Problem A. Asynchronous Processor

Идея: Виталий Аксёнов
Разработка: Максим Туревский

Давайте для удобства добавим в начало операцию $A := 0$ (sync). Посмотрим, как в итоге меняется величина A: она как-то меняется, потом происходит последняя операция присвоения, а потом к ней что-то добавляется. Эта последняя операция присвоения — это либо последняя синхронная операция присвоения, либо любая из асинхронных операций присвоения.

Давайте найдём все значения, которые в конце может принимать A, а в конце просто выведем их количество.

Если мы переберём позицию p среди изначальных операций, на которую мы в итоге поставим последнюю применяемую к A операцию присвоения, то мы знаем множество значений, которые мы могли в этой операции присвоить. Теперь все операции прибавления после позиции p обязательно должны примениться к A, а среди операций прибавления до позиции p — синхронные не применяются, а для каждой асинхронной мы можем выбрать, применять её или нет. С помощью динамического программирования мы можем для фиксированной позиции p найти, какие суммы достижимы в таком случае. Сложность решения тогда составит $O(n^3v)$. Используя битовое сжатие (bitset), можно получить решение за $O(\frac{n^3v}{64})$, но это всё ещё недостаточно эффективно.

Чтобы ускорить это решение, заметим следующее. Пусть последняя применяемая к A операция присвоения — асинхронная. Будем рассматривать её возможные позиции p в порядке возрастания. При увеличении этой позиции одна из операций прибавления переходит из состояния «после позиции p» в состояние «до позиции p». Посмотрим на эту операцию и на то, как меняется множество возможных итоговых значений A:

1. Если это операция “`+` v `sync`”, то это значит, что раньше это прибавление входило во все суммы, а теперь не входит ни в какие. Соответственно, все возможные суммы уменьшатся на v.

2. Если это операция “`+` v `async`”, то раньше v входило во все суммы, а теперь у нас есть выбор как применять эту операцию, так и не применять. Соответственно, для каждой возможной суммы x, сумма $x - v$ также становится возможной.

Случай, когда последняя применяемая к A операция присвоения — синхронная, обработаем отдельно.

Если поддерживать все допустимые значения с использованием битового множества (bitset), то это множество имеет размер $O(nv)$, и нужно совершить $O(n)$ действий с ним. Все множества, полученные в процессе выше, нужно объединить, чтобы получить итоговое множество. В итоге получим решение за $O(\frac{n^2v}{64})$.

Problem B. Bounding Boxes

Идея: Георгий Корнеев
Разработка: Георгий Корнеев

Как проверить, что коробка $w \times h \times d$ помещается в коробку $w' \times h' \times d'$? Отсортируем размерности обеих коробок по неубыванию: $w \le h \le d$ и $w' \le h' \le d'$. Тогда первая коробка помещается во вторую, если $w \le w'$, $h \le h'$ и $d \le d'$.

Из этого факта следует, что решение задачи следующее. Отсортируем по возрастанию размерности каждой коробки: $w_i \le h_i \le d_i$. Тогда максимальная коробка, помещающаяся во все упаковочные коробки, будет иметь размеры $\min_i w_i \times \min_i h_i \times \min_i d_i$.

Problem C. Compact Encoding

Идея: Дмитрий Штукенберг
Разработка: Дмитрий Штукенберг

В задаче (если опустить подробности) требуется преобразовать значение в 128-ричную систему счисления (7 битов как раз образуют одну 128-ричную цифру).

Это можно сделать стандартным образом, через печать остатков от последовательного деления на 128, но с двумя отличиями:

1. Ко всем результатам, кроме самого первого (младшие 7 бит), нужно прибавить 128.

2. Порядок байтов от старшего к младшему ("big-endian") требует в какой-то момент развернуть результат в обратном порядке (первый полученный остаток должен быть напечатан последним). Это можно сделать через векторы, а можно на уровне арифметики: сперва стандартным образом вычислим длину 128-ричного представления числа, а затем будем печатать отдельные значения в нужном порядке, тогда значение в 128-ричном разряде k будет равно $\lfloor \frac{n}{2^{7k}} \rfloor \bmod 128$.

