Разбор задачи «Удвоение»

В задаче просили проверить, может ли одно целое положительное число быть получено из другого с помощью нескольких (возможно, нуля или одного) последовательных применений операции удвоения.

Возможно, самое простое решение состоит в следующем: будем удваивать первое число, пока оно меньше второго. Этот процесс закончится и сделает это достаточно быстро. Если в результате числа окажутся равны, то ответ YES, иначе ответ NO. Ниже это решение объяснено более формально и скучно.

Пусть $f(n, m)$ — YES, если m можно получить из n последовательным применением операций удвоения, и NO иначе. То есть $f(n, m)$ — ответ на задачу для чисел n и m. Тогда

$$f(n,m) = \begin{cases} \text{YES, если } n = m \\ \text{NO, если } n > m \\ f(2n, m), \text{ если } n < m \end{cases}$$

Действительно, если $n = m$, то ответ YES (нужно применить ноль операций удвоения), если $n > m$, то ответ NO (так как, удваивая n, мы будем получать только числа, большие m). Если же $n < m$, то ответ для пары (n, m) такой же, как и для пары $(2n, m)$. Сейчас мы это объясним.

Из того, что ответ для $(2n, m)$ — YES, следует то, что ответ для (n, m) — YES: нужно просто применить операцию удвоения на один раз больше.

Более того, из того, что ответ для (n, m) — YES, следует то, что ответ для $(2n, m)$ — YES: если ответ для (n, m) — YES, то мы должны были применить к n хотя бы одну операцию удвоения (так как $n < m$) и получить при этом число m. Применив к исходному числу $2n$ на одну операцию удвоения меньше, мы получим из него число m, то есть ответ для $(2n, m)$ тоже YES, что мы и хотели доказать.

Стоит заметить, что вычисление $f(n, m)$ по указанной выше формуле не может продолжаться бесконечно, так как $n \geqslant 1$, и после не более, чем $\lceil \log_2 m \rceil$ удвоений мы получим число, не меньшее m, в качестве первого аргумента.

Благодаря маленьким ограничениям ($1 \leqslant n, m \leqslant 100$) хранить n и m можно в стандартном целочисленном типе данных практически любого языка программирования.

Разбор задачи «До гор»

Автор задачи:	Михаил Иванов
Подготовка тестов и решений:	Михаил Иванов
Автор разбора:	Михаил Иванов

Рассмотрим n строк: исходная Петина строка, а также Петина строка, в которой всеми способами поменяли местами какие-то два соседних символа: первый и второй, второй и третий, ..., $(n-1)$-й и n-й. Это всевозможные строки, которые может получить Вася. В каждой с помощью цикла за $n-1$ сравнение пары соседних символов можно проверить, получилась ли подстрока «`or`» и, если хоть где-то получилась, вывести «`Yes`», иначе — «`No`». Это решение требует $\mathcal{O}\left(n^2\right)$ операций и проходит ограничения всех подгрупп, кроме последней.

Чтобы справиться с последней подгруппой, заметим, что в предыдущем решении многие одинаковые пары символов мы излишне проверяли помногу. Действительно, если мы поменяли местами k-й и $(k+1)$-й символы, то, если подстроки «`or`» раньше не было, то она могла появиться лишь на местах $(k-1; k)$, $(k; k+1)$ или $(k+1; k+2)$. Поэтому можно действовать так: сначала проверить наличие подстроки «`or`» в Петиной строке, а дальше циклом с $n-1$ пробовать менять местами пару символов и проверять вышеуказанные три пары символов. В этом решении важно, во-первых, не забывать в конце итерации цикла менять символы назад, и, во-вторых, проверять лишь те из этих трёх пар символов, у которых оба индекса принадлежат строке (не выходят за её левую и правую границу). Другими словами, пару $(k-1; k)$ не нужно проверять, если k-й символ — первый в строке, а пару $(k+1; k+2)$ не нужно проверять, если $(k+1)$-й символ — последний в строке.

Наконец, можно просто заметить, что получить подстроку «or» можно тогда и только тогда, когда в исходной строке есть хотя бы одна из подстрок следующего вида:

- «or»;
- «ro»;
- «o?r», где ? — любая строчная латинская буква.

