

Лекции по предмету Алгоритмы 2

Группа лектория ФКН ПМИ 2016-2017
Данила Кутенин

2016/2017 учебный год

Содержание

| | | |
|----------|--|-----------|
| 1 | Программа. Орг моменты | 2 |
| 2 | Лекция 01 от 02.09.2016. Матроиды | 2 |
| 2.1 | Матроид | 2 |
| 2.2 | Приводимость одной базы к другой | 4 |
| 2.3 | Жадный алгоритм на матроиде | 4 |
| 3 | Лекция 2 от 06.09.2016. Быстрое преобразование Фурье | 6 |
| 3.1 | Применение преобразования Фурье | 6 |
| 3.2 | Алгоритм быстрого преобразования Фурье | 7 |
| 4 | Лекция 3 от 16.09.2016. Алгоритм Карацубы, алгоритм Штрассена | 9 |
| 4.1 | Перемножение 2 длинных чисел с помощью FFT | 9 |
| 4.2 | Алгоритм Карацубы | 9 |
| 4.3 | Перемножение матриц. Алгоритм Штрассена | 11 |
| 4.4 | Эквивалентность асимптотик некоторых алгоритмов | 12 |
| 5 | Лекция 4 от 20.09.2016. Простейшие теоретико числовые алгоритмы | 13 |
| 5.1 | Алгоритм Евклида | 13 |
| 5.2 | Расширенный алгоритм Евклида | 13 |
| 5.3 | Алгоритм быстрого возведения в степень по модулю | 14 |
| 5.4 | Китайская теорема об остатках и её вычисление | 15 |
| 5.5 | Решето Эратосфена | 15 |
| 5.6 | Решето Эйлера | 16 |
| 5.7 | Наивная факторизация числа за $O(\sqrt{n})$ | 17 |

| | | |
|----------|--|-----------|
| 6 | Лекция 5 от 27.09.2016. RSA, продолжение некоторых теоретико числовых алгоритмов, некоторые комбинаторные оптимизации | 18 |
| 6.1 | Предисловие | 18 |
| 6.2 | Тест Рабина-Миллера на проверку простоты числа | 18 |
| 6.3 | RSA, криптография | 20 |
| 6.4 | Комбинаторные оптимизации. Генетический алгоритм | 22 |

Программа. Орг моменты

Внимание: программа дополняется после каждой лекции.

1. Матроиды.
2. Быстрое преобразование Фурье.
3. Алгоритм Карацубы, алгоритм Штрассена.
4. Теоретико числовые алгоритмы.

Формула такая же, как и в прошлом году:

$$0.3 \cdot O_{\text{контесты}} + 0.25 \cdot O_{\text{семинарские листки}} + 0.15 \cdot O_{\text{кр}} + 0.3 \cdot O_{\text{экзамен}} + B.$$

Округление вверх.

Лекция 01 от 02.09.2016. Матроиды

Пока чуть отдаленно от матроидов.

У нас есть конечное множество A , которое в будущем мы будем называть *носителем*. Пусть $F \subset 2^A$, и F мы будем называть *допустимыми* множествами.

Также у нас есть весовая функция $c(w) \forall w \in A$. Для каждого $B \in F$ мы определим *стоимость* множества, как $\sum_{w \in B} c(w)$. Наша задача заключается в том, чтобы найти максимальный вес из всех допустимых множеств.

Пример 1 (Задача о рюкзаке). У каждого предмета есть вес и стоимость. Мы хотим унести как можно больше вещей максимальной стоимости с весом не более k .

Вес не более k нам задает ограничение, то есть множество F . А максимизация унесенной суммы нам и задаёт задачу.

Матроид

Множество F теперь будет всегда обозначаться как I .

Матроидом называется множество подмножеств множества A таких, что выполняются следующие 3 свойства:

1. $\emptyset \in I$
2. $B \in I \Rightarrow \forall D \subset B \Rightarrow D \in I$
3. Если $B, D \in I$ и $|B| < |D| \Rightarrow \exists w \in D \setminus B$ такой, что $B \cup w \in I$

Дальнейшее обозначение матроидов — $\langle A, I \rangle$.

Определение 1. Базой матроида называют множество всех таких элементов $B \in I$, что не существует B' , что $B \subset B'$, $|B'| > |B|$ и $B' \in I$. Обозначение \mathfrak{B} .

Свойство 1. Все элементы из базы имеют одну и ту же мощность. И все элементы из I , имеющие эту мощность, будут в базе.

Доказательство очевидно из определения.

Пример 2 (Универсальный матроид). Это все подмножества B множества A такие, что $|B| \leq k$ при $k \geq 0$. Все свойства проверяются непосредственно.

База такого матроида — все множества размера k .

Пример 3 (Цветной матроид). У элементов множества A имеются цвета. Тогда $B \in I$, если все элементы множества B имеют разные цвета. Свойства проверяются непосредственно, в 3 свойстве надо воспользоваться принципом Дирихле.

База такого матроида — множества, где присутствуют все цвета.

Пример 4 (Графовый матроид на n вершинах). $\langle E, I \rangle$. Множество ребер $T \in I$, если T не содержит циклов.

Докажем 3 свойство:

Доказательство. Пусть у нас есть T_1 и T_2 такие, что $|T_1| < |T_2|$. Разобьём граф, построенный на T_1 на компоненты связности. Так как ребер ровно $|T_1|$ на n вершинах, то компонент связности будет $n - |T_1|$. В другом случае компонент связности будет $n - |T_2| < n - |T_1|$. То есть во 2-ом графе будет меньше компонент связности, а значит по принципу Дирихле найдётся ребро, которое соединяет 2 компоненты связности в 1-ом графе.

Этот алгоритм чем-то отдаленно напоминает алгоритм Краскала. □

Базой в таком матроиде являются все остовные деревья.

Пример 5 (Матричный матроид). Носителем здесь будут столбцы любой фиксированной матрицы. I — множество всех подмножеств из линейно независимых столбцов. Все свойства выводятся из линейной алгебры (3-е из метода Гаусса, если быть точным).

Пример 6 (Трансверсальный матроид). $G = \langle X, Y, E \rangle$ — двудольный граф с долями X, Y . Матроид будет $\langle X, I \rangle$ такой, что $B \in I$, если существует паросочетание такое, что множество левых концов этого паросочетания совпадает с B .

Докажем 3 свойство:

Доказательство. Пусть есть 2 паросочетания на $|B_1|$ и $|B_2|$ ($|B_1| < |B_2|$) вершин левой доли. Тогда рассмотрим симметрическую разность этих паросочетаний. Так как во 2-ом паросочетании ребер больше, то существует чередующаяся цепь, а значит при замене ребер на этой чередующейся цепи с новой добавленной вершиной (а она найдётся по принципу Дирихле) получим паросочетание с ещё 1 добавленной вершиной. □

Базой в таком матроиде будут вершины левой доли максимального паросочетания.