Умножение и деление в данной задаче удобно производить с помощью битовых сдвигов, а взятие остатка от деления — через побитовую конъюнкцию.

Problem D. Defense Distance

Идея: Артем Васильев
Разработка: Михаил Первеев

Для начала заметим, что расстояние Левенштейна является метрикой, а значит для него выполняется неравенство треугольника. Поэтому, если $a > b + c$ или $b > a + c$ или $c > a + b$, построить требуемые строки невозможно.

Во всех остальных случаях, когда неравенство треугольника выполняется, построить требуемые строки возможно. Рассмотрим один из способов, как это можно сделать.

Для удобства будем строить строки s, t и u одинаковой длины. Разобьем каждую из трех строк на блоки длины x, y и z. Для удобства пронумеруем эти блоки слева-направо числами от 1 до 3. Таким образом, строки можно представить как конкатенации блоков: $s = s_1 + s_2 + s_3$, $t = t_1 + t_2 + t_3$, $u = u_1 + u_2 + u_3$, где $|s_1| = |t_1| = |u_1| = x$, $|s_2| = |t_2| = |u_2| = y$ и $|s_3| = |t_3| = |u_3| = z$.

Теперь построим строки по следующим правилам:

- Строки s и t будут различаться в каждом символе блоков 1 и 2, но совпадать в каждом символе блока 3;

- Строки s и u будут различаться в каждом символе блоков 1 и 3, но совпадать в каждом символе блока 2;

- Строки t и u будут различаться в каждом символе блоков 2 и 3, но совпадать в каждом символе блока 1.

Нетрудно заметить, что в таком случае $\mathrm{dist}(s,t) = x + y$, $\mathrm{dist}(s,u) = x + z$ и $\mathrm{dist}(t,u) = y + z$, где dist — расстояние Левенштейна.

Таким образом, осталось выбрать числа x, y и z таким образом, чтобы получились требуемые расстояния между строками. Для этого нужно решить систему из трех уравнений:

$$\begin{cases} x + y = a \\ x + z = b \\ y + z = c \end{cases} \tag{1}$$

Решение системы выглядит следующим образом:

$$\begin{cases} x = \dfrac{a+b-c}{2} \\ y = \dfrac{a+c-b}{2} \\ z = \dfrac{b+c-a}{2} \end{cases} \tag{2}$$

Если сумма $a+b+c$ четна, то x, y и z получатся целыми. В этом случае можно построить строки следующим образом: $s = a^x b^y b^z$, $t = b^x a^y b^z$, $u = b^x b^y a^z$, где a и b — произвольные различные символы, а запись c^p означает строку, равную символу c, повторенному p раз.

Теперь рассмотрим случай, когда сумма $a+b+c$ нечетна. В этом случае построим аналогичную конструкцию, округлив x, y и z вниз. Заметим, что теперь каждое из трех расстояний Левенштейна на 1 меньше, чем требуемое. Исправим это, дописав в конец каждой строки произвольный уникальный символ.

Также нужно не забыть о том, что строки должны быть непустыми, а x, y и z могут получиться равными нулю при $a = b = c = 0$. В этом случае достаточно вывести произвольные три непустые равные строки.

Problem E. Eight-Connected Figures

Идея: Захар Яковлев
Разработка: Захар Яковлев

Отметим, что максимальные ограничения в задаче $t = 50$, $n = 300$, а ограничение на количество запросов $30\,000$, то есть ровно $2nt$.

Первая идея, которая приходит в голову — поддерживать связную область одного цвета, но тогда только среднее число запросов будет $2nt$, а разброс значений приведет к превышению числа запросов.

Правильное решение — поддерживать одновременно связные области белого и черного цвета. При этом нужно делать запросы в клетки, соседние с обеими областями. Тогда область нужного размера получится меньше, чем за $2n$ запросов.