Проверить наличие таких подстрок можно за один цикл; здесь, опять же, необходимо следить, чтобы используемые номера символов лежали в пределах от минимального до максимального индекса в строке, чтобы избежать Runtime Error и Wrong Answer. Асимптотика последних двух решений — $\mathcal{O}(n)$.

Разбор задачи «Саша и задача про предков»

Рассмотрим сначала решение задачи для одного варианта выбора корня дерева — пусть это вершина v. Определим величину *размера поддерева* вершины $sz(u)$ как число вершин, достижимых (с учётом ориентации рёбер от корня) из вершины u. Отметим, что по построению ориентации из корня достижимы все вершины, то есть $sz(v) = n$. Тогда утверждается, что ответом будет $\max\limits_{u \neq v} sz(u) + 1$.

Действительно, если для некоторого множества вершин S и вершины u выполняется $\mathcal{A}(S) = u$, то $sz(u) \geqslant |S|$, ведь в таком случае все вершины из S достижимы из u. Значит, если мы ищем наименьшего общего предка для некоторого множества из k вершин, причём $k > \max\limits_{u \neq v} sz(u)$, то никакая вершина кроме v общим предком оказаться не может. Из этого следует, что ответ не превосходит $\max\limits_{u \neq v} sz(u) + 1$. Покажем, что он ещё и не меньше этой величины. Для этого требуется показать, что если для некоторой вершины $u \neq v$ выполняется $k \leqslant sz(u)$, то найдётся множество вершин S такое, что $|S| = k$ и $\mathcal{A}(S) \neq v$. В качестве такого множества вершин можно выбрать любые k вершин, достижимых из u. Из вершины u можно дойти до любой вершины этого множества, а глубина вершины u больше глубины вершины v, так как v — единственная вершина, имеющая нулевую глубину. Из этого следует, что вершина v не может быть наименьшим общим предком этого множества.

Научимся эффективно считать значения $sz(u)$ для всех вершин u. Будем называть детьми u вершины, в которые выходят ориентированные рёбра из u. Если из u не выходят никакие рёбра, то $sz(u) = 1$; единственная достижимая вершина — она сама. Иначе пусть у неё есть дети $w_1, w_2, \ldots, w_k$. Тогда $sz(u) = \sum\limits_{i=0}^{k} sz(w_i) + 1$. Ясно, что из вершины достижима она сама, а также все вершины, достижимые из её детей. Стоит отметить, что при расчёте по такой формуле никакая вершина не будет учтена дважды. Это могло бы случиться, если бы некоторая вершина t была бы достижима одновременно из вершин w_i, w_j, $i \neq j$. Предположим, это так, рассмотрим путь в исходном неориентированном графе, идущий из u сначала в t через w_i, затем возвращающийся в u через w_j. Если некоторая вершина отличная от u встретилась в пути хотя бы два раза, удалим всё между соседними её вхождениями и одно из них. Повторим эту процедуру сколько потребуется, и получим цикл, содержащий хотя бы три вершины: u, w_i, w_j. По нашей процедуре вершина может быть удалена, только если она заключена между вхождениями отличной от u вершины. А для каждой из этих трёх вершин верно, что с одной из сторон от них в пути нет отличных от u вершин. Существование же этого цикла должно означать, что граф не является деревом, а это противоречие.

Пользуясь найденными формулами, значения $sz(u)$ удобно вычислить с помощью рекурсии. Изначально запустимся от корня, затем будем спускаться в детей. Ввиду ацикличности мы не запустимся ни от какой вершины дважды, значит, такая процедура будет работать за линейное от числа вершин время.

