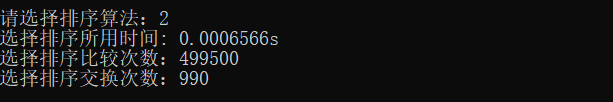


**3.3.2选择排序核心代码**

每趟从i到num-1中选择最小的一个数，放至到array[i]处，并将i+1重复之前的操作，直至i==num-1，排序完成。

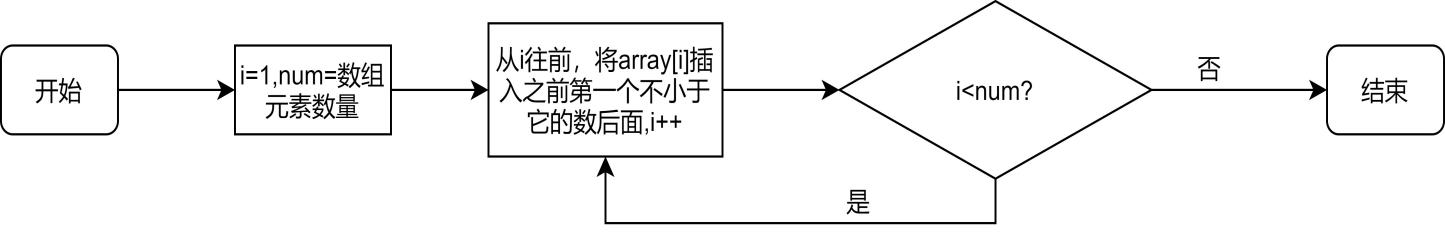
1. //选择排序
2. **void** SortSystem::SelectSort()
3. {
4. **int**\* copy = **new** **int**[num];
5. **for** (**int** i = 0; i < num; i++)
6. {
7. copy[i] = array[i];
8. }
9. steady\_clock::time\_point starttime, endtime;
10. steady\_clock::duration durationtime;
11. **long** **long** compare = 0;//记录比较次数
12. **long** **long** cnt=0;//记录交换次数
13. starttime = steady\_clock::now();
14. **for** (**int** i = 0; i < num-1; i++)
15. {
16. **int** min = i;
17. **for** (**int** j = i + 1; j < num; j++)
18. {
19. **if** (copy[j] < copy[min])
20. {
21. min = j;
22. }
23. compare++;
24. }
25. **if** (min != i)
26. {
27. Swap(copy[i], copy[min]);
28. cnt++;
29. }
30. }
31. endtime = steady\_clock::now();
32. durationtime = endtime - starttime;
33. cout << "选择排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
34. cout << "选择排序比较次数：" << compare << endl;
35. cout << "选择排序交换次数：" << cnt << endl;
36. }

**3.3.3选择排序截屏示例**



**3.4 直接插入排序的实现**

**3.4.1直接插入排序流程图**

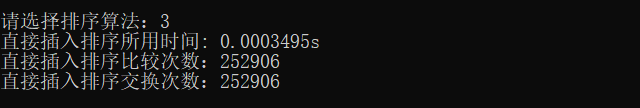


**3.4.2直接插入排序核心代码**

直接插入排序的思想是将一个数插入之前已经排好序的序列中。只要将要插入的数依次和之前的数比较，然后插入到第一个不小于它的数后面。当对序列中所有的元素都执行一遍这样的操作，排序也就完成了。

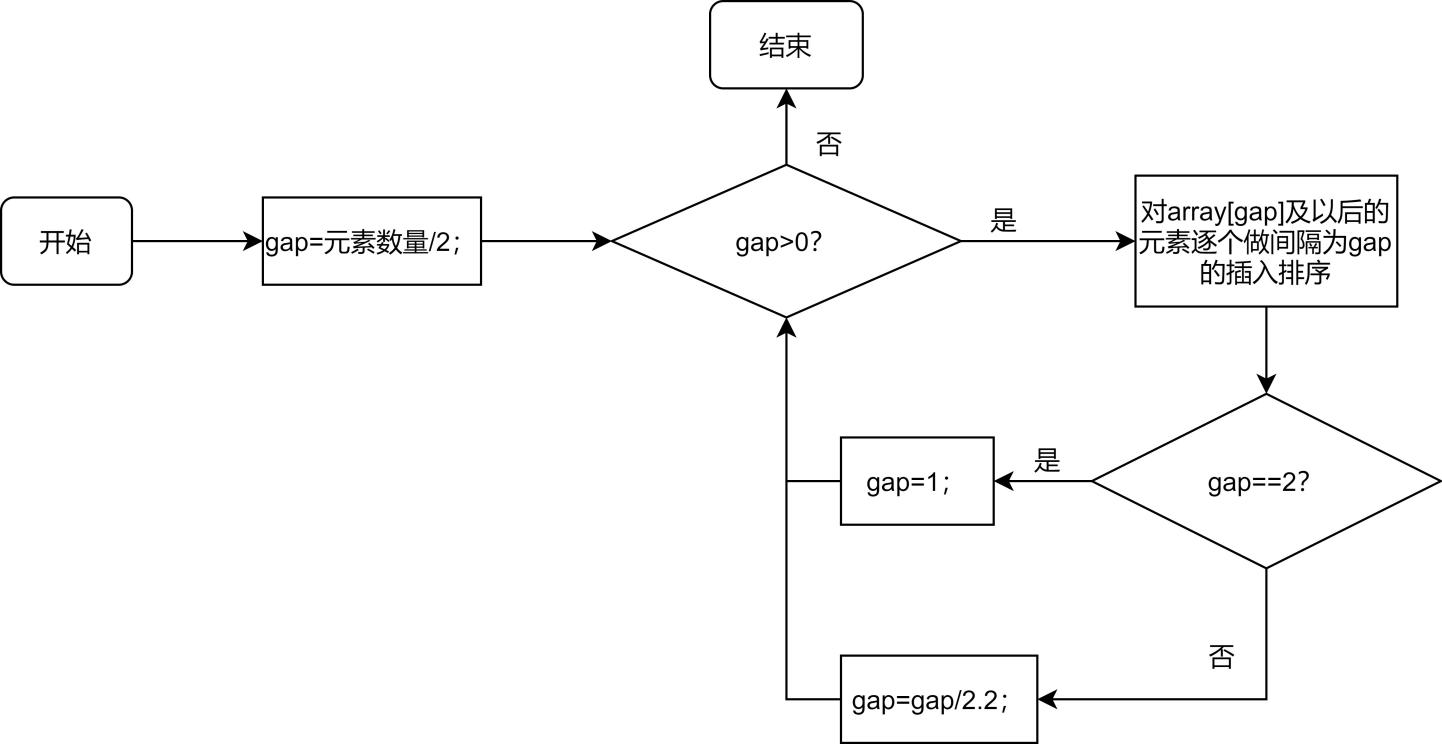
1. //直接插入排序
2. **void** SortSystem::InsertSort()
3. {
4. **int**\* copy = **new** **int**[num];
5. **for** (**int** i = 0; i < num; i++)
6. {
7. copy[i] = array[i];
8. }
9. steady\_clock::time\_point starttime, endtime;
10. steady\_clock::duration durationtime;
11. **long** **long** compare = 0;//记录比较次数
12. **long** **long** cnt = 0;//记录交换次数
13. starttime = steady\_clock::now();
14. **for** (**int** i = 1; i < num; i++)
15. {
16. **int** temp = copy[i]; **int** j = i;
17. **while** (j > 0 && temp < copy[j - 1])
18. {
19. compare++;
20. copy[j] = copy[j - 1];
21. cnt++;
22. j--;
23. }
24. compare++;
25. copy[j] = temp;
26. cnt++;
27. }
28. endtime = steady\_clock::now();
29. durationtime = endtime - starttime;
30. cout << "直接插入排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
31. cout << "直接插入排序比较次数：" << compare << endl;
32. cout << "直接插入排序交换次数：" << cnt << endl;
33. }

