Сложность алгоритмов

Количество операций, выполняемых алгоритмом, не играет существенной роли при анализе алгоритмов, а вот скорость роста количества операций, выполняемых алгоритмом при возрастании объемов входных данных (то, с чем мы будем работать) очень важна и называется скоростью роста или сложностью алгоритма. Для описания скорости роста количества операций (той скорости роста функций, зависящих от этой операции) используется аннотация **O** (Оннотация, так и произносится), Омега и Тета.

Обозначение	**Граница**	**Рост**
(Тета) Θ	Нижняя и верхняя границы, точная оценка	Равно
(O - большое) O	Верхняя граница, точная оценка неизвестна	Меньше или равно
(o - малое) o	Верхняя граница, не точная оценка	Меньше
(Омега - большое) Ω	Нижняя граница, точная оценка неизвестна	Больше или равно
(Омега - малое) ω	Нижняя граница, не точная оценка	Больше

Алгоритм	**Эффективность**
o(n)	< n
O(n)	≤ n
Θ(n)	= n
Ω(n)	≥ n
ω(n)	> n

O – не быстрее чем написано.

Θ (Тета) – точно, как написано.

Ω (Омега) – не медленнее чем написано.

O(n) – линейная сложность (цикл for), символическая форма записи (Функция).

O(n^2) – квадратичное время выполнение алгоритма.

O(n^3) – кубическое время выполнения алгоритма.

O(n!) – даже не рассматриваем, время стремится к бесконечности.

Алгоритмы обычно не поддаются точному анализу, тогда прибегают к аппроксимации. Например говорят, что быстродействие некоторого алгоритма характеризуется величиной O(n), где n - количество входных данных. Это значит, что быстродействие алгоритма измеряется величиной, пропорциональной n, возможно, за исключением нескольких небольших значений n. Перейдя к использованию системы обозначенной O(), мы получаем возможность воспользоваться так называемым асимптотическим анализом. Это в общем случае дает нам основания утверждать, скажем, что если n асимптотически приближается к бесконечности, то, например, алгоритм со сложностью O(n) проявит себя лучше, чем алгоритм со сложностью $O(n^2)$.

Основные классы сложности алгоритмов

O(1) - Большинство инструкций алгоритма запускается один или несколько раз, независимо от n, т. е. время выполнения программы постоянно.

O(n) - Время выполнения алгоритма линейно зависит от n. Каждый входной элемент подвергается небольшой обработке.

$O(n^2)$ - Время выполнения алгоритма является квадратичным. Такое время характерно для обработки всех пар элементов, например, в цикле двойного

уровня вложенности. Использовать такие алгоритмы имеет смысл для относительно небольших n.

$O(n^3)$ - Время выполнения алгоритма является кубическим. Алгоритм обрабатывает тройки элементов, например, в цикле тройного уровня вложенности. Использовать такие алгоритмы имеет смысл только для малых задач.

