О-символика. Интуитивное понятие сложности алгоритма

Оценкой эффективности реализации решения принято считать время работы программы и количество используемой памяти. Стандартные сообщения тестирующей системы о превышении максимальных лимитов обычно носят вид TL (Time Limit exeed, превышен лимит по времени) и ML (Memory Limit exeed, превышен лимит по памяти).

Эффективность программы обычно определяется, исходя из сочетания времени работы программы и используемых ресурсов, причем важность каждого из компонентов определяется в зависимости от задачи. В случае, например, олимпиадного программирования следует использовать еще один решающий компонент оценки эффективности – время, затраченное на написание программы.

Таким образом, наиболее эффективным решением можно считать решение, укладывающееся в TL и ML, и написанное за кратчайшее время.

Основная часть времени работы программы состоит из произведения количества итераций цикла (т.е. сколько раз были выполнены действия) на время выполнения команд в цикле. Время выполнения условно будем называть «константой» (не зависящей от входных данных), а количество итераций – «сложностью» алгоритма. Допустим, алгоритм линейный, т.е., например, на вход дается N чисел, и следует подсчитать их сумму. Тогда сложность алгоритма будет обозначаться как $O(N)$ (О-большое от N). Константа же для этой задачи может различаться в зависимости от конкретных условий. Например, для целых чисел константа будет в несколько раз меньше, чем для вещественных (т.к. целочисленные операции выполняются в несколько раз быстрее). Для квадратичного алгоритма (например, для сортировки пузырьком) сложность записывается как $O(N^2)$. Сложность может зависеть и от нескольких параметров, если они определяют количество циклов. Тогда сложность может записываться как $O(N^2 + M^3)$. Если в алгоритме нет циклов, зависящих от входных данных, то говорят, что он «работает за константу», т.е. за $O(1)$.

Обычно за О-большое обозначают сложность в худшем случае. Некоторые алгоритмы работают очень по-разному в зависимости от входных данных, в этом случае отдельно указывают «сложность в среднем».

Очень часто в процессе подсчета сложности алгоритма используется функция logN («логарифм от N»). В программировании мы обычно будем использовать двоичный логарифм. Он определяется так: $2^{logN} = N$. Интересная особенность логарифма заключается в том, что он растет очень медленно при росте N. Так при $N = 8$: $logN = 3$, $N = 65536$: $logN = 16$, $N = 4294967296$: $logN = 32$. Скорость роста логарифма намного меньше, чем даже у квадратного корня. Поэтому при больших значениях N алгоритм со сложностью logN и большой константой будет эффективнее линейного алгоритма с маленькой константой.

Элементы теории сложности алгоритмов

На первых порах теорию алгоритмов интересовали общие вопросы: определение основных понятий, поиск свойств алгоритмов, не зависящих от выбора модели, а также обнаружение алгоритмических неразрешимостей.

С развитием вычислительной техники было создано огромное количество разнообразных алгоритмов в различных прикладных областях и пришлось обратить серьезное внимание на вопросы их эффективности. Так как ресурсы памяти и времени

ЭВМ ограниченны, недостаточно знать, что существует алгоритм решения данной задачи. Нужно хотя бы в общих чертах представлять, какие ресурсы понадобятся ЭВМ для реализации алгоритма, сможет ли соответствующая программа поместиться в памяти и даст ли она результаты в приемлемое время. Исследователи этих вопросов создали новый раздел теории алгоритмов — **теорию сложности алгоритмов**.

Сложность алгоритма — количественная характеристика, которая говорит либо о том, сколько времени он работает (*временная сложность*), либо о том, какой объем памяти он занимает (*емкостная сложность*). Сложность рассматривается в основном для машинных алгоритмических моделей, поскольку в них время и память присутствуют в явном виде.

Физическое время выполнения алгоритма — это величина $\tau\ t$, где t — число действий (элементарных шагов, команд), а τ — среднее время выполнения одного действия. Число шагов t определяется описанием алгоритма в данной алгоритмической модели и не зависит от физической реализации модели; τ — величина физическая и зависит от скорости обработки сигналов в элементах и узлах ЭВМ. Поэтому объективной математической характеристикой трудоемкости алгоритма в данной модели является число действий t.