**3.4.3直接插入排序截屏示例**



**3.5 希尔排序的实现**

**3.5.1希尔排序流程图**

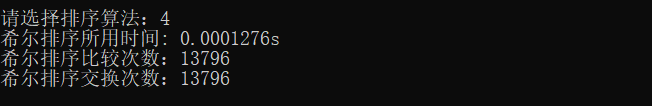


**3.5.2希尔排序核心代码**

希尔排序是优化过的直接插入排序。一开始设置一个间隔gap，使相邻gap个单位的元素之间做直接插入排序，由于一开始gap很大，元素数量较少，因此排序较快。然后逐步缩小gap范围直至1，由于gap变小后的序列已经相对有序了，所以排序速度也很快。

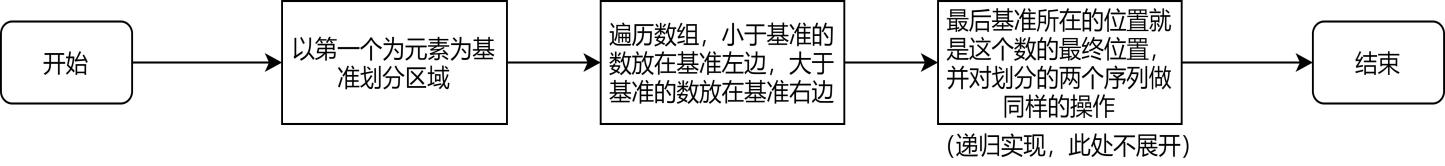
1. //希尔排序
2. **void** SortSystem::ShellSort()
3. {
4. **int**\* copy = **new** **int**[num];
5. **for** (**int** i = 0; i < num; i++)
6. {
7. copy[i] = array[i];
8. }
9. steady\_clock::time\_point starttime, endtime;
10. steady\_clock::duration durationtime;
11. **long** **long** compare = 0;//记录比较次数
12. **long** **long** cnt = 0;//记录交换次数
13. starttime = steady\_clock::now();
14. **int** gap = num / 2;
15. **while** (gap)
16. {
17. **for** (**int** i = gap; i < num; i++)
18. {
19. **int** temp = copy[i]; **int** j = i;
20. **while** (j >= gap && temp < copy[j - gap])
21. {
22. compare++;
23. copy[j] = copy[j - gap];
24. cnt++;
25. j -= gap;
26. }
27. compare++;
28. copy[j] = temp;
29. cnt++;
30. }
31. gap = (gap == 2) ? 1 : (**int**)(gap / 2.2);
32. }
33. endtime = steady\_clock::now();
34. durationtime = endtime - starttime;
35. cout << "希尔排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
36. cout << "希尔排序比较次数：" << compare << endl;
37. cout << "希尔排序交换次数：" << cnt << endl;
38. }

**3.5.3希尔排序截屏示例**



**3.6 快速排序的实现**

**3.6.1快速排序流程图**



**3.6.2快速排序核心代码**

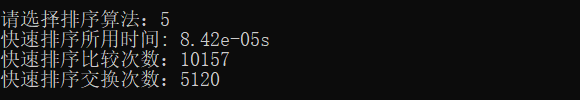
快速排序的思想是选取一个元素作为基准，依次和序列中所有的元素比较，比基准小的元素放至基准的左边，比基准大的元素放至基准右边，当遍历完一遍序列后，也就找到了这个基准的位置。然后通过递归对划分出来的两个序列做同样的操作。

1. //快速排序
2. **void** SortSystem::QuickSort()
3. {
4. **int**\* copy = **new** **int**[num];
5. **for** (**int** i = 0; i < num; i++)
6. {
7. copy[i] = array[i];
8. }
9. steady\_clock::time\_point starttime, endtime;
10. steady\_clock::duration durationtime;
11. **long** **long** compare = 0;
12. **long** **long** cnt = 0;
13. starttime = steady\_clock::now();
14. quicksort(copy, 0, num - 1, cnt, compare);
15. endtime = steady\_clock::now();
16. durationtime = endtime - starttime;
17. cout << "快速排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
18. cout << "快速排序比较次数：" << compare << endl;
19. cout << "快速排序交换次数：" << cnt << endl;
20. }

快排递归函数的参数有一个完整的数组用于比较元素，以及一个序列的左端点和右端点，通过左右两个端点来确定一个序列，以及交换次数和比较次数，用于记录排序中元素的比较情况和交换情况。

