Сложность сортировки пузырьком:

```
void BubbleSort(int * array, int size) {
    for(int i = 0; i < size - 1; i++) {
        for(int j = 0; j < size - 1 - i; j++) {
            if( array[j] > array[j + 1] ) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

Внешний цикл выполняется n - 1 раз, внутренний – n – 1 – i, где i изменяется от 0 до n – 2.

Суммарное количество выполнений внутреннего цикла: n-1 + n-2 + n-3 + … + 2 + 1.

То есть всего $\frac{(n-1)n}{2} = \frac{n^2-n}{2}$. Таким образом, сложность алгоритма составляет $O(n^2)$.

Сложность сортировки вставками:

```
void InsertSort(int * array, int size) {

    int place;

    for(int item = 1; item < size; item++ ) {
        place = item - 1;
        while(place >= 0) {
            if(array[place] > array[place + 1]) {
                swap(array[place], array[place + 1]);
            }
            else break;
            place--;
        }
    }
}
```

Внешний цикл выполняется n – 1 раз, внутренний в лучшем случае 1 раз, в худшем случае – i, где i изменяется от 1 до n – 1. Суммарное количество выполнений внутреннего цикла в худшем случае: n-1 + n-2 + n-3 + … + 2 + 1.

То есть всего $\frac{(n-1)n}{2} = \frac{n^2-n}{2}$. Таким образом, сложность алгоритма составляет $O(n^2)$.

Сложность сортировки слиянием:

```
void MergeSort(int * array, int size) {
    if(size < 2) return;
    MergeSort(array, size / 2);
    MergeSort(array + (size / 2), size - (size / 2));
    int temp[size];

    int i = 0;
    int counterLeft = 0;
    int counterRight = size / 2;

    for(i = 0; i < size; i++) {
        if(array[counterLeft] < array[counterRight]) {
            temp[i] = array[counterLeft];
            counterLeft++;
            if(counterLeft == (size / 2)) break;
        }
        else {
```

```
                temp[i] = array[counterRight];
                counterRight++;
                if(counterRight == size) break;
            }
        }

        for(counterLeft; counterLeft < size / 2; counterLeft++) {
            i++;
            temp[i] = array[counterLeft];
        }
        for(counterRight; counterRight < size; counterRight++) {
            i++;
            temp[i] = array[counterRight];
        }

        memcpy(array, temp, sizeof(int) * size);
    }
```

На каждой итерации массив делится на 2 части, то есть максимальная глубина рекурсивных вызовов $\log_2(n)$. На каждой итерации происходит процедура слияния двух подмассивов, каждый из которых в процессе оказывается пройден 1 раз, то есть слияние происходит за линейное время. При этом для каждой глубины вложенности i происходит слияние 2^i подмассивов, длина каждого из которых – $n/2^i$, то есть всего для каждой степени вложенности при слиянии по одному разу проходится каждый из n элементов. Таким образом, всего совершается $n\log_2 n$ действий, и сложность алгоритма O(nlogn).

Сложность пирамидальной сортировки:

```
static void downHeap(int * array, int size, int newIndex) {

    int child;
    int newElement = array[newIndex];

    while(newIndex < size / 2) {

        child = newIndex * 2 + 1;
        if( (child + 1 < size) && (array[child + 1] > array[child]) ) {
            child++;
        }
        if(array[child] <= newElement) {
            break;
        }
        array[newIndex] = array[child];
        newIndex = child;
    }
    array[newIndex] = newElement;
}

static void makeHeap(int * array, int size) {
    for(int i = size / 2; i >= 0; i--) downHeap(array, size, i);
}

void HeapSort(int * array, int size) {

    makeHeap(array, size);