Емкостная сложность алгоритма определяется числом ячеек памяти, используемых в процессе его вычисления. Эта величина не может превосходить числа действий t, умноженного на небольшую константу (число ячеек, используемых в данной модели на одном шаге); в свою очередь, число шагов может сколь угодно сильно превосходить объем памяти (за счет циклов по одним и тем же ячейкам). К тому же проблемы памяти технически преодолеваются легче, чем проблемы быстродействия, которое имеет физический предел — скорость распространения физических сигналов (300 тыс. км/с). Поэтому временная сложность (трудоемкость) считается более существенной характеристикой алгоритма.

Трудоемкость алгоритма, как и другие виды сложности, не является постоянной величиной, а зависит от размерности задачи. Под размерностью задачи понимается либо объем памяти, необходимой для записи данных, либо характеристики задачи, от которых зависит этот объем. В задачах обработки графов размерностью может считаться число вершин или дуг графа, в задачах преобразования логических выражений — число букв в выражении и т. д. Например, сложность простейшего алгоритма сложения двух чисел зависит от длины слагаемых. При сложении столбиком количество элементарных действий (сложений цифр) пропорционально количеству разрядов слагаемых. При сложении в ЭВМ, использующей параллельный *сумматор*, трудоемкость сложения равна 1 (одна машинная команда сложения) до тех пор, пока каждое слагаемое уменьшается в одной ячейке. Для больших чисел она пропорциональна числу ячеек, необходимых для размещения слагаемых.

Сложность алгоритма A — это функция, значение которой зависит от размерности данных n. Поскольку для разных конкретных данных одинаковой размерности n алгоритм может затрачивать разное число действий, функция сложности определяется различными способами. Наиболее употребительны два:

а) ее значением является сложность «худшего» случая (минимальное число действий, достаточное для обработки алгоритмом A любых данных размерности n);

б) значением является средняя сложность A, взятая по всем данным размерности n.

Чтобы сказанное было понятнее, рассмотрим четыре примера.

1. Алгоритм поиска объекта в множестве, состоящем из n объектов, последовательно перебирает объекты множества (так мы ищем нужную бумагу в стопке бумаг или определенную карту в колоде). Сложность худшего случая равна n (искомый объект может оказаться последним); средняя продолжительность поиска равна сумме всех номеров от 1 до n (т. е. сумме арифметической прогрессии), деленной на n, что составляет $\frac{n+1}{2}$. В обоих случаях функция сложности имеет вид $a_1n + a_2$, т. е. является линейной.

2. Рассмотрим алгоритм нахождения максимального числа в последовательности из n элементов.

Дано: последовательность чисел.

Требуется: найти в этой последовательности максимальное число.

Метод решения:

1. В некоторой памяти M запоминаем первое число.

2. Следующее число последовательности сравниваем с числом, лежащим в M. Записываем в M большее из этих чисел (т.е. либо сохраняем в M прежнее число — если оно больше, — либо записываем вместо него следующее).

3. Повторяем шаг 2 до конца данной последовательности.

Теперь описанный выше процесс зададим более точно. Для записи чисел в память будем пользоваться обычным для языков программирования оператором присвоения. Например, $M: = x$ означает, что переменной $\boldsymbol{M}$ присвоено значение переменной $\boldsymbol{x}$, в терминах машинной памяти это значит, что в ячейку памяти M записано содержимое ячейки x.

1. Ввести данные: исходную последовательность расположить в ячейках $p(1), ..., p(n)$.

2. $e:= n$ (в ячейке e лежит число элементов последовательности).

3. $i := 1$ (счетчик номеров устанавливаем в начальное положение).

4. $M := p(1)$ (в М — первое число).

5. $N := 1$ (в N — его номер).

6. $i := i + 1$ (счетчик увеличивается на 1).

7. Если $p(i) \le M$, то перейти к п. 10; иначе перейти к следующему шагу (п.8).

8. $M := p(i)$ (в M — новое число).

