На этом семинаре мы рассмотрим модели, формализующие случайность в теории сложности.

Вероятностые классы сложности

Вероятностная машина Тьюринга (ВМТ) представляет собой обычную МТ, которой в некоторых состояниях разрешено совершать переходы в зависимости от бросания монеты. В отличие от НМТ, можно довольно легко представить практическую реализацию такой конструкции. Как и в случае НМТ, мы можем избавиться от всякого индетерминизма, если будем считать, что кто-то уже совершил все броски монеты заранее и любезно предоставил нам слово r из случайных бит.

Определение 1. *Язык L принимается ВМТ в слабом смысле (стандарт Монте-Карло) если для любого $x \in \Sigma^*$ вероятность получения ошибочного ответа на вопрос $x \in^? L$ не больше $1/3$.*

Определение 2. *Язык L принимается ВМТ в сильном смысле (стандарт Лас-Вегас) если для любого $x \in \Sigma^*$ вероятность получения ошибочного ответа на вопрос $x \in^? L$ равна 0.*

Определение 3. *Языки, принимаемые ВМТ в слабом смысле за полиномиальное в среднем время образуют класс* **BPP** *(bounded-error probabilistic polynomial).*

$$L \in \mathbf{BPP} \iff \exists \mathcal{A}(\cdot,\cdot), \exists c \in \mathbb{N} : \forall w \in \Sigma^* \hookrightarrow \underset{r}{\mathbb{E}} T_{\mathcal{A}}(w,r) = O(|w|^c) \text{ и } \underset{r}{\mathbb{P}}(\mathcal{A}(w,r) = L(w)) \geq 2/3$$

Определение 4. *Языки, принимаемые ВМТ в сильном смысле за полиномиальное в среднем время образуют класс* **ZPP** *(zero-error probabilistic polynomial).*

$$L \in \mathbf{ZPP} \iff \exists \mathcal{A}(\cdot,\cdot), \exists c \in \mathbb{N} : \forall w \in \Sigma^* \hookrightarrow \underset{r}{\mathbb{E}} T_{\mathcal{A}}(w,r) = O(|w|^c) \text{ и } \underset{r}{\mathbb{P}}(\mathcal{A}(w,r) = L(w)) = 1$$

Определение 5. *Языки, для которых можно построить ВМТ, которая за полиномиальное в среднем время принимает каждое слово из языка с вероятностью не меньше $1/2$ и отвергает любое слово, не входящее в язык, образуют промежуточный класс* **RP** *(randomized polynomial).*

$$L \in \mathbf{RP} \iff \exists \mathcal{A}(\cdot,\cdot), \exists c \in \mathbb{N} : \forall w \in \Sigma^* \hookrightarrow \underset{r}{\mathbb{E}} T_{\mathcal{A}}(w,r) = O(|w|^c) \text{ и } \underset{r}{\mathbb{P}}(\mathcal{A}(w,r) = L(w)) \geq (1/2)^{L(w)}$$

Про эти классы известно, что они вложены следующим образом:

$$\mathbf{P} \subset \mathbf{ZPP} \subset \mathbf{RP} \subset \mathbf{BPP}$$

1. (1б) Покажите, что $\mathbf{RP} \cap \mathbf{co-RP} = \mathbf{ZPP}$.

2. (2б) Покажите, что класс **BPP** не изменится если:

1. В определении константу $1/3$ заменить на ε такой что $0 \leq \varepsilon < 1/2$.
2. Полиномиальное в среднем число шагов заменить на полиномиальное число шагов.

3. (4б) Рассмотрим числа $p, q \in \mathbb{Q} \cap (0, 1)$. Имеется генератор случайных чисел, который выдаёт 1 с вероятностью p и 0 с вероятностью $1 - p$. Предложите алгоритм, который, используя данный генератор, будет возвращать 1 с вероятностью q и 0 с вероятностью $1 - q$ для:

1. $p = 1/2$, $q = 1/3$. Алгоритм должен работать за $O(1)$ в среднем.
2. $p = 1/3$, $q = 1/2$. Алгоритм должен работать за $O(1)$ в среднем.
3. $p = a/b$, $q = c/d$. Алгоритм должен работать за $O(\log abcd)$ в среднем.

