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： 陈翔飞

学 号： 1851756

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 项目分析 4](#_Toc27754415)

[1.1 项目简介 4](#_Toc27754416)

[1.2文档要求 4](#_Toc27754417)

[2 辅助算法 4](#_Toc27754418)

[2.1随机数引擎 4](#_Toc27754419)

[2.2排序结果正确性检查 6](#_Toc27754420)

[3 实现 6](#_Toc27754421)

[3.1 冒泡排序的实现 6](#_Toc27754422)

[3.1.1 冒泡排序算法设计 6](#_Toc27754423)

[3.1.2 冒泡排序核心代码 7](#_Toc27754424)

[3.1.3 冒泡排序图解 8](#_Toc27754425)

[3.2 选择排序的实现 8](#_Toc27754426)

[3.2.1 选择排序算法设计 8](#_Toc27754427)

[3.2.2 选择排序核心代码 8](#_Toc27754428)

[3.2.3 选择排序图解 9](#_Toc27754429)

[3.3 直接插入排序的实现 9](#_Toc27754430)

[3.3.1 直接插入排序算法设计 9](#_Toc27754431)

[3.3.2 直接插入排序核心代码 10](#_Toc27754432)

[3.3.3 直接插入排序图解 10](#_Toc27754433)

[3.4 希尔排序的实现 11](#_Toc27754434)

[3.4.1 希尔排序算法设计 11](#_Toc27754435)

[3.4.2 希尔排序核心代码 11](#_Toc27754436)

[3.4.3 希尔排序图解 12](#_Toc27754437)

[3.5 堆排序的实现 12](#_Toc27754438)

[3.5.1 堆排序算法设计 12](#_Toc27754439)

[3.5.2 堆排序核心代码 13](#_Toc27754440)

[3.5.3 堆排序图解 14](#_Toc27754441)

[3.6 归并排序的实现 15](#_Toc27754442)

[3.6.1 归并排序算法设计 15](#_Toc27754443)

[3.6.2 归并排序核心代码 15](#_Toc27754444)

[3.6.3 归并排序图解 17](#_Toc27754445)

[3.7 快速排序的实现 17](#_Toc27754446)

[3.7.1 快速排序算法设计 17](#_Toc27754447)

[3.7.2 快速排序核心代码 18](#_Toc27754448)

[3.7.3 快速排序图解 19](#_Toc27754449)

[3.8 基数排序的实现 20](#_Toc27754450)

[3.8.1 基数排序算法设计 20](#_Toc27754451)

[3.8.2 基数排序核心代码 21](#_Toc27754452)

[4 较小范围测试 23](#_Toc27754453)

[4.1 100个随机数 23](#_Toc27754454)

[4.2 1000个随机数 24](#_Toc27754455)

[4.3 10000个随机数 25](#_Toc27754456)

[4.4 100000个随机数 26](#_Toc27754457)

[5较大范围测试 27](#_Toc27754458)

[5.1 1000000个随机数 27](#_Toc27754459)

[5.2 10000000个随机数 28](#_Toc27754460)

[5.3 50000000个随机数 29](#_Toc27754461)

[5.4 100000000个随机数 30](#_Toc27754462)

[5 实验结果分析 30](#_Toc27754463)

[6 一些思考 31](#_Toc27754464)

[6.1 希尔排序什么在实验过程中表现要优于堆排序？ 31](#_Toc27754465)

[6.2 快速排序什么在实验过程中表现要优于基数排序？ 31](#_Toc27754466)

# 1 项目分析

## 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移动次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的移动次数。

## 1.2文档要求

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。

# 辅助算法

## 2.1随机数引擎

由于标准库中自带的rand函数能产生的随机数范围太小，本项目选择自主实现了一个简单的随机数引擎，使用的随机数算法为MersenneTwisterMT19937算法，能产生32位的随机数。

**代码实现：**

1. **class** MT19937
2. {
3. **public**:
4. **int**  Extract()
5. {
6. **int** y;
7. **int**  i = index;
8. **if** (index >= N)
9. {
10. Twist();
11. i = index;
12. }
13. y = mt[i];
14. index = i + 1;
15. y ^= (y >> U);
16. y ^= (y << S) & B;
17. y ^= (y << T) & C;
18. y ^= (y >> L);
19. **if** (y <= 0) y \*= -1;
20. **else** **if** (y == 0) y = 1;
21. **return** y ;
22. }
23. **void** Initialize(**const** **int**  seed)
24. {
25. **int**  i;
26. mt[0] = seed;
27. **for** (i = 1; i < N; i++)
28. {
29. mt[i] = (F \* (mt[i - 1] ^ (mt[i - 1] >> 30)) + i);
30. }
31. index = N;
32. }
33. **private**:
34. **enum**
35. {
36. W = 32,
37. N = 624,
38. M = 397,
39. R = 31,
40. A = 0x9908B0DF,
42. F = 1812433253,
44. U = 11,
45. D = 0xFFFFFFFF,
47. S = 7,
48. B = 0x9D2C5680,
50. T = 15,
51. C = 0xEFC60000,
53. L = 18,
55. MASK\_LOWER = (1ull << R) - 1,
56. MASK\_UPPER = (1ull << R)
57. };
58. **void** Twist()
59. {
60. **int** i, x, xA;
61. **for** (i = 0; i < N; i++)
62. {
63. x = (mt[i] & MASK\_UPPER) + (mt[(i + 1) % N] & MASK\_LOWER);
64. xA = x >> 1;
65. **if** (x & 0x1)
66. {
67. xA ^= A;
68. }
69. mt[i] = mt[(i + M) % N] ^ xA;
70. }
72. index = 0;
73. }
74. **int**  mt[N];
75. **int**  index;
76. };

## 2.2排序结果正确性检查

遍历数组中的每个元素，比较它和它右边的元素的大小，来判断整个序列是否全部有序。

**代码实现：**

1. **void** Check(**int** v[], **int** Len)
2. {
3. **int** i;
4. **for** (i = 0; i < Len - 1; ++i)
5. {
6. **if** (v[i] > v[i + 1])
7. {
8. cout << "排序出错！" << endl;
9. **return**;
10. }
11. }
12. cout << "排序完成！" << endl;
13. }