Отдельно надо аккуратно обработать специальные случаи: одна область может оказаться полностью окруженной другой. В этом случае необходимо «выкинуть» внутреннюю область и начать строить заново область такого цвета снаружи. За счет множественных тестов неудача в одном тесте компенсируется запасом в других. Так, вероятность ошибки в авторском решении меньше 10^{-40}.

Также бывают особые случаи, когда у каждой из областей есть соседи, но нет общих соседей:

```
.......
.WWWWW.
.WBBBW.
.WB.BW.
.WBBBW.
.WWWWW.
.......
```

Но такие случаи крайне маловероятны и на практике практически не встречаются. Также можно менять стартовую точку для обхода таких случаев.

Problem F. Faulty Fraction

Идея: Николай Дубчук
Разработка: Николай Будин

Вам дана строка из цифр s и целое число c. Известно, что существуют два целых положительных числа a и b, такие что $a \div b = c$ и $concat(a, b) = s$. Требуется найти любые подходящие a и b.

Обозначим за $l(x)$ длину числа x. Тогда $10^{l(x)-1} \le x < 10^{l(x)}$.

$$\begin{cases} 10^{l(b)-1} \le b < 10^{l(b)} \\ 10^{l(c)-1} \le c < 10^{l(c)} \end{cases} \implies$$

$$\implies 10^{l(b)+l(c)-2} \le b \cdot c < 10^{l(b)+l(c)} \implies$$

$$\implies l(b) + l(c) - 1 \le l(b \cdot c) \le l(b) + l(c);$$

$$a \div b = c \implies a = b \cdot c \implies l(a) = l(b \cdot c) \implies$$

$$\implies l(b) + l(c) - 1 \le l(a) \le l(b) + l(c) \implies$$

$$\implies (l(s) - l(a)) + l(c) - 1 \le l(a) \le (l(s) - l(a)) + l(c) \implies$$

$$\implies l(s) + l(c) - 1 \le 2 \cdot l(a) \le l(s) + l(c)$$

Заметим, что $2 \cdot l(a)$ — четное, поэтому оно может быть равно только одному из двух последовательных значений $l(s) + l(c) - 1$ и $l(s) + l(c)$. Таким образом, $l(a) = \lfloor \frac{l(s)+l(c)}{2} \rfloor$. Решение единственно. А раз нам гарантируется, что такие a и b должны существовать, это именно они.

Problem G. Games of Chess

Идея: Фёдор Ушаков
Разработка: Фёдор Ушаков

Рассмотрим задачу в терминах теории графов. Нам дан неориентированный связный граф, и нужно раскрасить его вершины в некоторое число цветов так, чтобы для каждой вершины u число вершин v того же цвета, соединённых с u ребром, было нечётным.

При нечётном n ответа не существует. Действительно, предположим обратное: пусть существует какая-то допустимая раскраска. Оставим в графе только те рёбра, концы которых имеют одинаковый цвет. После этой операции у каждой вершины степень нечётна, и так как вершин нечётное число, сумма степеней всех вершин нечётна. А в неориентированном графе сумма степеней всех вершин должна быть чётной, так как каждое ребро прибавляет к этой сумме 2. Противоречие.

Для чётного n ответ всегда существует, и его можно найти с помощью следующего алгоритма:

1. Построим дерево обхода в глубину (DFS).
2. Повторяем, пока есть вершины:
   - Возьмём самый глубокий лист v. Посмотрим на его родителя u. У u есть k детей, все они — листья.
   - Если k нечётно, то покрасим поддерево u в один цвет и отрежем от остального дерева.
   - Если k чётно, но есть хотя бы один лист, из которого есть хотя бы одно обратное ребро, ведущее наверх в некоторого предка вершины u, то этот лист переподвесим к ближайшей вершине, в которую из него есть ребро, а остальную часть поддерева отрежем так же, как в предыдущем пункте.

- Если k чётно, но рёбер наверх из листьев нет, то отрежем все эти листья и пометим, что их в конце надо будет покрасить так же, как вершину u, а саму вершину u оставим в дереве.

