排序上机报告

学号：1500011398

姓名：秦光辉

题目描述：

利用 3 种以上的排序方法，随机产生一个序列（种子为 0），将序列排序。将序列排序之后再排序，或者反序之后再排序，讨论不同算法排序的速度，比较次数，以及排序的时间和数据量之间的关系。

题目分析：

题目要求种子为 0，意在使得每次产生的序列都是稳定的。不引入时间变量，这样我们每次启动程序的时候获得的序列都会是相同的。

不同算法适用于不用的数据，例如基数排序对于位数比较少，数据量比较大的数据效率很高，它的耗费时间和数据量成线性关系，而如果数据本身已经很有规律，则基数排序效果很差，效率极低。我们通过比较，可以获知在什么情况下使用什么样的算法最有利。

数据的排序速度取决因素有很多，除了算法之外，还有数据的疏密、数据的位数、数据的有序性等等。这些因素都会影响排序的效率，我们在排序算法中都需要予以考虑。

由于算法有不同，有的算法使用了顺序表，有的算法使用了链表。所以排序时间并不能非常好的体现算法的优劣，但是比较次数可以更加公正的比较算法优劣。我们在评价算法的效率的时候，比较次数也是一个重要因素。

基数排序是其中最不直观的排序方式。基数排序一般默认进制是十进制，如果采用 16 进制，32 进制，或者其他进制，效率会怎么样？我会在下面加以讨论。

关键函数和算法结构分析：

我一共使用了 7 种排序方式排序，4 种使用了链表，3 种使用了顺序表。我逐一介绍。

1、element<K, V> 类

在介绍函数之前，介绍一下 element 类。顾名思义 element 是一个元素，是链表的链结。我把它的成员列在下面。

三个数据成员分别是：

关键码

K key;

值

V value;

指向下一个元素的指针

element<K, V> *next = nullptr;

一些函数成员：

default constructor

element<K, V>() = default;

constructor

element<K, V>(K n_key, element<K, V> *n_next = nullptr, V n_value = V()) :

key(n_key), next(n_next), value(n_value) {}

destructor

~element();

assignment function

element<K, V> &operator=(element<K, V> &E);

类似于指针的 [] 操作，可以定位到之后的某个链结

element<K, V> &operator[](int n);

返回两个链结之间的距离

int operator-(const element<K, V> &e);

copy constructor

element<K, V>(element<K, V> &E);

添加一个元素到链表的末尾

```
bool add(K n_key, V n_value = V());
```

交换两个元素

```
void swap(element<K, V> *ptr);
```

冒泡排序，n_compare 记录比较次数

```
void bubbleSort(int &n_compare);
```

插入排序，n_compare 记录比较次数

```
void insertsort(int &n_compare);
```

选择排序，n_compare 记录比较次数

```
void select_sort(int &n_compare);
```

堆排序，n_compare 记录比较次数

```
void heap_sort(int &n_compare);
```

打印链表中的所有元素

```
void print();
```

2、二分插入排序

二分插入排序定义在 element 函数成员中的 insert_sort()中。

二分插入排序是所有排序中相对直观的一种，其思路是：

1、如果只有一个元素，不执行排序，直接返回。
2、将前两个元素排序。
3、如果待比较的元素小于第一个元素，将其定位在第一个
4、如果大于最后一个元素，放到最后一个元素，同时将 end 向后顺延一位
5、否则执行二分搜索，将元素插入已经排序好的序列的适当位置
6、如果当前指针为空，说明排序执行完毕，返回

这种思路简单易行，但是由于融合了二分搜索和插入排序的思路，代码比较长。

3、冒泡排序

冒泡排序是一种简单的比较排序。我把它定义在 element 成员函数的 bubble_sort() 中。

其思路如下：

1、统计元素个数。
2、对前 n 个元素进行一次搜索，搜索过程中比较相邻的两个元素，如果发现两个元素反序，则交换。
3、对前 n-1, n-2, n-3 … 2 个元素执行 2 的操作。
4、如果全部执行完，或者其中某一次执行过程中没有进行过交换，则直接返回。

冒泡排序很好理解，但是效率比较低。

4、选择排序

选择排序我定义在了 element 的函数成员 select_sort() 中。

选择排序的基本思路如下：

1、设当前操作元素是第一个元素。
2、在当前元素之后的所有元素中搜索，找到最小元素。
3、交换最小元素和当前元素。
4、递增当前元素，重复 2、3。
5、到达最后一个元素的时候，推出循环，排序执行完毕。

选择排序难度不高，思路简单，效率比较低。

5、堆排序

堆排序我定义在了 element 函数成员 heap_sort() 中。

堆排序思路如下：

1、统计元素个数。
2、创建一个堆。
3、将所有的元素入队。
4、将最小元素出队，记录在当前序列的第一个位置。
5、继续出队，依次记录。
6、全部出队之后结束排序，return。

这种思路比较快，但是需要一个辅助的优先队列。优先队列我定义在了 priority_queue 中。

6、归并排序

归并排序我定义在了 merge_sort.h 中。

1、将包含所有元素的 vector 放在另一个 vector 中，即 vector<vector<T> >。二维数组中的每个 vector 包含一个元素。
2、调用递归函数执行归并排序。
3、搜索当前数组，把数组中两个相邻的 vector 排序之后归并为一个 vector。
4、搜索结束后再次递归执行归并排序。
5、如果发现二维数组只剩下一个元素，说明排序执行完毕，return。
6、将二维数组恢复成一维数组，return。

归并排序比较复杂，但是效率相对较高。我这里没有考虑多路归并函数，仅写了二路归并函数。

7、快速排序函数

快速排序函数我定义在了 quick_sort() 中。

1、以第一个元素为标准，记作 tmp。
2、把所有小于 tmp 的元素放到 tmp 左边，其余放到右边。
3、对 tmp 左边的和右边的数组分别进行整理，调用自身的递归
4、当数列长度为 1 的时候，直接返回。