9. $N := i$ (в N — его номер).

10. Если $i < e$, то перейти к п. б; иначе — к следующему шагу.

11. Конец алгоритма.

Это описание уже близко к программе на языке программирования, его сложность также линейна (каждое из n чисел просматривается один раз, и при этом совершается не более 5 операций).

3. С помощью этого алгоритма (см. п.2) можно построить алгоритм упорядочения последовательности чисел: находим наибольшее число в исходной последовательности из n чисел, выписываем его и вычеркиваем из исходной последовательности; находим наибольшее число в оставшейся последовательности из n - 1 чисел, приписываем его справа к найденному ранее числу и вычеркиваем из исходной последовательности и т. д. Трудоемкость этого алгоритма равна выражению:

$$(b_1n + c_1) + (b_2(n - 1) + c_2) + (b_3(n - 2) + c_3) + \ldots + (b_n + c_n).$$

Так как сумма первых слагаемых в скобках имеет вид арифметической прогрессии, то все выражение можно привести к полиному (многочлену) второй степени.

4. Алгоритм вычисления определителя системы линейных уравнений имеет трудоемкость, выражаемую полиномом третьей степени.

Алгоритмы, сложность которых не превосходит некоторого полинома, называются **алгоритмами с полиномиальной трудоемкостью.**

На практике нас, в конечном счете, интересуют не сами алгоритмы, а задачи, которые можно решать с их помощью. Поскольку для решения задачи существуют различные алгоритмы, естественно искать для нее алгоритм с наименьшей сложностью и определять сложность задачи как минимум среди всех сложностей алгоритмов, решающих эту задачу. С этой точки зрения некоторые из приведенных выше алгоритмов не являются оптимальными. Например, в упорядоченном множестве поиск можно осуществить гораздо быстрее, чем за n шагов. Этот быстрый алгоритм, называемый алгоритмом двоичного поиска, работает так. Обращаемся к среднему элементу множества. Если искомый элемент меньше среднего, берем меньшую часть множества и обращаемся к ее середине; если он больше среднего, обращаемся к середине большей части — и так до тех пор, пока очередная середина не совпадет с искомым элементом. Поскольку на каждом шаге длина последовательности, в которой происходит поиск, уменьшается вдвое, общая трудоемкость этого алгоритма имеет порядок $\log_2 n$. Столь высокая скорость этого алгоритма возможна только для упорядоченных множеств и лишний раз говорит о том, как важен порядок там, где хочешь быстро найти то, что нужно. Так мы ищем слова в словарях, книги в каталогах, телефоны в справочниках и т. д. Поэтому чрезвычайно важно иметь хорошие алгоритмы для задачи упорядочивания, которая в программировании называется задачей сортировки. Описанный выше алгоритм сортировки (пример 3) не является оптимальным. Известны алгоритмы сортировки со сложностью в худшем случае порядка $n \log n$.

Число действий в алгоритме зависит от выбранной алгоритмической модели; поэтому трудоемкость одной и той же задачи на разных машинах будет, вообще говоря, различной. Однако в теории сложности показано, что переход от одной машинной модели к другой изменяет (да и то для известных моделей ненамного) только степень полинома.

Поэтому если задача имеет полиномиальную сложность, то этот факт не зависит от машины, на которой она решается.

Исследования сложности алгоритмов привели к осознанию важного факта: подобно тому, как существуют алгоритмические неразрешимые задачи, существуют и задачи, объективно сложные, т. е. такие, трудоемкость решения которых нельзя существенно уменьшить усовершенствованием ЭВМ. Повышение физического быстродействия элементов ЭВМ уменьшает физическое время в константное число раз (хотя и это на практике очень важно), однако не изменяет степень полинома в функции сложности. Изменение архитектуры ЭВМ иногда может эту степень ненамного уменьшить. Если же задача имеет сложность выше полиномиальной — а в теории графов и в исследовании операций есть много задач, для которых известные алгоритмы имеют в худшем случае трудоемкость порядка 2^n и более, то эта трудоемкость сохраняется при любом прогрессе вычислительной техники.