Заметим, что если w достижима из u, то $sz(u) \geqslant sz(v)$. Следовательно, при нахождения ответа не требуется рассматривать все вершины, достаточно ограничиться детьми корня, поскольку любая другая вершина достижима из кого-то из них. Воспользуемся этим соображением для того, чтобы получить способ найти за один проход по дереву ответы для всех вариантов выбора корня. Модифицируем наши обозначения: пусть теперь $sz_v(u)$ — число вершин, достижимых из u, если дерево

подвешено за v. Подвесим дерево за произвольную вершину v и запустим аналогичную описанной ранее рекурсивную процедуру. Пусть мы из вершины p пришли в вершину u, имеющую детей $w_1, w_2, \ldots, w_k$, и мы уже вычислили значения $sz_v(w_i)$ для всех i. Чтобы найти ответ для случая, когда в качестве корня выбрана u, нам потребуется знать значения $sz_u(p), sz_u(w_i)$ для всех i, поскольку именно эти вершины будут детьми u, если сделать u корнем. Во-первых, заметим, что из построения ориентации $sz_u(w_i) = sz_v(w_i)$. Действительно, по ранее доказанному не существует не проходящего через u пути из v в вершину, достижимую из u. А это означает, что когда при построении подвешенного за v дерева мы дойдём до вершины u, мы ещё ничего не сделаем с достижимыми из u вершинами. Тогда после этого мы построим для них ровно такую же структуру, как и в случае дерева, подвешенного за u. Остаётся определить $sz_u(p)$. Но мы знаем, что $sz_u(u) = n$, а также имеем связь между размером поддерева вершины и размерами поддеревьев её детей. Эти формулы дают результат: $sz_u(p) = n - 1 - \sum_{i=1}^{k} sz_u(w_i)$.

Разбор задачи «Коллективный пароль»

Автор задачи:	Иван Казменко
Подготовка тестов и решений:	Иван Казменко
Автор разбора:	Иван Казменко

Подзадача 1:

Пусть девятибуквенный пароль состоит из букв $w_1w_2w_3w_4w_5w_6w_7w_8w_9$. Сгенерируем такие три индивидуальных семибуквенных пароля: a$w_1w_2w_3w_4w_5w_6$, b$w_4w_5w_6w_7w_8w_9$ и c$w_7w_8w_9w_1w_2w_3$. Другими словами, в первой букве запишем номер индивидуального пароля, а в оставшихся шести — две трети букв исходного пароля.

Покажем, какие буквы исходного пароля куда попали, в виде таблицы:

a	*	*	*	*	*	*			
b				*	*	*	*	*	*
c	*	*	*				*	*	*

Можно видеть, что любые два из трёх полученных паролей содержат каждую из исходных букв хотя бы один раз.

Для восстановления исходного пароля посмотрим на первую букву каждого индивидуального пароля, который мы получили. Если это буква «a», скопируем следующие шесть букв в позиции $1, 2, 3, 4, 5, 6$. Если это буква «b», скопируем их в позиции $4, 5, 6, 7, 8, 9$. Наконец, если это буква «c», скопируем их в позиции $7, 8, 9, 1, 2, 3$. Произведя эти действия для обоих полученных паролей, мы восстановим все девять букв исходного, некоторые из них мы восстановим даже дважды.

Подзадача 2:

Можно решать эту подзадачу аналогично первой. А именно, сгенерируем пять паролей, содержащих следующие буквы:

a	*	*	*	*	*	*			
b				*	*	*	*	*	*
c	*	*	*				*	*	*
a	*	*	*	*	*	*			
b				*	*	*	*	*	*

Иными словами, просто продублируем первый и второй индивидуальные пароли из предыдущей подзадачи, чтобы индивидуальных паролей стало пять.

Можно видеть, что в любых трёх индивидуальных паролях каждая из исходных букв содержится хотя бы один раз.

Алгоритм восстановления не меняется.

Подзадача 3:

Воспользуемся *китайской теоремой об остатках*, которая формулируется так. Пусть задано целое число x. Пусть также выбраны r целых положительных чисел $a_1, a_2, \ldots, a_r$, причём они попарно взаимно просты, то есть наибольший общий делитель a_i и a_j равен единице при $i \neq j$. Тогда, зная остатки от деления $x \bmod a_1$, $x \bmod a_2$, …, $x \bmod a_r$, мы можем однозначно восстановить остаток от деления $x \bmod (a_1 \cdot a_2 \cdots a_r)$. В частности, если известно, что $0 \leqslant x < (a_1 \cdot a_2 \cdots a_r)$, то мы можем однозначно восстановить число x.