Можно показать, что раскраска графа по данному алгоритму является корректной. При аккуратной реализации сложность этого решения составит $O(n + m)$.

Problem H. High Score

Идея:	Дмитрий Якутов
Разработка:	Дмитрий Якутов

Посмотрим на состояние игры в произвольный момент времени. Пусть в мультисете в данный момент есть число $a = 2^t$. Сколько очков получил игрок за создание этого числа с нуля? Если $a = 2$, то игрок не получил очков. Иначе рассмотрим все числа, которые игрок добавлял в мультисет с помощью операции Insert, которые внесли вклад в текущее число a:

- Если при операциях Insert игрок добавлял только числа со значением 2, то игрок получил $Q(t) = (t - 1) \cdot 2^t$ очков: при операциях Merge с $x = 2$ игрок суммарно получил 2^t очков, с $x = 4$ – тоже 2^t очков, ..., с $x = a$ – тоже 2^t очков;
- Аналогично, если при операциях Insert игрок добавлял только числа 4, то игрок суммарно получил $P(t) = (t - 2) \cdot 2^t$ очков;
- Если часть операций Insert выполнялась со значением 2, а остальные – со значением 4, то игрок мог получить любое значение от $P(t)$ до $Q(t)$ с шагом 4 очка.

Из рассуждений выше есть одно исключение: если $t = k + 1$, то есть речь идет о максимальном возможном значении числа в мультисете. В этом случае невозможно получить это число, добавляя только 2 при Insert, так как для этого в какой-то момент на доске должны быть числа: $2, 2, 4, \ldots 2^k$, то есть $k + 1$ число. Максимальное возможное число очков, полученное при создании 2^{k+1} – это $R(t) = Q(t) - 4 = (t - 1) \cdot 2^t - 4$ очка.

Приведем алгоритм, с помощью которого можно брать любое число очков от 0 до $MaxScore(k) = \sum_{t=2}^{k+1} R(t)$, методом математической индукции по значению k.

Если $k = 2$, то максимальный возможный счет $\sum_{t=2}^{3}((t-1)\cdot 2^t - 4) = (1\cdot 4-4)+(2\cdot 8-4) = 0+12 = 12$. Способ набрать 12 очков проиллюстрирован в пояснении к сэмплу.

При $k \geq 3$. Пусть мы хотим набрать ровно h очков, где $h \mod 4 = 0$:

- Если $4 \leq h \leq MaxScore(k - 1)$, то наберем нужные очки, воспользовавшись индукционным предположением. Кол-во чисел в мультисете в любой момент времени не будет превышать $k - 1$, что нам подходит;
- Если $MaxScore(k - 1) + 4 \leq h \leq MaxScore(k - 1) + Q(k)$, то сначала сделаем так, в мультисете было только одно число 2^k, набрав максимальный возможный для этого числа счет $Q(k) = (k - 1) \cdot 2^k$ при создании этого числа, а затем доберем оставшиеся очки, используя не более $k - 1$ слотов в мультисете. Это можно сделать, так как $MaxScore(k - 1) + 4 \geq Q(k)$;
- Если $MaxScore(k-1) + Q(k) + 4 \leq h \leq MaxScore(k-1) + P(k+1)$, то сначала сделаем число 2^{k+1}, набрав счет $P(k+1)$, а затем доберем оставшиеся очки, воспользовавшись индукционным предположением. Это мы можем сделать, так как $MaxScore(k - 1) + Q(k) + 4 \geq P(k + 1)$.
- Если $MaxScore(k-1)+P(k+1) \leq h \leq MaxScore(k-1)+R(k+1) = MaxScore(k)$, то сначала сделаем число 2^{k+1}, набрав ровно $h_1 = h - MaxScore(k-1)$ очков, а затем доберем оставшиеся $MaxScore(k - 1)$ очков. Это мы можем сделать, так как $P(k + 1) \leq h_1 \leq R(k + 1)$.

Таким образом мы можем набрать любое число очков от 0 до $MaxScore(k)$ с шагом 4 очка.

