目录

一、数据集生成方法

考虑生成正序数据集，逆序数据集和随机数数据集首先规定数据集长度，然后通过文件指针写入到data.txt 数据集文件中。关闭文件，并以读取方式再次打开，每次打开都会清除上次的数据集

1、正序/逆序数据集的生成（以逆序为例）

```
#include<time.h>
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
  cout<<"How long is the data?  ";
  int len,i;/*用len来表示数据集长度*/
  ofstream data_in("data.txt",ios::trunc);/*每次删除并重建数据集*/
  cin>>len;
  for(i=len;i>=0;i--)
      data_in<<i<<" ";
  data_in.close();/*创建数据集完成，保存并退出*/
  return 0;
}
```

2、随机数据集的生成

```
#include<time.h>
#include<iostream>
#include<fstream>
#include<stdlib.h>
using namespace std;
int main()
{
  cout<<"How long is the data?   ";
  int len,i;/*用len来表示数据集长度*/
  int x;
    srand((unsigned)time(NULL));
  ofstream data_in("data.txt",ios::trunc);/*每次删除并重建数据集*/
  cin>>len;
  for(i=1;i<=len;i++)
  {
      x=rand()%len;
      data_in<<x<<" ";
  }
  data_in.close();/*创建数据集完成，保存并退出*/
  return 0;
}
```

二、时间测算方法

使用 time.h 中的方法来计算

```
#include <time.h>
clock_t start, stop; /* clock_t is a built-in type for processor time (ticks) */
double duration; /* records the run time (seconds) of a function */
int main ( )
{/* clock() returns the amount of processor time (ticks) that has elapsed since
the program began running */
      start = clock(); /* records the ticks at the beginning of the function call */
      function(); /* run your function here */
      stop = clock(); /* records the ticks at the end of the function call */
      duration = ((double)(stop - start))/CLK_TCK;
      /* CLK_TCK is a built-in constant = ticks per second */
return 1;
}
```

三、移动和比较的规定

均以数组中的数据移动和比较为准，其他元素的比较一概忽略

四、测试结果

```
测试数据规模为100,1000,5000,10000,40000

数据规模为100：
  插入排序：
    顺序数据    耗时:0s 移动次数:0       比较次数:99
  归并排序：
    顺序数据    耗时:0s 移动次数:672     比较次数:356
  快速排序：
    顺序数据    耗时:0s 移动次数:0       比较次数:5148
  插入排序：
    逆序数据    耗时:0s 移动次数:4950    比较次数:5049
  归并排序：
    逆序数据    耗时:0s 移动次数:672     比较次数:316
  快速排序：
    逆序数据    耗时:0s 移动次数:0       比较次数:5148
  插入排序：
    无序数据    耗时:0s 移动次数:2324    比较次数:2423
  归并排序：
    无序数据    耗时:0s 移动次数:672     比较次数:540
  快速排序：
    无序数据    耗时:0s 移动次数:228     比较次数:938

数据规模为1000：
  插入排序：
    顺序数据    耗时:0.001s      移动次数:0       比较次数:999
  归并排序：
    顺序数据    耗时:0s 移动次数:9976    比较次数:5044
  快速排序：
    顺序数据    耗时:0.002s      移动次数:0       比较次数:501498
  插入排序：
    逆序数据    耗时:0.006s      移动次数:499500  比较次数:500499
  归并排序：
    逆序数据    耗时:0s 移动次数:9976    比较次数:4932
  快速排序：
    逆序数据    耗时:0.003s      移动次数:0       比较次数:501498
  插入排序：
    无序数据    耗时:0.003s      移动次数:250190  比较次数:251189
  归并排序：
    无序数据    耗时:0.001s      移动次数:9976    比较次数:8699
  快速排序：
    无序数据    耗时:0s 移动次数:3763    比较次数:15176
```

```
数据规模为5000:
  插入排序:
    顺序数据    耗时:0s 移动次数:0        比较次数:4999
  归并排序:
    顺序数据    耗时:0.002s     移动次数:61808   比较次数:32004
  快速排序:
    顺序数据    耗时:0.052s     移动次数:0       比较次数:12507498
  插入排序:
    逆序数据    耗时:0.129s     移动次数:12497500        比较次数:12502499
  归并排序:
    逆序数据    耗时:0.001s     移动次数:61808   比较次数:29804
  快速排序:
    逆序数据    耗时:0.046s     移动次数:0       比较次数:12507498
  插入排序:
    无序数据    耗时:0.061s     移动次数:6254774         比较次数:6259773
  归并排序:
    无序数据    耗时:0.001s     移动次数:61808   比较次数:55167
  快速排序:
    无序数据    耗时:0.001s     移动次数:24378   比较次数:96563

数据规模为10000:
  插入排序:
    顺序数据    耗时:0.001s     移动次数:0       比较次数:9999
  归并排序:
    顺序数据    耗时:0.003s     移动次数:133616  比较次数:69008
  快速排序:
    顺序数据    耗时:0.184s     移动次数:0       比较次数:50014998
  插入排序:
    逆序数据    耗时:0.479s     移动次数:49995000        比较次数:50004999
  归并排序:
    逆序数据    耗时:0.002s     移动次数:133616  比较次数:64608
  快速排序:
    逆序数据    耗时:0.187s     移动次数:0       比较次数:50014998
  插入排序:
    无序数据    耗时:0.209s     移动次数:24857490        比较次数:24867489
  归并排序:
    无序数据    耗时:0.003s     移动次数:133616  比较次数:120366
  快速排序:
    无序数据    耗时:0.002s     移动次数:52699   比较次数:214998
```

