Оценка сложности алгоритмов

Память или время

Многие алгоритмы предлагают выбор между объёмом памяти и скоростью. Задачу можно решить быстро, используя большой объём памяти, или медленнее, занимая меньший объём. Типичным примером в данном случае служит алгоритм поиска кратчайшего пути. Представив кару города в виде графа, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками (вершинами) графа. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми вершинами и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными вершинами, мы можем просто взять готовое расстояние из таблицы. Результат будет получен мгновенно, но это потребует огромного объёма памяти. Карта большого города может содержать десятки тысяч точек. Тогда описанная выше таблица должна содержать более 10 млрд. ячеек. Т.е. для того, чтобы повысить быстродействие алгоритма, необходимо использовать дополнительные 10 Гбайт памяти. Из этой зависимости проистекает идея объёмно-временной сложности. При таком подходе алгоритм оценивается, как с точки зрении скорости выполнения, так и с точки зрения потреблённой памяти.

Оценка порядка

При сравнении различных алгоритмов важно знать, как их сложность зависит от объёма входных данных. Допустим, при сортировке одним методом обработка тысячи чисел занимает 1 сек., а обработка миллиона чисел — 10 сек., при использовании другого алгоритма может потребоваться 2 сек. и 5 сек. соответственно. В таких условиях нельзя однозначно сказать, какой алгоритм лучше.

Алгоритм — это программа, которая представляет из себя исключительно вычисление, без других часто выполняемых компьютером вещей — сетевых задач или пользовательского ввода-вывода. Анализ сложности позволяет нам узнать, насколько быстра эта программа, когда она совершает вычисления. Примерами чисто *вычислительных* операций могут послужить операции над числами (сложение и умножение), поиск заданного значения в находящейся в оперативной памяти базе данных.

Анализ сложности также позволяет нам объяснить, как будет вести себя алгоритм при возрастании входного потока данных. Если наш алгоритм выполняется одну секунду при 1000 элементах на входе, то как он себя поведёт, если мы удвоим это значение? Будет работать также быстро, в полтора раза быстрее или в четыре раза медленнее? В программировании такие предсказания крайне важны.

Начнём с простого примера: поиска максимального элемента в массиве. Дан входной массив a размера n:

```
int max = a[0];
for ( int i = 0; i < n; i++ )
    if ( a[i] > max )
        max = a[i];
```

Сначала подсчитаем, сколько здесь вычисляется фундаментальных инструкций. В процессе анализа данного кода, имеет смысл разбить его на простые инструкции — задания, которые могут быть выполнены процессором тотчас же или близко к этому. Предположим, что наш процессор способен выполнять как единые инструкции следующие операции:

- Присваивать значение переменной
- Находить значение конкретного элемента в массиве
- Сравнивать два значения
- Инкрементировать значение
- Основные арифметические операции (например, сложение и умножение)

Мы будем полагать, что ветвление (выбор между `if` и `else` частями кода после вычисления *if*-условия) происходит мгновенно, и не будем учитывать эту инструкцию при подсчёте.

Для первой строки в коде требуются две инструкции: для поиска $a[0]$ и для присвоения значения $\max$. Эти две инструкции будут требоваться алгоритму вне зависимости от величины n. Инициализация цикла *for* тоже будет происходить постоянно, что даёт нам ещё две команды: присвоение (`i = 0`) и сравнение (`i < n`). Всё это происходит до первого запуска *for*. После каждой новой итерации мы будем иметь на две инструкции больше: инкремент i (`i++`) и сравнение для проверки, не пора ли нам останавливать цикл (`i < n`).

Таким образом, если мы проигнорируем содержимое тела цикла, то количество инструкций у этого алгоритма $4 + 2n$ — четыре на начало цикла и по две на каждую итерацию, которых мы имеем n штук. Далее в теле цикла мы имеем операции поиска в массиве и сравнения, которые происходят всегда: `if (a[i] > max)`. Но тело *if* может запускаться, а может и нет, в зависимости от значения элемента массива. Если условие выполнено, то у нас запустятся две дополнительные команды: поиск в массиве и присваивание (`max = a[i]`). Теперь количество инструкций зависит не только от n, но и от конкретных входных значений. Например, для $a = \{1, 2, 3, 4\}$ программе потребуется больше команд, чем для $a = \{4, 3, 2, 1\}$. Когда мы анализируем алгоритмы, мы чаще всего рассматриваем наихудший сценарий. В нашем случае — это когда алгоритму потребуется больше всего инструкций до завершения, т.е. когда массив упорядочен по возрастанию, как, например, $a = \{1, 2, 3, 4\}$. Тогда max будет переприсваиваться каждый раз, что даст наибольшее количество команд. Это называется анализом наиболее неблагоприятного случая, который является ни чем иным, как просто рассмотрением максимально неудачного варианта.

