Карпов Юрий Глебович,
Трифонов Петр Владимирович

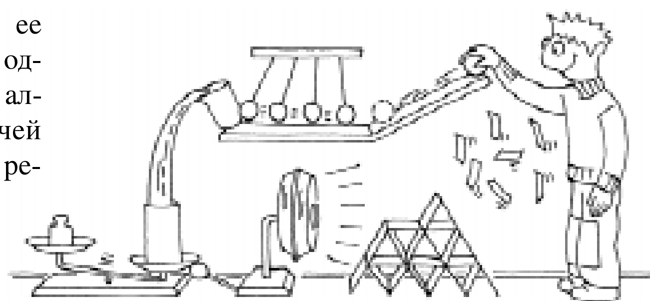
СЛОЖНОСТЬ АЛГОРИТМОВ И ПРОГРАММ

КЛАССЫ СЛОЖНОСТИ АЛГОРИТМОВ

Важнейшими характеристиками эффективности программы являются время и память, необходимые для ее выполнения. Временная сложность программы показывает, как долго она будет выполняться, и обычно измеряется числом элементарных шагов решения. Эффективность программы, выполняемой на компьютере, находится в прямой зависимости от алгоритма, который эта программа реализует. Более того, можно показать [1], что при некоторых вполне разумных допущениях о правилах кодирования, выбранном языке программирования и реалистических моделях компьютеров эта эффективность по существу не зависит ни от языка, ни от способа кодирования, ни от быстродействия компьютера и является имманентной (внутренне присущей) только самому алгоритму. Такая же ситуация имеет место и в отношении памяти. Таким образом, проблема определения эффективности программы сводится к проблеме определения эффективности ее алгоритма. Как правило, для решения одной и той же задачи можно построить алгоритмы различной сложности. Задачей профессионального программиста при решении проблемы является не кодирование первого пришедшего в голову алгоритма, а нахождение и реализация такого алгоритма, который сможет наиболее эффективно решить проблему при ее практических размерах.

Пусть A – алгоритм решения некоторого класса проблем, а N – размерность отдельной проблемы из этого класса. Натуральное число N может быть, например, размерностью обрабатываемого массива, числом вершин обрабатываемого графа и т. д. Обозначим $f_A(N)$ функцию, дающую верхнюю границу максимального числа основных операций (сложения, умножения и т. д.), которые должен выполнить алгоритм A при решении задачи размерности N.

Будем говорить, что алгоритм A *полиномиальный*, если $f_A(N)$ растет не быстрее, чем некоторый полином (в общем случае степенная функция) от N. Если $f_A(N)$ растет быстрее любой степенной функции от N, будем называть алгоритм A *экспоненциальным*. Для некоторых задач удается построить алгоритмы, для которых $f_A(N)$ растет медленнее любой степенной функции от N. Подобные алгоритмы называются *логарифмическими*. Оказывается, что между классом полиномиальных и клас-



Временная сложность программы ... обычно измеряется числом элементарных шагов решения.

сом экспоненциальных алгоритмов есть очень существенная разница: при больших размерностях проблем (которые чаще всего и интересны на практике), *полиномиальные* алгоритмы могут быть выполнены на современных компьютерах, тогда как *экспоненциальные* алгоритмы для практических размерностей проблем часто не могут быть реализованы вовсе. Обычно решение проблем, порождающих экспоненциальные алгоритмы, связанно с перебором всех или почти всех возможных вариантов, и, ввиду невозможности реализации таких алгоритмов, для нахождения решения разрабатываются другие подходы. Например, даже если существует экспоненциальный алгоритм для нахождения оптимального решения некоторой проблемы, то на практике применяются другие, более эффективные алгоритмы для нахождения не обязательно оптимального, а только приемлемого (допустимого) решения. Обычно полиномиальный алгоритм может быть построен для нахождения (даже и не всегда оптимального) решения проблемы лишь тогда, когда удается глубоко проникнуть в суть этой проблемы.