# 3 实现

## 3.1 冒泡排序的实现

### 3.1.1 冒泡排序算法设计

**算法流程：**

序列中有n个元素，第一趟排序从第0个元素开始，首先比较第0个和第1个元素，如 果前者大于后者，则交换两元素，把较大的元素排在两个中的后一个位置，如果前者小于等于后者，则保持不变。然后如此再比较第1个和第2个元素，直到比较完第n-2个元素和第n-1个元 素。这一趟可以把最大的元素排在最后一个位置。然后开始第二趟排序，从第1个元素开始，首 先比较第1个和第2个元素，和第一趟排序一样，直到比较完第n-2个元素和第n-1个元素。第二 趟排序可以把第二大的元素排在倒数第二个位置。如此继续下去，直到第n-2趟排序，比较第n2个元素和第n-1个元素，即可完成排序。但是不一定要完成n-2趟排序序列才有序，可能前面的排序中序列已经有序。可以设置一个标志，如果再某一趟排序中没有发生元素交换，则可以根据此标志来判断序列已经有序，便可以退出算法，不再进行后面的排序。

**性能评估：**

1、冒泡排序的时间复杂度为O()

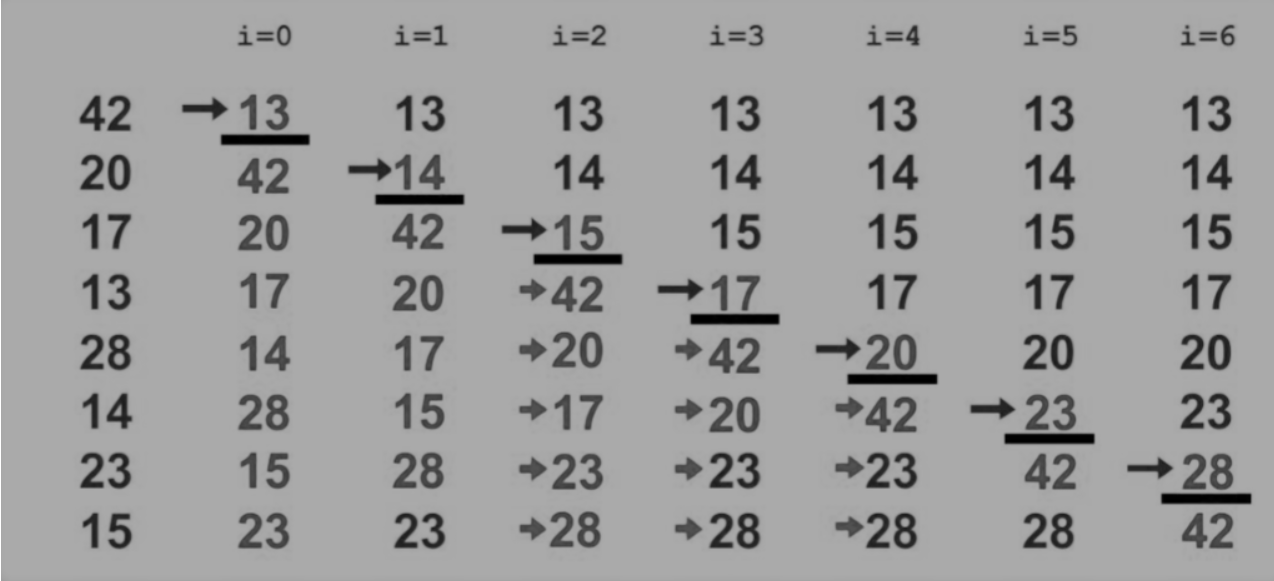
2、数据比较次数和输入序列中个排序元素的初始排列无关，数据的移动次数和各排序元素的初始序列有关。

3、冒泡排序是稳定的。

### 3.1.2 冒泡排序核心代码

1. **void** BubbleSort(**int** V[], **int** Len)
2. {
3. **int** i, j;
4. **bool** IsSwap = **false**;
5. **for** (i = 0; i < Len - 1; ++i)
6. {
7. IsSwap = **false**;
8. **for** (j = 0; j < Len - i - 1; j++)
9. {
10. ++CmpTime;
11. **if** (V[j + 1] < V[j])
12. {
13. Swap(V[j + 1], V[j]);
14. IsSwap = **true**;
15. }
16. }
17. **if** (IsSwap == **false**) **break**;
18. }
19. }

### 3.1.3 冒泡排序图解



## 3.2 选择排序的实现

### 3.2.1 选择排序算法设计

**算法流程：**

1. 找到数组中最小的那个元素
2. 将它和数组的第一个元素交换位置（如果第一个元素就是最小的元素就不执行交换）
3. 在剩下的元素中寻找最小的位置，将它与数组的第二个元素交换位置。
4. 如此往复，直到将整个数组排序。

**性能评估：**

1、选择排序的时间复杂度为O()

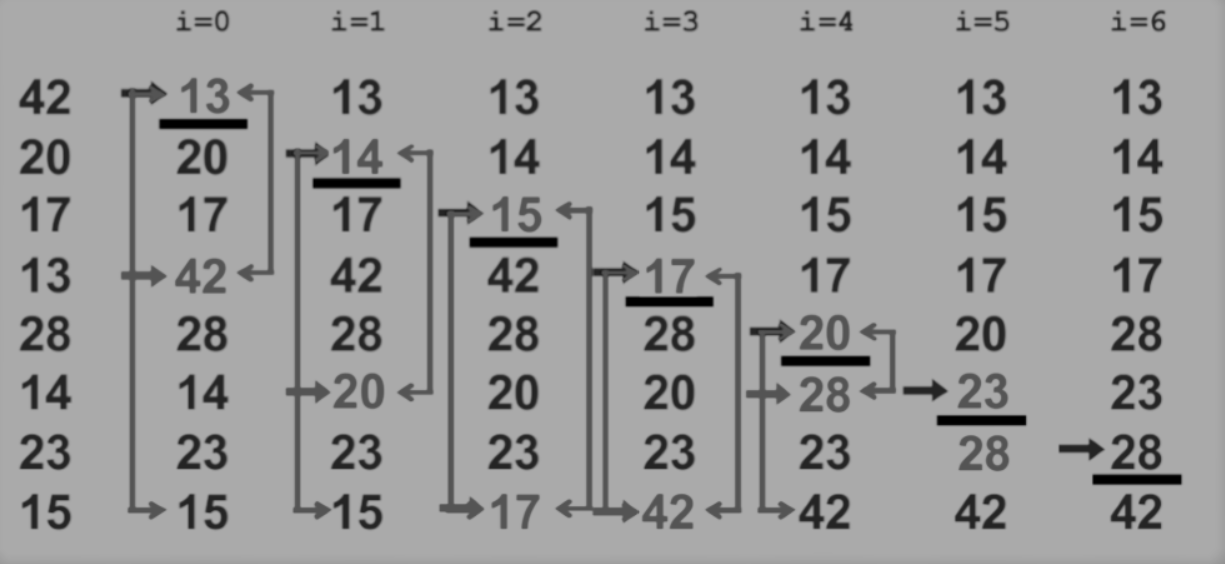
2、数据比较次数和输入序列中个排序元素的初始排列无关，数据的移动次数和各排序元素的初始序列有关。