Приводимость одной базы к другой

Лемма 1. Пусть $B, D \in \mathfrak{B}$. Тогда существует последовательность $B = B_0, B_1, \dots, B_k = D$ такие, что $|B_i \Delta B_{i+1}| = 2$, где Δ обозначает симметрическую разность множеств.

Доказательство. Будем действовать по шагам. Если текущее $B_i \neq D$, тогда возьмём произвольный элемент w из $B_i \setminus D$. Тогда по 2-ому пункту определения матроида следует, что $B_i \setminus w \in I$. Так как $|B_i \setminus w| < |D|$, то существует $u \in D$ такой, что $(B_i \setminus w) \cup u \in I$. И теперь $B_{i+1} \leftarrow (B_i \setminus w) \cup u$. Мы сократили количество несовпадающих элементов с D на 1, симметрическая разность B_i и B_{i+1} состоит из 2 элементов — w и u . \square

Наконец, мы подошли к основной теореме лекции — жадный алгоритм или теорема Радо-Эдмондса.

Жадный алгоритм на матроиде

Доказательство будет в несколько этапов.

Для начала определимся с обозначениями. $M = \langle A, I \rangle$, $n = |A|$, w_i — элементы множества A . Решаем обычную задачу на максимизацию необходимого множества.

Теорема 1 (Жадный алгоритм. Теорема Радо-Эдмондса). Если отсортировать все элементы A по невозрастанию стоимостей весовой функции: $c_1 \geq c_2 \geq \dots \geq c_n$, то такой алгоритм решает исходную задачу о нахождении самого дорогого подмножества:

Algorithm 1 Жадный алгоритм на матроиде.

```

 $B \leftarrow \emptyset$ 
for  $c_i$  do
  if  $B \cup w_i \in I$  then
     $B \leftarrow B \cup w_i$ 

```

Доказательство. Теперь поймём, что наш алгоритм в итоге получит какой-то элемент из базы. Пусть B_i — множество, которое мы получим после i шагов цикла нашего алгоритма. Действительно, если это не так, что существует множество из базы, которое его покрывает: формально $\exists D \in I : B_n \subset D$ и $|B_n| < |D|$, так как можно взять любой элемент из базы и добавлять в B_n по 1 элементу из пункта 3 определения матроида. Тогда у нас существует элемент w_i , который мы не взяли нашим алгоритмом, но $B_{i-1} \cup w_i \in I$, так как $B_{i-1} \cup w_i \subset B_n \cup w_i \subset D$, то есть это лежит в I по пункту 2 определения матроида. Значит мы должны были взять w_i , противоречие.

Рассмотрим последовательность d_i из 0 и 1 длины n такую, что $d_i = 1$ только в том случае, если мы взяли алгоритмом i -ый элемент. А оптимальное решение задачи пусть будет e_i — тоже последовательность из 0 и 1. Последовательности будут обозначаться d и e соответственно.

Если на каком-то префиксе последовательности d единиц стало меньше, чем в e , то возьмём все элементы, которые помечены последовательностью e единицами. Пусть это множество будет E . Аналогично на этом префиксе последовательности d определим множество D . $|D| < |E|$, $D \in I$, $E \in I$, поэтому мы можем дополнить D каким-то элементом из E , которого не было в D . То есть на этом префиксе у d стоит 0 (пусть это будет место i), но заметим, что на i -ом шаге мы обязаны были брать этот элемент, из-за рассуждений аналогичным рассуждению про базу (2 абзаца вверх).

Получаем, что на каждом префиксе d единиц не меньше, чем на этом же префиксе последовательности e . Значит 1-ая единица в d встретится не позже, чем в e , 2-ая единица в d не позже, чем 2-ая в e и т.д. по рассуждениям по индукции.

□

На лекции была теория про ранги. В доказательстве можно обойтись без неё, просто приложу то, что сказал Глеб. Может быть понадобится в задачах.

Рангом множества $B \subset A$ (обозн. $r(B)$) называют максимальное число k такое, что $\exists C \subset B$ такое, что $|C| = k, C \in I$.

Эта функция обладает таким свойством: для любого элемента $w \in A$ следует, что $r(B \cup w) \leq r(B) + r(w)$. Давайте поймём, почему так:

Если $r(B \cup w) = r(B)$, то всё хорошо, так как $r(w) \geq 0$. Если $r(B \cup w) = r(B) + 1$ (других вариантов не бывает из определения), то тогда $w \in I$, так как в $B \cup w$ найдётся такое $C \subset (B \cup w)$, что $|C| = r(B \cup w), w \in C$ (иначе C годилось бы для B и $r(B \cup w) = r(B)$), значит $r(w) = 1$, так как $C \in I$, а $\{w\} \subset C$.

Лекция 2 от 06.09.2016. Быстрое преобразование Фурье

Чтобы быть успешным программистом, надо знать 3 вещи:

- Сортировки;
- Хэширование;
- Преобразование Фурье.

Глеб

В этой лекции будет разобрано дискретное преобразование Фурье (Discrete Fourier Transform).

Применение преобразования Фурье

Допустим, что мы хотим решить такую задачу:

Пример 1. Даны 2 бинарные строки A и B длины n и m соответственно. Мы хотим найти, какая подстрока в A наиболее похожа на B . Наивная реализация решает эту задачу в худшем случае за $O(n^2)$. Преобразование Фурье поможет решить эту задачу за $O(n \log n)$, а именно научимся решать другую задачу:

Цель. Хотим научиться перемножать многочлены одной степени

$A(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ и $B(x) = b_0 + b_1x + \dots + b_{m-1}x^{m-1}$ так, что $C(x) = A(x)B(x)$, то есть считать свёртку (найти все коэффициенты, если по-другому) $\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_j b_{i-j} x^i$ за $O(n \log n)$.

Вернёмся к нашему примеру. Поймём как с помощью нашей **цели** решать задачу про бинарные строки.

Пусть $A = a_0 \dots a_{n-1}$, $B = b_0 \dots b_{m-1}$. Их можно считать одной длины (просто добавим нулей в конец b при надобности). Теперь задача переформулировывается как нахождение максимального скалярного произведения B и некоторых циклических сдвигов A (до $n - m + 1$).

Инвертируем массив B и припишем в конец n нулей, а к массиву A припишем самого себя. Посмотрим на все коэффициенты перемножения:

$$c_k = \sum_{i+j=k} a_i b_j$$

Но $b_i = 0$ при $i \geq n$, поэтому при $k \geq n$:

$$c_k = \sum_{i=0}^{n-1} b_i a_{k-i}$$

Выбрав нужные коэффициенты, мы решили эту задачу.

Алгоритм быстрого преобразования Фурье

Основная идея алгоритма заключается в том, чтобы представить каждый многочлен через набор n точек и значений многочлена в этих точках, быстро (за $O(n \log n)$) вычислить значения в каких-то n точках для обоих многочленов, потом перемножить за $O(n)$ сами значения. Потом применить обратное преобразование Фурье и получить коэффициенты $C(x) = A(x)B(x)$.