Полиномиальные алгоритмы различаются в зависимости от степени полинома, аппроксимирующего $f_A(N)$. Рассмотрим сначала, как оценить временную сложность алгоритма. Такая оценка производится с использованием символа O («*большого* O»): говорят, что $f_A(N)$ растет как $g(N)$ для больших N если существует положительная постоянная $C > 0$, такая, что $\lim_{N \to \infty} f_A(N)/g(N) = C$. Это записывается как $f_A(N) = O(g(N))$. Оценка $O(g(N))$ называется временной асимптотической сложностью алгоритма. Оценка $O(g(N))$ для функции $f_A(N)$ применяется, когда точное значение $f_A(N)$ неизвестно, а известен лишь порядок роста времени, затрачиваемого алгоритмом A на решение задачи размерностью N. Точные коэффициенты функции $f_A(N)$ зависят обычно от конкретной реализации, в то время как $O(g(N))$ является характеристикой самого алгоритма. Например, если временная асимптотическая сложность алгоритма есть $O(N^2)$ (такой алгоритм называется квадратичным), то при увеличении N время решения задачи увеличивается, как N^2. Факт экспоненциальной сложности алгоритма в терминах введенной символики можно записать так: $f_A(N) = O(k^N)$, где k – некоторое число, большее единицы.

Важность исследования асимптотической сложности алгоритмов видна из следующего примера, приведенного в [2]. Пусть A, B, C и D – алгоритмы такой теоретической сложности: A – экспоненциальной $O(2^n)$, B – полиномиальной $O(n^2)$, C – линейной $O(n)$ и, наконец, D – логарифмической $O(\log\ n)$. Пусть n_A, n_B, n_C и n_D – максимальные размеры задач, соответствующих этим четырем алгоритмам, которые можно решить за разумное время (например, за несколько часов работы современного компьютера). Предположим, что изобретен новый компьютер, в тысячу раз более быстродействующий, чем существующие. Конечно, на этом компьютере можно решать задачи большей размерности, чем на существующем. Зададимся вопросом, каким будет этот выигрыш для алгоритмов с разной оценкой сложности.

- Для экспоненциального алгоритма A на новом компьютере можно решать задачи размерности на 10 единиц большие, чем n_A.
- Для полиномиального алгоритма B можно теперь решать задачи в 30 раз большей размерности, чем n_B.
- Для линейного алгоритма C на новом компьютере можно решать задачи в 1000 раз более сложные, чем раньше.
- Для логарифмического алгоритма D можно теперь решать задачи размерности n_D^{1000}.

ПРИМЕР РАЗРАБОТКИ АЛГОРИТМОВ РАЗЛИЧНОЙ СЛОЖНОСТИ ДЛЯ РЕШЕНИЯ ОДНОЙ ПРОБЛЕМЫ

Сравним алгоритмы различной сложности на примере одной интересной задачи, которая часто предлагается на собеседовании кандидатам при найме на

Листинг 1. Алгоритм 1

	Код	Число выполнений
1:	`MaxSoFar = 0;`	1
2:	`for (int L=0; L<N; L++)`	N
3:	`  for (int U=L; U<N; U++) {`	$N + (N - 1) + \ldots + 1 = N(N+1)/2$
4:	`    Sum = 0;`	$N(N+1)/2$
5:	`    for (int i=L; i<U; i++)`	$(1 + \ldots + N) + (1 + \ldots + (N-1)) + \ldots + 1 = \Sigma_{i=1}^{N}(N - i + 1)*(N - i + 2)/2$
6:	`      Sum += X[i];`	$\Sigma_{i=1}^{N}(N - i + 1)*(N - i + 2)/2$
7:	`      MaxSoFar = max(MaxSoFar, Sum);}`	$N(N+1)/2$

работу программистов (в частности, в компании Майкрософт). Сравнение алгоритмов решения этой задачи приведено в [3].