На практике следующий жадный алгоритм приводит к тому же результату. Пока мы не набрали нужное число очков, будем находить максимальное t такое, что $P(t)$ не превосходит оставшегося числа очков, и создавать 2^t, вычитая максимальное возможное число очков, правильно выбирая между $Q(t)$ и $R(t)$.

Problem I. Infection Investigation

Идея: Фёдор Ушаков
Разработка: Леонид Данилевич

Будем отвечать на запросы оффлайн, применив принцип «разделяй и властвуй». А именно, напишем рекурсивную функцию, принимающую $1 \le \mathrm{lf} \le \mathrm{rg} \le n$, и получающую ответы для запросов (l, r) при условии $\mathrm{lf} \le l \le r \le \mathrm{rg}$. Пусть $\mathrm{m} = \lfloor \frac{\mathrm{lf}+\mathrm{rg}}{2} \rfloor$, тогда функция отвечает на запросы (l, r), не содержащие m, рекурсивно. Осталось разобраться с запросами (l, r), такими, что $\mathrm{lf} \le l \le m \le r \le \mathrm{rg}$.

Для этого насчитаем стандартными методами (при помощи двоичного поиска или дерева отрезков / дерева Фенвика) длины наибольшей возрастающей подпоследовательности на префиксах $[a_m, \ldots, a_{\mathrm{rg}}]$ и суффиксах $[a_{\mathrm{lf}}, \ldots, a_{m-1}]$. Пусть отрезок $(a_l, \ldots, a_r)$ бьётся на два, и длины возрастающей подпоследовательности в этих частях равны $x, y \ge 1$. Ясно, что в таком случае ответ на этот запрос заключён в отрезке $[\max(x, y), x + y]$. Если $x = y = 1$, то необходимо проверить, является ли подмассив $a_l, \ldots, a_r$ строго убывающим, а во всех остальных случаях $\lfloor \frac{\max(x,y)+(x+y)}{2} \rfloor$ является достаточно хорошим приближением.

В самом деле, с одной стороны $\lfloor \frac{\max(x,y)+(x+y)}{2} \rfloor \le \frac{\max(x,y)+(x+y)}{2} \le \frac{3}{2}\max(x, y)$, а с другой стороны при $\max(x, y) \ge 3$: $\lfloor \frac{\max(x,y)+(x+y)}{2} \rfloor \ge \frac{\max(x,y)+(x+y)-1}{2} \ge \frac{2}{3}(x + y)$; перебор случаев показывает, что при $\max(x, y) = 2$ тоже $\lfloor \frac{\max(x,y)+(x+y)}{2} \rfloor \ge \frac{2}{3}(x + y)$.

Итого асимптотика высчитывается по формуле $T(n) = 2T(n/2) + \mathcal{O}(n \log n)$, откуда алгоритм работает за $\mathcal{O}(n \log^2 n)$.

Problem J. Judging Problem

Идея: Захар Яковлев, Геннадий Короткевич
Разработка: Никита Голиков

Рассмотрим две задачи, выбранные в годы i и $i + 1$. Если их названия похожи, то при выборе названия в год $i + 1$ правило было выполнено.

В противном случае необходимо проверить, что на суффиксе, начинающемся с $i + 1$, нет задач, похожих на i-ю. Для этого пройдем по задачам в обратном порядке, начиная с n-й. Во время итерации мы будем хранить два множества слов: одно множество будет хранить все использованные первые слова в суффиксе, а другое — все использованные вторые слова.

При проверке задачи $i + 1$, если названия i-й и $(i + 1)$-й не были похожи, мы проверим, содержит ли какое-либо из множеств соответствующее слово из i-го названия. Если какое-либо из множеств содержит это слово, это означает, что правило не было соблюдено в год $i + 1$.

Если все проверки прошли успешно, это означает, что судьи правильно следовали правилу. Время работы решения $O(n)$ или $O(n \log n)$, в зависимости от реализации используемой структуры данных множества.