Итак, для начала будем считать, что $n = 2^k$ (просто добавим нулей до степени двойки).

Рассмотрим циклическую группу корней из 1 — $W_n = \{e^{i\frac{2\pi k}{n}} \mid k = 0, \dots, n-1\}$. Обозначим за $w_n = e^{i\frac{2\pi}{n}}$. Одно из самых главных свойств, что $w_n^p \cdot w_n^q = w_n^{p+q}$, которым мы будем пользоваться в дальнейшем.

Воспользуемся идеей метода «разделяй и властвуй».

Пусть $A(x) = a_0 + \dots + a_{n-1}x^{n-1}$.

Представим $A(x) = A_l(x^2) + xA_r(x^2)$ так, что

$$A_l(x^2) = a_0 + a_2x^2 + \dots + a_{n-2}x^{n-2}$$

$$A_r(x^2) = a_1 + a_3x^2 + \dots + a_{n-1}x^{n-2}$$

Определение 1. Назовём *Фурье-образом* многочлена $P(x) = p_0 + \dots + p_{m-1}x^{m-1}$ вектор из m элементов — $\langle P(1), P(w_m), P(w_m^2), \dots, P(w_m^{m-1}) \rangle$.

Теперь рекурсивно запускаемся от многочленов меньшей степени. Так как для любого целого неотрицательного k следует, что $2k$ четное число, то $w_n^{2k} = w_{n/2}^k \in W_{n/2}$, то есть мы можем уже использовать значения Фурье-образа для вычисления $A(x)$.

Если мы сможем за линейное время вычислить сумму $A_l(x^2) + xA_r(x^2)$, то суммарное время работы будет $O(n \log n)$, так как $A_l(x), A_r(x)$ имеют степень в 2 раза меньше, чем $A(x)$.

Действительно это легко сделать из псевдокода, который приведен ниже:

Algorithm 2 FFT

```

1: function FFT( $A$ )  $\triangleright$   $A$  — массив из комплексных чисел, функция возвращает Фурье-образ
2:    $n \leftarrow \text{length}(A)$ 
3:   if  $n == 1$  then
4:     return  $A$ 
5:    $A_l \leftarrow \langle a_0, a_2, \dots, a_{n-2} \rangle$ 
6:    $A_r \leftarrow \langle a_1, a_3, \dots, a_{n-1} \rangle$ 
7:    $\hat{A}_l \leftarrow \text{FFT}(A_l)$ 
8:    $\hat{A}_r \leftarrow \text{FFT}(A_r)$ 
9:   for  $k \leftarrow 0$  to  $\frac{n}{2} - 1$  do
10:     $A[k] \leftarrow \hat{A}_l[k] + e^{i\frac{2\pi k}{n}} \hat{A}_r[k]$ 
11:     $A[k + \frac{n}{2}] \leftarrow \hat{A}_l[k] - e^{i\frac{2\pi k}{n}} \hat{A}_r[k]$   $\triangleright$  Здесь минус перед комплексным числом из-за того,
    что мы должны найти другой угол, удвоенный которого на окружности будет  $\frac{2\pi(k+n/2)}{n}$ 
12:   return  $A$ 

```

Теперь поговорим про обратное FFT. Этого материала не было на лекции на момент написания:

Фактически, мы вычислили такую вещь за $O(n \log n)$:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

где y_i — Фурье-образ многочлена $A(x)$.

Фактически нам надо найти обратное преобразование. Магическим образом обратная матрица к квадратной матрице выглядит почти также:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Откуда получаем: $a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$.

Теперь напомним псевдокод обратного алгоритма:

Algorithm 3 FFT_inverted

```

1: function FFT_INVERTED( $A$ )  $\triangleright A$  — Фурье-образ, возвращает коэффициенты многочлена
2:    $n \leftarrow \text{length}(A)$ 
3:   if  $n == 1$  then
4:     return  $A$ 
5:    $A_l \leftarrow \langle a_0, a_2, \dots, a_{n-2} \rangle$ 
6:    $A_r \leftarrow \langle a_1, a_3, \dots, a_{n-1} \rangle$ 
7:    $\hat{A}_l \leftarrow \text{FFT\_inverted}(A_l)$ 
8:    $\hat{A}_r \leftarrow \text{FFT\_inverted}(A_r)$ 
9:   for  $k \leftarrow 0$  to  $\frac{n}{2} - 1$  do
10:     $A[k] \leftarrow \hat{A}_l[k] + e^{i\frac{-2\pi k}{n}} \hat{A}_r[k]$   $\triangleright$  Здесь угол идёт с минусом
11:     $A[k + \frac{n}{2}] \leftarrow \hat{A}_l[k] - e^{i\frac{-2\pi k}{n}} \hat{A}_r[k]$ 
12:     $A[k] \leftarrow A[k]/2$   $\triangleright$  Поделим на  $2 \log n$  раз, а значит поделим на  $n$  в итоге
13:     $A[k + \frac{n}{2}] \leftarrow A[k + \frac{n}{2}]/2$   $\triangleright$  Аналогично строчке выше
14:  return  $A$ 
```

Лекция 3 от 16.09.2016. Алгоритм Карацубы, алгоритм Штрассена

Перемножение 2 длинных чисел с помощью FFT

Пусть $x = \overline{x_1 x_2 \dots x_n}$ и $y = \overline{y_1 y_2 \dots y_n}$. Распишем их умножение в столбик:

$$\begin{array}{r}
\begin{array}{c}
\times \\
\frac{x_1 x_2 \dots x_n}{y_1 y_2 \dots y_n} \\
z_{11} z_{12} \dots z_{1n} \\
z_{21} z_{22} \dots z_{2n} \\
\vdots \\
z_{n1} z_{n2} \dots z_{nn}
\end{array} \\
+ \frac{\dots}{z_1 z_2 \dots z_{2n}}
\end{array}$$

Понятно, что наивное умножение 2 длинных чисел будет иметь сложность $O(n^2)$.

Давайте научимся перемножать 2 числа быстрым преобразованием Фурье за $O(n \log n)$.

Пусть $a = \overline{a_{n-1} \dots a_0}, b = \overline{b_{n-1} \dots b_0}$.

Тогда введём многочлены $f(x) = \sum_{i=0}^{n-1} a_i x^i$, $g(x) = \sum_{i=0}^{n-1} b_i x^i$.

За $O(n \log n)$ мы можем найти $h(x) = (f(x) \cdot g(x)) = \sum_{i=0}^{2n-2} c_i x^i$.

После этого надо аккуратно провести переносы разрядов таким образом и после этого развернуть полученное число, отбросив ненужные нули в начале:

Algorithm 4 Умножение 2 длинных чисел.

1: **function** УМНОЖЕНИЕ 2 ДЛИННЫХ ЧИСЕЛ($h(x)$) $\triangleright h(x)$ — перемножение 2 многочленов $f(x)$ и $g(x)$.