Пример 1. *Одной из известных задач, лежащих в классе* **BPP**, *о принадлежности классу* **P** *которой ничего известно не было, была задача проверки числа на простоту. Для этой задачи существовало множество полиномиальных тестов, таких как тест Миллера-Рабина, но только в 2002 году появился детерминированный полиномиальный алгоритм AKS, предложенный индийскими математиками Agrawal, Kayan и Saxena.*

Пример 2. *Пусть $f(x_1, \dots, x_n)$ – полиномиальное выражение с n переменными. Нужно проверить, что это выражение является тождественным нулём. Например, $x^2 - y^2 + (y-x)(x-y)$ – тождественный нуль, а $x^2 + y^2$ – нет. Вероятностное решение – подставить произвольный набор $x_1, \dots, x_n$ и проверить, что получился нуль показывает, что язык таких выражений лежит в* $\mathbf{co}-\mathbf{RP}$*, что обосновывается следующей леммой:*

Вероятностная проверка равенств

Лемма 1 (Шварца-Зиппеля)**.** *Пусть $f(x_1, \dots, x_n)$ – не равный тождественно нулю многочлен степени не выше k и пусть случайные величины $\xi_1, \dots, \xi_n$ независимо и равномерно распределены на $[0, N)$. Тогда $\mathbb{P}(f(\xi_1, \dots, \xi_n) = 0) \le \frac{kn}{N}$.*

Доказательство. Утверждение верно для $n = 1$, так как нетривиальный многочлен степени n имеет не больше n корней. Пусть $n > 1$, разложим многочлен по x_1:

$$f(x_1, \dots, x_n) = f_0 + f_1 x_1 + f_2 x_1^2 + \dots + f_t x_1^t$$

Многочлены $f_0, \dots, f_t$ не зависят от x_1, а f_t не равен нулю тождественно. Поэтому:

$$\mathbb{P}(f = 0) = \mathbb{P}(f = 0 | f_t = 0)\mathbb{P}(f_t = 0) + \mathbb{P}(f = 0 | f_t \ne 0)\mathbb{P}(f_t \ne 0) \le \mathbb{P}(f_t = 0) + \mathbb{P}(f = 0 | f_t \ne 0)$$

По индукции $\mathbb{P}(f_t = 0)$ не больше $\frac{t(n-1)}{N}$, а $\mathbb{P}(f = 0 | f_t \ne 0)$ не больше $\frac{t}{N}$, так как в каждом рассматриваемом элементарном исходе f будет нетривиальным многочленом от x_1 и применим случай $n = 1$. Итого получаем оценку в $\frac{tn}{N} \le \frac{kn}{N}$. □

Пример 3. *Рассмотрим матричное равенство $C = AB$. Научимся рандомизированно проверять его. Пусть x – случайный n-мерный вектор с компонентами из $\mathbb{Z} \cap [0, N)$. Для проверки равенства сравним векторы $A(Bx)$ и Cx. Это можно сделать за $O(n^2)$ арифметических операций над числами, длина записи которых составляет $O(\log nh)$, где n – порядок матрицы, h – размер наибольшего из чисел в матрице. Если равенство не выполнено, то сигнализируем об ошибке, в противном случае либо оно верно, либо мы подобрали плохой вектор x.*

4. (4б) Рассмотрим этот пример подробнее.

1. Принадлежность к какому классу сложности показывает описанный алгоритм?
2. Оцените вероятность того, что равенство не выполнено в случае $A(Bx) = Cx$.
3. Каким нужно выбрать N, чтобы гарантировать вероятность ошибки меньше p?
4. Тот же вопрос если можно провести проверку несколько раз и нужно минимизировать общую *битовую* сложность вычислений, сохранив ограничение вероятности в p.

Пример 4. *Рассмотрим двудольный граф, в каждой доле которого по n вершин. Научимся рандомизированно проверять, есть ли в нём полное паросочетание. Сопоставим этому графу матрицу, такую что a_{ij} равно 1 если между вершиной i первой доли и вершиной j второй доли есть ребро и 0 в противном случае. Такая матрица называется матрицей Эдмондса. Её перманент матрицы равен количеству полных паросочетаний в графе. Но задача вычисления перманента очень сложна, она является* $\#\mathbf{P}$*-полной*[1]*.*

Тем не менее если нам нужно только проверить, что перманент не равен 0 мы можем использовать следующий подход: Заменим a_{ij} равные единице на независимые переменные x_{ij}, а остальные оставим нулями. Тогда перманент матрицы равен нулю тогда и только тогда когда определитель матрицы как многочлен от x_{ij} равен 0 тождественно. Поэтому мы можем подставлять вместо x_{ij} случайные числа из $\mathbb{Z} \cap [0, N)$ и считать многочлен в этих точках.