Problem K. Keys and Grates

Идея: Михаил Иванов
Разработка: Михаил Иванов

Не умаляя общности, пусть $h > s$. Давайте представим себе оптимальный маршрут Китнисс, как он будет выглядеть? Китнисс будет чередовать хождение налево и хождение направо. Понятно, что если она добралась до какой-то точки в конце хождения налево, то следующее хождение налево должно закончиться строго дальше. В самом деле, зачем иначе Китнисс потребовалось идти налево? Если затем, чтобы подобрать ключ или выйти в люк, то она могла это же сделать и во время предыдущего хождения. А других причин быть и не могло — вскрывать замки на этом отрезке Китнисс не могла, они и так все были вскрыты ранее. Кроме того, понятно, что самое последнее хождение закончится в h, и это будет хождение направо, иначе Китнисс могла покинуть тоннель раньше. Формально, Китнисс выберет некоторый отрезок тоннеля $[L; R]$, содержащий и h, и s, выберет в нём несколько точек L_m, R_m, L_{m-1}, R_{m-1}, L_{m-2}, R_{m-2}, $\ldots$, L_0, R_0 и в этом порядке посетит их, по пути открывая все замки и подбирая все ключи от замков из отрезка $[L; R]$. При этом $L = L_0 < L_1 < \ldots < L_m \le s \le R_m < R_{m-1} < R_{m-2} < \ldots < R_0 = R = h$.

Теперь остаётся лишь выбрать эти отрезки и эти точки. Давайте постепенно это делать с конца. R_0 выбрать легко: это просто h. Как теперь выбрать L_0? Для того, чтобы добраться до R_0, Китнисс надо иметь на руках все ключи от замков из отрезка $[s; R_0]$. Рассмотрим все из этих ключей, которые находятся левее s, выберем самый левый такой ключ — его-то положение и будет точкой L_0. Далее аналогично: чтобы добраться до L_0, надо, чтобы все замки в $[L_0; s]$, ключи от которых правее s, можно было открыть; значит, в качестве R_1 надо взять самый правый ключ, открывающий замок из $[L_0; s]$. Продолжаем так делать, пока не окажется, что на очередном отрезке вообще нет замков, требующих ключей с другой половины тоннеля; тогда мы просто объявляем, что этот очередной отрезок и будет первым отрезком, который мы будем проходить.

Может оказаться, что такого момента никогда не настанет. Это окажется так, если мы обнаружим для какого-то j, что $L_j \ge L_{j+1}$ или $R_j \le R_{j+1}$ — это точно значит, что есть циклическая зависимость. А именно, давайте про пару из замка и люка/ключа (возможно, открывающего другой замок) будем говорить, что замок *запирает* этот люк/ключ, если замок находится между данным люком/ключом и s. Тогда понятно, что Китнисс не сможет подобрать ключ до того, как откроет все замки, запирающие этот ключ, и не сможет выйти в люк, пока не откроет все замки, запирающие люк. Неравенство вроде $R_j \le R_{j+1}$ гарантированно означает, что есть пара i, j, что i-й замок запирает j-й ключ, а j-й замок запирает i-й ключ, и при этом один из замков запирает люк. Тогда Китнисс не сможет открыть ни один из замков, не вскрыв сначала другой — это мы и назвали ранее циклической зависимостью. А, поскольку без вскрытия одного из этих замков Китнисс не добраться до люка, то задача в этом случае неразрешима. Также заметим, что если хоть один нужный нам замок запирает свой собственный ключ, то задачу тоже не решить.