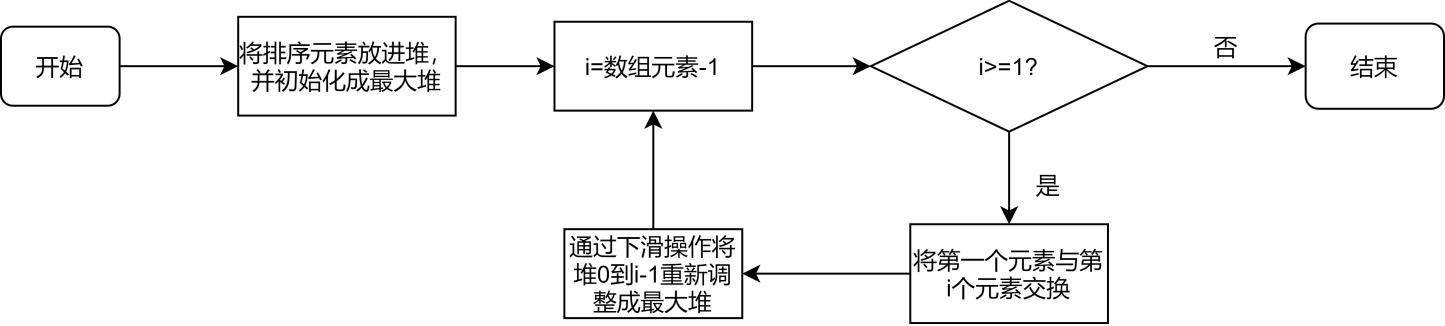
1. //快速排序递归函数，用于快速排序
2. **void** quicksort(**int**\* array, **int** left, **int** right, **long** **long**& cnt,**long** **long**&compare)
3. {
4. **if** (left < right)
5. {
6. //划分区域
7. **int** pivotpos = left; **int** pivot = array[left];
8. **for** (**int** i = left + 1; i <= right; i++)
9. {
10. compare++;
11. **if** (array[i] < pivot&&++pivotpos!=i)
12. {
13. Swap(array[pivotpos], array[i]);
14. cnt++;
15. }
16. }
17. Swap(array[left], array[pivotpos]);
18. cnt++;
19. //对划分的序列进行同样的操作
20. quicksort(array, left, pivotpos - 1, cnt,compare);
21. quicksort(array, pivotpos + 1, right, cnt,compare);
22. }
23. }

**3.6.3快速排序截屏示例**



**3.7 堆排序的实现**

**3.7.1堆排序流程图**

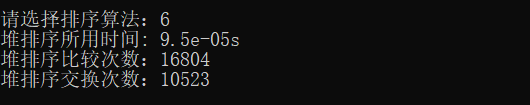


**3.7.2堆排序核心代码**

通过堆的下滑调整而实现的一种排序。首先建立一个最大堆，每次将堆顶的元素取出（即当前最大值），和当前堆的最后位置元素交换（即放到数组中该元素的对应位置），然后缩小堆的规模，通过对堆顶元素的下滑调整使堆重新成为最大堆，然后重复之前的操作，直至堆的规模为1。（下滑调整在之前第7题修理牧场中写过，此处就不再贴代码解释实现了）

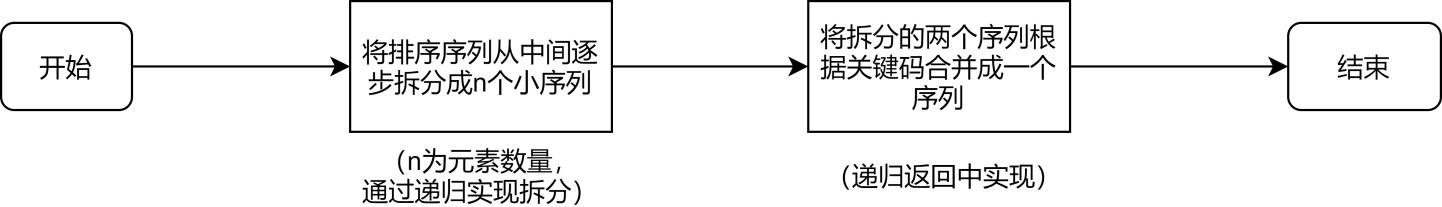
1. //堆排序
2. **void** SortSystem::HeapSort()
3. {
4. MaxHeap hp;
5. hp.heap = **new** **int**[num];
6. **for** (**int** i = 0; i < num; i++)
7. {
8. hp.heap[i] = array[i];
9. }
10. hp.cursize = hp.maxsize = num;
11. steady\_clock::time\_point starttime, endtime;
12. steady\_clock::duration durationtime;
13. **long** **long** compare = 0;//记录比较次数
14. **long** **long** cnt = 0;//记录交换次数
15. starttime = steady\_clock::now();
16. //将整个堆调整成最大堆
17. **for** (**int** i = (hp.cursize - 2) / 2; i >= 0; i--)
18. {
19. hp.Siftdown(i, hp.cursize - 1,cnt,compare);
20. }
21. //交换顺序
22. **for** (**int** i = hp.cursize - 1; i >= 1; i--)
23. {
24. Swap(hp.heap[0], hp.heap[i]);
25. cnt++;
26. hp.Siftdown(0, i - 1,cnt,compare);
27. }
28. endtime = steady\_clock::now();
29. durationtime = endtime - starttime;
30. cout << "堆排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
31. cout << "堆排序比较次数：" << compare << endl;
32. cout << "堆排序交换次数：" << cnt << endl;
33. }

**3.7.3堆排序截屏示例**



**3.8 归并排序的实现**

**3.8.1归并排序流程图**



**3.8.2归并排序核心代码**

归并排序入口函数。

1. //归并排序
2. **void** SortSystem::MergeSort()
3. {
4. **int**\* copy1 = **new** **int**[num];
5. **int**\* copy2 = **new** **int**[num];
6. **for** (**int** i = 0; i < num; i++)
7. {
8. copy1[i] = array[i];
9. }
10. steady\_clock::time\_point starttime, endtime;
11. steady\_clock::duration durationtime;
12. **long** **long** compare = 0;//记录比较次数
13. **long** **long** cnt = 0;//记录交换次数
14. starttime = steady\_clock::now();
15. mergesort(copy1, copy2, 0, num - 1, cnt,compare);
16. endtime = steady\_clock::now();
17. durationtime = endtime - starttime;
18. cout << "归并排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
19. cout << "归并排序比较次数：" << compare << endl;
20. cout << "归并排序交换次数：" << cnt << endl;
21. }