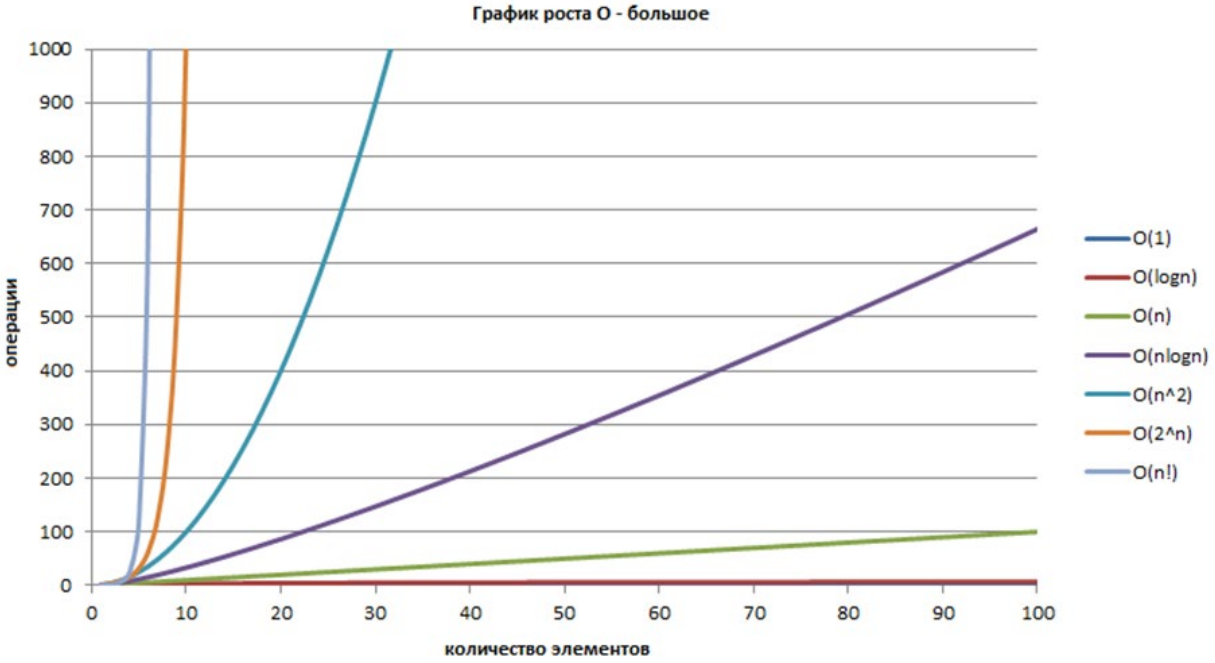
O(log(n)) - Логарифмический рост или логарифмическая зависимость. Это даже быстрее, чем O(n). Такое время характерно для алгоритмов, которые сводят большую задачу к набору меньших подзадач, уменьшая на каждом шаге размер подзадачи на некоторый постоянный коэффициент. Общее решение находится в одной из подзадач.

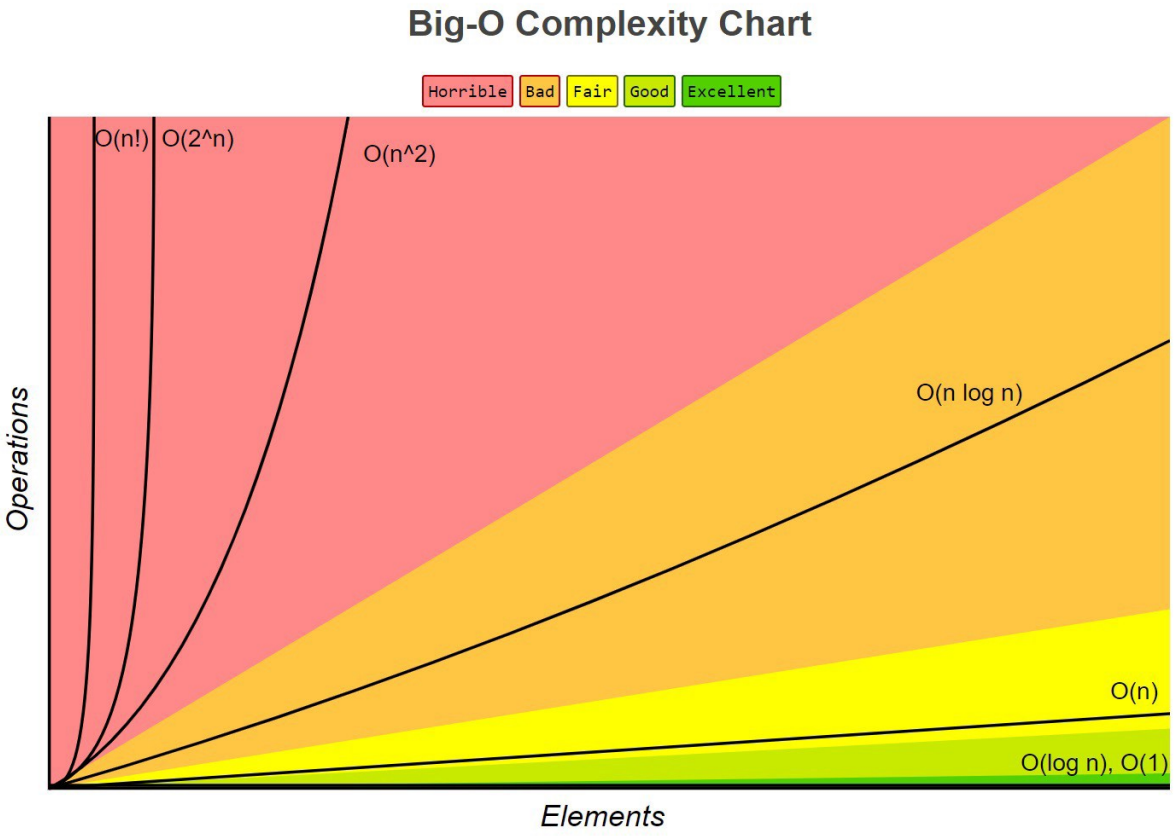
O(n*log(n)) - Квазилинейный рост. Время выполнения алгоритма пропорционально n*log(n) и возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя (комбинируя) решения подзадач.