快速排序适应性强，几乎可以处理所有情况，而且速度也很快。相对来说是一种比较好的排序算法。

8、基数排序法

基数排序法定义在 radix.h 中。

基数排序法思路如下：

1、找最大元素。
2、计算其位数。
3、创建一个队列的数组。个数由最大元素的位数决定。
4、按照最低位开始收集元素，之后再分配一次。
5、位数逐渐上升，到达最高位。
6、最高位分配之后，排序结束，return。

基数排序法除了接受排序的 vector，计数的 n_compare 以外，还接受一个 int 作为进制。进制不同，排序的速度也不同，下面加以讨论。

结果分析：

1、横向对比

我用相同的数据比较了几个算法的速度。

实验一：

数据量：1000

数据范围：1 ~ 32767

时间单位：ms

	乱序用时	比较次数	正序用时	比较次数	逆序用时	比较次数
BubbleSort	6203	496944	20	999	6229	499500
HeapSort	0	5113	1	4230	1	11519
InsertSort	43	4433	0	999	0	999
MergeSort	2033	8704	2019	5064	2045	4923
QuickSort	29	15378	553	501498	551	497327
RadixSort	1082	60	1072	69	1011	60
SelectSort	11	499500	11	499500	10	499500

表 1 第一次横向对比实验数据表

实验二：

数据量：5000

数据范围：1 ~ 32767

时间单位：ms

	乱序用时	比较次数	正序用时	比较次数	逆序用时	比较次数
BubbleSort	1012340	12483970	625	4999	1023423	13233243
HeapSort	4	32153	3	26891	5	76003
InsertSort	1481	27822	1	4999	0	4999
MergeSort	45771	56804	46064	33541	47066	29804
QuickSort	174	92603	Stack overflow	Stack overflow	Stack overflow	Stack overflow
RadixSort	1154	60	1284	60	1177	60
SelectSort	363	12497500	390	12497500	420	12497500

表 2 第二次横向对比实验数据表

实验三：

数据量：1000

数据范围：0 ~ 1073741823

时间单位：ms

	乱序用时	比较次数	正序用时	比较次数	逆序用时	比较次数
BubbleSort	8140	499200	22	999	6852	499500
HeapSort	1	5180	1	4227	1	11536
InsertSort	43	4476	0	999	0	999
MergeSort	2203	8754	2094	5052	2134	4932
QuickSort	29	15013	573	501498	576	501498
RadixSort	1040	110	1038	110	1047	110
SelectSort	13	499500	13	499500	12	499500