通过递归实现将原序列拆分成很多个规模为1的序列，并在递归返回的过程中合并相邻的两个序列，使其有序。

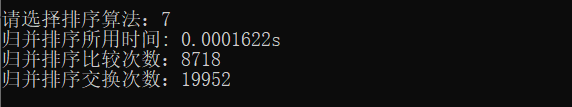
1. //归并排序递归函数，用于归并排序
2. **void** mergesort(**int**\* array, **int**\* copy, **int** left, **int** right, **long** **long**& cnt,**long** **long**&compare)
3. {
4. **if** (left >= right)
5. {
6. **return**;
7. }
8. **int** mid = (left + right) / 2;
9. mergesort(array, copy, left, mid, cnt,compare);
10. mergesort(array, copy, mid + 1, right, cnt,compare);
11. merge(array, copy, left, mid, right, cnt,compare);
12. }

将两个序列合并的函数。思想类似多项式加法，先创建两个分别指向序列开头的指针变量，然后通过不断地比较两个元素的大小来相应地移动指针，实现两个序列的有序合并。

比如说，若指针A的值小于指针B的值，则将A放进合并后的序列中，并将A往后移。若相等，则将A和B中的值依次放入合并序列中（保持相对稳定），并将A和B向后移。若大于，则将B值放入合并序列，并将B后移。当某一个序列已经为空时，只要将不为空的那个序列依次接在已形成的序列上就合并完成了。

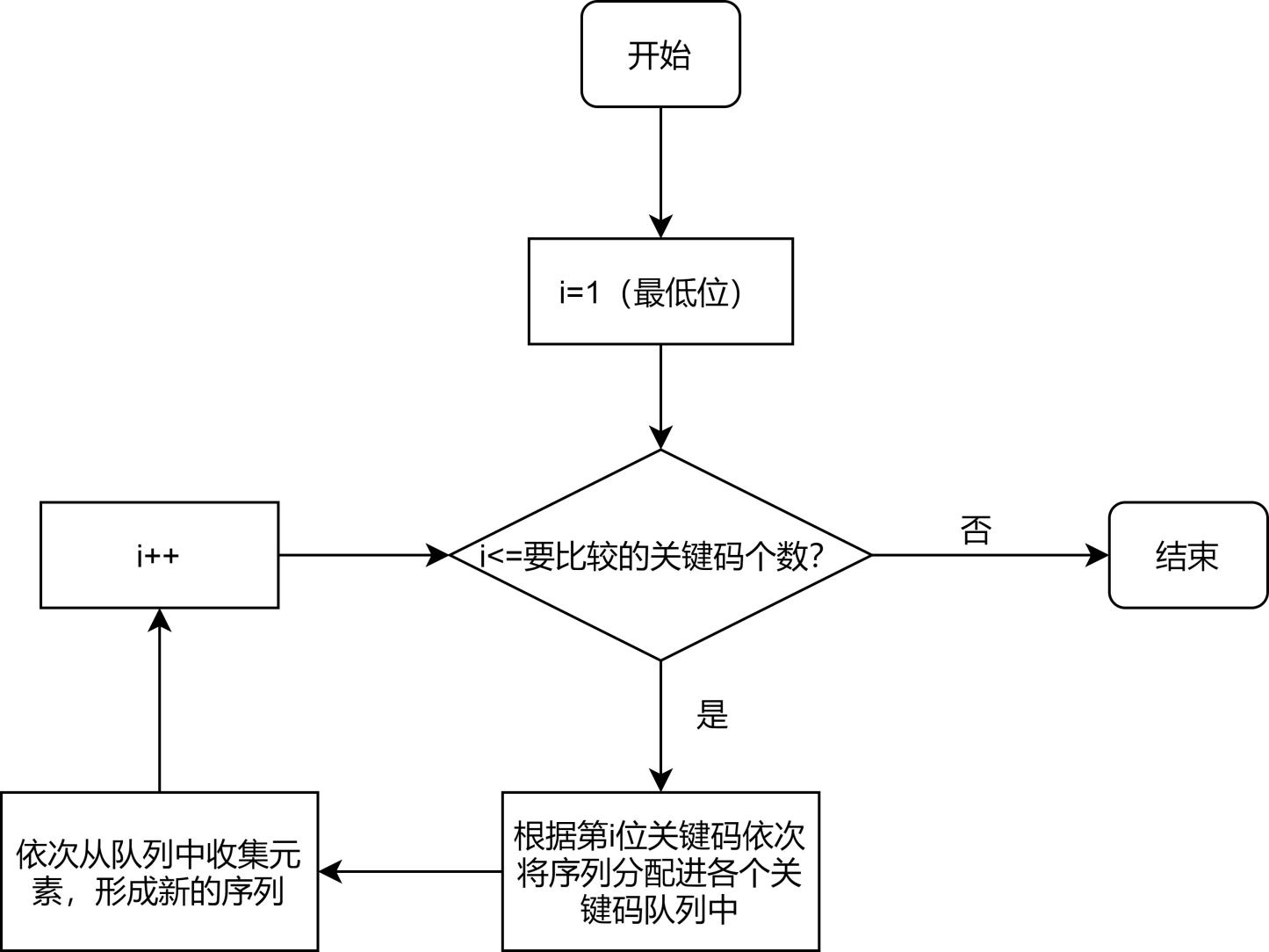
1. //合并两个序列的函数，用于归并排序
2. **void** merge(**int**\* array, **int**\* copy, **const** **int** left, **const** **int** mid, **const** **int** right, **long** **long**& cnt,**long** **long**&compare)
3. {
4. **for** (**int** k = left; k <= right; k++)
5. {
6. copy[k] = array[k];
7. cnt++;
8. }
9. **int** s1 = left, s2 = mid + 1, t = left;
10. **while** (s1 <= mid && s2 <= right)
11. {
12. compare++;
13. **if** (copy[s1] <= copy[s2])
14. {
15. array[t++] = copy[s1++];
16. cnt++;
17. }
18. **else**
19. {
20. array[t++] = copy[s2++];
21. cnt++;
22. }
23. }
24. **while** (s1 <= mid) { array[t++] = copy[s1++]; cnt++; }
25. **while** (s2 <= right) { array[t++] = copy[s2++]; cnt++; }
26. }