Такое решение работает за $\mathcal{O}(n \log n)$: сортируем все замки, потом для каждого ключа находим ближайший к нему замок, который запирает этот ключ (или флаг, что ключ не заперт никаким замком). Дальше для замков справа от s построим массив префиксных минимумов координат нужных для них ключей, а для замков слева от s — массив суффиксных максимумов координат нужных для них ключей. Теперь, как описывалось ранее, построим L_0 и R_0, переберём все замки на $[L_0; R_0]$ и проверим, что никакой из них не запирает свой собственный ключ. Наконец, построим $R_1, L_1, \ldots$, проверяя каждый раз, что не нарушается строгая монотонность L_i и R_i. Чтобы, скажем, зная R_i, найти L_i, найдём g_j — ближайший к R_i замок, запирающий R_i (в точке R_i всегда будет либо люк, либо ключ). Если g_j не существует, то процесс окончен, Китнисс может беспрепятственно идти от s к R_i. Если g_j существует и правее s, то мы находим в массиве префиксных минимумов самый левый ключ k_q, требующийся для вскрытия всех замков в $[s; g_j]$. Если $k_q \ge s$, то опять же Китнисс может просто идти к R_i, и все нужные для этого ключи будут лежать у неё на пути — другими словами, процесс построения последовательностей L_i, R_i окончен. В противном случае мы говорим, что $L_i = k_q$, и продолжаем процесс.

Наконец, чтобы найти ответ, вычислим $(s - L_m) + \sum_{i=0}^{m} (R_i - L_i) + \sum_{i=0}^{m-1} (R_i - L_{i+1})$ — длину пути Китнисс.

Problem L. Lucky Number Theory

Идея: Артем Васильев
Разработка: Артем Васильев

Начнем решать задачу со случая $d = 1$. Заметим, что стратегия зависит только от дробной части S (обозначается $\{S\}$): все действия для S и $S + 1$ можно делать одинаково, и результат всегда будет отличаться на единицу.

Важное наблюдение: вне зависимости от текущего распределения S, после броска, новое значение $\{S'\}$ будет иметь равномерное распределение от 0 до 1.

Благодаря этому наблюдению, мы можем предполагать, что на каждом шаге значение $\{S\}$ равномерно распределено от 0 до 1, что позволяет использовать динамическое программирование. Обозначим за $f_{n,k}$ матожидание оптимальной стратегии, если осталось n бросков, k возможностей вывести, в предположении, что в начале $\{S\}$ распределено равномерно от 0 до 1.

Обозначим $\{S\} = x$. В зависимости от x, есть два случая: бросить еще раз, либо вывести билеты и бросить еще раз. В первом случае матожидание числа билетов равно $x + f_{n-1,k}$: вероятность того, что после броска целая часть S увеличится, равна S, и мы переходим в состояние с $n - 1$ броском. Во втором случае матожидание равно $1 + f_{n-1,k-1}$: мы гарантированно получаем один билет сейчас, и переходим в состояние $(n - 1, k - 1)$. Все переходы корректны из-за наблюдения выше: несмотря на то, что мы используем x для принятия решения, после броска, $\{S\}$ все еще будет распределено равномерно от 0 до 1.

Так как x распределено равномерно от 0 до 1, нужно взять максимум из двух случаев, и проинтегрировать по x от 0 до 1: $f_{n,k} = \int\limits_0^1 \max(x + f_{n-1,k}, 1 + f_{n-1,k-1})dx$. Если обозначить $A = f_{n-1,k}$, $B = 1 + f_{n-1,k-1}$, то для $x \geq B - A$ максимум достигается в первом случае, а для $x < B - A$ — во втором. Опустив промежуточные вычисления, получаем формулу для ДП: $f_{n,k} = A + \frac{1}{2} + \frac{1}{2}(B - A)^2$, которую можно вычислить за $O(n^2)$ сразу для всех возможных пар (n, k). Итоговый ответ на задачу равен $f_{n-1,k-1} + 1$, учитывая первый бросок и последний вывод, который дает $+1$ к ответу.

Наконец, случай $d > 1$. Заметим, что сгенерировать случайное вещественное число в интервале $(0, d)$ это то же самое, что сгенерировать случайное вещественное число от 0 до 1, а заметим прибавить случайное *целое* число от 0 до $d-1$ включительно. Добавление случайного целого числа никак не влияет на стратегию, только на ответ: на каждый бросок в среднем прибавится $\frac{1}{d}(0 + 1 + \ldots + d - 1) = \frac{d-1}{2}$ билетов. Ответ на задачу равен решению для $d = 1$ плюс $\frac{d-1}{2}n$.