Задание 1.

Так как данные не превосходят 1000, то можно использовать сортировку подсчетом, которая работает за N. После этого останется взять только нужный элемент в отсортированном массиве.

Таким образом весь алгоритм будет работать за N при любых данных.

```
int count_stat(std::vector<int> & array, int k){
    int maxe = array[0];
    for(int i = 0; i < array.size(); i++) if(array[i] > maxe) maxe = array[i];
// максимальный элемент
    std::vector<int> counts(maxe+1, 0); // массив с подсчетами
    for(int i = 0; i < array.size(); i++) counts[array[i]]++; // подсчитываем
    int c = 0;
    for(int i = 0; i < counts.size(); i++) {
        for(int j = 0; j < counts[i]; j++) { // восстанавливаем массив
            array[c] = i;
            if(c == k) {
                return array[c];
            }
            c++;
        }
    }
}
```

Задание 2.

Как было показано в лекции, медиана медиан всегда работает за N. Таким образом, если бы заменим в быстрой сортировке выбор случайного элемента на выбор медианы, то тогда мы добавим лишь N операций на каждом уровне сортировки, а значит итоговое время работы все еще будет N*logN, но теперь гарантированно, так как мы всегда делим попалам.

```
template<class Iter>
Iter partition(Iter begin, Iter end, int t) { // функция для разбиения массива
на нужные нам подмассивы
    while(begin != end) { // идем с двух сторон, пока не сойдемся
        while( *begin < t ) { // пропускаем все уже корректно стоящие элементы
            ++begin;
            if(begin == end) return begin; // если сошлись
        }
        do { // симметрочно двигаемся с конца
            --end;
            if(begin == end) return begin;
        } while(*end > t);
        int temp = *begin; // меняем местами неправильно стоящие элементы
        *begin = *end;
        *end = temp;
        ++begin;
    }
    return begin; // возвращаем на место опорного элемента t в этом массиве (по
сути элемент разделения массива)
}

template <class Iter>
int median_of_medians(Iter begin, Iter end, int k) {
    if(end - begin <= 10) { // если осталось мало элементов, то просто
отсортируем и вернем k-ый элемент
        std::sort(begin, end);
        return *(begin + k);
```

```
    }
    std::vector<int> medians; // массив из медиан
    Iter prev = begin;
    Iter next = prev + 5; // делим по 5 элементов
    while(next < end) {
        medians.push_back(median_of_medians(prev, next, 2)); // добавляем
медиану 5-ти элементного множества
        prev += 5;
        next += 5;
    }
    if(next != end) { // если остались еще элементы, то дописываем их
        std::copy(prev, end, std::back_inserter(medians));
    }
    int M = median_of_medians(medians.begin(), medians.end(),
int(medians.size())/2); // находим медиану медиан
    // далее идет точь-в-точь предыдущий алгоритм
    Iter middle = std::partition(begin, end, [M](int s){return s < M;});
    std::partition(middle, end, [M](int s) {return s == M;});
    int left_size = middle - begin;
    if(left_size == k) { // нашли нашу статистику
        return M;
    } else if(left_size > k) {
        return median_of_medians(begin, middle, k);
    } else {
        return median_of_medians(middle+1, end, k-left_size-1);
    }
}
template<class Iter>
void quick_sort(Iter begin, Iter end) {
    if(end - begin <= 1) return ; // один элемент уже отсортирован
    int distance = end – begin;

    std::vector<int> temp;
    std::copy(begin, end, std::back_inserter(temp)); // если изначально не
скопировать массив, то тогда медиана медиан его изменит, а мы не хотим этого.
    int t = median_of_medians(temp.begin(), temp.end(), temp.size()/2); //
Будем выбирать медиану, а не случайный элемент

    Iter middle = partition(begin, end, t); // разбиваем массив
    quick_sort(begin, middle); // сортируем левую часть
    quick_sort(middle, end); // сортируем правую часть
}
```

Хоть мы и добились стабильной работы за N*logN, нужно понимать, что дополнительные затраты на поиск медианты неизбежны. Это означает, что данный алгоритм лучше использовать на больших массивах, а на маленьких гораздо лучше использовать стандартный алгоритм быстрой сортировки.