Исходными данными для задачи является вектор X из N целых чисел, выходом – максимальная сумма любого подмассива этого вектора. Например, если входной вектор есть $X[0..9] = <31, -41, 59, 26, -53, 58, 97, -93, -23, 84>$, то результатом решения задачи является сумма подмассива $X[2..6]$, которая равна 187. Это максимальная сумма среди всех возможных подмассивов массива X. Очевидным решением при всех положительных элементах вектора является сумма элементов всего входного вектора, а при всех отрицательных элементах – пустой подмассив с нулевой суммой.

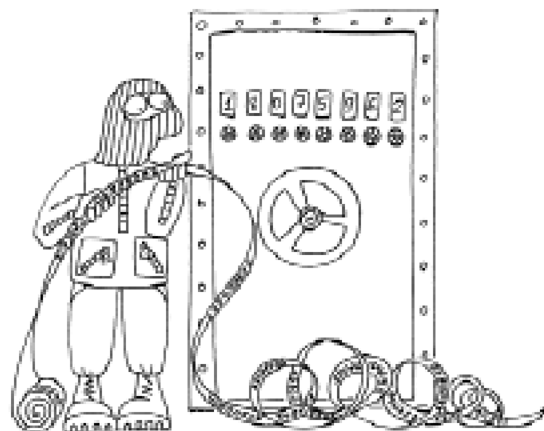
Эта проблема возникает при построении системы распознавания образов: после представления изображения в цифровой форме степень наилучшего совпадения некоторого фрагмента этого изображения с заданным образцом определяется именно как максимальная сумма членов подмассивов массива, представляющего образ.

Очевидный алгоритм решения этой проблемы, который сразу возникает в голове неглубокого программиста таков. Для всех пар границ подмассивов L, U, таких, что $1 \le L \le U \le N$, где N – длина вектора X, подсчитать суммы членов и выбрать среди них максимальную (листинг 1).

Оценим сложность *Алгоритма 1*. В листинге справа от каждого оператора этого алгоритма подсчитано число его выполнений. Будем считать, что все операторы одинаково сложны (если это не так, можно ввести соответствующий коэффициент). К оператору управления циклом (третья строка) производится N обращений; при первом обращении этот цикл выполняется N раз, при втором $N - 1$ раз, и т. д., всего $N(N+1)/2$ раз. Столько же раз выполняются операторы 4 и 7. К выполнению оператора цикла 5 управление передается $N + (N-1) + (N-2) + \ldots + 1$ раз. При первом обращении цикл выполняется повторно $[1, 2, \ldots, N]$ раз, затем $[1, 2, \ldots, N-1]$ раз и т. д. Всего операторы 5 и 6 выполняются

$$\sum_{i=1}^{N}[(N-i+1)*(N-i+2)/2] =$$
$$= N^3/6 + 2N^2 + 3N/4$$

раз, Общее число выполненных операторов алгоритма: $f_{A1}(N) = 1 + N + 3N(N-1)/2 + 2(N^3/6 + 2N^2 + 3N/4)$. Таким образом,



...степень наилучшего совпадения некоторого фрагмента этого изображения с заданным образцом...

Листинг 2. Алгоритм 2

	Код	
1:	`MaxSoFar = 0;`	1
2:	`for (int L=0; L<N; L++) {`	N
3:	`  Sum = 0;`	N
4:	`  for (int U=L; U<N; U++) {`	$N + (N-1) + (N-2) + \ldots + 1 = N(N+1)/2$
5:	`    Sum += X[U];`	$N(N+1)/2$
6:	`    MaxSoFar = max (MaxSoFar, Sum);}}`	$N(N+1)/2$

асимптотическая временная сложность этого *Алгоритма 1* равна $O(N^3)$.