2: $carry \leftarrow 0$ 3: **for** $i \leftarrow 0$ to $2n - 1$ **do**4: $h_i \leftarrow h_i + carry$ 5: $carry \leftarrow \left\lfloor \frac{h_i}{10} \right\rfloor$ 6: $h_j \leftarrow h_j \bmod 10$

Но этот метод плохо применим на практике из-за того, что быстрое преобразование Фурье имеет очень большую константу.

Алгоритм Карацубы

Какое-то время человечество не знало алгоритмов перемножения быстрее, чем за $O(n^2)$. А.Н. Колмогоров считал, что это вообще невозможно. В один момент собрались математики на мехмате МГУ и решили доказать, что это невозможно. Но один из аспирантов (Анатолий Алексеевич Карацуба) Колмогорова пришёл и сказал, что у него получилось сделать это быстрее. Давайте посмотрим, как:

Будем считать, что $n = 2^k$ (если это не так, дополним нулями, сложность вырастет лишь в константу раз).

Для начала просто попробуем воспользоваться стратегией «Разделяй и властвуй». Разобьём числа в разрядной записи пополам. Тогда

$$\begin{aligned} & \times \begin{cases} x = 10^{n/2}a + b \\ y = 10^{n/2}c + d \end{cases} \\ & \quad \downarrow \\ & xy = 10^n ac + 10^{n/2}(ad + bc) + bd \end{aligned}$$

Как видно, получается 4 умножения чисел размера $\frac{n}{2}$. Так как сложение имеет сложность $\Theta(n)$, то

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$

Чему равно $T(n)$? Если посмотреть на дерево исходов или воспользоваться индукцией, то получим, что $T(n) = O(n^2)$, что, конечно, неэффективно.

Анатолий Алексеевич проявил недюжие способности и предложил следующее:

Разложим $(a + b)(c + d)$:

$$(a + b)(c + d) = ac + (ad + bc) + bd \implies ad + bc = (a + b)(c + d) - ac - bd$$

Подставим это в начальное выражение для xy :

$$xy = 10^n ac + 10^{n/2}((a + b)(c + d) - ac - bd) + bd$$

Отсюда видно, что достаточно посчитать три числа размера $\frac{n}{2}$: $(a + b)(c + d)$, ac и bd . Тогда:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

Докажем, что $T(n) = O(n^{\log_2 3})$.

Рассмотрим дерево исходов: в каждой вершине дерева мы выполняем не более Cm действий, где C —какая-то фиксированная константа, а m — размер числа на данном шаге, поэтому

$$T(n) \leq Cn \left(1 + \frac{3}{2} + \dots + \frac{3^{\log_2 n}}{2^{\log_2 n}}\right), \text{ так как на каждом шаге мы запускаемся 3 раза от задачи в 2 раза}$$

$$\text{Откуда } T(n) \leq Cn \cdot \frac{3^{\log_2 n} - 1}{1/2} = 2Cn^{\log_2 3} = O(n^{\log_2 3}) \approx O(n^{1.5849})$$

Полученный алгоритм называется алгоритмом Карацубы.

Перемножение матриц. Алгоритм Штрассена

После идеи А.А. Карацубы, появились многие алгоритмы, использующие ту же идею. Одним из этих алгоритмов является алгоритм Штрассена. Будем считать, что $n = 2^k$ снова (оставляем читателю самим подумать, как дополнить матрицы $m \times t, t \times u$, чтобы потом легко восстановить ответ)

Пусть у нас есть квадратные матрицы

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \text{ и } B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}$$

Сколько операций нужно для умножения матриц? Умножим их по определению. Матрицу $C = AB$ заполним следующим образом:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Всего в матрице n^2 элементов. На получение каждого элемента уходит $O(n)$ операций (умножение за константное время и сложение n элементов). Тогда умножение требует $n^2 O(n) = O(n^3)$ операций.

Попробуем применить аналогичную стратегию «Разделяй и властвуй». Представим матрицы A и B в виде:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ и } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

где каждая матрица имеет размер $\frac{n}{2}$. Тогда матрица C будет иметь вид:

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Как видно, получаем 8 перемножений матриц порядка $\frac{n}{2}$. Тогда

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

По индукции получаем, что $T(n) = O(n^{\log_2 8}) = O(n^3)$.

Можно ли уменьшить число умножений до 7? Алгоритм Штрассена утверждает, что можно. Он предлагает ввести следующие матрицы (даже не спрашивайте, как до них дошли):

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}); \\ M_2 = (A_{21} + A_{22})B_{11}; \\ M_3 = A_{11}(B_{12} - B_{22}); \\ M_4 = A_{22}(B_{21} + B_{11}); \\ M_5 = (A_{11} + A_{12})B_{22}; \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}); \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}); \end{cases}$$

Тогда

$$\begin{cases} C_1 = M_1 + M_4 - M_5 + M_7; \\ C_2 = M_3 + M_5; \\ C_3 = M_2 + M_4; \\ C_4 = M_1 - M_2 + M_5 + M_6; \end{cases}$$

Можно проверить что всё верно (оставим это как ~~наказание~~ упражнение читателю). Сложность алгоритма:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \implies T(n) = O(n^{\log_2 7}) \approx O(n^{2.8073})$$

Доказательство времени работы такое же, как и в алгоритме Карацубы.

Также существует модификация алгоритма Штрассена, где используется лишь 15 сложений матриц на каждом шаге, вместо 18 предъявленных выше.

Эквивалентность асимптотик некоторых алгоритмов

Этот раздел не войдёт в экзамен.

Здесь мы поговорим об обращении и перемножении 2 матриц. Докажем, что асимптотики этих алгоритмов эквивалентны.

Теорема 1 (Умножение не сложнее обращения). *Если можно обратить матрицу размеров $n \times n$ за время $T(n)$, где $T(n) = \Omega(n^2)$, и $T(3n) = O(T(n))$ (условие регулярности), то две матрицы размером $n \times n$ можно перемножить за время $O(T(n))$*

Доказательство. Пусть A и B матрицы одного порядка размера $n \times n$. Пусть

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

Тогда легко понять, что

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

Матрицу D мы можем построить за $\Theta(n^2)$, которое является $O(T(n))$, поэтому с условием регулярности получаем, что $M(n) = O(T(n))$, где $M(n)$ — асимптотика перемножения 2 матриц. \square

С обратной теоремой предлагаем ознакомиться в книге Кормена или Ахо, Хопкрофта и Ульмана.

Лекция 4 от 20.09.2016. Простейшие теоретико-числовые алгоритмы

Числовые алгоритмы играют огромную роль в криптографии, фактически вся криптография держится на том, что не придуман до сих пор алгоритм, который умеет факторизовать числа за полиномиальное время от размера числа.

Алгоритм Евклида

Начнём, пожалуй, с одного из самых известных алгоритмов нахождения наибольшего общего делителя, а именно — алгоритм Евклида и его расширенную версию.