[1] Класс $\#\mathbf{P}$ образуют перечислительные задачи, ассоциированные с задачами класса $\mathbf{NP}$. Например, “сколько в графе есть различных гамильтоновых путей?”. Формально можно сказать, что если $\mathcal{A}(\cdot, \cdot)$ – верификатор для задачи из $\mathbf{NP}$, то порождаемая им задача в классе $\#\mathbf{P}$ формулируется как “сколько существует слов s для данного слова w таких что $\mathcal{A}(w, s) = 1$?”. В отличие от рассматриваемых нами классов это класс функциональных задач.

5. (6б) Рассмотрим этот пример подробнее.

1. Докажите, что перманент равен 0 $\iff$ детерминант как многочлен от x_{ij} равен 0.
2. Оцените вероятность ошибки алгоритма и битовую сложность его работы.
3. Предложите полиномиальный в среднем алгоритм, который не только проверит, есть ли полное паросочетание в графе, но и найдёт его в случае положительного ответа.

1*. (3б) Предложите рандомизированный полиномиальный алгоритм проверки того, что во взвешенном двудольном графе есть полное паросочетание, вес которого делится на заданное число $k = \text{const}$.

Тест Ферма

Напомним следующее утверждение, называемое *малой теоремой Ферма*:

Лемма 2. *Пусть p – простое число, тогда для любого целого числа $a \in [1; m)$ верно:*

$$a^{p-1} \equiv 1 \pmod p$$

Из этой теоремы естественным будет предложить следующий тест – возьмём произвольное число a от 1 до $m-1$ и проверим, что $a^{m-1} \equiv 1 \pmod m$. Эффективность такого теста можно оценить следующим утверждением:

Лемма 3. *Пусть есть хотя бы одно b взаимно простое с m такое что $b^{m-1} \not\equiv 1 \pmod m$. Тогда хотя бы половина чисел от 1 до $m-1$ обладают этим свойством.*

6. (2б) Докажите это.

Этот результат значит, что на большинстве составных чисел один запуск теста Ферма покажет, что число составное с вероятностью хотя бы 50%, то есть, 10 безрезультатных запусков дало бы гарантию в $\approx 99.94\%$, что число действительно простое.

К сожалению, это утверждение верно только если хотя бы одно такое число b есть, на деле же существуют так называемые *числа Кармайкла*, являющиеся составными, но для всех взаимно простых с ними числами выполнена теорема Ферма. Наименьшее из чисел Кармайкла – 561.

2*. (1б) Покажите, что чисел Кармайкла бесконечно много.

Тест Миллера-Рабина

К счастью, теорему Ферма можно усилить до следующего утверждения:

Лемма 4. *Пусть m – простое число и $m - 1 = 2^s d$, где d – нечётно. Тогда для любого целого числа $a \in [1; m)$ выполнено хотя бы одно из следующих условий:*

1. $a^d \equiv 1 \pmod m$
2. *Существует целое $r \in [0; s)$ такое что* $a^{2^r d} \equiv -1 \pmod m$

7. (2б) Докажите это.

По теореме Рабина у любого *нечётного* составного числа n не больше $\frac{\varphi(n)}{4}$ свидетелей простоты (чисел a, для которых выполнены условия леммы). Значит, если мы проверили k чисел и все они оказались свидетелями простоты, то вероятность того, что число на самом деле составное не больше 4^{-k}. Кроме того, для большинства чисел эта оценка будет составлять 8^{-k}. Реализация:

```
bool miller_rabin(int m) {
  int s = 0, d = m - 1;
  while(d % 2 == 0) {
    s++, d /= 2;
  }
  int a = 1 + rand() % (m - 1);
  int x = powmod(a, d, m);
  if(x == 1) {
    return true;
  }
  for(int r = 0; r < s; r++) {
    if(x == m - 1) {
      return true;
    }
    x = x * x % m;
  }
  return false;
}
```

Одна итерация теста требует $\log m$ умножений, соответственно время работы одной итерации на стандартных типах данных будет равно $O(\log m)$, а на длинных числах $O\left(\dfrac{\log^3 m}{\log \log m}\right)$ в случае умножения "в столбик" или $O\left(\log^2 m\right)$ в случае использования быстрого преобразования Фурье.