Существует возможность сделать программу значительно быстрее. Более внимательный программист может заметить, что если уже вычислена сумма *Sum* элементов подмассива $X[L \ldots U]$, то сумма элементов подмассива $X[L \ldots U+1]$ получается из *Sum* просто добавлением элемента $X[U+1]$. На этой идее основан следующий алгоритм (листинг 2).

Общее число выполненных операторов *Алгоритма 2*: $f_{A2}(N) = 1 + 2N + 3N(N-1)/2$. Поэтому его асимптотическая временная сложность равна $O(N^2)$.

Еще более проницательный разработчик программы может заметить, что прежде чем вычислять сумму элементов подмассива $X[L \ldots U]$, стоит перестроить исходный массив X так, чтобы каждый его элемент содержал бы сумму всех предшествующих элементов массива X. Новый вспомогательный массив назовем *CumArray*. Таким образом, *CumArray*$[i]$ будет равен $X[1] + X[2] + \ldots + X[i]$. Используя массив *CumArray*, вычислить сумму элементов подмассива $X[L \ldots U]$ легко: она равна *CumArray*$[U]$ – *CumArray*$[L-1]$. Таким образом, за одну операцию вычитания можно получить любую сумму, на получение которой в предыдущих алгоритмах тратилось $O(N)$ сложений. Новый алгоритм имеет следующий вид (листинг 3).

Асимптотическая временная сложность *Алгоритма 3* равна $O(N^2)$, она такая же, как и у *Алгоритма 2*. Действительно, сложность первой вспомогательной части – получения вспомогательного массива – требует $O(N)$ операций, а основная часть – это два вложенных цикла, и выполнение ее требует $O(N^2)$ операций.

Возможность построения еще более эффективного алгоритма для решения этой проблемы представляется сомнительной: действительно, существует порядка $O(N^2)$ подмассивов у массива длины N, поэтому любой алгоритм, сравнивающий суммы всех этих подмассивов, с неизбежностью будет иметь сложность не ниже, чем $O(N^2)$.

Однако глубокий, думающий разработчик обычно не останавливается на достигнутом. Он всегда будет искать наиболее эффективное, элегантное решение поставленной проблемы. После глубокого анализа он может найти следующий прием.

Листинг 3. Алгоритм 3

```
MaxSoFar = 0;
CumArray[0] = 0;
for (int i=0; i<N; i++)
  CumArray[i]:= CumArray[i-1] + X[i];
for (int L=1; L<N; L++) {
  for (int U=L; U<N; U++) {
    Sum = CumArray[U] - CumArray[L-1];
    MaxSoFar = max (MaxSoFar, Sum);}}
```

Пусть после анализа *X* [*i*], сканируя массив *X* слева направо, мы уже имеем значение *MaxSoFar* максимальной суммы среди всех подмассивов массива *X* [0 ... *i*]. Поставим вопрос: как можно найти значение *MaxSoFar* среди всех подмассивов массива *X* [1 ... *i* + 1] после анализа следующего элемента *X* [*i*] массива *X*? Ключевой идеей здесь является следующее соображение: *единственной альтернативой значению MaxSoFar на следующем шаге является максимальная сумма всех таких подмассивов массива X, которые имеют правую границу точно в позиции i* + 1. Назовем это значение *MaxEndingHere*. Если каким-то образом мы можем это значение вычислить на *i* + 1-м шаге, то новое значение *MaxSoFar* можно определить как максимальное среди двух значений: предыдущего значения *MaxSoFar* и текущего значения *MaxEndingHere*.

Можно подсчитывать значение *MaxEndingHere* каждый раз в цикле, и тем самым мы можем построить новый алгоритм, временная сложность которого будет квадратичной. Но простые соображения показывают, что величину *MaxEndingHere* можно получить итеративно из ее предыдущего значения и величины *X* [*i* + 1]. Таким образом, новый алгоритм имеет вид (листинг 4).