3、选择排序是稳定的。

### 3.2.2 选择排序核心代码

1. **void** SelectionSort(**int** V[], **int** Len)
2. {
3. **int** i, j;
4. **int** MinIndex;
5. **for** (i = 0; i < Len - 1; ++i)
6. {
7. MinIndex = i;
8. **for** (j = i+1; j < Len ; j++)
9. {
10. ++CmpTime;
11. **if** (V[MinIndex] > V[j])
12. {
13. MinIndex = j;
14. }
15. }
16. **if** (MinIndex != i)
17. {
18. Swap(V[MinIndex], V[i]);
19. }
20. }
21. }

### 3.2.3 选择排序图解



## 3.3 直接插入排序的实现

### 3.3.1 直接插入排序算法设计

**算法流程：**

1、当插入第i（i>=1)个元素时，前面的元素都已经排好序。

2、用第i个元素的值与前面的元素一次比较，找到插入的位置就插入其中

3、将原来的元素及其后面的元素向后顺移。

4、将最后一个元素插入，则序列最终有序

**性能评估：**

1、直接插入排序的时间复杂度为O()

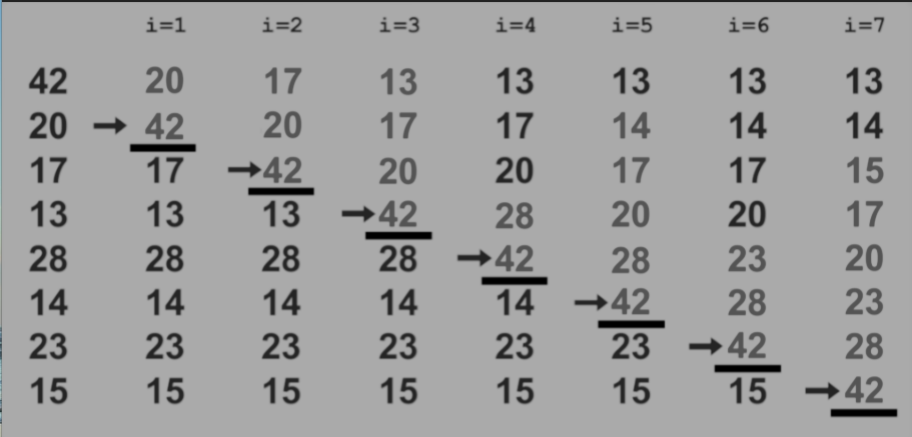
2、数据比较次数和移动次数与输入序列中个排序元素的初始排列有关。

3、直接插入排序是稳定的。

### 3.3.2 直接插入排序核心代码

1. **void** InsertionSort(**int** V[], **int** Len)
2. {
3. **int** i, j;
4. **int** Tmp;
5. **for** (i = 1; i < Len; ++i)
6. {
7. Tmp = V[i];
8. **for** (j = i; j > 0 && V[j - 1] > Tmp; j--)
9. {
10. ++CmpTime;
11. ++MoveTime;
12. V[j] = V[j - 1];
13. }
14. V[j] = Tmp;
15. MoveTime++;
16. }
17. }

### 3.3.3 直接插入排序图解



## 3.4 希尔排序的实现

### 3.4.1 希尔排序算法设计

**算法流程：**

1. 取一个整数gap < n作为间隔，将全部元素划分为gap个子序列。
2. 所有距离为gap的元素放在同一个子序列中，在每一个子序列中分别实施插入排序。
3. 然后缩小间隔gap，重复上述的子序列划分和排序工作。
4. 直到最后gap == 1, 将所有的元素放在同一个序列中排序，即可完成整个序列的排序。

**性能评估：**

1、选择不同的Gap序列会使希尔排序具有不同的时间复杂度，本项目中选择的是Sedgewick序列，其平均时间复杂度被估计为O()，最坏时间复杂度为O（）。

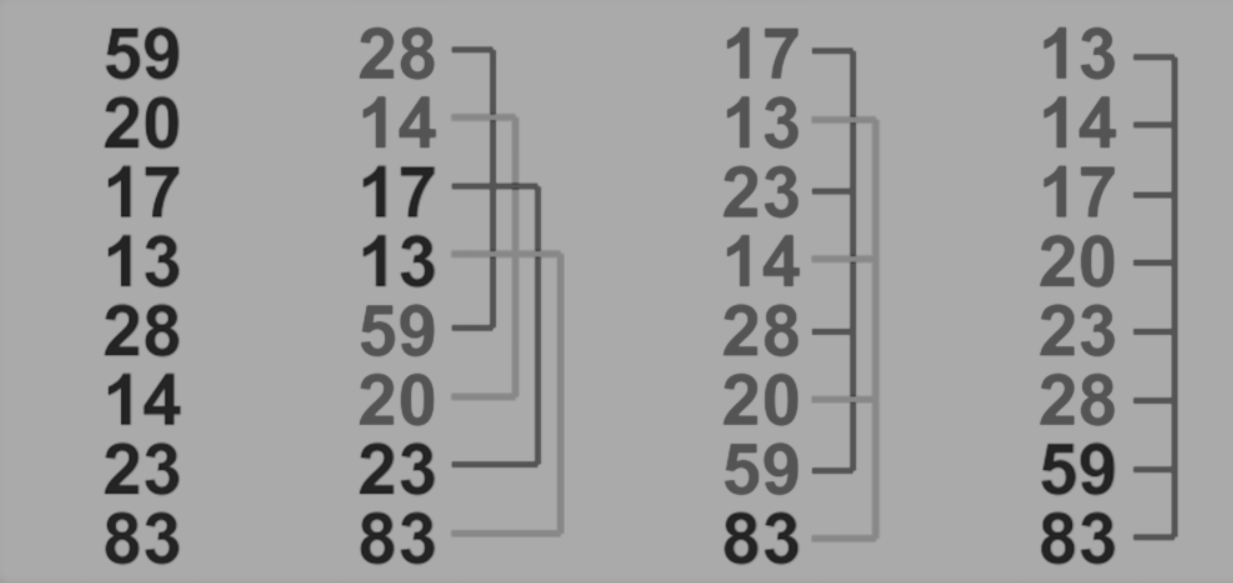
2、数据比较次数和移动次数与输入序列中个排序元素的初始排列有关。

3、希尔排序是不稳定的。

### 3.4.2 希尔排序核心代码

1. **void** ShellSort(**int** V[], **int** Len)
2. {
3. **int** Sedgewick[30] =
4. { 0, 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289,
5. 64769, 146305, 260609, 587521, 1045505, 2354689, 4188161, 9427969, 16764929,
6. 37730305, 67084289, 150958081, 268386305, 603906049, 1073643521 };
7. **int** i, j;
8. **int** Tmp;
9. **int** Gap;
10. **int** GapIndex = 0;
11. **while** (Sedgewick[GapIndex] < Len/2) GapIndex++;
12. Gap = Sedgewick[GapIndex];
13. **while** (Gap > 0)
14. {
15. **for** (i = Gap; i < Len; ++i)
16. {
17. Tmp = V[i];
18. **for** (j = i; j >= Gap; j -= Gap)
19. {
21. **if** (Tmp < V[j - Gap])
22. {
23. ++CmpTime;
24. ++MoveTime;
25. V[j] = V[j - Gap];
26. }
27. **else**
28. **break**;
29. }
30. V[j] = Tmp;
31. ++MoveTime;
32. }
33. Gap = Sedgewick[--GapIndex];
34. }
35. }

