Lab 4

1 实验目的

1. 掌握快速排序随机算法的设计思想与方法，
2. 熟练使用高级编程语言实现不同的快速排序算法，
3. 利用实验测试给出不同快速排序算法的性能以理解其优缺点。

2 实验问题

快速排序。

3 实验步骤

3.1 《算法导论》伪代码实现

```
void QuickSort(vector<int>& nums, int l, int r) {
    if (l < r) {
        int p = Partition(nums, l, r);
        QuickSort(nums, l, p - 1);
        QuickSort(nums, p + 1, r);
    }
}

int Partition(vector<int>& nums, int l, int r) {
    uniform_int_distribution<int> unif(l, r);

    int t = unif(rd);
    int pivot = nums[t];

    swap(nums[t], nums[l]);
    int p = l, q = r;
    while (p < q) {
        while (p < q && pivot < nums[q]) {
            --q;
        }
        swap(nums[p], nums[q]);
        while (p < q && nums[p] <= pivot) {
            ++p;
        }
        swap(nums[p], nums[q]);
    }
    nums[p] = pivot;
    return p;
}
```

考虑极端情况，面对1e6个重复数据，此算法每次选择pivot后，所有数据都会排在pivot的左侧，这样就达到了最坏时间复杂度$O(n^2)$，因为数据量过大会导致栈溢出。

3.2 优化

由3.1分析可知，每次划分都是非常不均衡，所以考虑在划分上做优化。具体的思想是用随机打败随机，可以把比较函数随机化，进而避免比较后导致所有数据排在pivot一侧。

```
// Bernoulli分布
random_device rd;
mt19937 gen(rd());
bernoulli_distribution d(0.5);

bool Compare(int a, int b) { return (d(rd)) ? a < b : a <= b; }

void QuickSort(vector<int>& nums, int l, int r) {
    if (l < r) {
        int p = Partition(nums, l, r);
        QuickSort(nums, l, p - 1);
        QuickSort(nums, p + 1, r);
    }
}

int Partition(vector<int>& nums, int l, int r) {
    uniform_int_distribution<int> unif(l, r);

    int t = unif(rd);
    int pivot = nums[t];
    swap(nums[t], nums[l]);

    int p = l, q = r;
    while (p < q) {
        while (p < q && Compare(pivot, nums[q])) {
            --q;
        }
        swap(nums[p], nums[q]);
        while (p < q && Compare(nums[p], pivot)) {
            ++p;
        }
        swap(nums[p], nums[q]);
    }
    nums[p] = pivot;
    return p;
}
```

性能测试

与c++ stl标准库函数sort一起针对数据规模为1e6对不同数据分布（重复百分比）进行测试，测试结果如下。虽然经过了简单的优化使得优化后的代码可以在有限时间内给出结果，但是与标准库封装的sort函数还是有较大差距。