К примеру, если известны величины $x \bmod 5$ и $x \bmod 7$, и при этом $0 \leqslant x < 35$, то x восстанавливается по двум остаткам однозначно. В следующей таблице в строках указаны остатки от деления на 5, в столбцах — на 7, а в клетках — соответствующие числа x:

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	15	30	10	25	5	20
1	21	1	16	31	11	26	6
2	7	22	2	17	32	12	27
3	28	8	23	3	18	33	13
4	14	29	9	24	4	19	34

Как найти x, зная $y = x \bmod a$ и $z = x \bmod b$? Самое простое — рассмотреть числа y, $y+a$, $y+2a$, $y+3a$, $y+4a$, …, имеющие остаток y от деления на a, и найти среди них то, которое имеет остаток z от деления на b. Это удастся сделать не более чем за b шагов, так как первое число с правильным остатком от деления на b будет не больше, чем $y + b \cdot a$. Например, если $x \bmod 5 = 3$ и $x \bmod 7 = 4$, то мы рассматриваем числа 3, 8, 13, 18, 23, 28, 33, и ищем то из них, у которого остаток от деления на 7 равен 4: это число 18.

Есть и более быстрые методы, например, расширенный алгоритм Евклида, работающий за $\log(a + b)$ шагов, но в задаче они не требовались.

Как же воспользоваться в нашей задаче китайской теоремой об остатках? Решим с её помощью последнюю подзадачу.

Будем рассматривать строки из английских букв как числа в позиционной системе счисления с основанием 26: буква «`a`» будет соответствовать числу 0, «`b`» — числу 1, и так далее до буквы «`z`», которая будет соответствовать числу 25.

Например, строка «`cat`» будет соответствовать числу $2 \cdot 26^2 + 0 \cdot 26^1 + 19 \cdot 26^0 = 1371$.

При таком подходе девятибуквенный пароль — это просто целое число от 0 до $26^9 - 1 = 5\,429\,503\,678\,975$, а индивидуальные четырёхбуквенные пароли — целые числа от 0 до $26^4 - 1 = 456\,975$.

Выберем какие-нибудь семь чисел так, чтобы они были попарно взаимно просты, а произведение любых четырёх из них было не меньше 26^9. Например, можно взять следующие семь простых чисел за $26^3 = 17\,576$: это числа $a_0 = 17\,579$, $a_1 = 17\,581$, $a_2 = 17\,597$, $a_3 = 17\,599$, $a_4 = 17\,609$, $a_5 = 17\,623$ и $a_6 = 17\,627$.

Имея девятибуквенный пароль — число x — сгенерируем семь индивидуальных четырёхбуквенных паролей. Каждый будет хранить свой номер — число от 0 до 6 — а также остаток по соответствующему модулю из семи чисел, выбранных выше. А именно, запишем семь чисел вида $m \cdot i + x \bmod a_i$, где $m = a_6$ — максимальное из a_i:

$$
\begin{aligned}
y_0 &= 17\,627 \cdot 0 + x \bmod 17\,579,\\
y_1 &= 17\,627 \cdot 1 + x \bmod 17\,581,\\
y_2 &= 17\,627 \cdot 2 + x \bmod 17\,597,\\
y_3 &= 17\,627 \cdot 3 + x \bmod 17\,599,\\
y_4 &= 17\,627 \cdot 4 + x \bmod 17\,609,\\
y_5 &= 17\,627 \cdot 5 + x \bmod 17\,623,\\
y_6 &= 17\,627 \cdot 6 + x \bmod 17\,627.
\end{aligned}
$$

Эти числа меньше $17\,627 \cdot 7 = 123\,389$, что в свою очередь меньше, чем $26^4 = 456\,976$. А значит, каждое из них получится записать четырьмя буквами.

Чтобы восстановить исходный девятибуквенный пароль, посмотрим на каждое из полученных чисел y_*. Разобьём его на две части: $u = [y_*/m]$ и $v = y_* \bmod m$. Первая часть — это просто номер ключа: $y_* = y_u$. После этого мы знаем, что $x \bmod a_u = v$.