表 3 第三次横向对比实验数据表

从横向对比试验来看，在数据量相同，数据范围确定的时候，冒泡排序用时最长，归并排序也很慢，最快的是堆排序，选择排序。

当数据本身比较有序的情况下，冒泡排序，二分插入排序用时大大减小，快速排序用时大大增加。其他排序时间没有变化。

当数据基本是逆序的情况下，二分插入排序用时大大减小，快速排序用时大大增加。

当数据的范围扩大的时候，各种排序所用时间都有所延长，但是并没有看出明显规律。

2、纵向对比

我单独讨论了每个算法的性质。

1、冒泡排序

我测试了在数据量增加的情况下冒泡排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

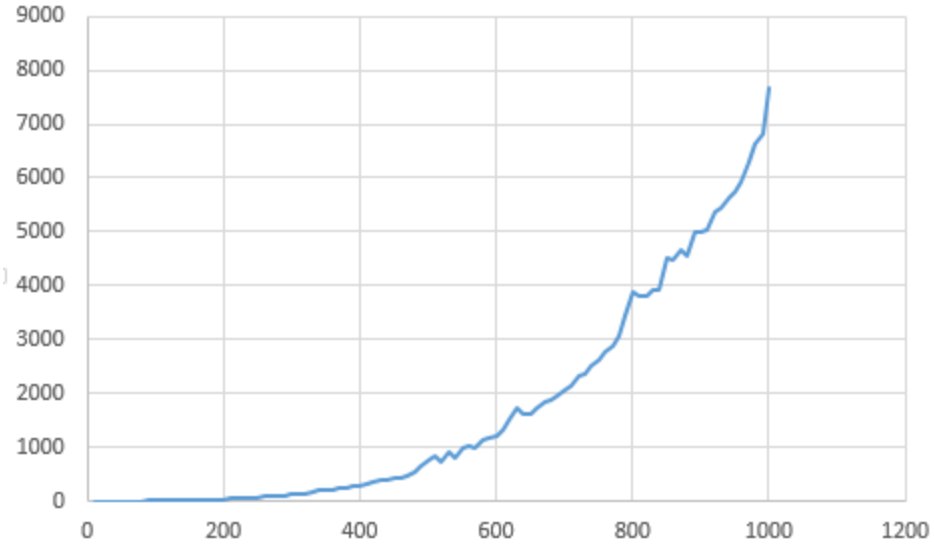

图 1 冒泡排序时间消耗和数据量的关系图

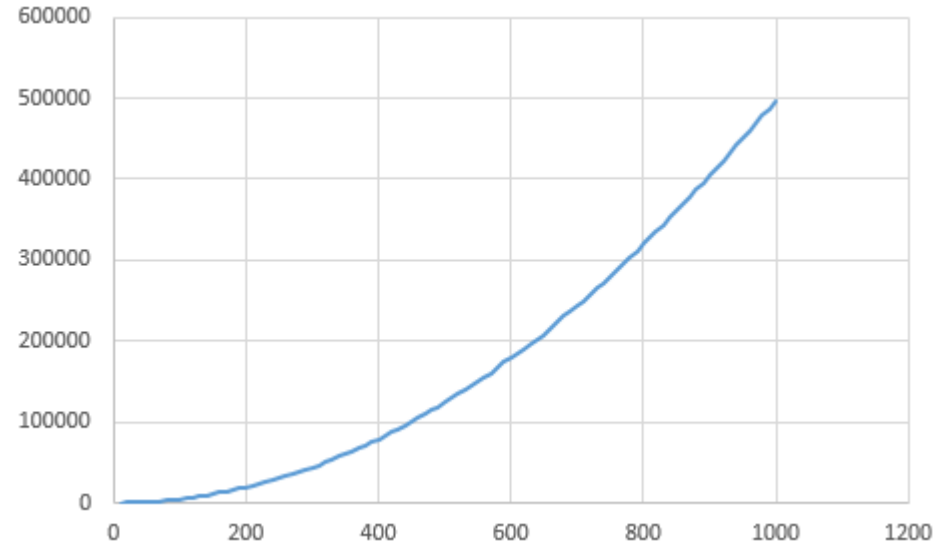

图 2 冒泡排序比较次数和数据量的关系图

上述图像并不直观，我们使用双对数图来观察这三个量之间的关系。

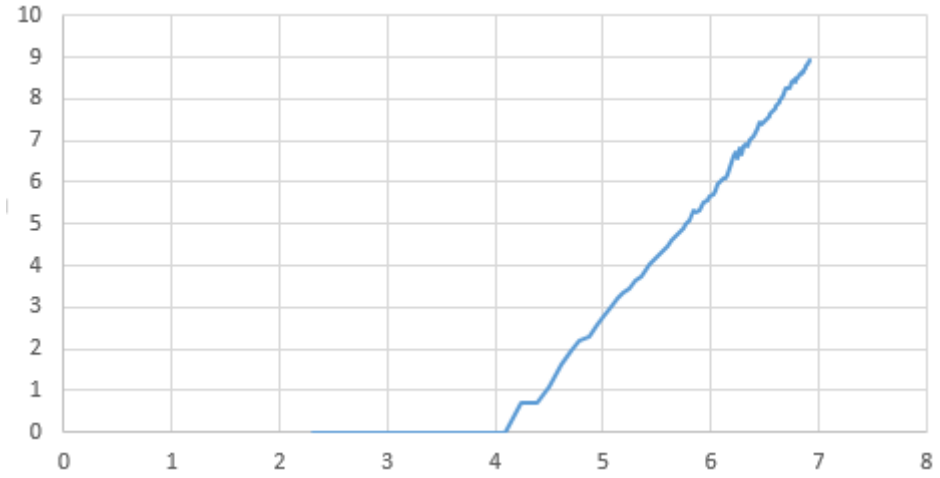

