

姓名：吴立凡

学号：PB19061323

1. 编程实现快速排序

算法思想：

快速排序算法：快速排序算法是一个分治的算法，通过设定一个分界值，将输入数据分成两部分，前一部分小于分界值，后一部分大于分界值，不断递归，来进行排序。

此时时间复杂度公式为： $T[n] = 2T[n/2] + f(n)$ ， $f(n)$ 为将数组分为两部分的时间代价。

在这种算法下， $f(n)=n$ ，从而可以推导出整个算法的复杂度为 $O(n\log n)$ ，是一种很优秀的算法。

快速排序具体实现：

```
void quick_sort(int array[], int x, int y)
{
    int i = x + 1, j = y, temp;
    int key = array[x]; //基准选择为数组的第一个元素
    if (x >= y)
        return; //当数组只有一个元素的时候，直接返回
    while (true) //对整个数组按照第一个元素进行 partition，将小于 x 的数放在最后返回的 j
    的左边，大于 x 的数放在最后返回的 j 的右边
    {
        for (; array[i] <= key && i < y; i++);
        for (; array[j] > key && j > x; j--);
        if (i >= j)
            break;
        temp = array[i]; //交换左边大于 x 的元素和右边小于 x 的元素
        array[i] = array[j];
        array[j] = temp;
    }
    temp = array[j]; //将第一个指标元素与第 j 个元素交换，此时，第 j 个元素左边的元素均
    小于 j，右边的元素均大于 j
    array[j] = array[x];
    array[x] = temp;

    quick_sort(array, x, j - 1); //递归对左侧和右侧进行排序
    quick_sort(array, j + 1, y);
}
```

第二部分：

快速排序的优化。

算法思想：

- 1) 基准的选择：快速排序的运行时间与划分是否对称有关。最坏情况下，每次划分过程产生两个区域分别包含 $n-1$ 个元素和 1 个元素，其时间复杂度会达到 $O(n^2)$ 。在最好的情况下，每次划分所取的基准都恰好是中值，即每次划分都产生两个大小为 $n/2$ 的区域。此时，快排的时间复杂度为 $O(n\log n)$ 。

2) 所以基准的选择对快排而言至关重要。快排中基准的选择方式主要有以下三种:

- ① 固定基准; (见 quicksort_base1.cpp)
- ② 随机基准; (见 quicksort_baserandom.cpp)
- ③ 三数取中。(见 quicksort_base3num.cpp)

随即基准的代码与固定基准类似, 只有下面这一处选择基准的地方不同:

```
int i = x + 1, j = y, temp;
    if (x >= y)
        return;
    int keypos=x+rand()%(y-x);//随机选取 x 到 y 之间的数作为基准, 将其与第一个
元素交换, 后面就可以复用前面的代码
    int key = array[keypos];
    temp=array[x];
    array[x]=array[keypos];
    array[keypos]=temp;
```

三数取中的代码与固定基准类似, 只有下面这一处选择基准的地方不同:

```
int i = x + 1, j = y, temp;
    if (x >= y)
        return;
    int a, b, c;
    int keypos =0;//选择开头、中间、结尾三个数, 找出中位数, 将其与第一个元
素交换, 后面的就可以复用第一次实验的代码
    a =x;
    b =y-1;
    c =(x+y-1)/2;
    if(array[a]>array[b])
    {
        if(array[b]>array[c])keypos=b;
        else keypos=c;
    }
    else
    {
        if(array[a]>array[c])keypos=a;
        else keypos=c;
    }

    int key = array[keypos];
    temp = array[x];
    array[x] = array[keypos];
    array[keypos] = temp;
```

3) 当输入数据已经“几乎有序”时, 使用插入排序速度很快。我们可以利用这一特点来提高快速排序的速度。当对一个长度小于 k 的子数组调用快速排序时, 让她不做任何排序就