3*. (2б) Докажите теорему Рабина. Оценка в $\frac{\varphi(n)}{2}$ оценивается в 1 балл.

Рандомизированная факторизации

Прежде, чем говорить о факторизацию рассмотрим, казалось бы, не связанную с ней задачу.

Нахождение цикла

Рассмотрим функцию $f(x)$, отображающую конечное множество в себя и последовательность чисел $x_0 = \text{const}$, $x_k = f(x_{k-1})$. В какой-то момент найдутся i и j такие что $x_i = x_j$ после чего числа будет повторяться с периодом $|i - j|$. Нужно найти период этой последовательности.

Для этого можно использовать метод двух указателей, при этом первый будет идти с шагом 1, а второй – с шагом 2. Таким образом, на каждом шаге мы будем сравнивать элементы x_i и x_{2i} расстояние между которыми будет на каждом шаге увеличиваться на единицу. В какой-то момент мы обнаружим, что $x_i = x_{2i}$, что будет означать, что настоящий период является делителем i.

ρ-метод Полларда

Пусть мы хотим факторизовать число $m = pq$, где p – некоторый нетривиальный делитель n. Рассмотрим некоторый многочлен $g(x)$, с помощью которого генерируется псевдослучайная последовательность: $x_0 = \text{const}$, $x_k \equiv g(x_{k-1}) \pmod m$. Помимо данной последовательности будем рассматривать последовательность $y_k \equiv x_k \pmod p$, которую мы не можем считать в явном виде, так как не знаем p. Тем не менее, обе последовательности будут периодическими (возможно, с предпериодом) в силу конечности множеств значений x_k и y_k.

Так как мы считаем последовательность псевдослучайной, согласно парадоксу дней рождения первое повторение в x_k случится когда у нас будет $O(\sqrt{m})$, а в y_k – ещё раньше, на индексах порядка $O(\sqrt{p})$. Когда же последовательность y_k зациклится мы, хоть и не будем в точности знать её, сможем это определить – если $y_i \equiv y_j \pmod p$, то $x_i - x_j \equiv y_i - y_j \equiv 0 \pmod p$, то есть, число $x_i - x_j$ будет делиться на p, но не на m, так как последнее значило бы, что в x_k уже случилось повторение, которое ожидается значительно позже. Из этого следует, что посчитав

$\gcd(x_i - x_j, m)$ мы найдём некоторое число p', которое делит m и делится на p. Найдя его, мы можем рекурсивно запуститься от p' и m/p', чтобы найти все простые делители m. Само же повторение можно искать описанным выше алгоритмом. Реализация:

```
int g(int x, int m) {
    return (x * x + 1) % m;
}

int rho(int m) {
    int x = 1 + rand() % (m - 1), y = x, d = 1;
    do {
        x = g(x, m);
        y = g(g(y, m), m);
        d = gcd(abs(x - y), m);
    } while(d == 1);
    return d;
}
```

Обратите внимание, что описанные выше рассуждения предполагают, что числа $\sqrt{m}$ и $\sqrt{p}$ заметно отличаются, значит, подаваемые на вход алгоритму числа должны быть достаточно большими. В противном случае последовательность x_k может зациклиться одновременно с y_k, что будет означать неудачу для нас (мы вернём $d = m$). Рекомендуется также предварительно проверить число на наличие сравнительно небольших делителей. Если алгоритм все равно потерпел неудачу, стоит попробовать запустить его ещё раз с другим x_0 или другим $g(x)$.

Алгоритм работает до первого повторения в последовательности y_k, что в нашей вероятностной трактовке значит $O(\sqrt{p})$, где p – наименьший простой делитель m, что эквивалентно $O(m^{1/4})$.

8. (4б) Считая x_k последовательностью независимо и равномерно распределённых случайных величин из $\mathbb{Z} \cap [0, m)$, найдите вероятность неудачи ρ-метода на составном числе m.