C++快速排序

例：程序huawei\_test1

反馈结果：不是纯粹的快速排序

例：程序huawei\_test2

// 分区

int Partition(*vector*<int>& array, int left, int right)

{

int pivot = array[left];

int pivot\_position = left;

for (int i = left + 1; i <= right; ++i)

{

// 如果array[i]<pivot，则首先++pivot\_position

// pivot\_position初始在最左边(默认没有元素小于pivot)，当array[i]<pivot时

// 由于pivot\_position需要满足:pivot\_position左边元素比pivot小，右边比pivot大

// 所以++pivot\_position

if (array[i] < pivot)

{

++pivot\_position;

// ++pivot\_position之后，如果i!=pivot\_position，则array[pivot\_position]>=pivot

// 否则i肯定等于pivot\_position

// 交换array[i]和array[pivot\_position]，这样是为了将所有小于pivot的元素连续放置

// 在循环结束后，只需交换array[left]和array[pivot\_position]，pivot\_position左边的元素

// 肯定小于pivot，右边的大于等于pivot

if (i != pivot\_position)

{

*swap*(array[i], array[pivot\_position]);

}

}

}

if (left != pivot\_position)

{

// 执行一次交换，枢纽元pivot原始在left位置

// 根据上面for的比较规则，array[pivot\_position]肯定小于pivot

// 执行交换后，array[pivot\_position]就是pivot

*swap*(array[left], array[pivot\_position]);

}

return pivot\_position;

}

void QuickSort(*vector*<int>& array, int left, int right)

{

if (left >= right)

{

return;

}

// 找出一个分隔位置，left-pivot\_position-1中的

// 元素肯定比array[pivot\_position]小，pivot\_position+1-right

// 肯定比array[pivot\_position]大

int pivot\_position = Partition(array, left, right);

QuickSort(array, left, pivot\_position - 1);

QuickSort(array, pivot\_position + 1, right);

}

模板实现快速排序

例：程序huawei\_test3

template<typename T, typename Comparator>

int Partition(*vector*<T>& array, int left, int right, Comparator compare)

{

T pivot = array[left];

int pivot\_position = left;

for (int i = left + 1; i <= right; ++i)

{

if (compare(array[i], pivot))

{

++pivot\_position;

// ++pivot\_position之后，如果i!=pivot\_position，则array[pivot\_position]>=pivot

// 否则i肯定等于pivot\_position

// 交换array[i]和array[pivot\_position]，这样是为了将所有小于pivot的元素连续放置

// 在循环结束后，只需交换array[left]和array[pivot\_position]，pivot\_position左边的元素

// 肯定小于pivot，右边的大于等于pivot

if (i != pivot\_position)

{

*swap*(array[i], array[pivot\_position]);

}

}

}

if (left != pivot\_position)

{

// 执行一次交换，枢纽元pivot原始在left位置

// 根据上面for的比较规则，array[pivot\_position]肯定小于pivot

// 执行交换后，array[pivot\_position]就是pivot

*swap*(array[left], array[pivot\_position]);

}

return pivot\_position;

}

// 参考std::sort的实现方式

template<typename T, typename Comparator>

void QuickSort(*vector*<T>& array, int left, int right, Comparator compare)

{

if (left >= right)

{

return;

}

int pivot\_position = Partition(array, left, right, compare);

QuickSort(array, left, pivot\_position - 1, compare);

QuickSort(array, pivot\_position + 1, right, compare);

}

// 比较函数是默认的

template<typename T>

void QuickSort(*vector*<T>& array, int left, int right)

{

QuickSort(array, left, right, *less*<T>());

}

class Test

{

public:

Test()

{

number\_ = 0;

}

Test(int number):number\_(number)

{}

bool operator()(const Test& lhs, const Test& rhs) const

{

return lhs.number\_ < rhs.number\_;

}

public:

int number\_;

};

int main(int argc, char\* argv[])

{

*vector*<int> array1{ 1, 6, 7, 8, 2, 9, 10, 12, 23, 5, 89, 24, 25, 26, 27, 34, 56 };

*vector*<int> array{ 100, 8, 4, 12, 13, 1, 6, 7, 8, 2, 9, 10, 12, 23, 5, 89, 24, 25, 26, 27, 34, 56 };

*cout* << "原始数组：" << *endl*;

for (int i = 0; i < array.*size*(); ++i)

{

*cout* << array[i] << ",";

}

*cout* << *endl*;

QuickSort(array, 0, array.*size*() - 1);

*cout* << "排序后的数组：" << *endl*;

for (int i = 0; i < array.*size*(); ++i)

{

*cout* << array[i] << ",";

}

*cout* << *endl*;

*cout* << *endl*;

// 排序对象为类对象

*vector*<Test> test\_vec;

test\_vec.*push\_back*(Test(1));

test\_vec.*push\_back*(Test(8));

test\_vec.*push\_back*(Test(4));

test\_vec.*push\_back*(Test(12));

test\_vec.*push\_back*(Test(2));

test\_vec.*push\_back*(Test(29));

*cout* << "原始数组：" << *endl*;

for (int i = 0; i < test\_vec.*size*(); ++i)

{

*cout* << test\_vec[i].number\_ << ",";

}

*cout* << *endl*;

// 自定义的排序规则

QuickSort(test\_vec, 0, test\_vec.*size*() - 1, Test());

*cout* << "排序后的数组：" << *endl*;

for (int i = 0; i < test\_vec.*size*(); ++i)

{

*cout* << test\_vec[i].number\_ << ",";

}

return 0;

}

输出为：

原始数组：

100,8,4,12,13,1,6,7,8,2,9,10,12,23,5,89,24,25,26,27,34,56,

排序后的数组：

1,2,4,5,6,7,8,8,9,10,12,12,13,23,24,25,26,27,34,56,89,100,

原始数组：

1,8,4,12,2,29,

排序后的数组：

1,2,4,8,12,29,

快速排序的时间复杂度和空间复杂度

平均时间复杂度为：

最坏时间复杂度为：

空间复杂度：使用就地排序的快速排序的空间复杂度为，但由于有递归调用：

平均空间复杂度为：

最坏空间复杂度为：