Задача 1.

Добавим флаг об отсортированности массива. Вначале он будет равен true. После этого, во время прохождения по всему массиву если нам необходимо будет сделать перестановку элементов, то тогда обнуляем флаг. Если к концу итерации флаг останется равным true, это значит, что нам не пришлось делать перестановки, а значит массив уже отсортирван.

```
void bubble_sort(std::vector<int> & array) {
    for(int i = array.size()-1; i>= 0; i--) { // N внешних итераций
        bool sorted = true;
        for(int j = 0; j < i+1; j++) { // проход по всему массиву и
соответствующий обмен
            if(array[j] > array[j+1]) { // меняем двух соседних, если они стоят
в неправильном порядке
                sorted = false;
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
        if(sorted){
            break;
        }
    }
}
```

Задача 2.

Алгоритм уже описан в задании. Осталось его только реализовать.

```
class Tree{
public:
    Tree(int v) { // создание вершины со значением
        value = v;
        left = nullptr;
        right = nullptr;
    }
    void add(int x){ // добавление новой вершины
        if(x < value){
            if(left == nullptr) {
                left = new Tree(x);
            }else {
                left->add(x);
            }
        }else {
            if(right == nullptr) {
                right = new Tree(x);
            } else {
                right->add(x);
            }
        }
    }
    void build(std::vector<int> & array) { // восстановление отсортированного
массива
        if(left != nullptr) {
            left->build(array);
        }
        array.push_back(value);
        if(right != nullptr) {
            right->build(array);
```

```
        }
    }
private:
    int value;
    Tree* left;
    Tree* right;
};

void tree_sort(std::vector<int> & array) {
    Tree * root = new Tree(array[0]);
    for(int i = 1; i < array.size(); i++) root->add(array[i]);
    array.clear();
    root->build(array);
}
```

Время работы данного алгоритма похоже на быструю сортировку → при хорошем разбиении за N*log(N), при плохом → за N*N. Однако проигрывает ему, так как возникают доп расходы на построение дерева, его хранение в памяти и восстановление массива из дерева.