ТЕОРИЯ СЛОЖНОСТИ АЛГОРИТМОВ

Глава 6, стр. 135

Понятие временной и емкостной 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,
$C(n)$ — емкостная сложность.
Определение 6.1. 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 $g(n)$"

Зависимость времени работы программы от сложности задачи

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.

Функция временной сложности	$n = 10$	$n = 30$	$n = 60$
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

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 размеров задач от быстродействия ЭВМ

Примеры роста размеров задач при увеличении скорости компьютера для некоторых полиномиальных и экспоненциальных зависимостей функции временной сложности приведены в таблице

Функция временной сложности	На современной ЭВМ	На ЭВМ в 100 раз более быстрых	На ЭВМ в 1000 раз более быстрых
n	n_1	$100n_1$	$1000n_1$
n^2	n_2	$10n_2$	$31.6n_2$
n^3	n_3	$4.64n_3$	$10n_3$
n^5	n_4	$2.5n_4$	$3.98n_4$
2^n	n_5	$n_5 + 6.64$	$n_5 + 9.97$
3^n	n_6	$n_6 + 4.19$	$n_6 + 6.29$

Данные получены для задач, решаемых за один час машинного времени, если быстродействие ЭВМ возрастает в 100 или 1000 раз по сравнению с современными компьютерами.

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**.
- При такой схеме классификации задачи, решаемые алгоритмами полиномиальной сложности, будут легко решаемыми.
- 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

Время работы цикла любого типа можно оценить по формуле

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

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:

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

Практическая оценка временной сложности

- Рассмотрим, например, два программных фрагмента, реализующих вычисление суммы элементов матрицы $A[100][3]$.

```
for (i=0; i< 100; i++)
    for (j=0; j<3; j++ ) S=S+A[i][j];
```

```
for (j=0; j<3; j++)
     for (i=0; i<100; i++) S=S+A[i][j];
```

Цикл типа $for(i = V1;\ i <= V2;\ i++)\ O;$

Требует при выполнении число операций

$$T_{for} = T_{V1} + 1 + \sum_{i=V1}^{V2} (T_O + 4 * T_{V2})$$

Тогда алгоритмы потребуют

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

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

операций.

Анализ временной сложности рекурсивных алгоритмов

- Функция определяется рекурсивно:

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.

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

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

P-задачи и NP–задачи

- Если задача решается за полиномиальное время

$$T(n) = P_k(n) = \sum_{i=0}^{k} a_i n^i$$

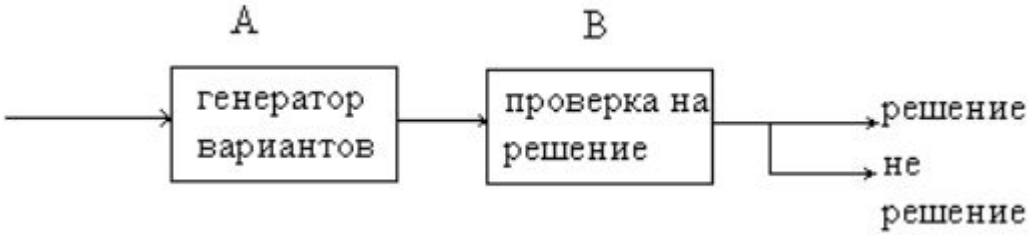
то обычно считается, что эта задача является легко решаемой. Поэтому среди множества всех задач выделен **класс *P* –задач**, для которых существует детерминированный алгоритм, решающий эту задачу за полиномиальное время.

Будем называть задачу ***труднорешаемой***, если для ее решения не существует полиномиального алгоритма.

Определение 6.2. Класс задач, для решения которых существует недетерминированный алгоритм, решающий эту задачу за полиномиальное время, называется **классом *NP* –задач**.

Недетерминированный алгоритм

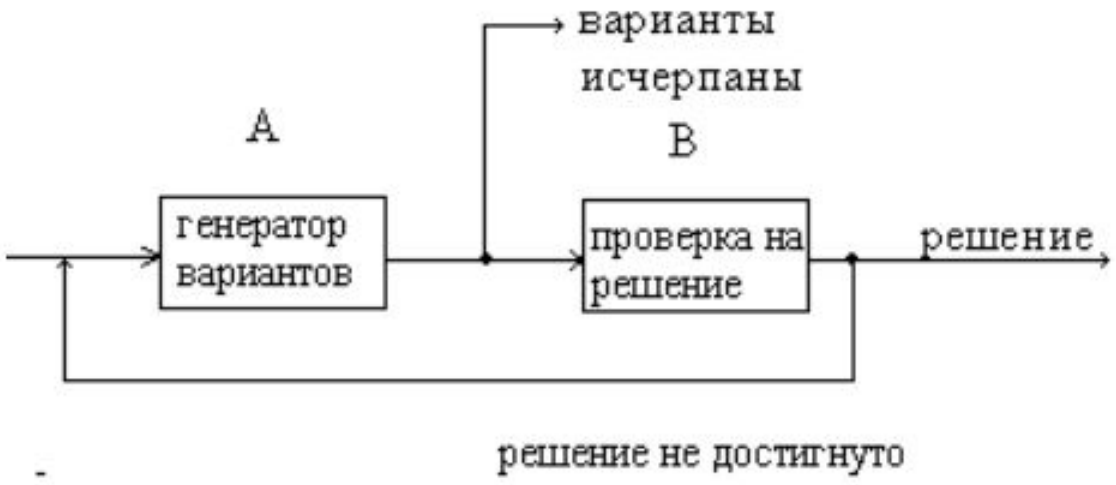
Недетерминированный алгоритм всегда должен выдавать на выходе одно из двух сообщений: "получено решение" или "решение не получено"



