三种快速排序以及快速排序的优化

1、快速排序的基本思想：

快速排序使用分治的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小。之后分别对这两部分记录继续进行排序，以达到整个序列有序的目的。

2、快速排序的三个步骤：

(1)选择基准：在待排序列中，按照某种方式挑出一个元素，作为 "基准"（pivot）

(2)分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准左边的元素都比该基准小，在基准右边的元素都比基准大

(3)递归地对两个序列进行快速排序，直到序列为空或者只有一个元素。

3、选择基准的方式

对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效率会达到最大。也就是说，基准的选择是很重要的。选择基准的方式决定了两个分割后两个子序列的长度，进而对整个算法的效率产生决定性影响。

最理想的方法是，选择的基准恰好能把待排序序列分成两个等长的子序列

我们介绍三种选择基准的方法

方法(1)：固定位置

思想：取序列的第一个或最后一个元素作为基准

基本的快速排序

```
1. int SelectPivot(int arr[],int low,int high)
2. {
3.     return arr[low];//选择选取序列的第一个元素作为基准
4. }
```

注意：基本的快速排序选取第一个或最后一个元素作为基准。但是，这是一直很不好的处理方法。

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms

测试数据分析：如果输入序列是随机的，处理时间可以接受的。如果数组已经有序时，此时的分割就是一个非常不好的分割。因为每次划分只能使待排序序列减一，此时为最坏情况，快速排序沦为起泡排序，时间复杂度为Θ(n^2)。而且，输入的数据是有序或部分有序的情况是相当常见的。因此，使用第一个元素作为枢纽元是非常糟糕的，为了避免这个情况，就引入了下面两个获取基准的方法。

方法(2)：随机选取基准

引入的原因：在待排序列是部分有序时，固定选取枢轴使快排效率底下，要缓解这种情况，就引入了随机选取枢轴

思想：取待排序列中任意一个元素作为基准

随机化算法

```
/*随机选择枢轴的位置，区间在low和high之间*/
int SelectPivotRandom(int arr[],int low,int high)
{
    //产生枢轴的位置
    srand((unsigned)time(NULL));
    int pivotPos = rand()%(high - low) + low;

    //把枢轴位置的元素和low位置元素互换，此时可以和普通的快排一样调用划分函数
    swap(arr[pivotPos],arr[low]);
    return arr[low];
}
```

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms

测试数据分析：:这是一种相对安全的策略。由于枢轴的位置是随机的，那么产生的分割也不会总是会出现劣质的分割。在整个数组数字全相等时，仍然是最坏情况，时间复杂度是O(n^2)。实际上，随机化快速排序得到理论最坏情况的可能性仅为1/(2^n)。所以随机化快速排序可以对于绝大多数输入数据达到O(nlogn)的期望时间复杂度。一位前辈做出了一个精辟的总结：“随机化快速排序可以满足一个人一辈子的人品需求。”

方法(3)：三数取中（median-of-three）

引入的原因：虽然随机选取枢轴时，减少出现不好分割的几率，但是还是最坏情况下还是O(n^2)，要缓解这种情况，就引入了三数取中选取枢轴

分析：最佳的划分是将待排序的序列分成等长的子序列，最佳的状态我们可以使用序列的中间的值，也就是第N/2个数。可是，这很难算出来，并且会明显减慢快速排序的速度。这样的中值的估计可以通过随机选取三个元素并用它们的中值作为枢纽元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。显然使用三数中值分割法消除了预排序输入的不好情形，并且减少快排大约14%的比较次数

举例：待排序序列为：8 1 4 9 6 3 5 2 7 0

我们这里取三个数排序后，中间那个数作为枢轴，则枢轴为6

注意：在选取中轴值时，可以从由左中右三个中选取扩大到五个元素中或者更多元素中选取，一般的，会有（2t＋1）平均分区法（median-of-(2t+1)，三平均分区法英文为median-of-three）。