### 3.4.3 希尔排序图解



## 3.5 堆排序的实现

### 3.5.1 堆排序算法设计

**算法流程：**

1. 首先对根据初始数组建立最大堆，从下向上逐步调整为堆
2. 建立好最大堆之后，堆的第一个元素具有最大的排序码，将其与最后一个元素对调，即是把最大的元素排到了最后。
3. 对前面的元素运用从下向上调整为堆的算法，重新建立最大堆。
4. 重新建立最大堆后，次最大的元素处于堆顶，将其与倒数第二个元素对调，就排好了次最大的元素。
5. 反复执行与前两次相同的步骤，直到排好堆中所有的元素。

**性能评估：**

1、堆排序的时间复杂度为O() 。

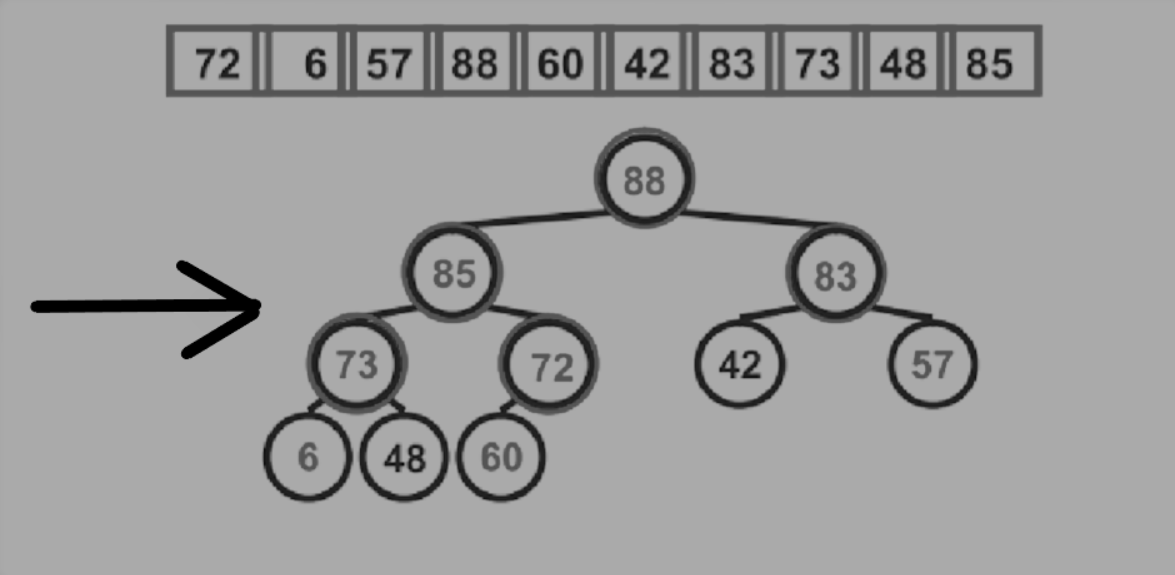
2、数据比较次数和移动次数与输入序列中个排序元素的初始排列有关。

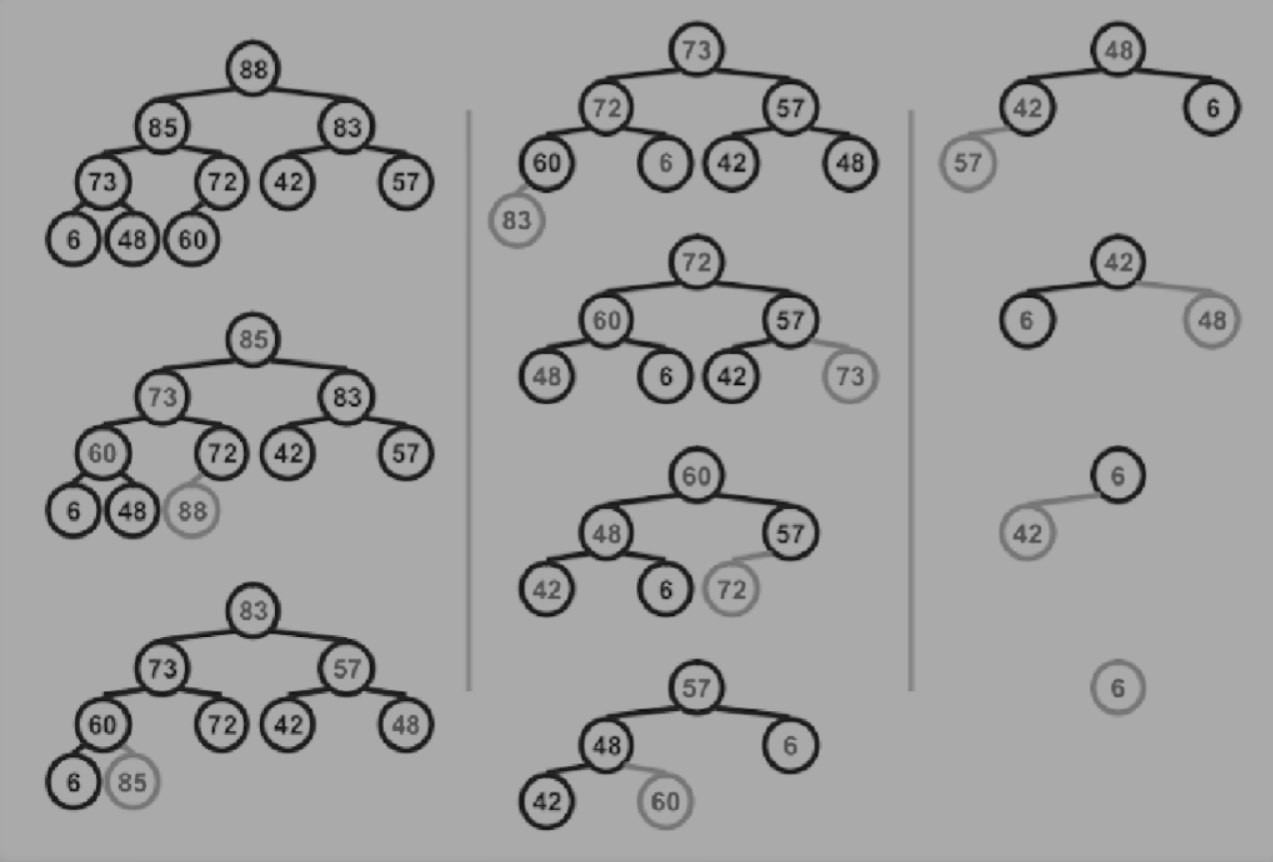
3、堆排序是不稳定的。

### 3.5.2 堆排序核心代码

1. **void** PercDown(**int** V[], **int** i, **int** N)
2. {
3. **int** Child;
4. **int** Tmp;
5. **for** (Tmp = V[i]; 2 \* i + 1 < N; i = Child)
6. {
7. Child = 2 \* i + 1;
8. MoveTime++;
9. CmpTime += 3;
10. **if** (Child != N - 1 && V[Child + 1] > V[Child]) ++Child;
11. **if** (Tmp < V[Child]) V[i] = V[Child];
12. **else** **break**;
13. }
14. V[i] = Tmp;
15. MoveTime++;
16. }
17. **void** HeapSort(**int** V[], **int** Len)
18. {
19. **int** i;
20. **for** (i = Len / 2; i >= 0; --i)
21. {
22. PercDown(V, i, Len);
23. }
24. **for** (i = Len - 1; i > 0; --i)
25. {
26. Swap(V[0], V[i]);
27. PercDown(V, 0, i);
28. }
29. }

### 3.5.3 堆排序图解





## 3.6 归并排序的实现

### 3.6.1 归并排序算法设计

**算法流程：**

**①排序**：

1、分解待排序的n个元素的序列成各具n/2个元素的两个子序列

2、使用归并排序递归地排序两个子序列，当排序的序列长度为1时，递归开始回升，这种情况下不要做任何工作，因为长度为1的每个序列都已经排好序。 3、合并两个已排序的子序列以产生已排序的答案。

**②合并：**

1、有两个子数组，每个数组都已经排好序。

2、首先分别取出两个位置的首元素，选取其中较小的元素，并将其移出该数组，而作为合并结果数组的第一个元素。

3、重复以上步骤，直到一个子数组为空，将剩余的元素按照从前到后的顺序加入结果数组中。