Algorithm 5 Алгоритм Евклида.

```

1: function gcd(int  $a$ , int  $b$ )
2:   if  $b = 0$  then
3:     return  $a$ ;
4:   else
5:     return gcd( $b, a \bmod b$ );

```

Практически очевидно, что данный алгоритм возвращает нужное нам число. Вспомните курс дискретной математики или выпишите на бумаге то, что делает данный алгоритм.

Асимптотика такого алгоритма $O(\log n)$ (где n — максимальное значение числа) — легко проверить, что каждое число уменьшается хотя бы в 2 раза за 2 шага алгоритма.

Расширенный алгоритм Евклида

Пусть даны числа a, b, c , мы хотим найти хотя бы одну пару решений x, y таких, что $ax + by = c$. Понятно, что $\gcd(a, b) \mid c$, поэтому если это условие не выполняется, то найти решение мы не сможем. Пусть $c = k \gcd(a, b)$. Сейчас мы предъявим хотя бы одну пару чисел x, y , что $ax + by = \gcd(a, b)$ — после этого мы просто домножим x, y на k и получим, что сможем представить c в таком виде.

Algorithm 6 Расширенный алгоритм Евклида.

```

1: function EXTENDED_gcd(int  $a$ , int  $b$ )      ▷ — возвращаем тройку чисел  $(x, y, \gcd(a, b))$ .
2:   if  $b = 0$  then
3:     return  $(1, 0, a)$ ;
4:    $(x', y', d) \leftarrow \text{EXTENDED\_gcd}(b, a \bmod b)$ 
5:   return  $(y', x' - \lfloor \frac{a}{b} \rfloor y', d)$ 

```

Лемма 1. Для произвольных неотрицательных чисел a и b ($a \geq b$) расширенный алгоритм Евклида возвращает целые числа x, y, d , для которых $\gcd(a, b) = d = ax + by$.

Доказательство. Если не рассматривать x, y в алгоритме, то такой алгоритм полностью повторяет обычный алгоритм Евклида. Поэтому алгоритм 3-им параметром действительно вычислит $\gcd(a, b)$.

Про корректность x, y будет вести индукцию по b . Если $b = 0$, тогда мы действительно вернём верное значение. Шаг индукции: заметим, что алгоритм находит $\gcd(a, b)$, произведя рекурсивный вызов для $(b, a \bmod b)$. Поскольку $(a \bmod b) < b$, мы можем воспользоваться предположением индукции и заключить, что для возвращаемых рекурсивным вызовом чисел x', y' выполняется равенство:

$$\gcd(b, a \bmod b) = bx' + (a \bmod b)y'$$

Понятно, что $a \bmod b = a - \lfloor \frac{a}{b} \rfloor b$, поэтому

$$\begin{aligned} d = \gcd(a, b) &= \gcd(b, a \bmod b) = bx' + (a \bmod b)y' = \\ &= bx' + \left(a - \left\lfloor \frac{a}{b} \right\rfloor b\right)y' = ay' + b\left(x' - \left\lfloor \frac{a}{b} \right\rfloor y'\right) \end{aligned}$$

□

Пример 1. Мы умеем с помощью расширенного алгоритма Евклида вычислять обратные остатки по простому модулю (в поле \mathbb{F}_p). Действительно, если $(a, p) = 1$ то существуют x, y , что $ax + py = 1$, а значит в поле \mathbb{F}_p — $ax = 1$, откуда $x = a^{-1}$.

Алгоритм быстрого возведения в степень по модулю

Хотим вычислить $a^b \bmod p$. Основная идея в том, чтобы разложить b в двоичную систему и вычислять только $a^{2^i} \bmod p$. Здесь будем предполагать, что операции с числами выполняются достаточно быстро. Приведём ниже псевдокод такого алгоритма:

Algorithm 7 Алгоритм быстрого возведения в степень.

```

1: function FAST_POW(int  $a$ , int  $b$ , int  $p$ ) ▷ — возвращаем  $a^b \bmod p$ .
2:   if  $b = 0$  then
3:     return 1;
4:   if  $b \bmod 2 = 1$  then
5:     return FAST_POW( $a, b - 1, p$ ) ·  $a \bmod p$ 
6:   else
7:      $c \leftarrow$  FAST_POW( $a, b/2, p$ )
8:     return  $c^2 \bmod p$ 

```

Корректность этого алгоритма следует из того, что $a^b = a^{b-1} \cdot a$ для нечетных b и $a^b = a^{b/2} \cdot a^{b/2}$ для четных b . Также мы здесь неявно пользуемся индукцией по b , в которой корректно возвращается база при $b = 0$.

От каждого числа b , если оно четно, мы запускаем наш алгоритм от $b/2$, а если оно нечетно, то от $b - 1$, откуда получаем, что количество действий, совершенным нашим алгоритмом будет не более, чем $2 \log b = O(\log b)$.

Замечание 1. На самом деле быстрое возведение в степень работает на всех ассоциативных операциях. Например, если вы хотите вычислить A^n , где A — квадратная матрица, то это можно сделать тем же самым алгоритмом за $O(T(m) \log n)$, где $T(m)$ асимптотика перемножения матриц $m \times m$.

Китайская теорема об остатках и её вычисление

Китайская теорема об остатках звучит так — пусть даны попарно взаимно простые модули и числа r_1, \dots, r_n . Тогда существует единственное с точностью по модулю $a_1 \dots a_n$ решение такой системы:

$$\begin{cases} x \equiv r_1 \pmod{a_1} \\ x \equiv r_2 \pmod{a_2} \\ \vdots \\ x \equiv r_n \pmod{a_n} \end{cases}$$

Доказательство. Докажем и предьявим сразу алгоритм вычисления за $O(n \log \max(a_1, \dots, a_n))$.

Пусть $x = \sum_{i=1}^n r_i M_i M_i^{-1}$, где $M_i = \frac{a_1 \dots a_n}{a_i}$, M_i^{-1} это обратное к M_i по модулю a_i (такое всегда найдётся из попарной взаимной простоты). Прошу заметить, что такое число мы можем вычислить за $O(n \log \max(a_1, \dots, a_n))$ (см. пример в расширенном алгоритме Евклида).

Докажем, что это число подходит по любому модулю a_i .

$$x \equiv \sum_{j=1}^n r_j M_j M_j^{-1} \equiv r_i M_i M_i^{-1} \equiv r_i \pmod{a_i}$$

Второе равенство следует из того, что $a_i \mid M_j$ при $j \neq i$ (из построения).

Докажем единственность решения по модулю. Пусть x, x' — различные решения данной системы, тогда $0 < |x - x'| < a_1 \dots a_n$ и $|x - x'|$ делится на $a_1 \dots a_n$, что невозможно, так как ни одно положительное число до $a_1 \dots a_n$ не делится на $a_1 \dots a_n$. \square

Решето Эратосфена

Решето Эратосфена — это один из первых алгоритмов в истории человечества. Он позволяет найти все простые числа на отрезке от $[1; n]$ за $O(n \log \log n)$, а разложить все числа на простые множители за $O(n \log n)$

В первом случае у нас задача состоит в том, чтобы вернуть 1, если число простое и 0, если непростое.