Смоделировать такой недетерминированный алгоритм *T* можно, формируя этот алгоритм из двух частей *A* и *B*, которые работают последовательно одна за другой и $T = A \cdot B$. Эти составные части представляют собой недетерминированный алгоритм угадывания и детерминированный алгоритм проверки. Стадия *A* — недетерминированное начало алгоритма, стадия *B* — его детерминированное завершение, как правило, отвечающее на вопрос, построил ли недетерминированный алгоритм угадывания *A* решение или нет.

Детерминированная модель недетерминированного алгоритма

Начальный участок алгоритма *A* формирует какой–то (очередной) вариант данных, которые могут быть (или не быть) решением задачи. Вторая часть — алгоритм *B* — получает сгенерированный вариант и проверяет, является ли он решением или нет. Если решение не достигнуто, вновь инициируется алгоритм *A* для получения нового варианта решения



Всякая задача, разрешимая за полиномиальное время детерминированным алгоритмом, разрешима также за полиномиальное время недетерминированным алгоритмом, т.е. класс *P* –задач входит в класс *NP* –задач.

Теорема 6.1. Если задача $Z \in NP$, то существует такой полином p(n), что задача Z может быть решена детерминированным алгоритмом с временной сложностью $O\left(2^{p(n)}\right)$.

Доказательство.

Пусть T — полиномиальный недетерминированный алгоритм решения задачи Z. Тогда существует полином q(n), ограничивающий временную сложность алгоритма T .

По определению класса NP , для каждого набора исходных данных длины n найдется некоторая последовательность данных, представляющая собой слово–догадку, длины не более q(n). При обработке этой последовательности–догадки алгоритм T на стадии проверки работает и выдает ответ "да" или "нет" за q(n) шагов. Таким образом, общее число догадок, которые нужно рассмотреть, не превосходит $k^{q(n)}$, где k - число символов, из которых состоит слово–догадка (если слово–догадка короче q(n), его можно дополнить пустыми символами и всегда рассматривать как слово длины q(n)).

Доказательство (продолжение)

Теперь построим детерминированыый алгоритм решения задачи *Z*. Для этого достаточно последовательно генерировать все слова–догадки в количестве $k^{q(n)}$ и для каждой из них запустить детерминированную стадию проверки алгоритма *T* , который работает не более *q*(*n*) шагов.

Этот алгоритм даст ответ "да" или "нет" и всегда выполняет действия проверки за время, представляющее собой некоторую константу. Теперь достаточно добавить алгоритм анализа ответа, останавливающий процесс генерации очередного слова–догадки и, следовательно, завершающий весь алгоритм.

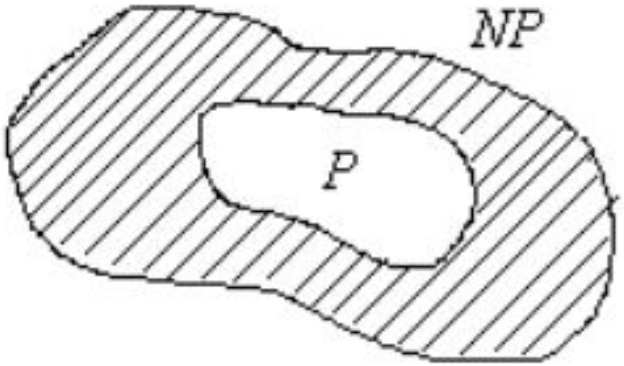
Построенный нами алгоритм, очевидно, будет детерминированным алгоритмом, работающим с временной сложностью $q(n) \cdot k^{q(n)}$

Логарифмируя эту экспоненту, а затем переходя к двоичным логарифмам, получим, что эта сложность не превосходит $O(2^{p(n)})$, где *p*(*n*) — полином.

$$P = NP\ ?$$

Вопрос о равенстве классов сложности P и NP

Проблема равенства классов P и NP является одной из семи задач тысячелетия, за решение которой Математический институт Клэя назначил премию в миллион долларов США.



На основе накопленного опыта будем представлять себе класс *NP* так, как от изображен на рис., ожидая, что затененная область, обозначающая *NP\P,* не пуста.

Что если P=NP?