**性能评估：**

1、归并排序的时间复杂度为O() 。

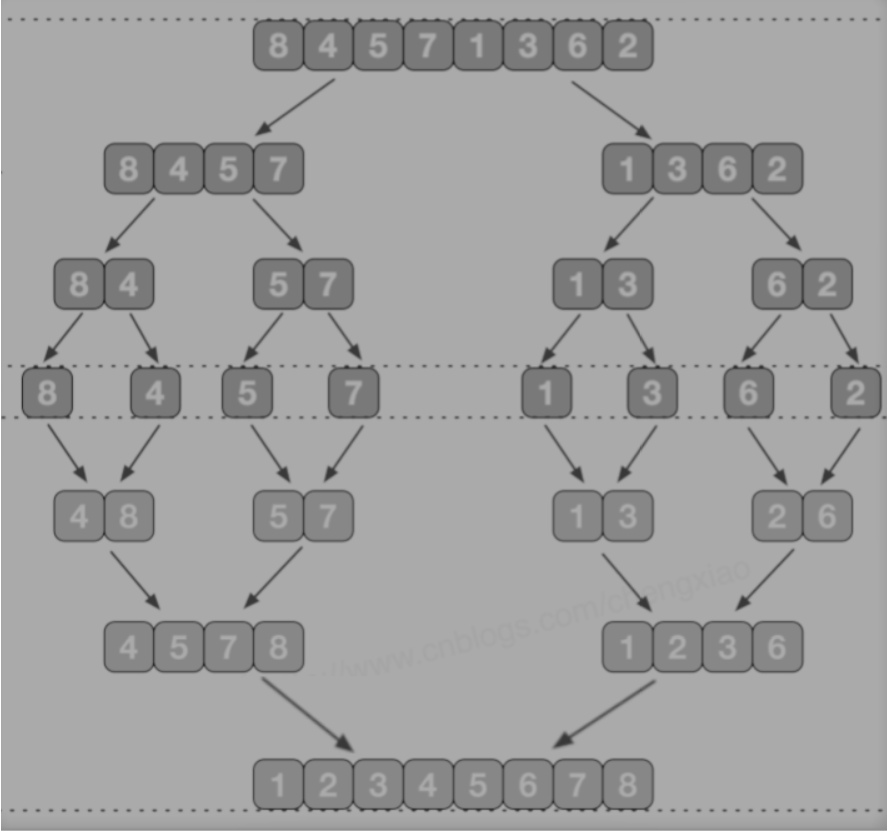
2、数据比较次数和移动次数与输入序列中个排序元素的初始排列无关。

3、归并排序是稳定的。

### 3.6.2 归并排序核心代码

1. **void** Merge(**int** V[], **int** TmpArray[], **int** Lpos, **int** Rpos, **int** RightEnd)
2. {
3. **int** i, LeftEnd, NumElements, TmpPos;
4. LeftEnd = Rpos - 1;
5. TmpPos = Lpos;
6. NumElements = RightEnd - Lpos + 1;
8. **while** (Lpos <= LeftEnd && Rpos <= RightEnd)
9. {
10. ++MoveTime;
11. ++CmpTime;
12. **if** (V[Lpos] <= V[Rpos])
13. TmpArray[TmpPos++] = V[Lpos++];
14. **else**
15. TmpArray[TmpPos++] = V[Rpos++];
16. }
17. **while** (Lpos <= LeftEnd)
18. {
19. TmpArray[TmpPos++] = V[Lpos++];
20. ++MoveTime;
21. }
22. **while** (Rpos <= RightEnd)
23. {
24. TmpArray[TmpPos++] = V[Rpos++];
25. ++MoveTime;
26. }
27. **for** (i = 0; i < NumElements; ++i)
28. {
29. V[RightEnd] = TmpArray[RightEnd];
30. ++MoveTime;
31. --RightEnd;
32. }
33. }
34. **void** Msort(**int** V[], **int** TmpArray[], **int** Left, **int** Right)
35. {
36. **int** Center;
37. **if** (Left < Right)
38. {
39. Center = (Left + Right) / 2;
40. Msort(V, TmpArray, Left, Center);
41. Msort(V, TmpArray, Center + 1, Right);
42. Merge(V, TmpArray, Left, Center + 1, Right);
43. }
44. }
45. **void** MergeSort(**int** V[], **int** Len)
46. {
47. **int** \*TmpArray = **new** **int**[Len];
48. Msort(V, TmpArray, 0, Len - 1);
49. **delete** TmpArray;
50. }