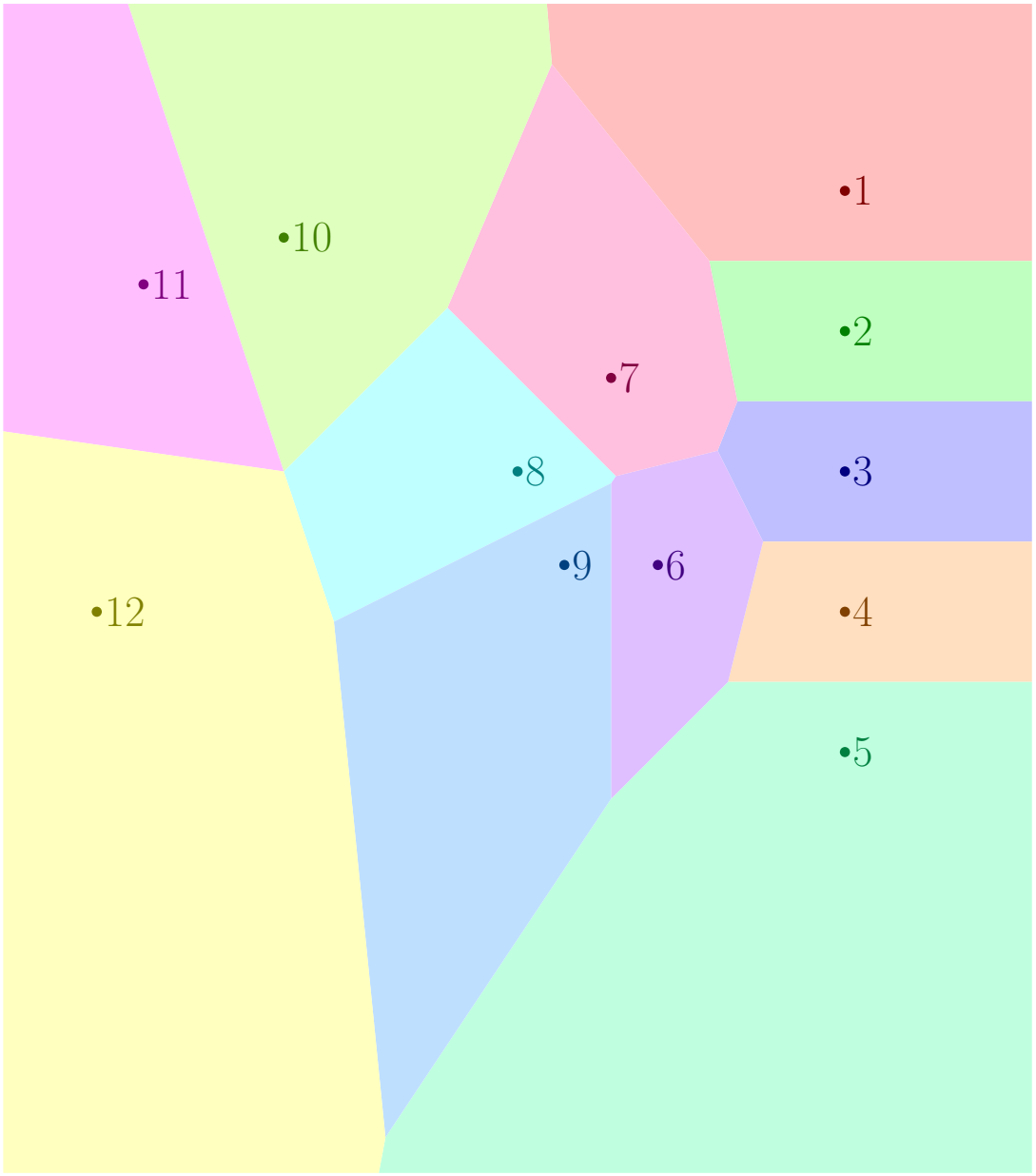
Итак, мы знаем числа $x \bmod a_i$, $x \bmod a_j$, $x \bmod a_k$ и $x \bmod a_l$ для каких-то четырёх различных индексов i, j, k и l. Сначала, пользуясь $(x \bmod a_i)$ и $(x \bmod a_j)$, по китайской теореме об остатках найдём $x' = x \bmod (a_i \cdot a_j)$. Далее, пользуясь x' и $(x \bmod a_k)$, найдём $\tilde{x} = x \bmod (a_i \cdot a_j \cdot a_k)$. Наконец, пользуясь $\tilde{x}$ и $(x \bmod a_l)$, найдём $\bar{x} = x \bmod (a_i \cdot a_j \cdot a_k \cdot a_l)$. Поскольку $a_i \cdot a_j \cdot a_k \cdot a_l \geqslant 26^9$, мы можем заключить, что найденное значение $\bar{x}$ — это и есть само число x.

