Математическая логика и теория алгоритмов. Лекция 8. Теория вычислительной сложности.

2020

Понятие временной и ёмкостной сложности алгоритмов

Для того, чтобы оценить сложность алгоритма, необходимо сначала выбрать характеристику, которая определяет величину исходных данных или их количество. Такая характеристика называется размером задачи. Выбор размера задачи главным образом определяется формулировкой этой задачи.
Например, для любой машины Тьюринга в качестве размера задачи удобно выбрать длину исходной цепочки, а в задаче, алгоритм решения которой построен на основе обработки графа, для этой цели обычно используется число вершин соответствующего графа.

Понятие временной и ёмкостной сложности алгоритмов

Время работы алгоритма и используемую алгоритмом память можно рассматривать как функции размера задачи n.
Обычно рассматривают следующие функции сложности алгоритма:
$T(n)$ — временная сложность,
$C(n)$ — ёмкостная сложность.
Единицы измерения $T(n)$ и $C(n)$ зависят от типа исследуемой алгоритмической модели.

Понятие временной и ёмкостной сложности алгоритмов

Обычно при выполнении алгоритма над разными исходными данными, имеющими один размер, время работы и используемая память зависят еще и от значения этих данных. Поэтому, чтобы сделать функции сложности независимыми от конкретных значений данных, как правило, рассматривают не точную аналитическую зависимость T или C от n, а оценки сложности:
максимальную $T_{\max}(n)$ или $C_{\max}(n)$, среднюю $T_{\text{mid}}(n)$ или $C_{\text{mid}}(n)$. Причем часто представляют интерес даже не точное аналитическое представление функциональной зависимости, например $T_{\max}(n)$, а просто порядок сложности алгоритма.

Понятие временной и ёмкостной сложности алгоритмов

Определение

Функция $f(n)$ есть $O(g(n))$, если существует константа C такая, что

$$|f(n)| < C|g(n)|$$

для всех $n > 0$.

Запись $f(n) = O(g(n))$ читается: "функция $f(n)$ имеет порядок $g(n)$". Полиномиальным алгоритмом (или алгоритмом полиномиальной временной сложности) называется алгоритм, у которого

$$T(n) = O(p(n)),$$

где $p(n)$ — некоторая полиномиальная функция. Алгоритмы, временная сложность которых не поддается подобной оценке, называются экспоненциальными.

Зависимость времени работы программы от сложности задачи

Различие между двумя указанными типами алгоритмов становится особенно заметным при решении задач большого размера. В следующей таблице приведены скорости роста некоторых типичных полиномиальных и экспоненциальных функций.

Зависимость времени работы программы от сложности задачи

Функция вр. слож.	$n = 10$	$n = 30$	$n = 60$
n	0,00001 сек.	0,00003 сек.	0,00006 сек.
n^2	0,0001 сек.	0,0009 сек.	0,0036 сек.
n^3	0,001 сек.	0,027 сек.	0,216 сек.
n^5	0,1 сек.	24,3 сек.	13,0 мин.
2^n	0,001 сек.	17,9 мин.	366 столетий
3^n	0,059 сек.	6,5 лет	$13 \cdot 10^{13}$ столетий

Зависимость времени работы программы от сложности задачи

Разные алгоритмы имеют разную временную сложность, и выяснение того, какие алгоритмы "достаточно эффективны" , а какие "совершенно неэффективны" всегда будет зависеть от конкретной ситуации. Однако различие между полиномиальными и экспоненциальными алгоритмами становится настолько заметным при решении задач большого размера, что становятся ясными причины, по которым понятие "труднорешаемости" отождествляется с экспоненциальным характером функции временной сложности. Именно временная сложность алгоритма определяет в итоге размер задач, которые можно решить этим алгоритмом. Разные алгоритмы имеют различную временную сложность $T(n)$ и влияние того, какие алгоритмы достаточно эффективны, а какие нет, всегда зависит как от размера задачи, так и от порядка временной сложности, а при небольших размерах еще и от коэффициентов в выражении $T(n)$.

Зависимость времени работы программы от сложности задачи

Можно было бы подумать, что колоссальный рост скорости вычислений, вызванный появлением нового поколения компьютеров, уменьшит значение эффективных алгоритмов. Однако происходит в точности противоположное. Так как вычислительные машины работают все быстрее и мы можем решать все большие задачи, именно сложность алгоритма определяет то увеличение размера задачи, которое можно достичь с увеличением скорости машины.

Зависимость времени работы программы от сложности задачи

Вопрос

Сколько вычислений должна потребовать задача, чтобы мы сочли ее труднорешаемой?

Общепринято, что если задачу нельзя решить быстрее, чем за полиномиальное время, то ее следует рассматривать как труднорешаемую. Тогда при такой схеме классификации задачи, решаемые алгоритмами полиномиальной сложности, будут легко решаемыми.