Предьявим псевдокод такого алгоритма:

Лемма 2. Алгоритм *Sieve_of_Eratosthenes* корректно оставит все простые числа.

Доказательство. Докажем по индукции по n . База $n = 2$ очевидна.

Переход $n \rightarrow n + 1$. Заметим, что наш алгоритм и корректно завершит для n чисел, потому что мы только расширяем область рассматриваемых чисел.

Если $n + 1$ составное, тогда $n + 1 = p \cdot t$ для какого-то простого $p < n + 1$. По предположению индукции мы рассмотрим простое число p правильно, то есть удалим из массива все числа, которые кратны p , а значит и $n + 1$ мы правильно уберём.

Если $n + 1$ простое, то если мы его убрали на каком-то шаге, то оно делилось на то простое, которые мы рассматривали до этого, но это противоречит определению простых чисел. \square

Algorithm 8 Решето Эратосфена.

```

1: function SIEVE_OF_ERATOSTHENES(int  $n$ )           ▷ найти — массив  $prime_i$ , означающий
   характеристическую функцию простых чисел от 1 до  $n$ .
2:   for  $i \leftarrow 1$  to  $n$  do
3:      $prime_i \leftarrow true$ 
4:    $prime_1 \leftarrow false$ 
5:   for  $i \leftarrow 2$  to  $n$  do
6:     if  $prime_i = true$  then
7:        $j \leftarrow 2i$ 
8:       while  $j \leq n$  do
9:          $prime_j \leftarrow false$ 
10:         $j \leftarrow j + i$ 

```

Заметим, что алгоритм будет выполняться за время

$$\sum_{\substack{p \leq n, \\ p - \text{простое}}} \frac{n}{p}$$

Потому что для каждого простого числа мы рассматриваем в таблице все числа, кратные p . Можно оценить очень грубо и получим, что

$$\sum_{\substack{p \leq n, \\ p - \text{простое}}} \frac{n}{p} \leq \sum_{i=1}^n \frac{n}{i} \approx n \ln n + o(n) = O(n \log n)$$

Но используя свойства ряда $\sum_{\substack{p \leq n, \\ p - \text{простое}}} \frac{n}{p} \approx n \ln \ln n + o(n)$, следует, что алгоритм работает за

$O(n \log \log n)$, но факт про асимптотику этого ряда мы оставим без доказательства.

Если теперь первый раз, приходя в составное число в алгоритме, хранить его наименьший простой делитель, то рекурсивно мы можем разложить число на простые множители. Всего количество простых делителей у числа не может превышать $O(\log n)$ (так как самый наименьший простой делитель это двойка), поэтому разложение на множители будет выполняться за $O(n \log n)$.

Решето Эйлера

Составим двусвязный список из чисел от 2 до n , а также ещё массив длиной n с указателями на каждый элемент.

Будем идти итеративно: первый непросмотренный номер в списке берётся как простое число, и определяются все произведения с последующими элементами в списке (само на себя тоже умножим), пока не выйдем в произведении за пределы n . После этого удаляются все числа, которые мы вычислили (смотрим в массив указателей и удаляем по указателю за $O(1)$) и повторяем процедуру.

Лемма 3. После k шагов алгоритма останется первых k простых чисел в начале и в списке будут только числа взаимно простые с первыми k .

Доказательство. База при $k = 1$ очевидна. Просто убираем все четные числа.

Переход $k \rightarrow k + 1$.

Докажем, что следующим нерассмотренным элементом списка мы возьмём p_{k+1} . Действительно, простые числа мы не выкидываем, а значит следующим шагом после p_k мы возьмём число, не большее p_{k+1} , но по предположению индукции все числа от (p_k, p_{k+1}) были убраны, так как они составные и содержат в разложении только простые, меньшие p_{k+1} .

Предположим, что после ещё одного шага алгоритма у нас осталось число, кратное p_{k+1} (и большее p_{k+1}) (все числа, делящиеся на предыдущие простые до этого были убраны).

Тогда пусть это будет $m = p_{k+1} \cdot a$, $a > 1$. Если a содержит в разложении на простые хотя бы одно число, меньшее p_{k+1} , то получим противоречие, так как все числа не взаимно простые с p_1, \dots, p_k по предположению индукции были убраны.

Значит a содержит в разложении на простые числа, не меньшие p_{k+1} , а значит $a \geq p_{k+1}$ и это число ещё было в списке, значит мы это число уберем, противоречие. \square

Если мы вдруг на шаге алгоритма получили в умножении число, которое мы уже убрали, то значит у этого числа есть меньший простой делитель, чем p_k , но по доказанной лемме у нас все такие числа к k -ому шагу были убраны. Значит каждое составное число мы рассмотрим ровно 1 раз и ровно 1 раз уберем за $O(1)$.

Также по лемме получаем, что в начале списка останутся только простые числа.

Простые числа мы тоже рассматриваем по 1 разу в нашем алгоритме, значит общая сложность решета Эйлера будет $O(n)$.

Наивная факторизация числа за $O(\sqrt{n})$

На данный момент не существует алгоритма факторизации числа за полином от размера числа, а не от значения. Здесь мы рассмотрим наивный алгоритм факторизации числа. На следующей лекции рассмотрим ρ -метод Полларда (**UPD** так и не рассмотрели), который работает за $O(\sqrt[4]{n})$.

Пусть $n' = n$. Будем перебирать от 2 до $\lceil \sqrt{n'} \rceil$ числа и пока текущее n делится на данное число, делим n на это число.

Легко показать, что делим мы только на простые числа (иначе мы поделили бы на меньшее простое несколькими шагами раньше).

В конце n будет либо 1 (тогда факторизация удалась), либо простым. Составным оно не может быть, иначе $n = ab$, $a, b > 1$ и $a, b > \lceil \sqrt{n'} \rceil$, так как на все числа, меньшие корня, мы поделили.

Осталось оценить, сколько операций раз мы обращаемся к циклу *while*. В нём мы делаем суммарно не более, чем $O(\log n + \sqrt{n})$ действий, так как сумма степеней при разложении числа не более, чем $O(\log n)$ (см. выше). Ну а также обращаемся по 1 разу каждый шаг внешнего цикла.

Лекция 5 от 27.09.2016. RSA, продолжение некоторых теоретико числовых алгоритмов, некоторые комбинаторные оптимизации

«На самом деле, если этот алгоритм (разложение на простые) придумают за полиномиальное время, можно спокойно идти и покупать попкорн и смотреть, как рушится этот мир. Только не забудьте перед этим снять все деньги с банковских карточек.»

Глеб

Прим. Я в этой лекции поменял местами темы, чтобы было легче воспринимать материал.