1. Целый ряд задач, связанных с поиском в экспоненциальном пространстве, получит эффективное полиномиальное решение. К таким задачам относятся многие задачи комбинаторной оптимизации, в том числе проектирование цифровых схем с оптимальным энергопотреблением, задачи составления расписаний, искусственного интеллекта, проверки корректности программного обеспечения, доказательства математических теорем и пр.
2. Можно будет доказать, что полиномиальные рандомизированные алгоритмы, использующие во время работы датчики случайных чисел, могут быть выполнены за полиномиальное же время без использования случайных чисел. То есть случайность оказывается ненужной.
3. Многие широко используемые в настоящее время криптографические алгоритмы и технологии (RSA, SSL, PGP, цифровые подписи и пр.) перестанут работать, так как любая зашифрованная с их помощью информация может быть эффективно расшифрована без знания секретных ключей.

NP –полные задачи

- В классе *NP* содержатся ***NP* –полные задачи**. Это *NP* – задачи, для решения которых не существует детерминированного алгоритма, работающего за полиномиальное время.
- Для доказательства *NP* –полноты некоторой задачи *A* можно использовать несколько различных методов:
  - провести независимое доказательство для задачи *A*;
  - воспользоваться известным доказательством *NP* –полноты некоторой задачи *B* и провести доказательство *NP* –полноты задачи *A* по аналогии;
  - воспользоваться методом сужения задачи, который заключается в установлении того факта, что поставленная задача *A* включает в качестве частного случая известную *NP* –полную задачу;
  - воспользоваться полиномиальной сводимостью.
- Первые два пути сложны, на практике обычно используется третий или четвертый метод

Примеры NP –полных задач

1. ***(Выполнимость).*** Дан набор $C = C_1, \ldots, C_m$ дизъюнкций на конечном множестве переменных U. Существует ли на U набор значений истинности, при котором выполняются все дизъюнкции из C?

   Теорема Кука. *Задача "выполнимость" есть NP–полная задача.*

2. ***(Трехмерное сочетание).*** Дано множество $M \subseteq W \times X \times Y$, причем W, X и Y — непересекающиеся множества, содержащие одинаковое число элементов q,
   q =|W|=|X|=|Y|. Содержится ли в M подмножество N ⊆ M, такое, что |N|= q и никакие два разных элемента N не имеют ни одной равной координаты?

3. ***(Гамильтонов цикл).*** Дан граф G с n вершинами. Существует ли в графе простой цикл, проходящий через все вершины графа? Простым называется цикл, в котором вершины не повторяются. Таким образом, гамильтонов цикл — это последовательность вершин и дуг (ребер) графа, содержащая все вершины графа G по одному разу, но, может быть, содержащая не все дуги.

Примеры NP –полных задач

4. ***(Раскрашиваемость).*** Задан граф *G* = (*V, E*) и положительное целое число $k \leq |V|$. Является ли данный неориентированный граф *k*–раскрашиваемым?
   Граф называется *k*–раскрашиваемым, если каждой вершине графа можно поставить в соответствие такое число *j* (называемое "цветом" вершины), что любые две соседние вершины графа имеют разный цвет.
5. ***(Клика).*** Содержит ли данный граф *G* = (*V, E*) некоторую клику мощности не менее заданного целого *N*.
   Кликой мощности не менее *N* называется такое подмножество вершин $\widetilde{V} \subseteq V$, что $\left|\widetilde{V}\right| \geq N$ и любые две вершины из $\widetilde{V}$ соединены ребром в *G*.
6. ***(Разбиение).*** Задано конечное множество *A* и вес *S*(*a*) каждого элемента $a \in A$. Существует ли множество $\widetilde{A} \subseteq$ A такое, что

$$\sum_{a \in \tilde{A}} S(a) = \sum_{a \in A \setminus \tilde{A}} S(a)?$$

Методы решения NP-полных задач

- В соответствии с представлением алгоритма решения *NP*–полных задач с помощью алгоритма угадывания и алгоритма проверки программы, реализующие *NP*–полные задачи, требуют полного перебора вариантов и решаются рекурсивно, так, что алгоритм поиска решения на каждом их шаге рассматривает все возможные варианты решений на глубину 1 и оставшуюся задачу меньшего размера.

Пример

Пусть имеется произвольное клеточное поле и плитки размером 1×2. Необходимо покрыть данное поле такими плитками. Очевидно, что произвольное клеточное поле можно представить матрицей, в которой клетки заданного поля имеют следующие значения:

а) 0 — свободны,

б) число $n > 0$ — клетка занята плиткой с номером n,

в) число -1 — не принадлежащие полю клетки.

Сначала все свободные клетки заняты нулями, а все клетки, не подлежащие заполнению, числом -1. Приведем пример подлежащего заполнению поля:

0	0	-1	-1	-1
-1	0	-1	-1	-1
-1	0	0	0	0
-1	-1	0	0	0

1	1	-1	-1	-1	-1
-1	2	-1	-1	-1	-1
-1	2	3	4	4	-1
-1	-1	3	5	5	-1
-1	-1	-1	-1	-1	-1

Если полный перебор укладки плиток не привел к решению, следовательно задача решения не имеет.