Зависимость времени работы программы от сложности задачи

Но нужно иметь в виду, что хотя экспоненциальная функция (например, 2^n) растет быстрее любой полиномиальной функции от n, для небольших значений n алгоритм, требующий $O(2^n)$ времени, может оказаться эффективнее многих алгоритмов с полиномиально ограниченным временем работы. Например, функция 2^n не превосходит n^{10} до значения n, равного 59. Тем не менее, скорость роста экспоненциальной функции столь стремительна, что обычно задача называется труднорешаемой, если у всех решающих ее алгоритмов сложность по меньшей мере экспоненциальна.

Зависимость времени работы программы от сложности задачи

Замечание

Существуют задачи, для которых в принципе не может существовать полиномиальный алгоритм - это задачи, для которых сама постановка влечет экспоненциальность алгоритма.

Например, перечислить все перестановки некоторого множества из n элементов, найти все подмножества заданного множества, найти все каркасы заданного графа и т.п. Такие задачи называются экспоненциальными по постановке.

Практическая оценка временной сложности

Любая программа состоит из элементов трех типов: последовательно выполняющихся участков, циклов и условных операторов, каждый из которых, в свою очередь, может иметь сложную структуру и представлять собой такие же элементы. Очевидно, что время работы последовательного участка равно сумме времени выполнения всех его элементов. Время работы цикла любого типа можно оценить по формуле

$$T_{while} = T_{begin} + \sum_{i} (T_{body} + T_{next}),$$

где T_{begin} и T_{next} предназначены для выполнения начальных действий подготовки цикла и перехода к очередному шагу цикла и зависят от типа цикла, а i — условие выполнения цикла. Время T_{body} — это время выполнения тела цикла.

Практическая оценка временной сложности

Время работы условного оператора вычисляется как сумма времени $T_{expression}$ вычисления условного выражения и максимального времени, которое может потребоваться для вычисления одной из ветвей:

$$T_{if} = T_{expression} + \max\{T_{then} + 1, T_{else}\}.$$

В выражении "$T_{then} + 1$" одна дополнительная операция означает выполнение одной команды перехода после реализации ветви $<then>$.

Пример

Рассмотрим два программных фрагмента, реализующих вычисление суммы элементов матрицы A[100][3].
Первый из них имеет вид
for $(i = 0; i < 100; i++)$
for $(j = 0; j < 3; j++) S = S + A[i][j];$
Второй реализуется последовательностью операторов
for $(j = 0; j < 3; j++)$
for $(i = 0; i < 100; i++) S = S + A[i][j];$

Пример

Цикл типа

$$\text{for } (i = V_1; i <= V_2; i++)O;$$

требует при выполнении число операций

$$T_{for} = T_{V_1} + 1 + \sum_{i=V_1}^{V_2} (T_0 + 4T_{V_2}).$$

Одна операция перед циклом соответствует начальному присваиванию, а дополнительные четыре операции в цикле — это сравнение i и V_2, условный переход, увеличение i и безусловный переход на начало цикла.

Пример

Тогда первый алгоритм потребует

$$1 + \sum_{i=1}^{100}(4 + 1 + \sum_{j=1}^{3}(4 + 2)) = 2301$$

операций, а второй

$$1 + \sum_{j=1}^{3}(4 + 1 + \sum_{i=1}^{100}(4 + 2)) = 1816$$

операций, что существенно меньше.

Практическая оценка временной сложности

Анализ временной сложности рекурсивных алгоритмов приводит к рекурсивному определению этой функции:
a) $T(n_0) = const$, т.к. начальном значении $n = n_0$ нет рекурсивного хода;
b) $T(n) = f(T(g(n)))$ при рекурсивном вызове.

Практическая оценка временной сложности

В зависимости от вида рекурсивной схемы можно либо попытаться подобрать вид точного аналитического выражения для $T(n)$, либо воспользоваться грубой оценкой функций. Если сложность рекурсивного алгоритма представляется следующей рекурсивной функцией

$$T(1) = d,$$

$$T(n) = aT\left(\frac{n}{c}\right) + bn, n > 1,$$

то в зависимости от a и c выражение для сложности имеет вид

$$T(n) \leq \begin{cases} O(n,) & a < c; \\ O(n \log_2 n), & a = c \\ O(n^{\log_c a}), & a > c. \end{cases}$$

Пример.

Для функции, заданной рекурсивной схемой

$$T(0) = 10,$$

$$T(k+1) = 8T\left(\frac{k+1}{2}\right) + 41$$

грубая оценка имеет вид $T(k) = O(k^{log_2 8}) = O(k^3)$.

Пример.

Не всегда следует пользоваться предложенной формулой вычисления оценки порядка сложности. Иногда можно вывести по индукции более точную формулу. Пусть, например, алгоритм имеет формулу сложности

$$T(0) = 1,$$

$$T(n+1) = (n+1)T(n).$$

В соответствии с формулой грубой оценки имеем

$$a = n+1, c = \frac{n+1}{n}, T(n) = O(n^{\log_{\frac{n}{n-1}}(n)}) \leq O(n^n).$$

Однако, анализ рекурсивной зависимости позволяет получить более точную формулу $O(n!)$. Действительно, допустим, соотношение $T(n) = n!$ справедливо для любого n. Докажем справедливость этого равенства для $n+1$:

$$T(n+1) = (n+1)T(n) = (n+1) \cdot n! = (n+1)!$$