Напоследок заметим, что это решение можно адаптировать и для первых двух подзадач. Нужно лишь выбрать другие числа a_i в соответствии с их ограничениями.

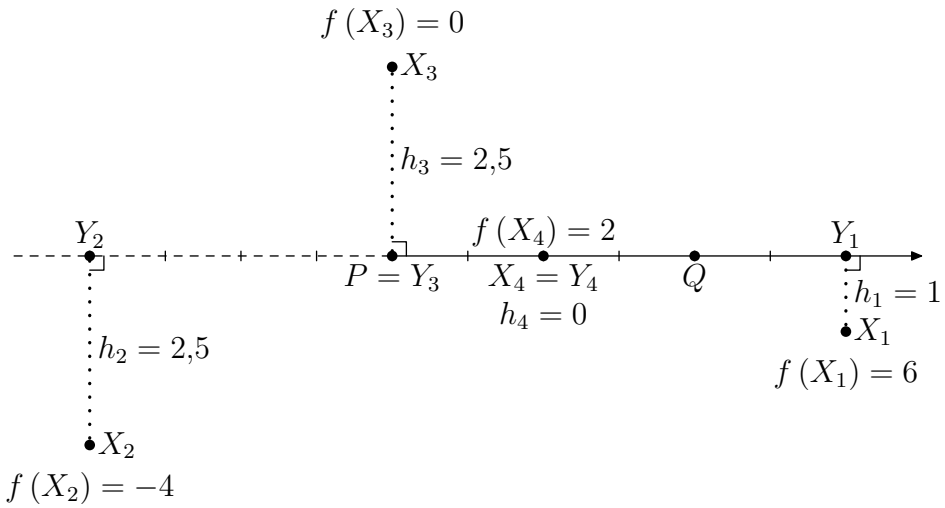
Разбор задачи «Лазерный луч»

Автор задачи:	Михаил Иванов
Подготовка тестов и решений:	Михаил Иванов
Автор разбора:	Михаил Иванов

Напомним, задача заключалась в следующем: на плоскости выбирались точки $C_1, C_2, \ldots, C_n$, красились в n разных цветов (мы пронумеруем эти цвета так же, как и точки C_i), и дальше каждая из остальных точек X плоскости красилась так: выбиралась такая из точек $C_1, \ldots, C_n$, которая располагалась ближе всего в X, и если это была точка C_i, то X красилась в цвет i. Некоторые точки (например, середины отрезков между некоторыми парами C_i) при этом не получали цвета, так как были равноудалены от нескольких ближайших точек, а не от одной-единственной. Наконец, на плоскости выбирался луч PQ, и надо было определить, в какой цвет окрашена бесконечная часть этого луча. Ниже на картинке нарисован пример: несколько точек C_i и задаваемая при них раскраска плоскости. Видно, что она разбилась на несколько многоугольных областей, некоторые из которых бесконечны. Ломаные, разделяющие разные области, не покрашены ни в какой цвет. Данная раскраска плоскости называется *диаграммой Вороного*. Быстрое построение такой диаграммы — довольно непростое дело, однако в данной задаче строить диаграмму Вороного не требовалось и даже было излишне.



Любой луч задаёт на плоскости *координату* — специальную функцию $f(X)$, сопоставляющую вещественное число каждой точке плоскости. Мы сейчас определим $f(X)$ для нашего луча PQ. А именно, возьмём точку X на нашей плоскости и спроецируем её на прямую PQ. Основание перпендикуляра Y окажется либо на самом луче PQ, либо на его продолжении за точку P. В первом случае координатой считается расстояние PY, а во втором — то же расстояние, но взятое со знаком минус: $-PY$. Кроме того, у каждой точки есть *расстояние* h до PQ — это длина перпендикуляра XY. Ниже на картинке показаны четыре разных положения точки X и соответствующие им Y, $f(X)$ и h.