Предисловие

Теория чисел с появлением алгоритмов, а особенно криптографии, приобрела новую «жизнь». Теперь простые числа, разложение на простые множители являются важными алгоритмами даже в нашей повседневной жизни. Сначала поговорим о самих простых числах, потом о криптографии.

Тест Рабина-Миллера на проверку простоты числа

В той же самой криптографии есть необходимость в генерировании длинных простых чисел. Благо простые числа встречаются не так редко. Пусть функция распределения простых чисел будет $\pi(n)$ — количество чисел, не больших n , которые являются простыми. Есть теорема, о которой мы уже упоминали

$$\lim_{n \rightarrow +\infty} \frac{\pi(n)}{n / \ln(n)} = 1$$

То есть если взять случайно $\ln(n)$ чисел до n , то математическое ожидание, что хотя бы одно из них будет простым равно 1.

Поэтому, чтобы сгенерировать большое простое число, надо уметь проверять за полином от числа (было бы вообще прекрасно), является ли число простым. Был найден алгоритм, который проверяет это свойство за полином от размера числа.

Но мы рассмотрим достаточно эффективный вероятностный алгоритм проверки числа на простоту. Вспомним малую теорему Ферма, которая нам гласит:

$$a^{p-1} \equiv 1 \pmod{p} \text{ для всех простых чисел } p.$$

Определение 1. Назовём число n псевдопростым по основанию a , если $(a, n) = 1$ и $a^{n-1} \equiv 1 \pmod{n}$.

Поэтому первый наш алгоритм будет такой (будем проверять нечётные n и $a = 2$):

Algorithm 9 Проверка на псевдопростоту по основанию 2.

```

1: function PSEUDOPRIME(int  $n$ )
2:   if FAST_POW(2,  $n - 1$ ,  $n$ )  $\neq 1$  then
3:     return Составное ▷ 100% составное
4:   return Простое ▷ молимся и надеемся, что тут действительно простое

```

Но, к сожалению, существуют и составные числа, которые удовлетворяют этому алгоритму и нам выведется, что число простое, а на самом деле нет. Наименьшие из них это 341, 561... и так далее. Поменять основание тоже не вариант, так как существуют числа Карлмайка, которые псевдопростые по любому основанию. Но было доказано, что с ростом n вероятность, что непростое число является псевдопростым стремится к 0.

К ещачеью сожалению, мы не будем доказывать вероятность ошибки этого алгоритма (она составляет не более 0.5), поэтому если запустить этот алгоритм k раз, то вероятность ошибки будет равна 2^{-k} . Собственно, алгоритм:

Algorithm 10 Тест Миллера-Рабина, считаем, что $n - 1 = 2^t u$, где u нечетно и $t \geq 1$

```

1: function MILLER_RABIN(int  $n$ )
2:    $a \leftarrow \text{random\_integer}(1, n - 1)$ 
3:    $x_0 \leftarrow \text{FAST\_POW}(a, u, n)$ 
4:   for  $i \leftarrow 1$  to  $t$  do
5:      $x_i \leftarrow x_{i-1}^2 \bmod n$ 
6:     if  $x_i = 1$  и  $x_{i-1} \neq 1$  и  $x_{i-1} \neq n - 1$  then
7:       return Составное
8:   if  $x_t \neq 1$  then
9:     return Составное
10:  return Простое

```

Лемма 1. Если алгоритм как-то вышел на строчки 7 или 9, то число действительно составное.

Доказательство. Заметим, что $x_i \equiv a^{2^i u} \bmod n$, так как $x_0 \equiv a^u \bmod n$ (база индукции) и $x_i = x_{i-1}^2 \bmod n$ (переход индукции), а значит $x_i \equiv a^{2^{i-1} u} \cdot a^{2^{i-1} u} \equiv a^{2^i u} \bmod n$.

Поэтому если $x_i = 1$ и $x_{i-1} \neq 1$ и $x_{i-1} \neq n - 1$, то $n \mid x_{i-1}^2 - 1$, то есть $(x_{i-1} - 1)(x_{i-1} + 1) = 0$ в \mathbb{Z}_n . То есть у нас нетривиальные делители нуля, а из курса алгебры известно, что в поле (а при простых n — \mathbb{Z}_n поле) их нет, поэтому перед нами составное число.

Если $x_t \neq 1$, то просто-напросто не выполняется малая теорема Ферма и тогда n точно составное. □

Лемма 2. Количество таких a , на которых алгоритм выдаст «Составное» не меньше $\frac{n-1}{2}$ при составном нечетном n .

Именно этот факт мы оставим без доказательства (Прим. на самом деле, он не очень сложный, видимо, Глебу было лень). И именно он нам даёт ошибку не более 0.5.

RSA, криптография

Криптографическую систему с открытым ключом можно использовать для шифровки сообщений, которыми обмениваются 2 партнера (Алиса и Боб), чтобы посторонние люди (Ева в дальнейшем), даже перехватившие сообщения, не могли его расшифровать. Также некоторая система позволяет подписывать свои подписи. Кто угодно без труда может её проверить, но подделать никак.

Давайте уже перейдём к обсуждению различных систем.

Но для начала несколько определений. У Алисы есть ключи P_A, R_A , у Боба P_B, R_B — публичные и приватные соответственно (на самом деле это функции, которые что-то вычисляют). Алиса и Боб хранят приватные ключи у себя, а с открытыми можно делать что угодно. Будем считать, что Алиса и Боб передают двоичные последовательности. Также будем считать, что $M = P_A(R_A(M)) = R_A(P_A(M))$ и $M = P_B(R_B(M)) = R_B(P_B(M))$. Также, чтобы шифрование имело смысл, надо, чтобы секретные ключи владельцы умели вычислять быстро, и чтобы по открытому ключу было очень сложно вычислить обратное преобразование. На этом и держится весь алгоритм. Рассмотрим пример:

Боб хочет отправить сообщение M Алисе, зашифрованное так, чтобы для Евы оно выглядело как ужасный набор символов:

- Боб получает открытый ключ Алисы P_A любым способом;
- Боб шифрует сообщение, которое знает только он, как $C = P_A(M)$;
- Алиса, когда получает сообщение C , расшифровывает своё сообщение с помощью секретного ключа.

Функции обратные, поэтому вычисления будут корректными. Но, к сожалению, такая система плоха тем, что, перехватив сообщения, Ева может их подменивать. Поэтому часто используют ещё и цифровые подписи:

Пусть Алиса хочет отправить сообщение M' Бобу:

- Алиса вычисляет свою подпись с помощью своего секретного ключа. $\sigma = R_A(M')$;
- Алиса отправляет пару Бобу (M', σ) ;
- Боб может легко убедиться, что это действительно Алиса, с помощью открытого ключа, вычислив $P_A(\sigma)$ и сравнив с M' .

В данном случае никакая Ева не страшна в подмене сообщения, так как она не может вычислить $S_A(M')$ ни для какого M' .