Временная сложность *Алгоритма 4* равна *O*(*N*). Очевидно, что это наилучший возможный алгоритм, поскольку любой алгоритм обязательно должен просмотреть все *N* значений массива *X* хотя бы раз.

Приведенный пример демонстрирует часто возникающую ситуацию: тщательное продумывание задачи ведет к уменьшению сложности решающего ее алгоритма, а это, в свою очередь, дает существенное уменьшение времени решения. Так, например, время решения этой проблемы для входного массива *X* длиной 10^7 на машине Dell XPS M1710 для первого алгоритма – несколько недель, а для последнего – менее секунды.

КРИПТОГРАФИЯ: ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ РЕЗУЛЬТАТОВ ТЕОРИИ СЛОЖНОСТИ АЛГОРИТМОВ

Теория сложности кажется довольно абстрактной ветвью информатики, имеющей весьма небольшое прикладное значение. Однако существуют целые разделы практической информатики, целиком основанные на результатах этой теории. Одним из таких разделов является криптозащита в компьютерных сетях.

ШИФРОВАНИЕ С ОТКРЫТЫМ КЛЮЧОМ

Много лет люди были уверены, что защита от несанкционированного доступа при обмене конфиденциальной информацией может быть организована только в случае, если обе обменивающиеся информацией стороны разделяют один и тот же секретный ключ. Для компьютерных

...один и тот же секретный ключ.

Листинг 4. Алгоритм 4

```
MaxSoFar = 0;
MaxEndingHere = 0;
for (int i=0; i<N; i++) {
  MaxEndingHere = max (0, MaxEndingHere + X[i]);
  MaxSoFar = max (MaxSoFar, MaxEndingHere);}}
```

сетей, где обычным является оперативно организуемый обмен конфиденциальной информацией между абонентами, никогда друг друга не видевшими, встала следующая проблема: как двум сторонам договориться об общем секретном ключе, как передать сам ключ по сети? Открытая передача секретного ключа в каналах связи сопряжена с риском потери секретности при его перехвате. Поэтому ключ для передачи тоже надо зашифровать с помощью нового ключа. Как разорвать этот порочный круг?

В 1976 г. было обнаружено, что можно построить обмен конфиденциальной информацией на основе так называемого «*открытого ключа*», без передачи общего для двух сторон секретного ключа. Введем две функции: функцию (ключ) шифрования E и обратную ей функцию (ключ) дешифрования D. Как правило, общий вид этих функций общеизвестен, а секретность обеспечивается введением соответствующих параметров. Если M – сообщение, то $E(M)$ – зашифрованное сообщение, и $D(E(M))$ – дешифрированное сообщение, то есть M. Идея криптографии с открытым ключом состоит в том, что если по известной функции E очень трудно (практически невозможно) найти обратную ей функцию D, то пользователь A может опубликовать свою функцию E, храня функцию D в секрете. Любой может направить пользователю A зашифрованную информацию, воспользовавшись открытой функцией E, но только сам A может

Рассмотрим схему для удостоверения... электронной подписи.

расшифровать ее, используя свой секретный ключ D.

Слова «*трудно (практически невозможно) найти*» имеют здесь смысл именно алгоритмической трудности поиска, а не трудности розыскных мероприятий. Достаточно быстро криптографы отказались от таких шифров, как, например, подстановочные (как в «Пляшущих человечках» А. Конан-Дойля, где каждая буква текста заменяется специальным знаком). Подстановочные шифры легко раскрываются вычислением частотного словаря зашифрованного сообщения. В настоящее время криптография рекомендует системы шифрования, для которых сложность всех известных в настоящее время атак достаточно велика. Шифр считается надежным, если затраты на взлом (выражаемые в объеме необходимых компьютерных ресурсов) превышают выигрыш от получения зашифрованной информации. С течением времени разрабатываются новые атаки, совершенствуется вычислительная техника, что приводит к снижению криптостойкости систем шифрования. Для обеспечения адекватной защиты приходится или пересматривать параметры криптосистем (например, увеличивать длину ключа), или разрабатывать новые криптосистемы.