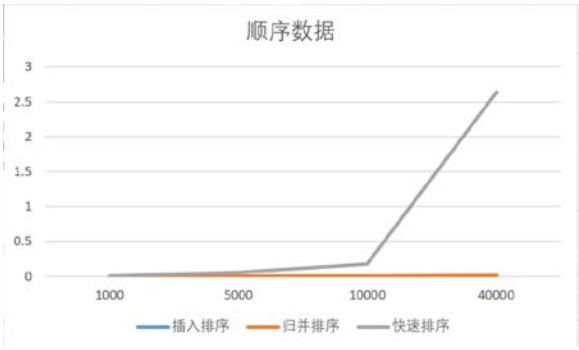
```
数据规模为40000:
  插入排序:
    顺序数据    耗时:0s 移动次数:0        比较次数:39999
  归并排序:
    顺序数据    耗时:0.012s     移动次数:614464  比较次数:316032
  快速排序:
    顺序数据    耗时:2.659s     移动次数:0       比较次数:800059998
  插入排序:
    逆序数据    耗时:6.838s     移动次数:799980000       比较次数:800019999
  归并排序:
    逆序数据    耗时:0.011s     移动次数:614464  比较次数:298432
  快速排序:
    逆序数据    耗时:2.721s     移动次数:0       比较次数:800059998
  插入排序:
    无序数据    耗时:3.464s     移动次数:398968600       比较次数:399008599
  归并排序:
    无序数据    耗时:0.021s     移动次数:614464  比较次数:561691
  快速排序:
    无序数据    耗时:0.012s     移动次数:240280  比较次数:1023650


----------------------比较完成------------------------
```