具体思想：对待排序序列中low、mid、high三个位置上数据进行排序，取他们中间的那个数据作为枢轴，并用0下标元素存储枢轴。

即：采用三数取中，并用0下标元素存储枢轴。

```
/*函数作用：取待排序序列中low、mid、high三个位置上数据，选取他们中间的那个数据作为枢轴*/
int SelectPivotMedianOfThree(int arr[],int low,int high)
{
    int mid = low + ((high - low) >> 1);//计算数组中间的元素的下标
    //使用三数取中法选择枢轴
    if (arr[mid] > arr[high])//目标: arr[mid] <= arr[high]
    {
            swap(arr[mid],arr[high]);
    }
    if (arr[low] > arr[high])//目标: arr[low] <= arr[high]
    {
            swap(arr[low],arr[high]);
    }
    if (arr[mid] > arr[low]) //目标: arr[low] >= arr[mid]
    {
            swap(arr[mid],arr[low]);
    }
    //此时，arr[mid] <= arr[low] <= arr[high]
    return arr[low];
    //low的位置上保存这三个位置中间的值
    //分割时可以直接使用low位置的元素作为枢轴，而不用改变分割函数了
}
```

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms

测试数据分析：使用**三数取中选择枢轴**优势还是很明显的，但是还是处理不了重复数组

优化1、当待排序序列的长度分割到一定大小后，使用插入排序。

原因：对于很小和部分有序的数组，快排不如插排好。当待排序序列的长度分割到一定大小后，继续分割的效率比插入排序要差，此时可以使用插排而不是快排

截止范围：待排序序列长度N = 10，虽然在5~20之间任一截止范围都有可能产生类似的结果，这种做法也避免了一些有害的退化情形。摘自《数据结构与算法分析》Mark Allen Weiness 著

```
if (high - low + 1 < 10)
{
    InsertSort(arr,low,high);
    return;
}//else时，正常执行快排
```

测试数据：

随机生成 1 百万个数字，进行排序
重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	125ms	63ms	250 ms	699516 ms

测试数据分析：针对随机数组，使用**三数取中选择枢轴+插排，**效率还是可以提高一点，真是针对已排序的数组，是没有任何用处的。因为待排序序列是已经有序的，那么每次划分只能使待排序序列减一。此时，插排是发挥不了作用的。所以这里看不到时间的减少。另外，三数取中选择枢轴+插排还是不能处理重复数组