Как правило, при передаче больших объемов данных собственно их шифрование производится с помощью некоторого симметричного шифра, а ключ шифрования защищается с помощью криптосистемы с открытым ключом. Это позволяет существенно снизить сложность вычислений.

ЭЛЕКТРОННАЯ ПОДПИСЬ

Рассмотрим схему для удостоверения сообщений – *электронной подписи*. Очевидно, что если для функций E и D выполняется $D(E(M)) = M$, то справедливо и $E(D(M)) = M$, то есть функции D и E взаимно обратны. Пусть A и B – два корреспондента, с соответствующими ключами шифрования и дешифрования E_A, D_A, E_B,

D_B и пусть E_A и E_B – открыты. Тогда A «подписывает» свое сообщение M, направляемое к B, следующим образом. Он шифрует M, используя сначала свой секретный ключ D_A, а затем получившийся зашифрованный код еще раз шифруется открытым ключом E_B с получением закодированного сообщения $E_B(D_A(M))$. Это сообщение может быть дешифровано только корреспондентом B, имеющим свой секретный ключ дешифрования D_B, так: $D_B(E_B(D_A(M)))$ с получением $D_A(M)$. При этом B уверен, что сообщение M пришло именно от A, если он может полученный код $D_A(M)$ дешифровать открытым ключом для A: $E_A(D_A(M)) = M$. Необходимо отметить, что электронная подпись может использоваться и без шифрования.

ПРИМЕР КРИПТОСИСТЕМЫ С ОТКРЫТЫМ КЛЮЧОМ

Рассмотрим один из простейших примеров решения проблемы криптозащиты с открытым ключом. Пусть a и b – заданные параметры-константы, z – переменная, и $h(z) = a^z \, mod \, b$ – функция от a, b и z. Очевидно, что вычисление этой функции не представляет трудностей, она имеет линейную сложность по отношению к размеру z. В то же время вычисление обратной функции, а именно, определение z при заданных a, b и $x = h(z)$ является трудным делом. Эта задача носит название проблемы дискретного логарифма. В настоящее время отсутствуют полиномиальные алгоритмы решения этой задачи. Это значит, что увеличивая размеры параметров можно сделать проблему нахождения обратной функции для $h(z)$ практически неразрешимой.

Эти результаты могут быть использованы в криптозащите, например, следующим образом. Пусть банк и его клиенты имеют одни и те же известные им всем параметры a и b. Банк должен организовать обмен конфиденциальной информацией со своими клиентами. Любой клиент A банка имеет свой уникальный[1] секретный код x. Этот клиент при регистрации посылает в банк не сам секретный код x, а значение функции $h(x) = a^x \, mod \, b$, которое и хранится в банке.

Рассмотрим протокол Диффи-Хеллмана, позволяющий организовать построение так называемого общего разделяемого ключа для одной сессии обмена конфиденциальной информацией. Для этого банк генерирует случайное число y и вычисляет функцию $h(y)$. Именно этот код посылается от банка клиенту A как открытый ключ. Клиент, получив значение $h(y) = a^y \, mod \, b$, использует свое секретное число x для того, чтобы вычислить $K = (h(y))^x = a^{xy} \, mod \, b$. Банк находит это же значение как $(h(x))^y$. Эта величина используется далее в качестве ключа шифрования некоторого симметричного шифра.

Защита передаваемой информации обеспечивается здесь тем, что общий секрет K не передается открыто, и даже имея *«открытые ключи»* $h(y)$ и $h(x)$, величину K можно построить только при знании персонального ключа x клиента или же числа y, которые не передаются открыто. На стороне банка секретный ключ x неизвестен, и для обеспечения защиты информации со стороны нечестных сотрудников банка, имеющих доступ к значению $h(x)$, следует только скрыть (забыть, стереть) сразу после генерации случайное число y. Конечно, сам ключ K должен быть также уничтожен сразу после сессии – кодирования сообщения M. Для этого можно разработать защищенную аппаратуру, которая будет генерировать случайную величину y, вычислять функцию $h(y)$ и сразу же после получения кода K текущей сессии стирать сгенерированное значение y.