$O(n^m)$ - Полиноминальный рост.

$O(2^n)$ - Экспоненциальный рост. Такие алгоритмы возникают естественным образом при попытке прямого решения задач (перебор и сравнение различных решений). Тем не менее, лишь несколько алгоритмов с экспоненциальным временем имеет смысл применять на практике. В остальных случаях алгоритм с такой сложностью сводят к приближенному алгоритму с приближенными значениями, но с меньшей сложностью.

Обычно время выполнения в результате анализа выглядит как многочлен, где главным членом является один из вышеприведенных. Коэффициент при главном члене зависит от количества инструкций во внутреннем цикле. Поэтому так важно максимально упрощать количество инструкций во внутреннем цикле. При росте n начинает доминировать главный член, поэтому остальными слагаемыми в многочлене можно пренебречь.





//Проигрываем в ресурсах, но выигрываем в скорости и наоборот.

//Разделяют сложность по времени и по памяти.

//Алгоритмы делятся на 2 типа, устойчивые и неустойчивые.

Устойчивость алгоритма сортировки (**Stable sorting**) — это сортировка, которая не меняет относительный порядок элементов, имеющих одинаковые ключи, по которым происходит сортировка.

//Часто подбирают алгоритм под то, какие данные будут доминировать.

Сортировка массивов

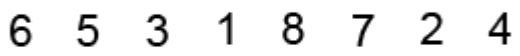
//Самый известный и бесполезный алгоритм сортировки это Пузырьковый.

все $O(n^2)$

Пузырьковая сортировка (**Bubble sort**)

По памяти O(1). Он устойчивый.

Берем предпоследний (верхний) и последний (нижний) элементы массива и сравниваем их. Если нижний элемент меньше верхнего, то меняем их местами. Затем рассматриваем пред предпоследний и предпоследний и делаем то же самое. И так далее до самого верха. Таким образом, самый «легкий» элемент из еще неотсортированных «всплывет» как пузырек.



(Если GIF не воспроизводится, то нажми сюда)

```
// Bubble sort
for (int i = 0; i < arraySize; i++)
{
    for (int j = arraySize - 1; j > i; j--)
    {
        if (a[j - 1] > a[j])
        {
            int temp = a[j - 1];
            a[j - 1] = a[j];
            a[j] = temp;
        }
    }
}
```

Сортировка выбором (**Selection sort**)

Значительно быстрее пузырька.

