Асимптотическая сложность

Оценка алгоритмов одна из важных задач информатики, и ее принято решать при помощи О-символики, позволяющей в простой форме быстро оценить эффективность алгоритма.

Начнём с математической стороны вопроса, которая представляет собой О-символику.

! По определению *О*-большого,

или, переводя на русский, функция g ведёт себя на бесконечности почти так же, как и функция f, с поправкой на умножение на какой-то конечный коэффициент.

Для лучшего понимания нам пригодятся следующие свойства:

! Сумма двух *О* равна *О*, произведение двух *О* равно *О*.

В информатике в общем случае употребление этого понятия сводится к написанному выше вольному определению.

Эту запись используют для оценки *временной* (количество операций) и *пространственной* (количество памяти) сложностей. Так как большинство алгоритмов взаимодействуют с коллекциями (массив, список, стек, дерево и т. д.), то за в условной записи  берётся количество элементов в коллекции.

Пример 1*.* Оценим алгоритм пузырьковой сортировки (*bubble sort*)

/\* Отсортируем массив по убыванию \*/

for (int i=1; i<n; ++i)

{

for (int r=0; r<n-i; r++)

{

if (mass[r] < mass[r+1])

{

//Обмен местами

Int temp = mass[r];

mass[r] = mass[r+1];

mass[r+1] = temp;

}

}

}

! Для оценки асимптотической сложности нам нужно максимально условно получить функцию количества операций от элементов в массиве в худшем случае. Для сортировки это будет отсортированный в обратном порядке массив.

Пройдемся по алгоритму. Мы видим, что цикл перебирает элементов массива.

Следовательно, сложность только этого цикла (т. к. ,

т. е. конечному числу).

*Примечание.* Будь в цикле, к примеру, 4 фиксированные

операции (арифметические операции, операции вывода), то можно было бы

утверждать, что сложность такого алгоритма составила бы , т. к. мы раз делаем 4 операции, следовательно алгоритм имел бы асимптотикой (т. к. т. е. конечному числу).

В алгоритме из примера всё чуть сложнее.

Рассмотрим тело первого цикла. Он содержит вложенный цикл, количество

итераций в котором зависит от внешнего . Следовательно, количество итераций в худшем случае равно , и . Можно утверждать, что сложность этой части

В теле второго цикла мы делаем фиксированное количество операций в том смысле, что они не зависят от нашего – количества элементов в массиве, следовательно асимптотика этой части равна .

Пользуясь свойством произведения, получаем, что наша сложность приблизительна равна следующему выражению:

Предел отношения будет стремиться к конечному числу.

Пример 2. Алгоритм с *логарифмической сложностью* – бинарный поиск (*binary search*)

В отсортированном массиве берется центральный элемент. Если искомый

элемент меньше центрального, то аналогичным образом проводим поиск левой части, иначе в правой. То есть проводится деление массива на 2 до тех пор, пока не останется один элемент в промежутке , где - количество делений. Таким образом будет произведено приблизительно делений (операций). Следовательно, асимптотика бинарного поиска .

! *Экспоненциальная сложность* в основном встречается в *рекурсивных* алгоритмах.

Int foo(int n) {

if n == 1 { // 1

return 1;

}

return foo(n-1) + foo(n-1)

}

Каждый раз функция делает 2 вызова самой себя, каждый из которых делает еще 2 вызова этой функции, которые делают еще по 2 вызова этой функции и т.д. до выхода из рекурсии. Таким образом сложность составит (алгоритм возвращает количество вызовов функции, так что можно проверить).

*Пространственная сложность* считается так же. К примеру в выше рассмотренной сортировке мы выделяем фиксированное количество переменных для работы алгоритма, которое не зависит от размера массива, так что пространственная сложность равна . В случае рекурсивной функции foo пространственная сложность составит (вызов функции тоже требует память, не забываем).

Конец?