图 3 冒泡排序时间消耗和数据量的关系双对数图

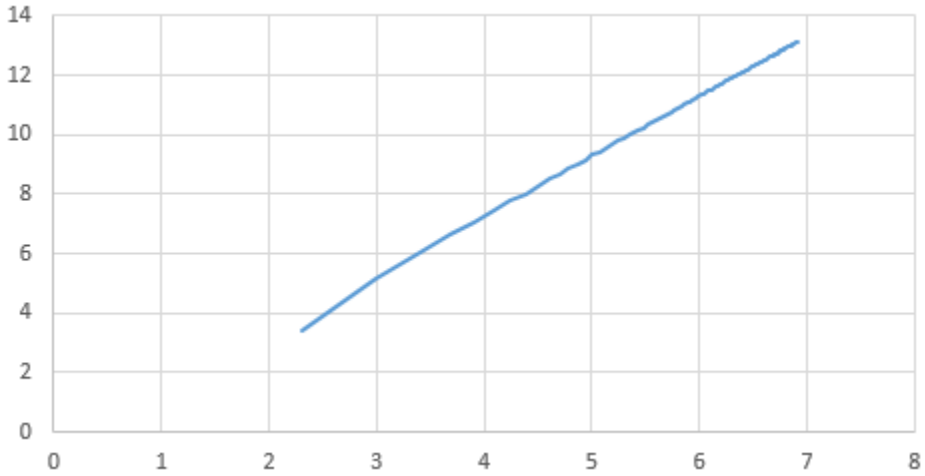

图 4 冒泡排序比较次数和数据量的关系双对数图

虽然实验次数比较少，但是我们可以大致看出，耗时大约和数据量的三次方成正比，比较次数和数据量的二次方成正比。

起泡排序是稳定的。

2、二分插入排序

我测试了在数据量增加的情况下冒泡排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

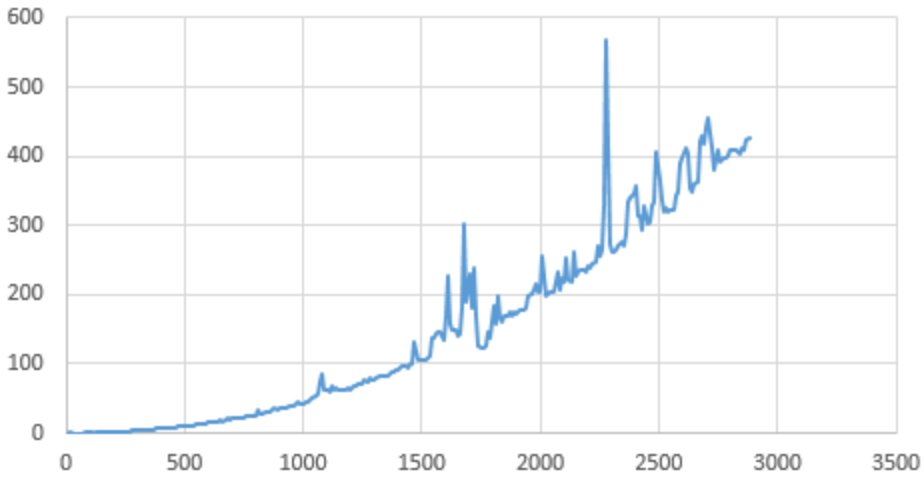

图 5 二分插入排序时间消耗和数据量的关系图

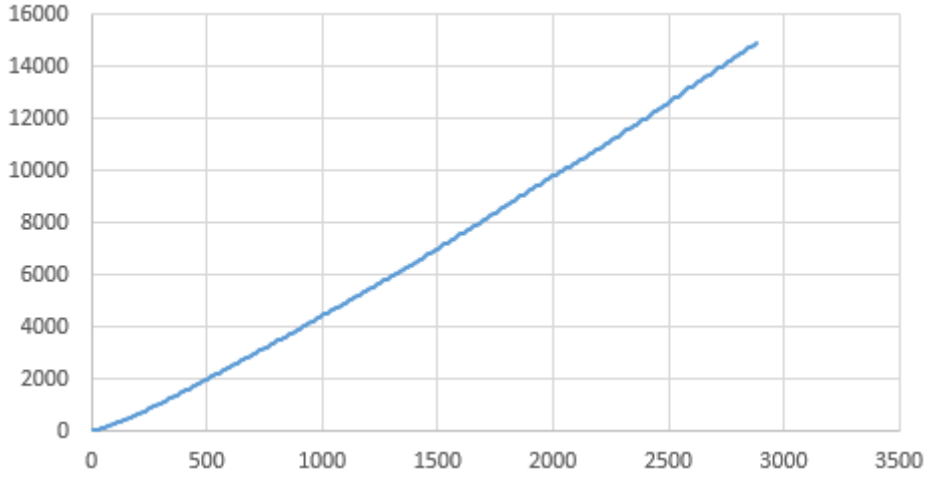

图 6 二分插入排序比较次数和数据量的关系图

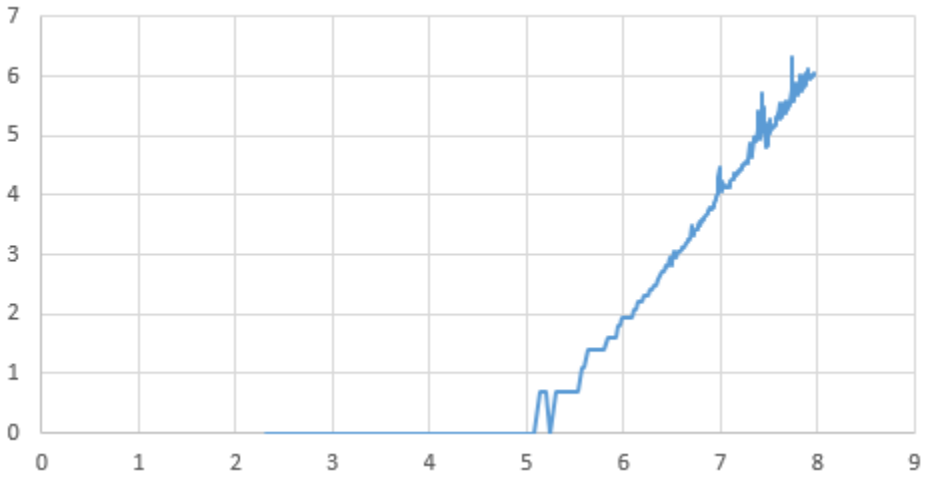

图 7 二分插入排序时间消耗和数据量的双对数关系图

从图中可以看出，二分插入排序的时间消耗和数据量成一次正比，比较次数和数据量成一次正比。

函数控制适当的情况下，二分插入排序是稳定的。

3、堆排序