    for(int i = size - 1; i > 0; i--) {

        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;
```

```
            downHeap(array, i, 0);
        }
    }
```
Высота пирамиды составляет $k = \log_2 n$, одно просеивание занимает в худшем случае k – m, где m – высота просеиваемого элемента от корня пирамиды. В лучшем случае просеивание выполняется за постоянное время. При создании пирамиды просеивание выполняется n/2 раз, причём размер куча размера m встречается 2^m раз. Всего просеивание в худшем случае займёт $\sum_{m=0}^{k} 2^m(\mathrm{k}-\mathrm{m}) =$

$k\sum_{m=0}^{k} 2^m - \sum_{m=0}^{k} m2^m = k(2^{k+1}-1) - \sum_{m=0}^{k} m2^m$

$\sum_{m=0}^{k} mx^m = \sum_{m=0}^{k}((\mathrm{x}^{m+1})' - \mathrm{x}^m) = \left(\frac{x^{k+2}-1}{x-1}\right)' - \frac{x^{k+1}-1}{x-1} = (k+2)x^{k+1}(x-1) - (x^{k+2}-1) - \frac{x^{k+1}-1}{x-1}$

$\sum_{m=0}^{k} mx^m\Big|_{x=2} = (k+2)2^{k+1} - (2^{k+2}-1) - 2^{k+1} - 1 = 2^{k+1}(k+2-2-1) + 1 - 1 = 2^{k+1}(k-1)$

$\sum_{m=0}^{k} 2^m(\mathrm{k}-\mathrm{m}) = k(2^{k+1}-1) - \sum_{m=0}^{k} m2^m = k(2^{k+1}-1) - 2^{k+1}(k-1) = 2^{k+1}(k-k+1) - \mathrm{k} = 2^{k+1} - \mathrm{k} = 2\cdot 2^{log_2 n} - log_2 n = 2n - log_2 n$, то есть сложность составляет O(n).

При сортировке просеивание производится n-1 раз, причём при каждом следующем просеивании длина массива уменьшается на 1. Всего действий: $\sum_{i=1}^{n-1}\log_2(\mathrm{n}-\mathrm{i}) = \sum_{m=1}^{n-1}\log_2(\mathrm{m}) = \log_2(1) + \log_2(2) + \log_2(3) + \cdots + \log_2(\mathrm{n}-1) = \log_2((\mathrm{n}-1)!)$. При больших n по формуле Стирлинга: $\log_2((\mathrm{n}-1)!) = \frac{\ln((\mathrm{n}-1)!)}{\ln(2)} \approx \frac{(n-1)\ln(n-1)-(n-1)}{\ln(2)}$, то есть сложность сортировки составляет O(nlogn).

Таким образом, сложность алгоритма составляет O(nlogn).

Сложность простого поиска:

```
int findForNotSorted(int * array, int x, int size){
    int i;
    for (i = 0; i < size; i++){
        if(array[i] == x) break;
    }
    return i;
}
```
Цикл проходится n раз, то есть сложность алгоритма составляет O(n).

Сложность бинарного поиска:

```
int findForSorted(int * array, int x, int size, int start, int stop){
    if(start >= stop) return start;
    int center = (int) ( (start + stop) / 2 );
    if(array[center] == x) return center;
    if(array[center] > x){
        stop = center - 1;
    }
    else start = center + 1;
    return findForSorted(array, x, size, start, stop);
}
```
Поиск в худшем случае продолжается до тех пор, пока не останется участок массива длиной 1 элемент. При этом на каждой итерации длина исследуемого участка массива уменьшается примерно в 2 раза. Таким образом, для того чтобы достичь длины участка массива равной 1, его

нужно поделить напополам $\log_2(n)$ раз. Таким образом, рекурсивная функция выполняется примерно $\log_2(n)$ раза и сложность алгоритма – O(log(n)).

Сложность замены элемента с последующей сортировкой:

```
void change(int * array, int index, int newValue, int size) {

    int newPosition = findForSorted(array, newValue, size, 0, size - 1);

    printf("\n%d", newPosition);

    if(index < newPosition) {
        for(int i = index; i < newPosition; i++) {
            array[i] = array[i + 1];
        }
    }
    else {
        for(int i = newPosition; i < index; i++) {
            array[i + 1] = array[i];
        }
    }

    if(array[newPosition] > newValue) array[newPosition - 1] = newValue;
    else array[newPosition] = newValue;
}
```

Сначала выполняется бинарный поиск, имеющий сложность O(logn), а затем производится сдвиг части элементов массива, лежащей между заменяемым элементом и позицией нового элемента в отсортированном массиве, который в худшем случае требует n действий, то есть имеет сложность O(n). Таким образом, сложность всего алгоритма O(n).