Такие подписи позволяют проверять целостность сообщений. Но всё равно есть проблема — Ева знает содержания сообщений. Можно взять ещё 1 ключ, который шифрует по 1-ой схеме сообщения, которые мы передаём по 2-ой схеме. И тогда Ева, даже получив перехваченное сообщение, во-первых, не сможет понять, какой парой оно было зашифровано, то есть дешифровка невозможна за разумное время, да и подмена тоже, так как там применяется к сообщению сложный ключ.

Проблема остаётся одна — что Алиса и Боб должны обмениваться ключами, чтобы Ева не могла подменить открытые ключи. Но, к сожалению, невозможно спрятаться от Евы, если быть совсем параноиком. Она всегда может подменять вам ключи, где бы вы ни находились. Поэтому фактор личной встречи должен быть. И самая большая «insecurity» находится именно в том, что Ева внедряется работать к Алисе, чтобы разузнать, а то и подменять ключи для Боба.

Давайте уже, наконец-то поговорим о способах шифрования:

- Самая старая система это шифр Цезаря. Она просто переставляет по циклу символы в алфавите, что конечно, же ломается за $O(k)$, где k — размер алфавита
- Взять случайную перестановку алфавита. Да, задача уже сложнее, но если это какой-нибудь язык, то можно из статистических параметров восстанавливать символы, что значительно сократит перебор. Не годится.
- Например, выравнивать двоичные сообщения и брать случайную перестановку, применяя её ко всем сообщениям. Тогда нетрудно убедиться, что за $O(\log n)$ действий мы сможем понять, какой символ, где стоит. Просто посмотреть, куда переходят единицы и нули. На непонятных случайных сообщениях математическое ожидание того, чтобы разобрать, где что стоит, будет $O(\log n)$.

Все примеры сверху так или иначе зависят от человеческого фактора или для них легко подобрать обратную функцию. Рассмотрим криптографическую систему RSA (Rivest–Shamir–Adleman public-key cryptosystem).

1. Случайным образом выбираются 2 больших простых числа $p \neq q$. Мы уже обсуждали выше, что это сделать легко достаточно.
2. Вычисляется $n = pq$ (что можно сделать тоже не очень сложно алгоритмом Карацубы или преобразованием Фурье).
3. Выбирается маленькое нечетное число e , взаимно простое с $\varphi(n) = (p-1)(q-1)$ из-за мультипликативности.
4. Вычисляем число $d = e^{-1} \bmod \varphi(n)$. Это можно сделать расширенным алгоритмом Евклида.
5. Пара $P = (e, n)$ будет открытым ключом.
6. Пара $S = (d, n)$ закрытым.

Теперь в качестве сообщения мы передаём сообщение $P(M) = M^e \bmod n$.

Обратное шифрование будет равно $S(C) = C^d \bmod n$.

Лемма 3 (Корректность RSA). *Докажем, что это взаимно обратные функции.*

Доказательство. Видно, что $P(S(M)) = S(P(M)) = M^{ed} \bmod n$.

Так как e, d взаимно обратные по модулю $\varphi(n)$, то $ed = 1 + k(p-1)(q-1)$

$$M^{ed} \equiv M^{1+k(p-1)(q-1)} \equiv M \cdot M^{k(q-1)\varphi(p)} \bmod p$$

$$M^{ed} \equiv M^{1+k(p-1)(q-1)} \equiv M \cdot M^{k(p-1)\varphi(q)} \bmod q$$

Малая теорема Ферма имеет очень простое следствие, что для любых чисел $M^p \equiv M \pmod p$ (предлагается это доказать самостоятельно). Поэтому в обоих равенствах в арифметике это просто эквивалентно M , то есть

$$M^{ed} \equiv M \pmod p \text{ и } M^{ed} \equiv M \pmod q$$

А значит по легкому следствию из основной теоремы арифметики $M^{ed} \equiv M \pmod n$. \square

Основная сложность в том, что зная $n, e, M^e \pmod n$ практически невозможно найти M . Перебрать все M может занять экспоненциальную сложность, а разложение на множители n и вычисление d оказалось очень сложной задачей, которая пока не решается за полиномиальное время.

Байка от Глебаса: На самом деле вся теория по шифрованию в интернете появилась лет 5-6 назад. Раньше кто угодно мог перехватывать сообщения вашей почты, платить в интернете было опасно (если вообще можно было). Я только однажды покупал что-то не через безопасное соединение в интернете. Я очень хотел ту пиццу, мне было без разницы тогда на безопасность.

Комбинаторные оптимизации. Генетический алгоритм

На этой лекции был треш. Я честно не знаю, как это конспектировать. Напишу то, в чём я разобрался.

На лекции были рассмотрены метод Ньютона, отжига и генетический алгоритм. Всегда есть Google, поэтому первые 2 ищите там.

Генетический алгоритм позволяет решать некоторые трудные задачи методом ошибок за разумное время.

Есть несколько фаз алгоритма:

- **Создание популяции.** Обычно это какие-то случайные решения нашей задачи, которые могут иметь много ошибок.
- **Размножение.** Тут всё как у людей. Мы скрещиваем некоторые особи (обычно сильные особи) вместе, чтобы получить лучшее поколение.
- **Мутация.** Тут природа говорит, что мутации иногда хорошо влияют на организмы. Мы просто берём некоторые особи и мутируем их с помощью какого-то заранее определённого алгоритма. Да, могут получиться плохие особи, но есть вероятность, что получатся хорошие.
- **Отбор.** Мы отбираем самых лучших, те, кто пойдут дальше повторять этот процесс.

Иногда такой алгоритм приносит правильные решения.

Далее материала не было на лекции.

Приведём пример работы генетического алгоритма в задаче о правильной расстановке ферзей.

Берём какую-нибудь перестановку, что в строках и столбцах ровно по 1 ферзю. Генерируем, например, 100 таких перестановок.

Потом считаем количество ошибок, то есть количество пар, которые бьют друг друга (это можно сделать за $O(n)$, пройдясь по диагоналям).

После этого выбираем хороших особей — примерно половину, у которых ошибок меньше всего. Берем любые 2, смотрим, какие элементы у них совпадают, оставляем их в предположении, что они являются «сильными» генами, а остальное всё перемешиваем между собой. Так делаем с каким-то количеством пар, потом выбираем, например, 1 особь, мутируем её — меняем 2 любых элемента местами.

Опять же считаем количество ошибок и выбираем лучших 100. Повторяем алгоритм, пока не найдём нужное решение.

Генетический алгоритм ничего не доказывает, он лишь может работать как природа. Мы можем находить какие-то хорошие решение с его помощью за более разумное время.

Байка от Глебаса: В конце 40-ых годов появилась компания RAND, которая одна из первых вообще придумала работать со случайными числами (*Прим. и даже сгенерировала огромный список случайных цифр*). Вообще, это была первая компания, которая моделировала процессы с помощью случайных чисел.