我测试了在数据量增加的情况下堆排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

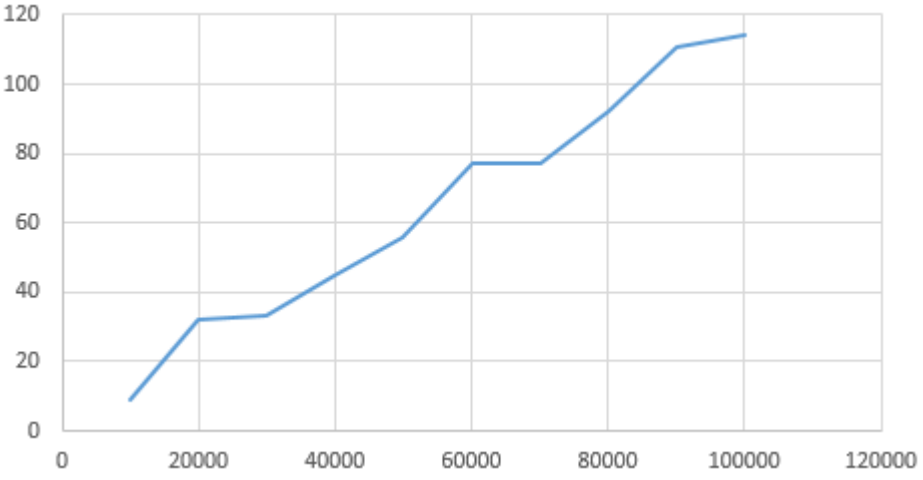

图 8 堆排序时间消耗和数据量的关系图

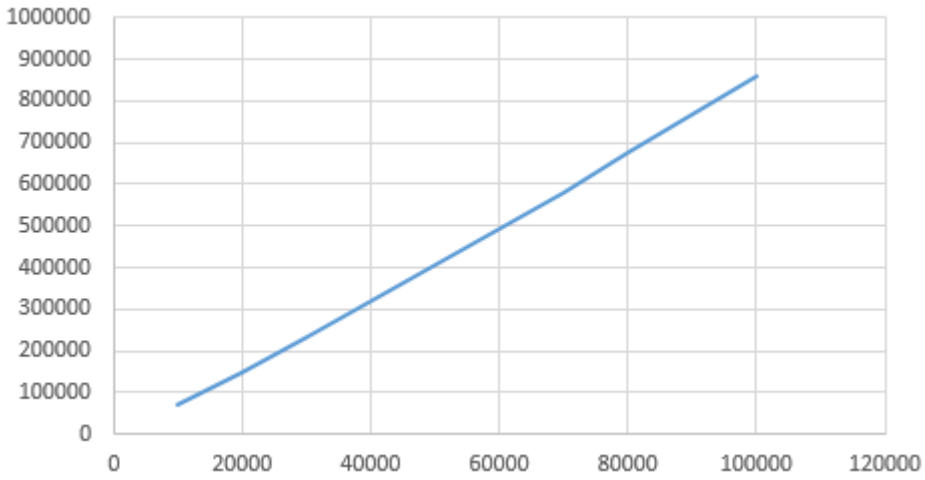

图 9 堆排序比较次数和数据量的关系图

从图上我们可以看出，堆排序所用时间，比较次数和数据量成正比关系。

堆排序是不稳定的。

4、选择排序

我测试了在数据量增加的情况下选择排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

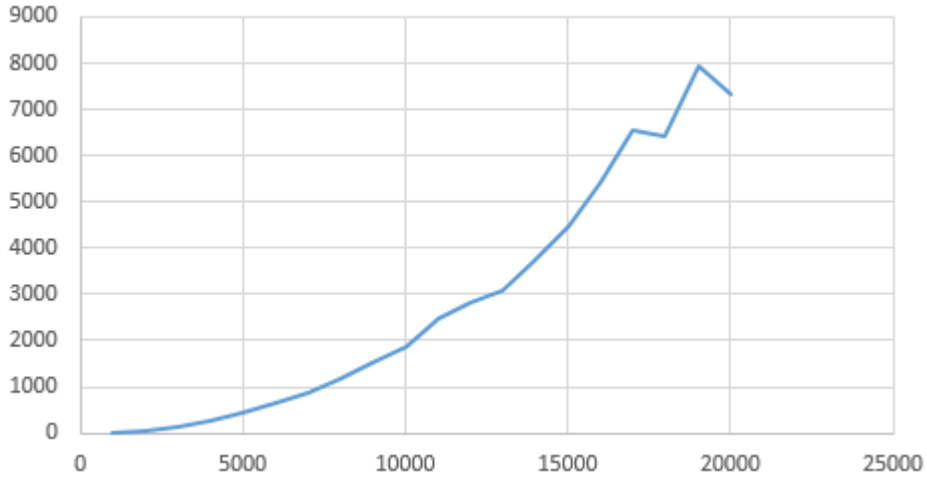

图 10 选择排序时间消耗和数据量的关系图

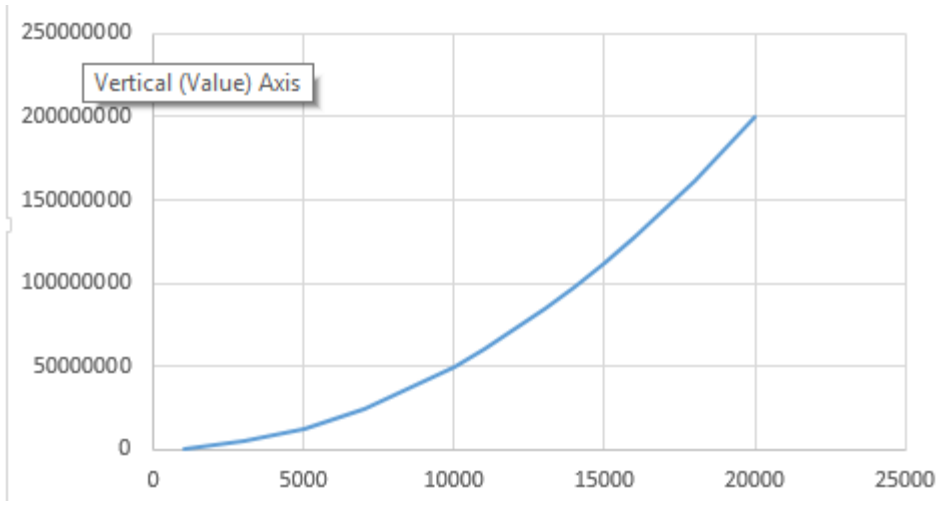

图 11 选择排序比较次数和数据量的关系图

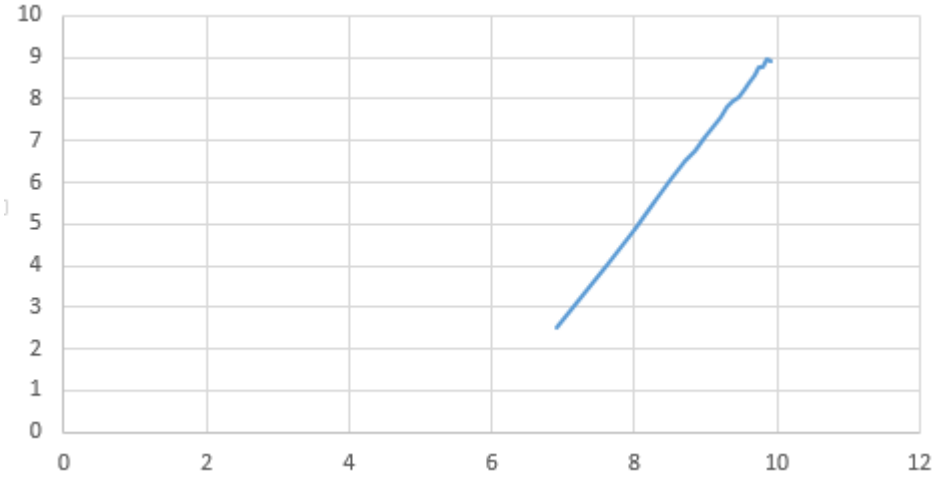

图 12 选择排序时间消耗和数据量的双对数关系图