**3.8.3归并排序截屏示例**



**3.9 基数排序的实现**

**3.9.1基数排序流程图**



**3.9.2基数排序核心代码**

基数排序入口函数。基数排序核心思想是通过对最低位关键码到最高位关键码的依次比较分配再收集的过程实现序列的有序化。

1. //基数排序
2. **void** SortSystem::RadixSort()//LSD
3. {
4. Staticlinkedlist\* copy = **new** Staticlinkedlist[num+1];
5. **for** (**int** i = 1; i <= num; i++)
6. {
7. copy[i].data = array[i-1];
8. copy[i].totallength = num;
9. }
10. cout << endl;
11. steady\_clock::time\_point starttime, endtime;
12. steady\_clock::duration durationtime;
13. **long** **long** compare = 0;//记录比较次数
14. **long** **long** cnt = 0;//记录交换次数
15. starttime = steady\_clock::now();
16. radixsort(copy,8,cnt,compare);
17. endtime = steady\_clock::now();
18. durationtime = endtime - starttime;
19. cout << "基数排序所用时间: " << **double**(durationtime.count()) \* microseconds::period::num / microseconds::period::den / 1000 << "s" << endl;
20. cout << "基数排序比较次数：" << compare << endl;
21. cout << "基数排序交换次数：" << cnt << endl;
22. }

基数排序所需的静态链表结点。其成员包括储存的数据，指向下一个结点的地址，以及一个记录静态链表总长度的整型变量。

1. //静态链表节点
2. **class** Staticlinkedlist {
3. **public**:
4. **int** data;
5. **int** link;
6. **int** totallength;
7. Staticlinkedlist() { data = 0; link = 0; totallength = 0; }
8. ~Staticlinkedlist() {};
9. };

用于获取数字第d位关键码的辅助函数。

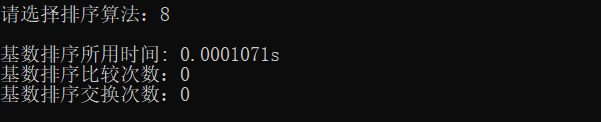
1. //基数排序辅助函数
2. **int** GetDigit(**int** num, **int** d)
3. {
4. **return** (**int**)(num / pow(10, d - 1)) % 10;
5. }

先对最低位关键码进行比较，分配进不同的队列中，然后再依次从队列中取出相应的元素，形成一个新的序列。然后对这个新序列，比较再高一级的关键码，重复分配收集的工作。直至所有的关键码都比较完，排序就完成了。

1. //基数排序函数
2. **void** radixsort(Staticlinkedlist\* copy, **int** d,**long** **long**&cnt,**long** **long**&compare)
3. {
4. **int** rear[10], front[10];
5. **int** i, j, k, last, current, n = copy[1].totallength;
6. **for** (i = 0; i < n; i++)
7. {
8. copy[i].link = i + 1;
9. }
10. copy[n].link = 0;
11. current = 1;
12. **for** (i = 1; i <=d; i++)
13. {
14. **for** (j = 0; j < 10; j++)
15. {
16. front[j] = 0;
17. }
18. **while** (current!=0)
19. {
20. k = GetDigit(copy[current].data, i);
21. **if** (front[k] == 0)
22. {
23. front[k] = current;
24. }
25. **else**
26. {
27. copy[rear[k]].link = current;
28. }
29. rear[k] = current;
30. current = copy[current].link;
31. }
32. j = 0;
33. **while** (front[j] == 0)
34. {
35. j++;
36. }
37. copy[0].link = current = front[j];
38. last = rear[j];
39. **for** (k = j + 1; k < 10; k++)
40. {
41. **if** (front[k] != 0)
42. {
43. copy[last].link = front[k];
44. last = rear[k];
45. }
46. }
47. copy[last].link = 0;
48. **int** temp = current;
49. }

