#3: QuickSort

При реализации алгоритмов в виде библиотечных функций одним из требований к реализации является универсальность, чтобы однажды реализованный алгоритм можно было применять при необходимости к разным данным. Универсальность достигается через параметры, с их помощью можно настроить однажды реализованный алгоритм под конкретную задачу. В случае с алгоритмами сортировки такой настройкой является указание сортируемого массива и правила упорядочивания его элементов. Однако, в языках со статической типизацией (к каким относится и C++) возникает проблема, что при записи параметров функции требуется указать также их тип, в случае же универсального алгоритма хотелось бы иметь реализацию, не зависящую от типа сортируемых данных, и применять одну и ту же реализацию ко всем типам данных. Поэтому, большинство библиотек реализует алгоритмы в обобщенном (generic) виде. Механизмом обобщенного программирования в C++ являются шаблоны (templates). Общая идея заключается в том, что вместо использования конкретных типов данных при описании функции используются шаблонные параметры, вводимые при помощи специального синтаксиса.

Для функции сортировки сортируемый массив может передаваться при помощи указателей на начало и конец сортируемого интервала (или, например, как указатель на начало интервала и количество элементов); а правило сортировки - как указатель на функцию (function pointer), функциональный объект (functional object). Начиная с C++11 для этого также можно применять лямбды (lambda).

Задание

В вашу задачу входит разработка обобщенной функции `sort()`, реализующей алгоритм быстрой сортировки со следующими оптимизациями:

- Выбор опорного элемента, как медианы из первого, среднего и последнего элемента сортируемого интервала;

- Исключение хвостовой рекурсии: функция должна определять длины получившихся интервалов после разбиения и рекурсивно сортировать интервал меньшей длины, итеративно обрабатывая интервал большей длины;
- Использование алгоритма сортировки вставками для коротких интервалов (длину такого интервала необходимо подобрать экспериментально).
- Использовать move-семантику для обмена элементов в процессе разбиения и при сортировке вставками.

Функция должна принимать три аргумента: начало `first` и конец `last` сортируемого интервала и предикат сортировки `comp`, который принимает два аргумента – элементы массива `a` и `b`, и возвращает `true`, если `a < b`, то есть в отсортированном массиве элемент `a` должен располагаться *перед* `b`.

```
template<typename T, typename Compare>
void sort(T *first, T *last, Compare comp)
```

где предикат упорядочивания должен удовлетворять прототипу

```
template<typename T>
bool comp(const T &a, const T &b)
```

Сортировка массива из 100 целых чисел при помощи разработанной функции будет выглядеть следующим образом:

```
int a[100];
sort(a, a + 100, [](int a, int b) { return a < b; });
```

Для разработанной функции необходимо создать набор модульных тестов (например, Google Test или CppUnit). Тесты должны быть репрезентативными, покрывать общий и максимальное количество граничных случаев. Тесты должны покрывать случаи сортировки массивов из элементов как примитивных, так и нетривиальных типов.

(*) Можно ли использовать такую функцию `sort()` для сортировки динамического массива из работы 2? Модифицируйте функцию сортировки/итератор динамического массива, чтобы обеспечить такую возможность.