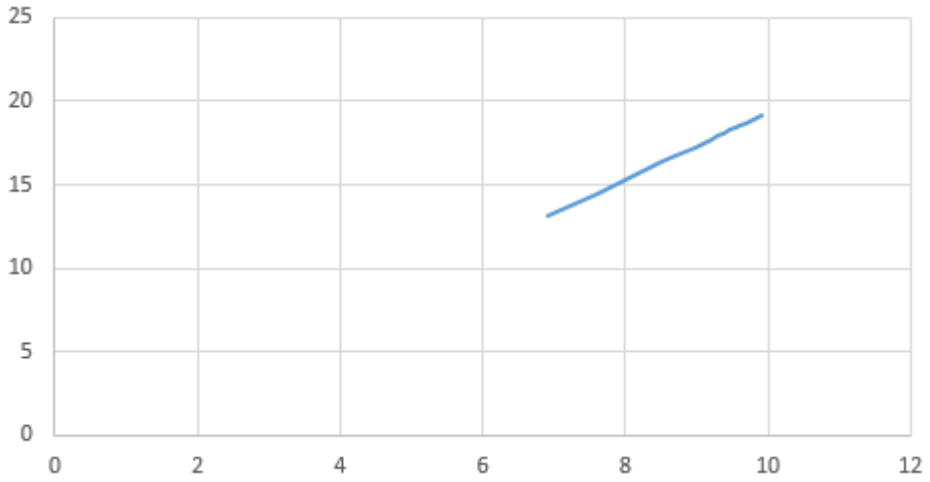

图 13 选择排序比较次数和数据量的双对数关系图

从图上我们可以看出，选择排序所用时间，比较次数和数据量二次方成正比关系。

选择排序是不稳定的。

5、归并排序

我测试了在数据量增加的情况下归并排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

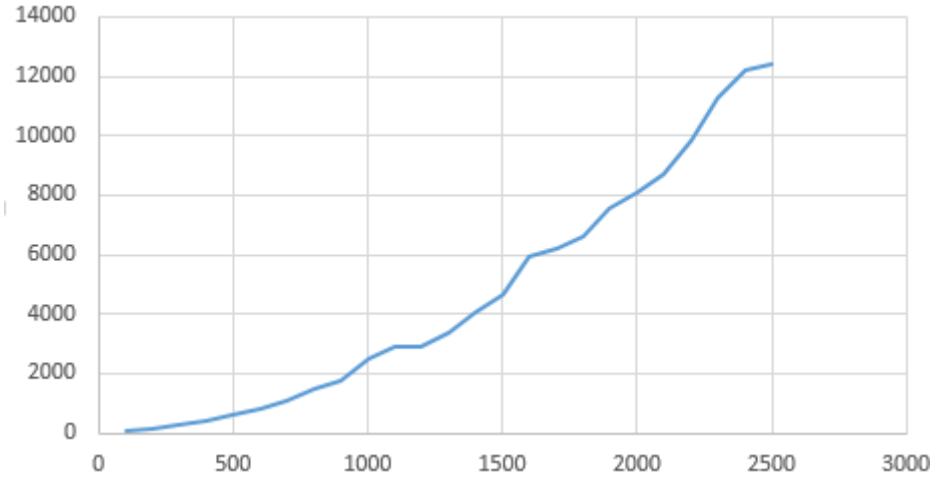

图 14 归并排序时间消耗和数据量的关系图

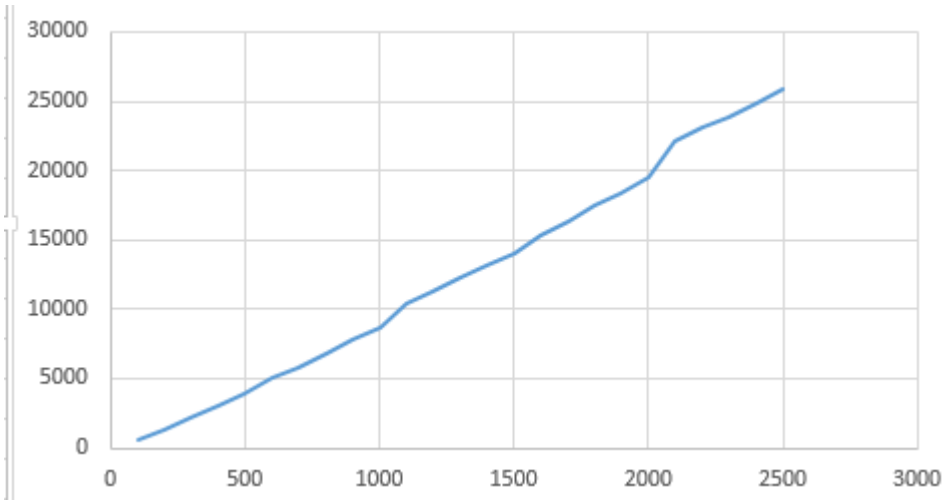

图 15 归并排序比较次数和数据量的关系图

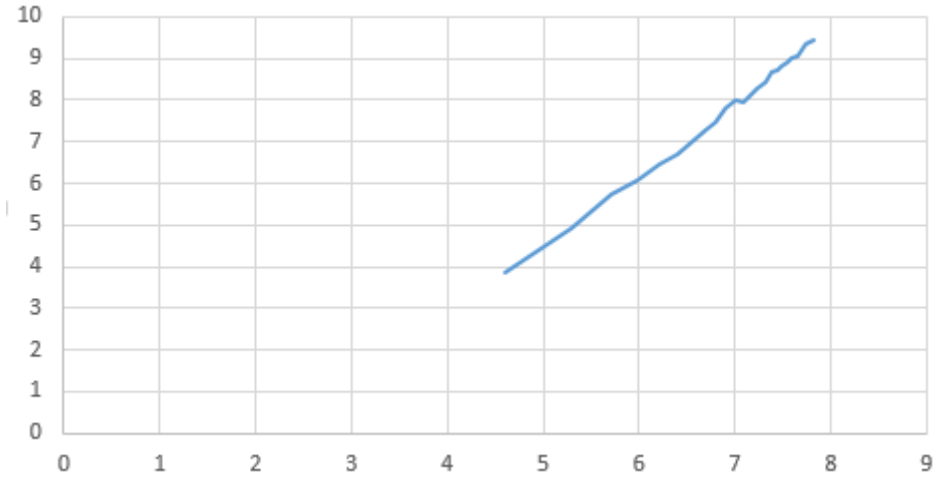

图 16 归并排序时间消耗和数据量的双对数关系图

从图上我们可以看出，归并排序所用时间和数据量的二次方成正比，比较次数和数据量成正比关系。

归并排序是稳定的。

6、基数排序

我测试了在数据量增加的情况下基数排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