优化2、在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割

举例：

待排序序列 1 4 6 7 6 6 7 6 8 6

三数取中选取枢轴：下标为4的数6

转换后，待分割序列：6 4 6 7 1 6 7 6 8 6

枢轴key：6

本次划分后，未对与key元素相等处理的结果：1 4 6 6 7 6 7 6 8 6

下次的两个子序列为：1 4 6 和 7 6 7 6 8 6

本次划分后，对与key元素相等处理的结果：1 4 6 6 6 6 6 7 8 7

下次的两个子序列为：1 4 和 7 8 7

经过对比，我们可以看出，在一次划分后，把与key相等的元素聚在一起，能减少迭代次数，效率会提高不少

具体过程：在处理过程中，会有两个步骤

第一步，在划分过程中，把与key相等元素放入数组的两端
第二步，划分结束后，把与key相等的元素移到枢轴周围
举例：
待排序序列 1 4 6 7 6 6 7 6 8 6
三数取中选取枢轴：下标为4的数6
转换后，待分割序列：6 4 6 7 1 6 7 6 8 6
枢轴key：6
第一步，在划分过程中，把与key相等元素放入数组的两端
结果为：6 4 1 6(枢轴) 7 8 7 6 6 6
此时，与6相等的元素全放入在两端了
第二步，划分结束后，把与key相等的元素移到枢轴周围
结果为：1 4 66(枢轴) 6 6 6 7 8 7
此时，与6相等的元素全移到枢轴周围了
之后，在1 4 和 7 8 7两个子序列进行快排
代码

```
void QSort(int arr[],int low,int high)
{
    int first = low;
    int last = high;

    int left = low;
    int right = high;

    int leftLen = 0;
    int rightLen = 0;

    if (high - low + 1 < 10)
        InsertSort(arr,low,high);
        return;
    }

    int key = SelectPivotMedianOfThree(arr,low,high);//使用三数取中法选择枢轴

    while(low < high)
    {
        while(high > low && arr[high] >= key)
        {
```

```
            if (arr[high] == key)//处理相等元素
            {
                right--;
                rightLen++;
            }
            high--;
        }
        arr[low] = arr[high];
        while(high > low && arr[low] <= key)
        {
            if (arr[low] == key)
            {
                swap(arr[left],arr[low]);
                left++;
                leftLen++;
            }
            low++;
        }
        arr[high] = arr[low];
    }
    arr[low] = key;

    //一次快排结束
    //把与枢轴key相同的元素移到枢轴最终位置周围
    int i = low - 1;
    int j = first;
    while(j < left && arr[i] != key)
    {
        i--;
        j++;
    }
    i = low + 1;
    j = last;
    while(j > right && arr[i] != key)
    {
        swap(arr[i],arr[j]);
        i++;
    }
```

```
    QSort(arr,first,low - 1 - leftLen);
    QSort(arr,low + 1 + rightLen,last);
}
```

测试数据：

随机生成 1 百万个数字，进行排序
重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	125ms	63ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms

测试数据分析：**三数取中选择枢轴+插排+聚集相等元素**的组合，效果竟然好的出奇。
原因：在数组中，如果有相等的元素，那么就可以减少不少冗余的划分。这点在重复数组中体现特别明显啊。
其实这里，插排的作用还是不怎么大的。

优化3：优化递归操作

快排函数在函数尾部有两次递归操作，我们可以对其使用尾递归优化
优点：如果待排序的序列划分极端不平衡，递归的深度将趋近于n，而栈的大小是很有限的，每次递归调用都会耗费一定的栈空间，函数的参数越多，每次递归耗费的空间也越多。优化后，可以缩减堆栈深度，由原来的O(n)缩减为O(logn)，将会提高性能。
代码：

```
void QSort(int arr[],int low,int high)
{
    int pivotPos = -1;
    if (high - low + 1 < 10)
    {
        InsertSort(arr,low,high);
        return;
    }
    while(low < high)
    {
        pivotPos = Partition(arr,low,high);
        QSort(arr,low,pivot-1);
        low = pivot + 1;
    }
}
```

注意：在第一次递归后，low就没用了，此时第二次递归可以使用循环代替

测试数据：

三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms
三数取中+插排+聚集相等元素+尾递归	110ms	32ms	32 ms	10 ms

测试数据分析：其实这种优化编译器会自己优化，相比不使用优化的方法，时间几乎没有减少

优化4：使用并行或多线程处理子序列（略）

所有的数据测试：

随机生成 1 百万个数字，进行升序排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	133ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	131ms	63ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms
三数取中+插排+聚集相等元素+尾递归	110ms	32ms	32 ms	10 ms
STL 中的 Sort 函数	**125ms**	**27ms**	**31ms**	**8ms**

概括：这里效率最好的快排组合 是：三数取中+插排+聚集相等元素,它和STL中的Sort函数效率差不多

注意：由于测试数据不稳定，数据也仅仅反应大概的情况。如果时间上没有成倍的增加或减少，仅仅有小额变化的话，我们可以看成时间差不多。

参考文献

http://blog.sina.com.cn/s/blog_5a3744350100jnec.html

http://www.blogjava.net/killme2008/archive/2010/09/08/331404.html

http://www.cnblogs.com/cj723/archive/2011/04/27/2029993.html

http://blog.csdn.net/zuiaituantuan/article/details/5978009