Теперь мы можем определить математическую функцию $f(n)$ такую, что зная n, мы будем знать и необходимое алгоритму количество инструкций. В наихудшем случае в теле цикла из нашего кода запускается четыре инструкции, и мы имеем $f(n) = 4 + 2n + 4n = 6n + 4$.

Фактически, любая программа, не содержащая циклы, имеет $f(n) = 1$, потому что в этом случае требуется константное число инструкций (при отсутствии рекурсии). Одиночный цикл от 1 до n, даёт асимптотику $f(n) = n$, поскольку до и после цикла выполняет неизменное число команд, а постоянное же количество инструкций внутри цикла выполняется n раз. Два вложенных цикла дадут нам асимптотику вида $f(n) = n^2$.

В реальной жизни бывает проблематично выяснить точно поведение алгоритма. Особенно для более сложных примеров. Однако, мы можем сказать, что поведение нашего алгоритма никогда не пересечёт некой границы. Тогда всё, что нужно — найти эту границу.

В общем случае сложность алгоритма можно оценить по порядку величины. Алгоритм имеет сложность $O(f(n))$, если при увеличении размерности входных данных n, время выполнения алгоритма возрастает с той же скоростью, что и функция $f(n)$.

Оценка сложности алгоритма до порядка является верхней границей сложности алгоритмов. Если программа имеет большой порядок сложности, это вовсе не означает, что алгоритм будет выполняться действительно долго. На некоторых наборах данных выполнение алгоритма занимает намного меньше времени, чем можно предположить на основе их сложности. Оценивая порядок сложности алгоритма, необходимо использовать только ту часть, которая возрастает быстрее всего. Предположим, что рабочий цикл описывается выражением $f(n) = 3n^3 + 2n$. В таком случае его сложность будет равна $O(n^3)$. Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении n. При вычислении O можно не учитывать постоянные множители в выражениях. Алгоритм с рабочим шагом $3n^3$ рассматривается, как $O(n^3)$. Это делает зависимость отношения $O(n)$ от изменения размера задачи более очевидной.

Общие функции оценки сложности

Перечислим некоторые функции, которые чаще всего используются для вычисления сложности. Функции перечислены в порядке возрастания сложности. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

1. C — константа
2. $\log(\log(n))$
3. $\log(n)$
4. n^C, $0<C<1$
5. n
6. $n \cdot \log(n)$
7. n^C, $C>1$
8. C^n, $C>1$
9. $n!$

Если мы хотим оценить сложность алгоритма, уравнение сложности которого содержит несколько этих функций, то уравнение можно сократить до функции, расположенной ниже в таблице. Например, $O(\log(n)+n!)=O(n!)$.

Если алгоритм вызывается редко и для небольших объёмов данных, то приемлемой можно считать сложность $O(n^2)$. Обычно алгоритмы со сложностью $n \cdot \log(n)$ работают с хорошей скоростью. Алгоритмы со сложностью n^C можно использовать только при небольших значениях C. Вычислительная сложность алгоритмов, порядок которых определяется функциями C^n и $n!$ очень велика, поэтому такие алгоритмы могут использоваться только для обработки небольшого объёма данных.

Таким образом, обозначение $O(n)$ говорит о том, что при увеличении в 2 раза размера массива данных количество операций тоже увеличивается примерно в 2 раза. Сложность $O(n)$ имеет алгоритм с одним или несколькими простыми (не вложенными!) циклами в каждом из которых выполняется n шагов (как при поиске минимального элемента). Количество операций для алгоритма, имеющего сложность $O(n)$, вычисляется по формуле $k = a \cdot n + b$, где a и b — некоторые постоянные. Если в одном алгоритме решения задачи используется несколько циклов от 1 до n, а во втором — только один цикл, то алгоритм с одним циклом, как правило, эффективнее (хотя оба алгоритма имеют сложность $O(n)$, постоянная a в каждом случае своя, для алгоритма с несколькими циклами она будет больше).

Для алгоритма, имеющего сложность $O(n^2)$, количество операций пропорционально квадрату размера массива, то есть, если n увеличить в 2 раза, то количество операций увеличивается примерно в 4 раза (например, в программе используется два вложенных цикла, в каждом из которых n шагов). Сложность $O(n^2)$ имеют простые способы сортировки массивов: метод выбора, метод вставки... При больших n функция $f_1(n) = a_1 n^2$ растет значительно быстрее, чем $f_2(n) = a_2 n$, поэтому алгоритм, имеющий сложность $O(n^2)$ всегда менее эффективен, чем алгоритм сложности $O(n)$. Алгоритм сложности $O(n^2)$ (например, сортировку) нужно использовать только тогда, когда нет алгоритма сложности $O(n)$.

Иногда встречаются алгоритмы сложности $O(n^3)$ (три вложенных цикла от 1 до n), при больших n они работают медленнее, чем любой алгоритм сложности $O(n^2)$, то есть, менее эффективны.

Для многих задач известны только алгоритмы экспоненциальной сложности, когда размер массива входит в показатель степени, например $O(2^n)$, для больших n такие задачи не решаются за приемлемое время.