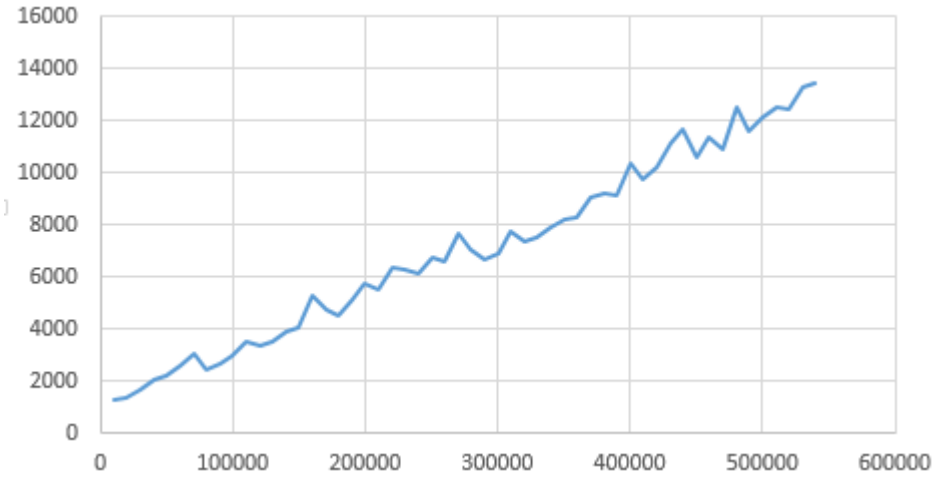

图 17 基数排序耗时和数据量的关系图

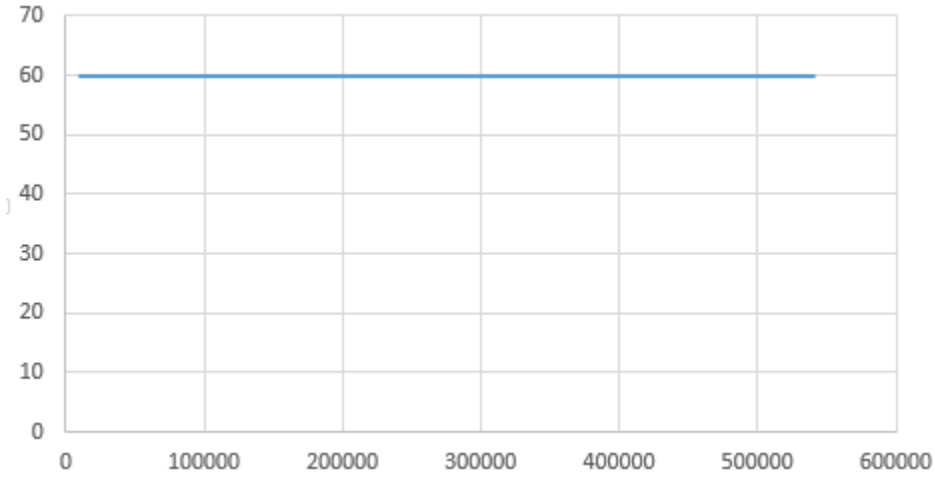

图 18 基数排序比较次数和数据量关系图

从图上我们可以看出，基数排序所用时间和数据量成正比关系。它的比较次数这个数据并没有什么意义，在位数确定的情况下，它是固定的。

基数排序是稳定的。

7、快速排序

我测试了在数据量增加的情况下快速排序的时间消耗和比较次数增长。

数据在附件中，绘制图像如下。

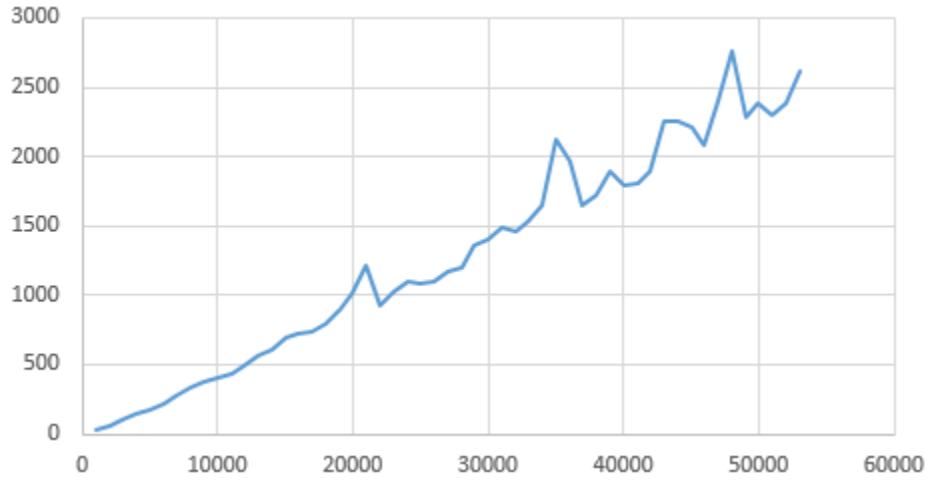

图 19 快速排序时间消耗和数据量的关系图

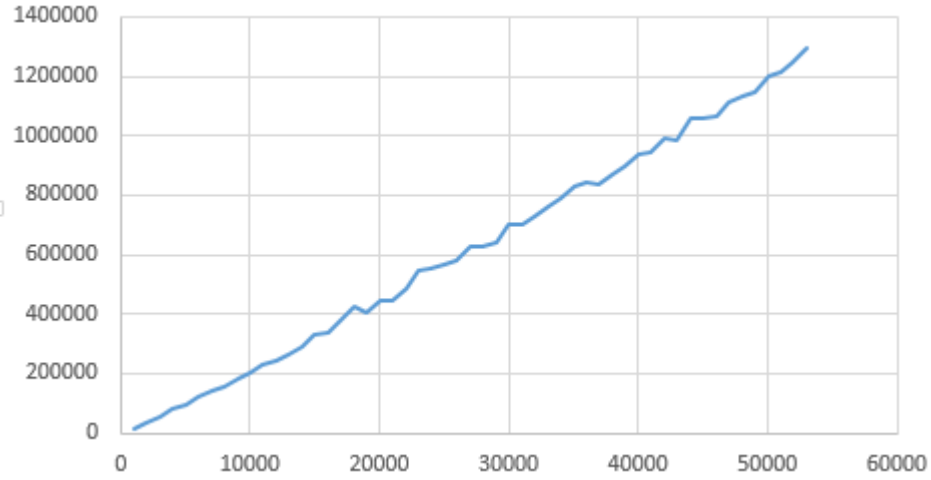

图 20 快速排序比较次数和数据量的关系图

从图上我们可以看出，快速排序所用时间，比较次数和数据量成正比关系。

快速排序是不稳定的。

改进和讨论：