### 3.6.3 归并排序图解



## 3.7 快速排序的实现

### 3.7.1 快速排序算法设计

**算法流程：**

1、取头尾中间三个元素，选择其中的中位数位基准

2、遍历第一个元素之后的元素，如果元素值小于基准，就把元素移到数组的左侧。如果元素值大于基准，就把元素移到数组的右侧。

3、把基准的值移到序列的中间 然后对两个子序列重复使用上述的划分序列的方法，直到所有的元素都已经排好序。

**性能评估：**

1、快速排序的平均时间复杂度为O() 。

2、数据比较次数和移动次数与输入序列中个排序元素的初始排列有关。

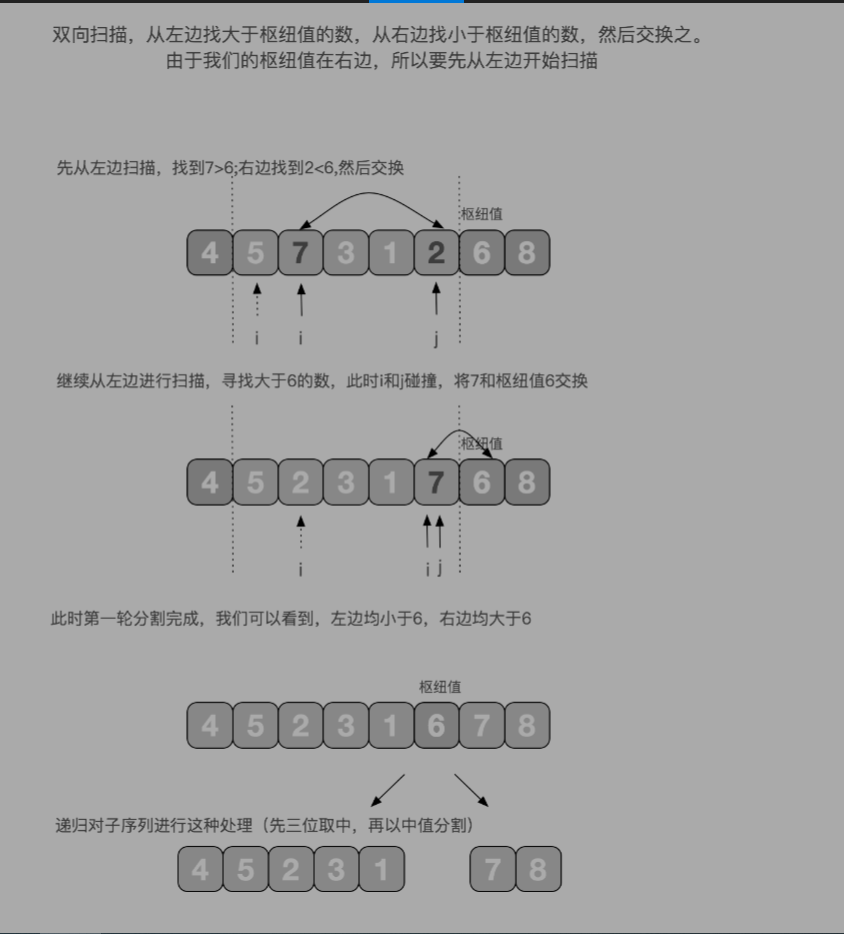
3、快速排序是不稳定的。

### 3.7.2 快速排序核心代码

1. **void** Qsort(**int** V[], **int** Left, **int** Right)
2. {
3. **int** i, j;
4. **int** Pivot;
5. **int** Center = (Left + Right) / 2;
6. CmpTime += 3;
7. **if** (V[Center] < V[Left]) Swap(V[Left], V[Center]);
8. **if** (V[Right] < V[Left]) Swap(V[Left], V[Right]);
9. **if** (V[Right] < V[Center]) Swap(V[Center], V[Right]);
10. **if** (Left + 3 <= Right)
11. {
13. Swap(V[Center], V[Right - 1]);
14. Pivot = V[Right - 1];
15. i = Left; j = Right - 1;
16. **while** (**true**)
17. {
18. **while** (V[++i] < Pivot)++CmpTime;
19. **while** (V[--j] > Pivot)++CmpTime;
20. **if** (i < j) Swap(V[i], V[j]);
21. **else** **break**;
22. }
23. Swap(V[i], V[Right - 1]);
24. Qsort(V, Left, i - 1);
25. Qsort(V, i + 1, Right);
26. }
27. }
28. **void** QuickSort(**int** V[], **int** Len)
29. {
30. Qsort(V, 0, Len - 1);
31. }

### 3.7.3 快速排序图解





## 3.8 基数排序的实现

### 3.8.1 基数排序算法设计

**算法流程：**

本项目中选取2为基数，以便用位运算简化代码。基数排序第i趟将待排数组里的每个数的i位数放到Bucket[j]（j=0或1）队列中，然后再从这两个队列中取出数据，重新放到原数组里，直到i大于待排数的最大位数。