}

**3.9.3基数排序截屏示例**

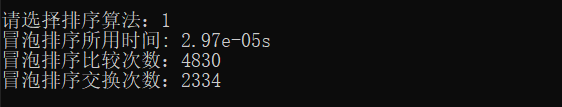


**4 测试（release模式下运行）**

**4.1 100个随机数排序测试**

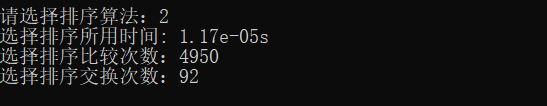
**4.1.1 冒泡排序**

**实验结果：**



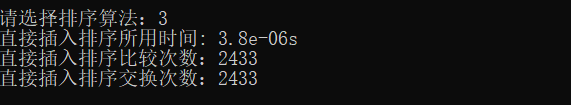
**4.1.2 选择排序**

**实验结果：**



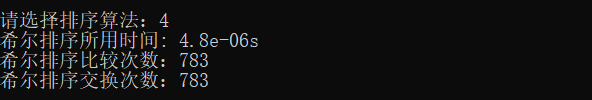
**4.1.3 直接插入排序**

**实验结果：**



**4.1.4 希尔排序**

**实验结果：**



**4.1.5 快速排序**

**实验结果：**



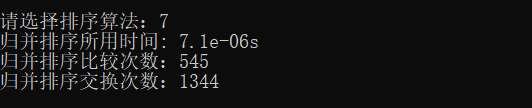
**4.1.6 堆排序**

**实验结果：**



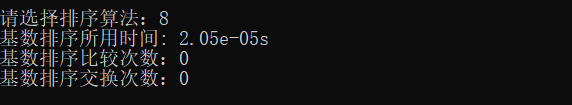
**4.1.7 归并排序**

**实验结果：**

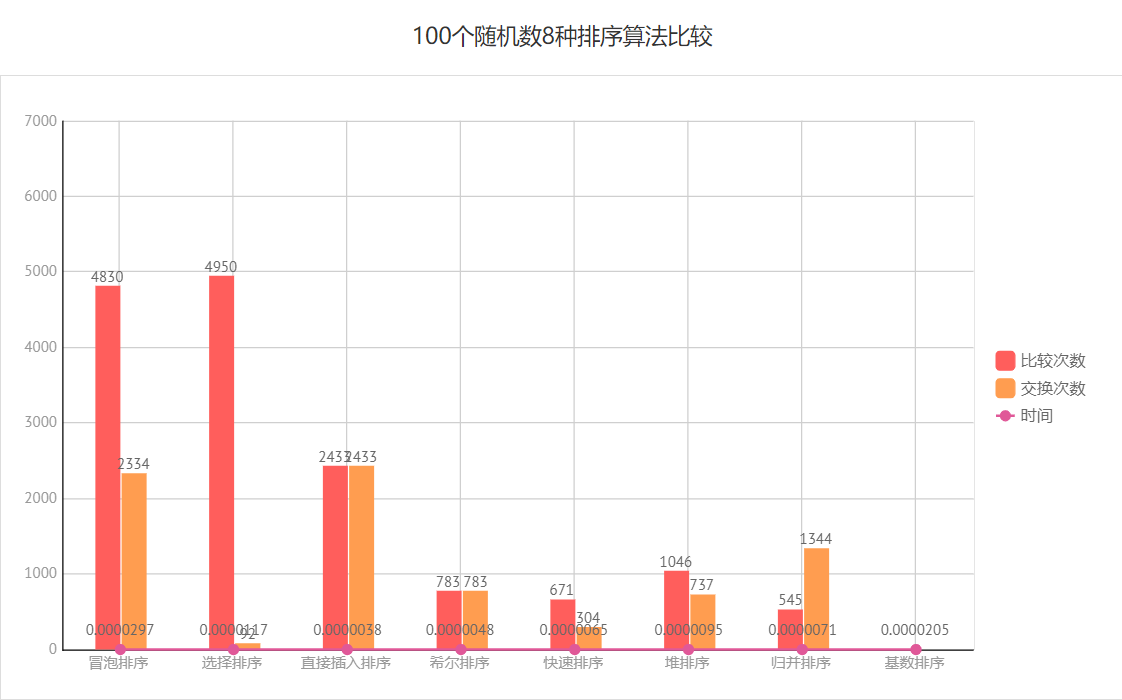


**4.1.8 基数排序**

**实验结果：**



**4.1.9实验结论**

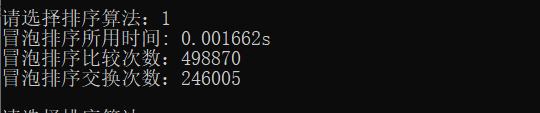


当排序规模较小时，各个排序算法之间的差异并不是很大，最快的是直接插入排序，最慢的是冒泡排序，但都远小于人能感知的范围。比较交换次数方面，冒泡排序的选择交换次数最多，选择排序和直接插入排序比冒泡排序略少，但仍很多。希尔排序，快速排序，堆排序和归并排序的比较交换次数并驾齐驱。基数排序由于是采用分配收集的策略，比较交换次数都是0。