1、冒泡排序占用了大量的时间，我认为这是因为冒泡排序使用了错误的数据结构所致。在我的程序中，冒泡排序使用了链表的结构。这是十分不适当的。链表结构在进行定位的时候非常非常慢，导致冒泡排序在数据量达到 5000 的时候居然需要十分钟来排序。

这提醒我，以后需要注意数据结构的选择。链表结构有很多优势，但不是所有的情况都适合使用链表结构。

2、快速排序在数据有规律的时候，可能会使用大量的时间，因为它是以第一个元素作为参照去比较其他元素的，如果元素整体有单调递减或者单调递增的趋势，这种算法往往非常耗时，但是一些基础的算法，比如冒泡，插入排序，往往非常快。

3、基数排序是一种耗时稳定的排序方法，它时间消耗量和数据的性质没什关系，无论是顺序还是逆序。

4、基数排序默认的进制是十进制，如果我们不使用十进制会怎么样？我做了一个测试。数据在附件中，绘制图像如下：

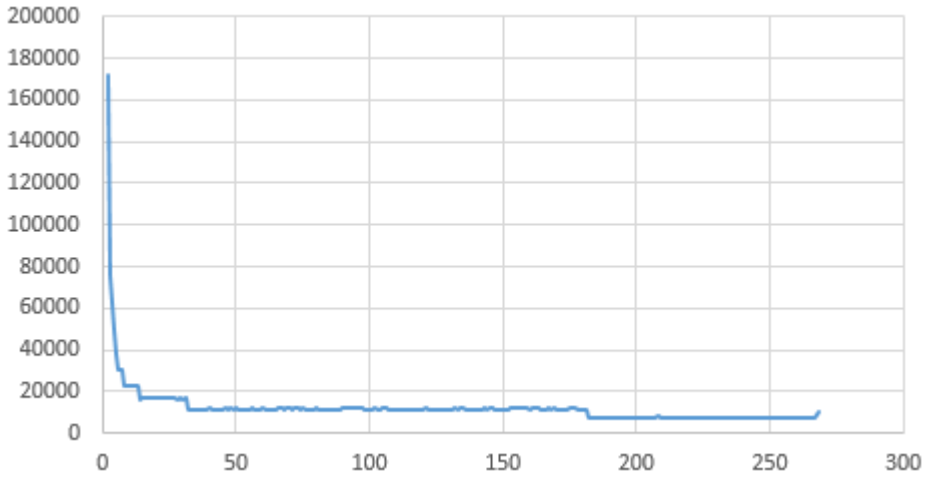

图 21 基数排序处理时间和进制的关系

可见十进制并不是最好的选择。当进制比最大数的二次方根稍小的时候，这个处理速度达到最高。

我们在使用基数排序的时候，不妨测试一下在各种进制下的处理效率，然后再选择最好的排序方式。

附件：

文件夹的 sort_data 是我的实验数据。