1. *Среди еще неотсортированных элементов находим наименьший.*
2. *Меняем его с первым из этих еще неотсортированных элементов. Этот элемент уже отсортирован.*
3. *Повторяем алгоритм для оставшихся неотсортированных элементов до тех пор, пока они еще есть.*

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

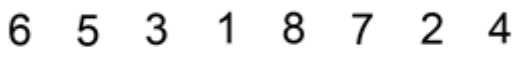
(Если GIF не воспроизводится, то нажми сюда)

```
// Selection sort
for (int i = 0; i < arraySize - 1; i++)
{
    int minIndex = i;
    for (int j = i + 1; j < arraySize; j++)
        if (a[minIndex] > a[j])
            minIndex = j;
    if (minIndex != i)
    {
        int temp = a[i];
        a[i] = a[minIndex];
        a[minIndex] = temp;
    }
}
```

Сортировка вставками (**Insertion sort**)

По скорости +- как сортировка выбором, чем больше вводных данных, тем он будет быстрее чем сортировка выбором.

1. *Берем элемент x, являющийся первым из еще неотсортированных элементов.*
2. *Движемся от него влево (т. е. среди уже отсортированных элементов). Если очередной левый элемент больше x, меняем их местами. Останавливаемся, когда выполняется одно из условий:*
    - *Найден элемент массива, меньший, чем x;*
    - *Достигнута левая граница массива.*
3. *Повторяем алгоритм для оставшихся неотсортированных элементов до тех пор, пока они еще есть.*



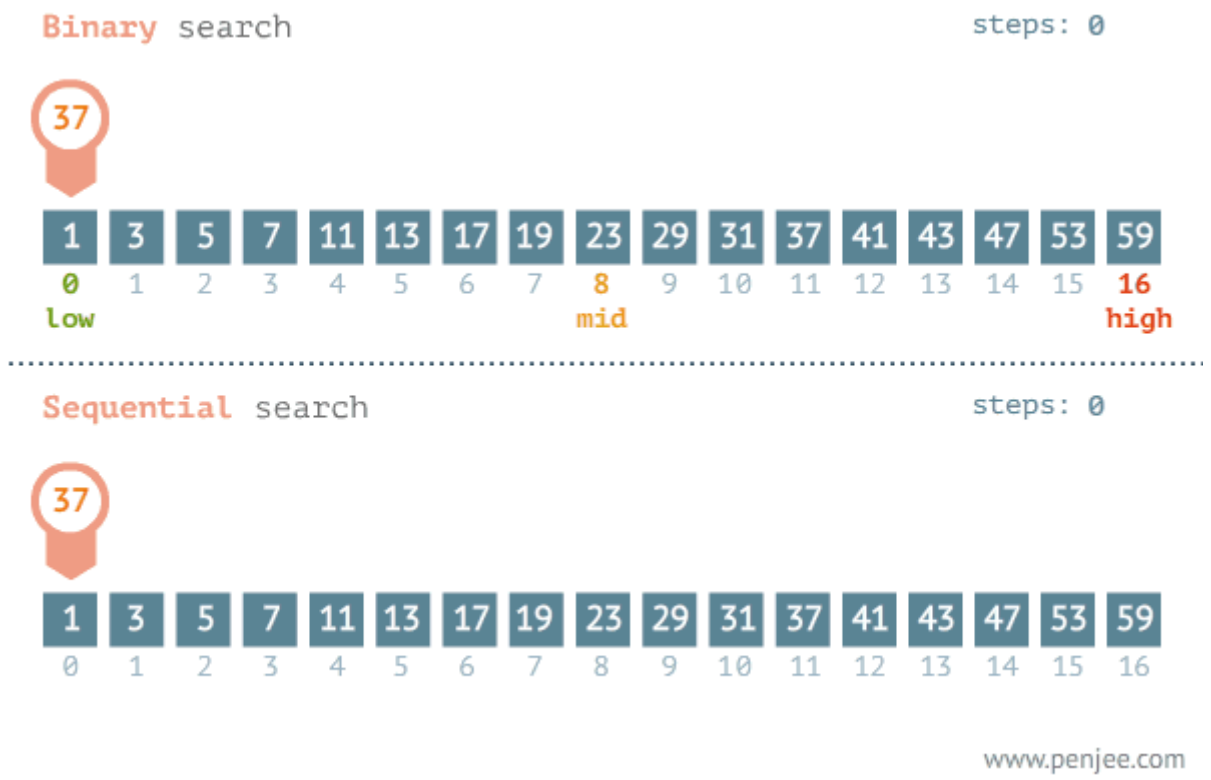
(Если GIF не воспроизводится, то нажми сюда)

```
// Insert sort
for (int i = 1; i < arraySize; i++)
{
    int temp = a[i];
    int j = i - 1;
    while (j >= 0 && a[j] > temp)
    {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = temp;
}
```

Бинарный поиск

Алгоритм бинарного поиска предназначен для эффективного поиска элемента в отсортированном массиве. Основывается на методе половинного деления.