返回。上层的快速排序调用返回后，对整个数组运行插入排序来完成排序过程。
与第一次实验的代码类似，只是要加入以下代码：（见 quicksort_optimize.cpp）

```
void insertsort(int a[],int n)//插入排序算法
{
    for(int i= 1;i<n;i++){
        if(a[i] < a[i-1]){
            int j= i-1;
            int x = a[i];
            while(j>-1 && x < a[j]){
                a[j+1] = a[j];
                j--;
            }
            a[j+1] = x;
        }
    }
}

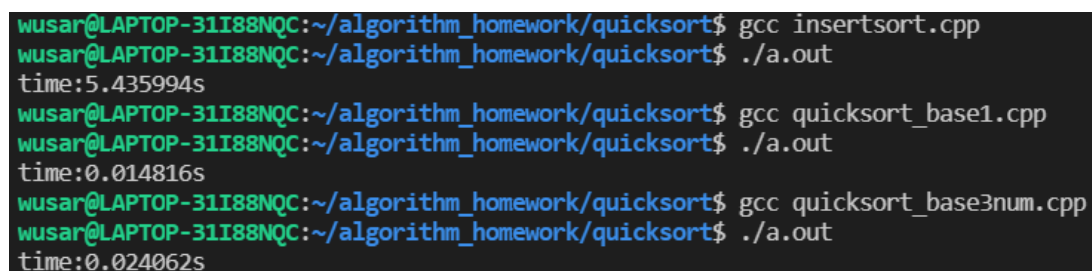
void quick_sort(int array[], int x, int y)
{
    int i = x + 1, j = y, temp;
    int key = array[x];
    if (y-x<=K-1) //在数组长度小于 k 时， 直接进行插入排序并返回
    {
        insertsort(&array[x],y-x);
        return;
    }
}
```

3、实验结果分析。

我们对各种代码进行算法性能分析，为了更好的效果，我使用如下的 python 代码产生一个大小为 100000 的数据集（见 randomgenerate.py）

```
import random
max_num=100000
with open("data.txt","wb") as datafile:
    datafile.write(str(max_num)+'\n')
    for i in range(max_num):
        datafile.write(str(random.randint(0,max_num))+ ' ')
```

使用不同的算法对其经行排序，排序过程如图：



```
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc insertsort.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:5.435994s
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc quicksort_base1.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:0.014816s
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc quicksort_base3num.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:0.024062s
```

可以看出，单纯使用 insertsort 需要 5.4s 的时间，而 quicksort 只需要 0.01 秒的时间。
但是由于 quicksort 的时间太短，必须加大数据量才能看出后续算法的性能差别。
将数据集的大小增加到 10000000
对后面的四个算法分别测试，结果如下：

```
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc quicksort_base1.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:1.545634s
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc quicksort_baserandom.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:1.721038s
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc quicksort_base3num.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:1.572523s
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ gcc quicksort_optimize.cpp
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ ./a.out
time:1.534522s
wusar@LAPTOP-31I88NQC:~/algorithm_homework/quicksort$ []
```

可以看出，这几种算法的性能其实都差不多。我们采用多次测量取平均值的办法，依次统计每种算法的平均时间（同时统计了 k 不同情况下使用插入排序优化的运行）：

次数	以第一个为基准	以随机数为基准	以第三个数为基准	使用插入排序优化 (K=3)	使用插入排序优化 (K=4)	使用插入排序优化 (K=5)
1	1.549035	1.756704	1.693124	1.566108	1.540424	1.524596
2	1.559388	1.751689	1.632973	1.584859	1.572083	1.529193
3	1.600865	1.764985	1.616806	1.563210	1.568559	1.515932
4	1.556182	1.743360	1.699316	1.614260	1.601459	1.516749
5	1.552772	1.743218	1.593781	1.573180	1.659372	1.530078
6	1.545218	1.736130	1.602695	1.594950	1.586976	1.524426
7	1.549760	1.780189	1.598304	1.604109	1.663587	1.528368
平均值	1.559031	1.753754	1.633857	1.585811	1.598923	1.524192

经过比较，最后使用插入排序优化（K=5）的算法胜出，以 1.524192s 的时间成为了最快的一个算法。
最慢的是以随机数为基准的算法，这是因为随机数生成本身开销很大。