**4.2 1000个随机数排序测试**

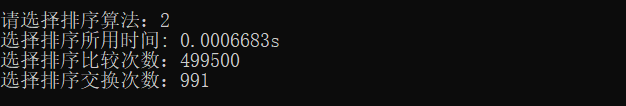
**4.2.1 冒泡排序**

**实验结果：**



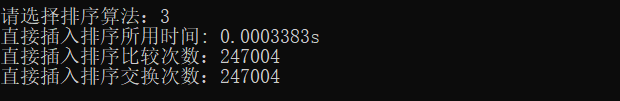
**4.2.2 选择排序**

**实验结果：**



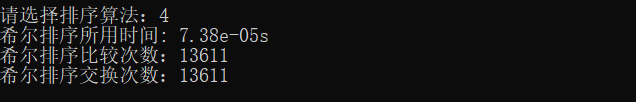
**4.2.3 直接插入排序**

**实验结果：**



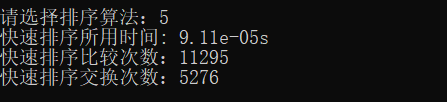
**4.2.4 希尔排序**

**实验结果：**



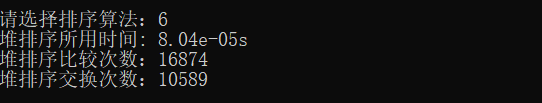
**4.2.5 快速排序**

**实验结果：**



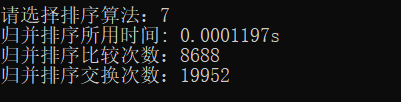
**4.2.6 堆排序功**

**实验结果：**



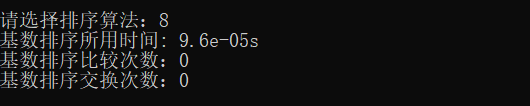
**4.2.7 归并排序**

**实验结果：**

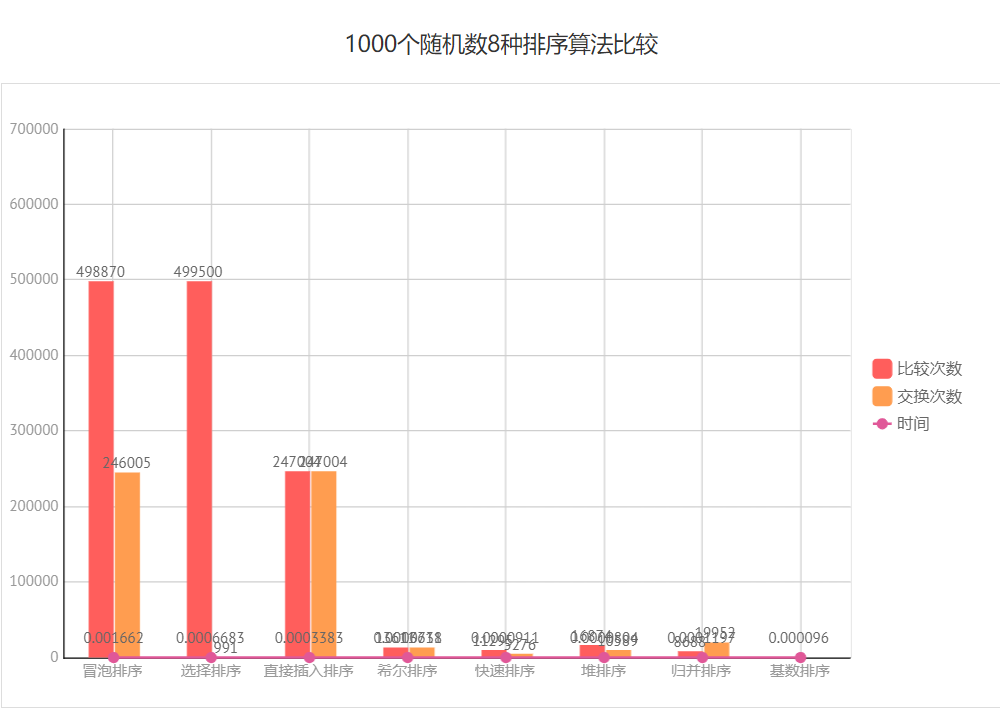


**4.2.8 基数排序**

**实验结果：**



**4.2.9实验结论**

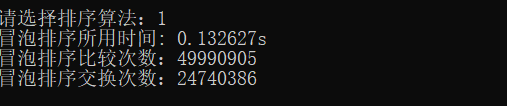


当排序规模略微增大时，可以明显发现冒泡排序和选择排序的比较交换次数都是比较多的。直接插入排序略好于前两种排序。但前三种排序的比较交换次数都远远多于后面的其他排序方法。时间方面，虽然都很快，但是冒泡排序相比之下依然处于劣势，而希尔排序在这次排序中是最快的。

**4.3 10000个随机数排序测试**

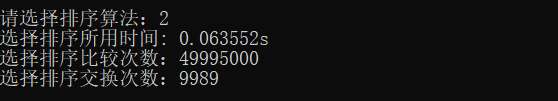
**4.3.1 冒泡排序**

**实验结果：**



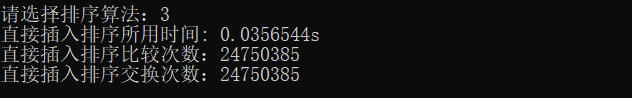
**4.3.2 选择排序**

**实验结果：**



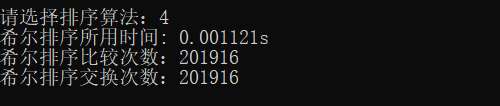
**4.3.3 直接插入排序**

**实验结果：**



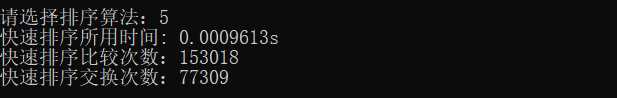
**4.3.4 希尔排序**

**实验结果：**



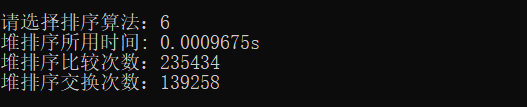
**4.3.5 快速排序**

**实验结果：**



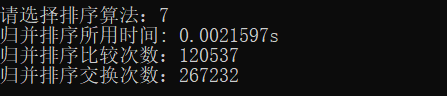
**4.3.6 堆排序功**

**实验结果：**



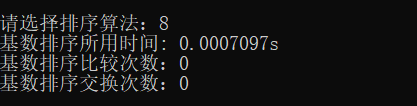
**4.3.7 归并排序**

**实验结果：**

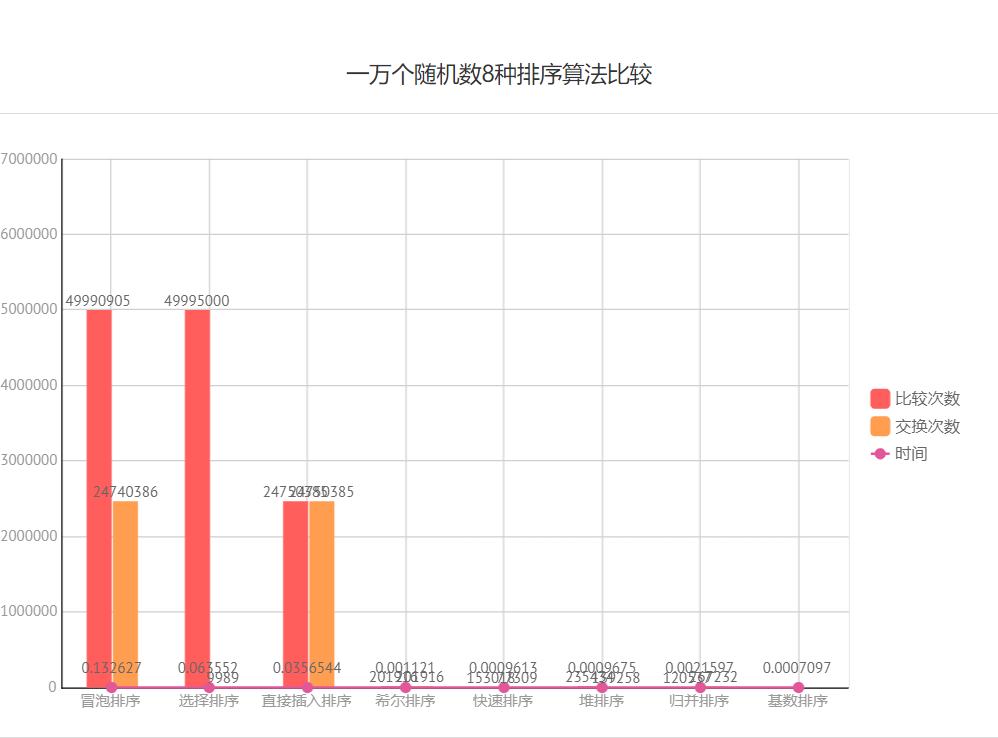


**4.3.8 基数排序**

**实验结果：**



**4.3.9实验结论**

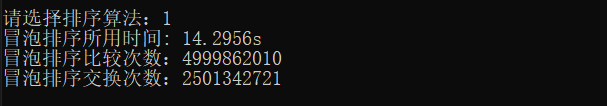


当排序规模进一步增大时，比较交换情况依然与之前类似，冒泡排序的总比较交换次数最大，选择排序和直接插入排序次之，其他排序都比较少。时间方面，最慢的依然是冒泡排序，最快的是基数排序。