Основная идея – выбираем точку начала срединный элемент и сравниваем его с ключом поиска. Если они равны, то поиск закончен. Если ключ меньше, то, очевидно, он находится в массиве левее срединного элемента. Тогда продолжаем поиск в левой половине массива. Соответственно, если ключ больше, то он находится в массиве правее срединного элемента и поиск продолжаем в правой половине массива. Поиск продолжается до тех пор, пока ключ не станет равным срединному элементу или пока оставшийся подмассив содержит хотя бы один элемент, не равный ключу.



(если GIF не воспроизводится, то нажим сюда)

```
int low = 0, high = arraySize - 1, middle;
bool found = false;
while (low <= high)
{
    middle = (low + high) / 2;
    if (key == a[middle])
    {
        found = true;
        break;
    }
    else if (key < a[middle])
        high = middle - 1;
    else
        low = middle + 1;
}

if (found)
    cout << "Index of element = " << middle << endl;
else
    cout << "Element not found" << endl;
```

Быстрая сортировка (**Quick Sort**)

Сложность в худшем случае - $O(n^2)$, в среднем - $O(n*\log(n))$.

*Быстрая сортировка представляет собой усовершенствованный метод сортировки, основанный на принципе обмена (**Bubble Sort**). Пузырьковая сортировка является самой неэффективной из всех алгоритмов прямой сортировки. Однако усовершенствованный алгоритм является лучшим из известных методом сортировки массивов.*

Для достижения наибольшей эффективности желательно производить обмен элементов на больших расстояниях. В массиве выбирается некоторый элемент, называемый разрешающим. Затем он помещается в то место массива, где ему полагается быть после упорядочивания всех элементов. В процессе отыскания подходящего места для разрешающего элемента производятся перестановки элементов так, что слева от них находятся элементы, меньшие разрешающего, и справа — большие (предполагается, что массив сортируется по возрастанию).

Тем самым массив разбивается на две части:

- *не отсортированные элементы слева от разрешающего элемента;*
- *не отсортированные элементы справа от разрешающего элемента.*

Чтобы отсортировать эти два меньших подмассива, алгоритм рекурсивно вызывает сам себя.

Если требуется сортировать больше одного элемента, то нужно

- *выбрать в массиве разрешающий элемент;*
- *переупорядочить массив, помещая элемент на его окончательное место;*
- *отсортировать рекурсивно элементы слева от разрешающего;*
- *отсортировать рекурсивно элементы справа от разрешающего.*

Ключевым элементом быстрой сортировки является алгоритм переупорядочения.

6 5 3 1 8 7 2 4

(если GIF не воспроизводится, то нажми сюда)

```
void QuickSortFunc(int array[], const int low, const int high)
{
    int i = low, j = high;
    const int middle = array[(low + high) / 2];
    do
    {
        while (array[i] < middle) i++;
        while (middle < array[j]) j--;

        if (i <= j)
        {
            int tmp = array[i];
            array[i] = array[j];
            array[j] = tmp;
            i++;
            j--;
        }
    }
    while (i <= j);

    if (low < j) QuickSortFunc(array, low, j);
    if (i < high) QuickSortFunc(array, i, high);
}

//Для "удобства" вызова алгоритма. При вызове не нужно каждый
//раз указывать минимальный и максимальный элемент.
void QuickSort(int array[], const int arraySize)
{
    QuickSortFunc(array, 0, arraySize - 1);
}
```