1、数组里的数最大位数是位，就需要排趟，例如数组里最大的数是3位数，则需要排3趟。

2、若数组里共有n个数，则需要两个长度为n的数组Bucket[j]（j=0或1）用来暂存i位上数为j的数。

3、分配结束后，再依次从Bucket[j]数组中取出数据，遵循先进先进原则。

4、循环到趟后结束，排序完成

**性能评估：**

1、基数排序的平均时间复杂度为O() 。

2、基数排序不是基于比较的排序，故其比较次数始终为0。移动次数与输入序列中个排序元素的初始排列无关，与序列中的最大值有关。

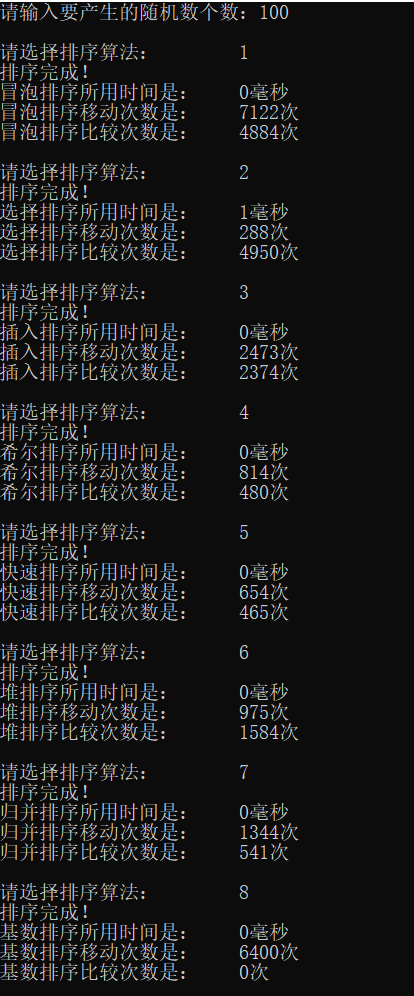
3、基数排序是稳定的。

### 3.8.2 基数排序核心代码

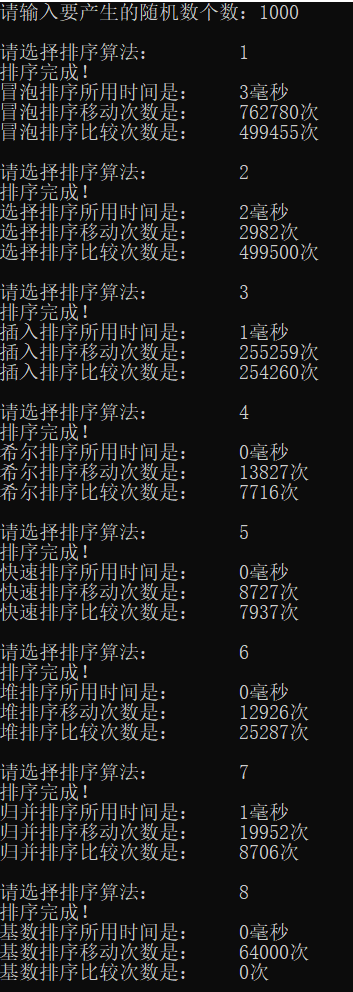
1. **void** RadixSort(**int** V[], **int** Len)
2. {
3. **int** Max = V[0];
4. **int** i, j, k;
5. **for** (i = 0; i < Len; ++i) **if** (Max < V[i])Max = V[i];
6. **int** \*\*Buckets;
7. Buckets = **new** **int**\*[2];
8. Buckets[0] = **new** **int**[Len];
9. Buckets[1] = **new** **int**[Len];
10. **int** BucketsSize[2] = { 0,0 };
11. **int**  TotalCnt = 0;
12. **while** (Max > 0)
13. {
14. TotalCnt++;
15. Max >>= 1;
16. }
17. **int** CurBit, CurNum, CurIndex;
18. **for** (CurBit = 0; CurBit <= TotalCnt; ++CurBit)
19. {
20. **for** (i = 0; i < Len; ++i)
21. {
22. CurNum = V[i];
23. CurIndex = (CurNum >> CurBit) & 1;
24. Buckets[CurIndex][BucketsSize[CurIndex]++] = CurNum;
25. ++MoveTime;
26. }
27. k = 0;
28. **for** (i = 0; i < 2; ++i)
29. {
30. **for** (j = 0; j < BucketsSize[i]; ++j)
31. {
32. V[k++] = Buckets[i][j];
33. ++MoveTime;
34. }
35. BucketsSize[i] = 0;
36. }
37. }
38. **delete** Buckets[0];
39. **delete** Buckets[1];
40. **delete** Buckets;
41. }