**4.4 100000个随机数排序测试**

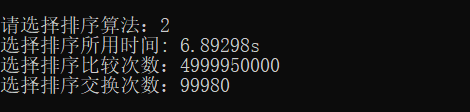
**4.4.1 冒泡排序**

**实验结果：**



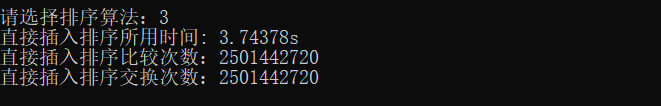
**4.4.2 选择排序**

**实验结果：**



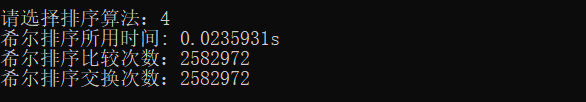
**4.4.3 直接插入排序**

**实验结果：**



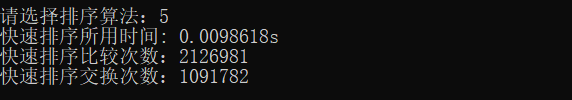
**4.4.4 希尔排序**

**实验结果：**



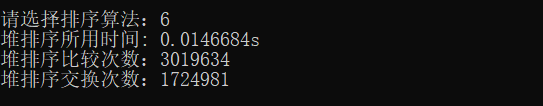
**4.4.5 快速排序**

**实验结果：**



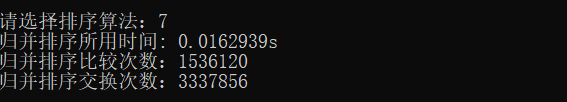
**4.4.6 堆排序功**

**实验结果：**



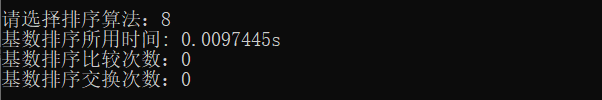
**4.4.7 归并排序**

**实验结果：**

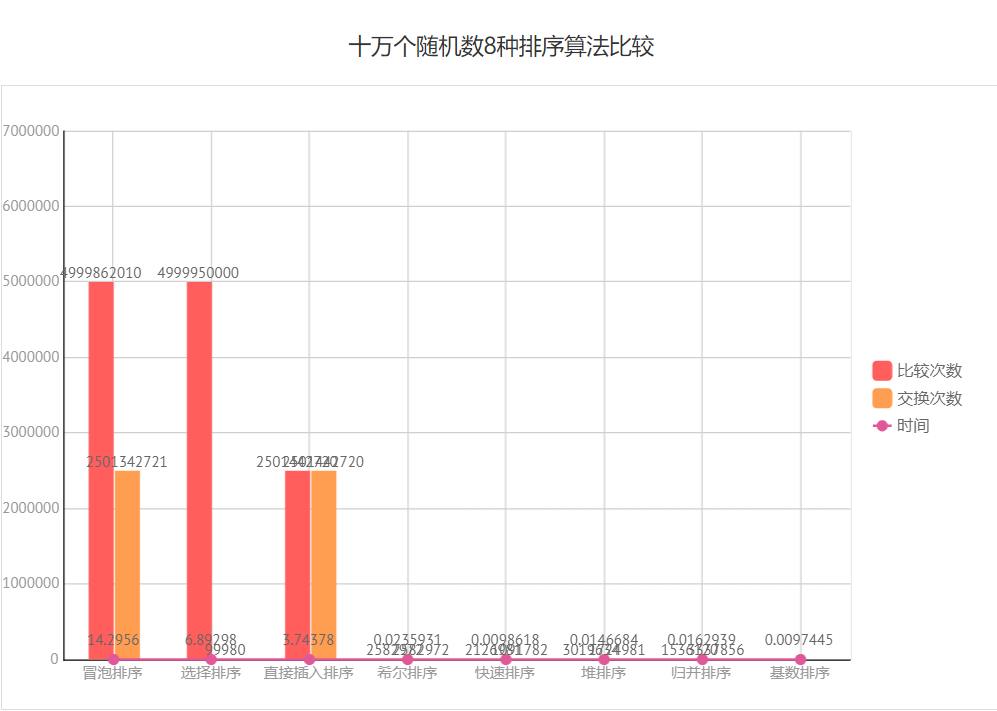


**4.4.8 基数排序**

**实验结果：**



**4.4.9实验结论**



当序列规模增大到10万时，比较交换次数冒泡排序最多，选择排序和直接插入排序略少，其他排序都很少。时间方面，冒泡排序最长，有14秒（Debug模式下甚至有70秒），选择排序次之，有7秒，是冒泡排序时间的一半，直接插入排序紧随其后，花费了3秒，是选择排序的一半。其他排序都在1毫秒左右，其中快速排序和基数排序是最快的两个，都在0.0098秒左右。

**5 综合实验结论**

综合各个实验结论来说，不难将这8种排序分为4个等级。第4级，冒泡排序，当规模逐渐增大时，时间花费远超其他的所有排序。第3级，选择排序和直接插入排序，虽然时间花费比冒泡排序少一点，但相比于其他排序还是不太理想。第2级，希尔排序，堆排序和归并排序，基本上时间花费不会随着数据规模的增大而增大很多，是优秀的排序算法。第1级，快速排序和基数排序，这两种排序是最快的两种排序方法，不过快排用的是分治递归的思想，而基数排序是分配收集，用空间换时间的思想。