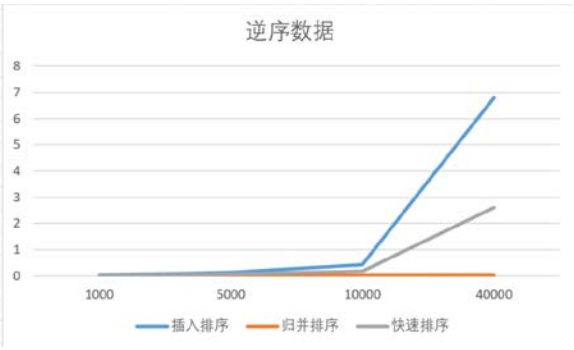
五、三种方法的总结

1、从算法时间复杂度角度考虑，选择排序为O(n^2)，快速排序和归并排序则为O(nlogn)。

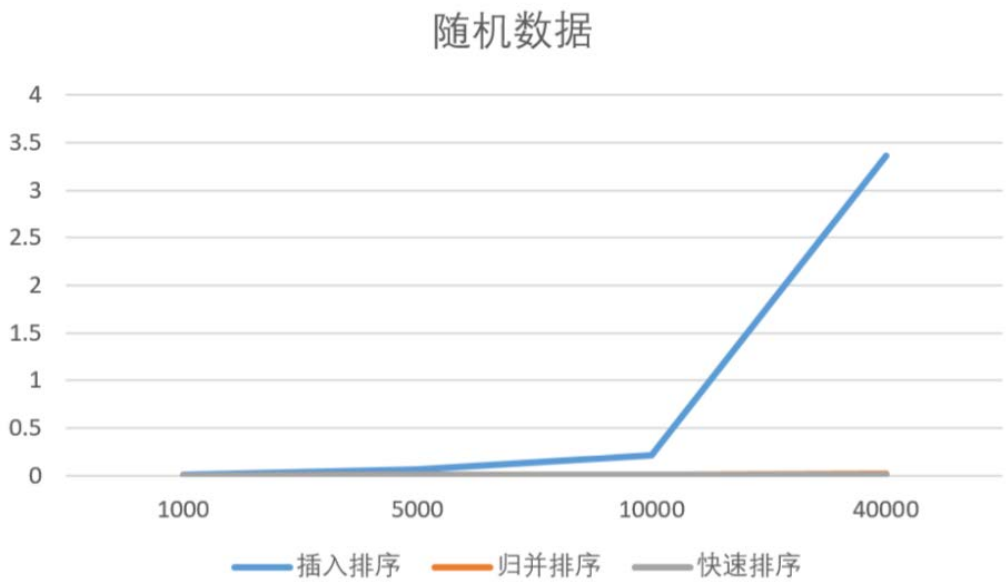
2、在测试中，一旦数据集是已经排好序的，快速排序的耗时会接近于O(n^2)的插入排序

然而归并排序则完全不受影响。这是因为快速排序在面对已经排好序的数组时，分治法并不能起到分治的作用，无法将问题的规模减小，最终导致和选择排序的耗时一样。这也是快速排序最差情况。同时也易于看出对于插入排序来讲，逆序是最差的情况，顺序是最好的情况（参照比较次数）但是归并排序则一直很稳定。





3、但是，当遭遇随机数组的时候，之前快速排序的问题相对减小，随机数的存在，虽然不能使每次分治达到最好，但是效果已经很明显了，快速排序和归并排序的耗时非常接近，插入排序则一如既往的慢。同时，从比较和交换的次数也能看出这个结论（程序会输出这两个结果）



4、从比较和移动次数的统计结果来看，依然会得到相同的结论

移动次数	顺序/插入	顺序/归并	顺序/快排	逆序/插入	逆序/归并	逆序/快排	无序/插入	无序/归并	无序/快排
100	0	672	0	4950	672	0	2324	672	228
1000	0	9976	0	499500	9976	0	250190	9976	3763
5000	0	61808	0	12497500	61808	0	6254774	61808	24378
10000	0	133616	0	49995000	133616	0	24857490	133616	52699
40000	0	614464	0	799980000	614464	0	398968600	614464	240280
比较次数	顺序/插入	顺序/归并	顺序/快排	逆序/插入	逆序/归并	逆序/快排	无序/插入	无序/归并	无序/快排
100	99	356	5148	5049	316	5148	2423	540	938
1000	999	5044	501498	500499	4932	501498	251189	8699	15176
5000	4999	32004	12507498	12502499	29804	12507498	6259773	55167	96563
10000	9999	69008	50014998	50004999	64608	50014998	24867489	120366	214998
40000	39999	316032	800059998	800019999	298432	800059998	399008599	561691	1023650

六、遇到的问题和改进

1、数据集的设计应该再丰富一些，比如数据全都为一个整数，这样的话，或许会得到新的结论

2、虽然整体的趋势没有问题，但在移动和比较数据的次数统计方面仍然有偏差。

3、按理来讲，快速排序应该比归并排序更快，但是实际上并不是这样。我认为原因有两个

（1）编译器对归并有优化，归并排序收益更大。

（2）快速排序不能保证每次都能以最好的情况进行分治。

4、根据 3，说明快速排序有很大的改进空间，比如不能总用第一个元素当成标杆，应该尽量选择中位数。同时，快速排序在遭遇大数据量的时候会爆栈，所以还要手动消除递归。