# 4 较小范围测试

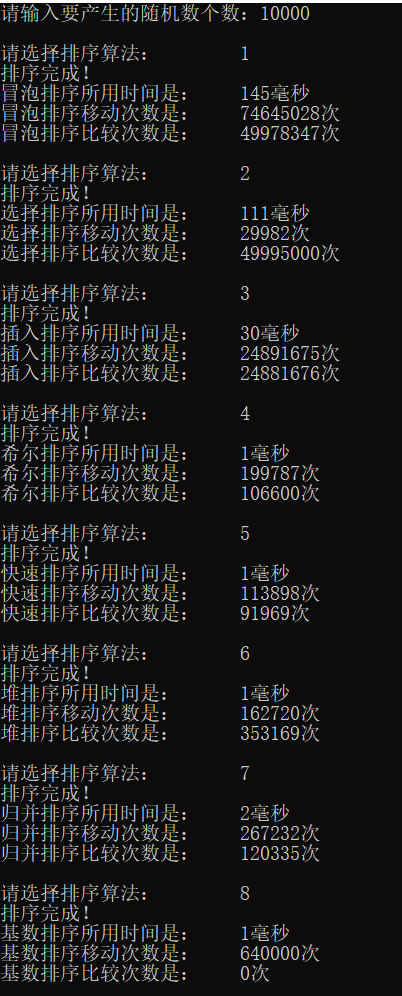
## 4.1 100个随机数



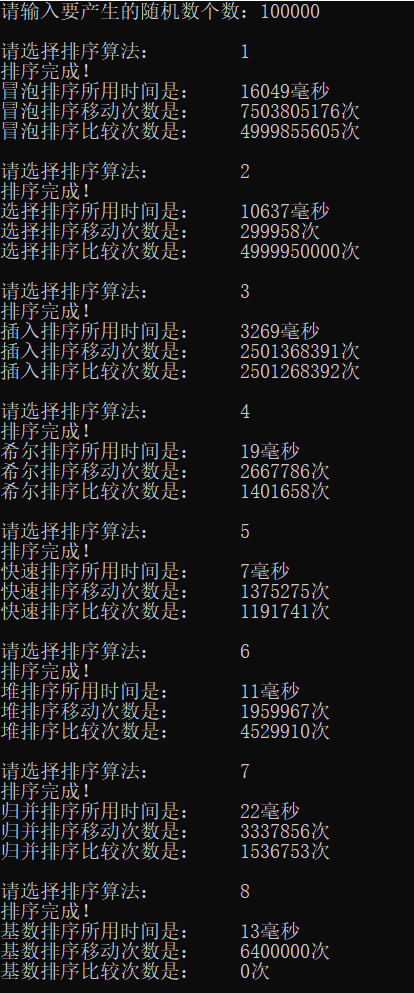
## 4.2 1000个随机数



## 4.3 10000个随机数



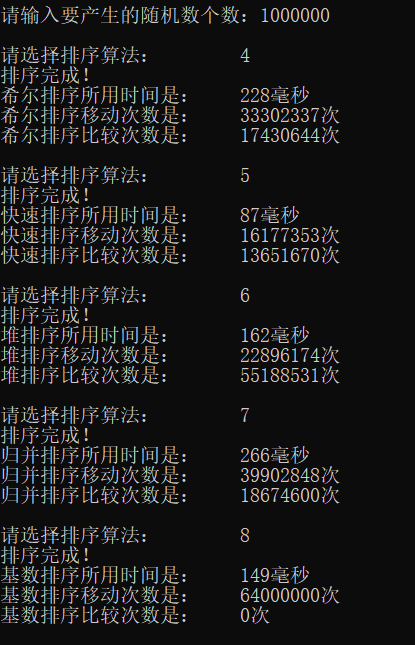
## 100000个随机数



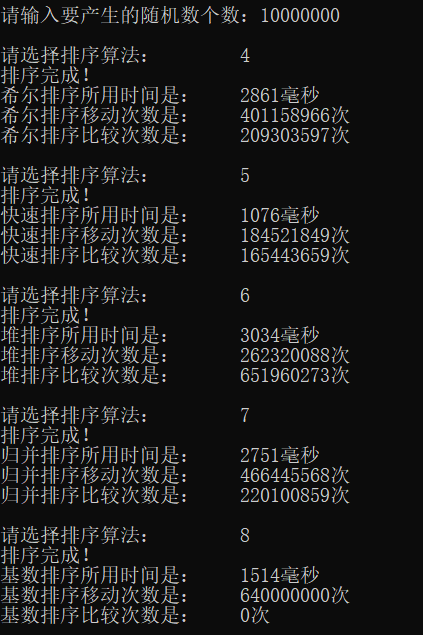
# 5较大范围测试

由于接下来的测试数据范围较大，故时间复杂度为O()的冒泡、选择、插入排序不参与测试。

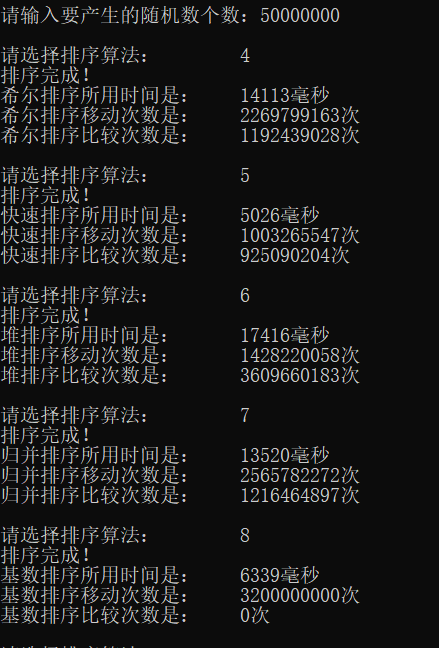
## 5.1 1000000个随机数



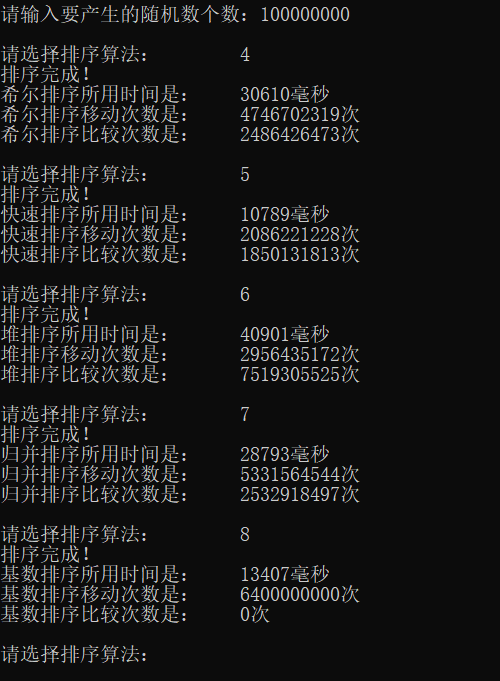
## 5.2 10000000个随机数



## 5.3 50000000个随机数



## 5.4 100000000个随机数



# 实验结果分析

通过对八种算法在不同排序序列规模下的数据比较，可以得到以下结论：

1、直接插入排序，起泡排序和直接选择排序是基本的排序方法,它们对于规模很小的序列非常高效，与其他复杂的排序方法相比，耗时并没有非常大的差距。

2、从内存复杂度来看，这三种基本的排序方法除了一个辅助元素之外，都不需要其他额外的内存空间。

3、当规模非常大时，希尔排序，快速排序，堆排序，归并排序，基数排序的速度优势非常明显，三种基本排序算法的时间复杂度和移动次数呈平方级增长。

4、在不同规模以及稳定性要求方面，需要选择不同的算法。一般来说，当排序规模较大时，考虑采用复杂排序方法，当排序规模较小，而且对内存有限制，可以采用基本排序方法。

5、当排序规模较大并且要求排序稳定，则采用归并排序或者基数排序。

6、当排序规模较大，关键码随机分布，并且不要求稳定性，可以采用快速排序。 7、当排序规模较小，序列已接近有序或者随机分布时，而且还要求稳定性，可以采用直接插入排序。

8、当排序规模较小，而且对稳定性不做要求时，可以采用直接选择排序。

9、当数据规模较大，并且对内存要求较高时，可以选择堆排序、快速排序、希尔排序。

# 一些思考

## 6.1 希尔排序什么在实验过程中表现要优于堆排序？

虽然使用Sedgewick增量的平均时间复杂度还是学术界的一个开放问题，但是基于测试的估计表明其平均时间复杂度约为O()，堆排序的时间复杂度为O()，单单从时间复杂度上来看，直觉上堆排序应该优于希尔排序，但是实验中堆排序却是五个复杂排序算法中表现最差的一个。

先单就时间复杂度上来计算，两种排序算法的性能差距应该体现在与这两项的数值差距上，由于内存性能限制，本项目测试最大数据范围仅为一亿，故这里的n取来估计：

≈21.544<≈26.575

所以，在一亿这个量级上，实际上对数复杂度是不如六分之一次方的。

当然，考虑实际性能时不能只看复杂度，还需要考虑复杂度时间常数，堆排序的常数很大，所以在常数和n的量级的影响下，堆排序的表现在本项目实验中劣于希尔排序。

实际上，随着n的量级的增大，堆排序的性能又会反超使用Sedgewick增量的希尔排序。

## 6.2 快速排序什么在实验过程中表现要优于基数排序？

本项目采用MersenneTwisterMT19937算法生成随机数，可以生成32位的随机数，所以基数排序大部分情况都需要进行32趟的单关键词排序，而在前一个问题分析中也计算过，一亿量级的对数也才仅仅26.5左右。如果修改代码，让产生的随机数范围缩小，就会发现基数排序变得快于快速排序。