Именно отсутствие эффективных алгоритмов получения z по $h(z)$, то есть решения задачи дискретного логарифма, дает возможность организации такой схемы криптозащиты.

[1] Иногда величина x генерируется случайным образом.

КРИПТОСИСТЕМА RSA

Реально используемая в настоящее время криптографическая система с открытым ключом RSA, названная так по фамилиям ее разработчиков (Rivest, Shamir, Adleman, 1978), также построена на основе алгоритмически сложных проблем теории чисел. Здесь функции шифрования E: $E(M) = M^e \; mod \; n$ и дешифрования D: $D(P) = P^d \; mod \; n$ строятся так:

– выбираются два больших простых числа p и q;

– вычисляются $n = pq$ и $z = (p - 1)(q - 1)$;

– выбирается число d, взаимно простое с z;

– подбирается e, такое, что $de = 1 \; mod \; z$.

Доказано, что $M = (M^e \; mod(n))^d \; mod(n)$, то есть две функции – E и D – взаимно обратны и могут использоваться для криптографической защиты с открытым ключом. При шифровании исходный текст (рассматриваемый как строка битов) разбивается на блоки по k бит каждый так, чтобы $2^k < n$. Для шифрования блока M вычисляется $Mк = E(M)$. Для дешифрования блока $Mк$ вычисляется $D(Mк) = M$. Для шифрования необходима пара чисел e и n, для дешифрования нужны d и n. Поэтому открытый ключ состоит из пары (e, n), а закрытый – из пары (d, n).

Рассмотрим пример. Выберем $p = 3$, $q = 11$, что дает $n = 33$ и $z = 20$. Выберем $d = 7$, как простое к 20. Решая сравнение $7e = 1 \; mod \; 20$, найдем $e = 3$. Таким образом, каждый шифрованный блок P по исходному блоку M строится по правилу $P = M^3 \; mod \; 33$, а дешифрирование каждого блока M проводится по формуле $M = P^7 \; mod \; 33$. В этом простейшем примере независимо шифруются блоки из 5 битов, поскольку $2^5 < n$.

Для вскрытия шифра необходимо разложить опубликованное число n на множители. Защита передаваемой информации основана на том, что существующие алгоритмы разложения на множители имеют высокую сложность, хотя невозможность построения более эффективных алгоритмов не доказана формально. Попытки математиков в течение почти 300 лет найти полиномиальный алгоритм такого разложения оказались безуспешными.

В заключение можно сказать, что фактически все проблемы защиты информации в компьютерных сетях связаны с поиском специфических вычислительных проблем, имеющих очень высокую сложность (и, конечно, обладающих рядом дополнительных свойств).

Литература

1. *М. Гэри, Д. Джонсон.* Вычислительные машины и труднорешаемые задачи. М.: «Мир», 1982.
2. *А. Ахо, Дж. Хопкрофт, Дж. Ульман.* Построение и анализ вычислительных алгоритмов. М.: «Мир», 1979.
3. *J. Bentley.* Algorithm design techniques // Communications of the ACM. V. 27, № 9, 1984.
4. *С. Гудман, С. Хидетниеми.* Введение в разработку и анализ алгоритмов. М.: «Мир», 1981.

Карпов Юрий Глебович,
доктор технических наук,
профессор, заведующий кафедрой «Распределенные вычисления и компьютерные сети» СПбГПУ,

Трифонов Петр Владимирович,
кандидат технических наук,
доцент кафедры «Распределенные вычисления и компьютерные сети» СПбГПУ.

Наши авторы, 2007
Our authors, 2007