Таким образом, координата как бы показывает нам, насколько далеко точка находится от P в смысле движения вдоль направления PQ (то, насколько далеко при этом точка от прямой PQ, координатная функция полностью игнорирует) Предположим для начала, что у всех точек C_i различные координаты. Нас интересует, какая точка при этом будет ближайшей к очень далёким точкам на луче. Пусть на луче взята точка A с координатой a, а у точки C_i координата $f(C_i)$ и расстояние h_i до луча. Тогда по теореме Пифагора, чтобы найти расстояние AC_i, надо понять, насколько далеки точки друг от друга по двум перпендикулярным направлениям, сложить два квадрата полученных расстояний и извлечь из суммы корень. По направлению PQ расстояние между точками, то есть разность их координат, равно $|a - f(C_i)|$. По направлению, перпендикулярному PQ, одна из точек лежит на луче, а другая находится на расстоянии h_i от содержащей этой луч прямой. Таким образом, расстояние равно $\sqrt{(a - f(C_i))^2 + h_i^2}$. Нас интересует поведение этого числа, наводящего страх, при всех достаточно больших a. Ну, возьмём две точки C_i и C_j, пусть для определённости $f(C_i) < f(C_j)$. Какое из этих двух расстояний тогда окажется меньшим для бесконечной части луча? Будем сравнивать эти два выражения, а знаком ? будет обозначаться знак $>$ или $<$, который будет одинаковым на протяжении всех преобразований. Мы пытаемся сравнить

$$\sqrt{(a - f(C_i))^2 + h_i^2} \quad ? \quad \sqrt{(a - f(C_j))^2 + h_j^2}.$$

Возведём обе части в квадрат. Поскольку они обе неотрицательны, это не повлияет на результат сравнения.

$$(a - f(C_i))^2 + h_i^2 \quad ? \quad (a - f(C_j))^2 + h_j^2.$$

Раскроем скобки.

$$a^2 - 2af(C_i) + f(C_i)^2 + h_i^2 \quad ? \quad a^2 - 2af(C_j) + f(C_j)^2 + h_j^2.$$

Вычтем a^2 из обеих частей.

$$-2af(C_i) + f(C_i)^2 + h_i^2 \quad ? \quad -2af(C_j) + f(C_j)^2 + h_j^2.$$

Перегруппируем члены сравнения, чтобы в левой части оказалось всё с a, а в правой — всё без этого множителя.

$$2af(C_j) - 2af(C_i) \quad ? \quad f(C_j)^2 + h_j^2 - f(C_i)^2 - h_i^2.$$

Вынесем множитель при a за скобку.

$$2\big(f(C_j) - f(C_i)\big)a \quad ? \quad f(C_j)^2 + h_j^2 - f(C_i)^2 - h_i^2.$$

В такой записи уже совсем ясно, что правая часть вообще не зависит от a, а левая неограниченно возрастает с ростом a (поскольку $f(C_j) - f(C_i) > 0$). Поэтому при достаточно больших a (то есть

для всех точек A луча, достаточно далёких от его начала) в этом неравенстве стоит знак $>$, что значит, что в самом первом неравенстве стоит знак $>$ — таким образом, точка C_j ближе к A, чем C_i.

Мы только что поняли, что если все координаты различны, то областью содержащей бесконечную часть луча, будет та, которая содержит точку C_i с наибольшей координатой. Что же делать, если среди координат есть равные? Чтобы понять, вернёмся к нашей формуле.

$$2\big(f\,(C_j) - f\,(C_i)\big)a \quad ? \quad f\,(C_j)^2 + h_j^2 - f\,(C_i)^2 - h_i^2.$$

Понятно, что если у двух точек разные координаты, то у той, которая имеет меньшую координату, точно нет никаких шансов быть ответом. Поэтому принимать решение надо лишь о тех точках, которые имеют максимальную координату из всех C_i (если таких точек с максимальной координатой больше одной). Но если у точек C_i и C_j одинаковы координаты $f\,(C_i)$ и $f\,(C_j)$, то выражение сильно упрощается:

$$0 \quad ? \quad h_j^2 - h_i^2.$$

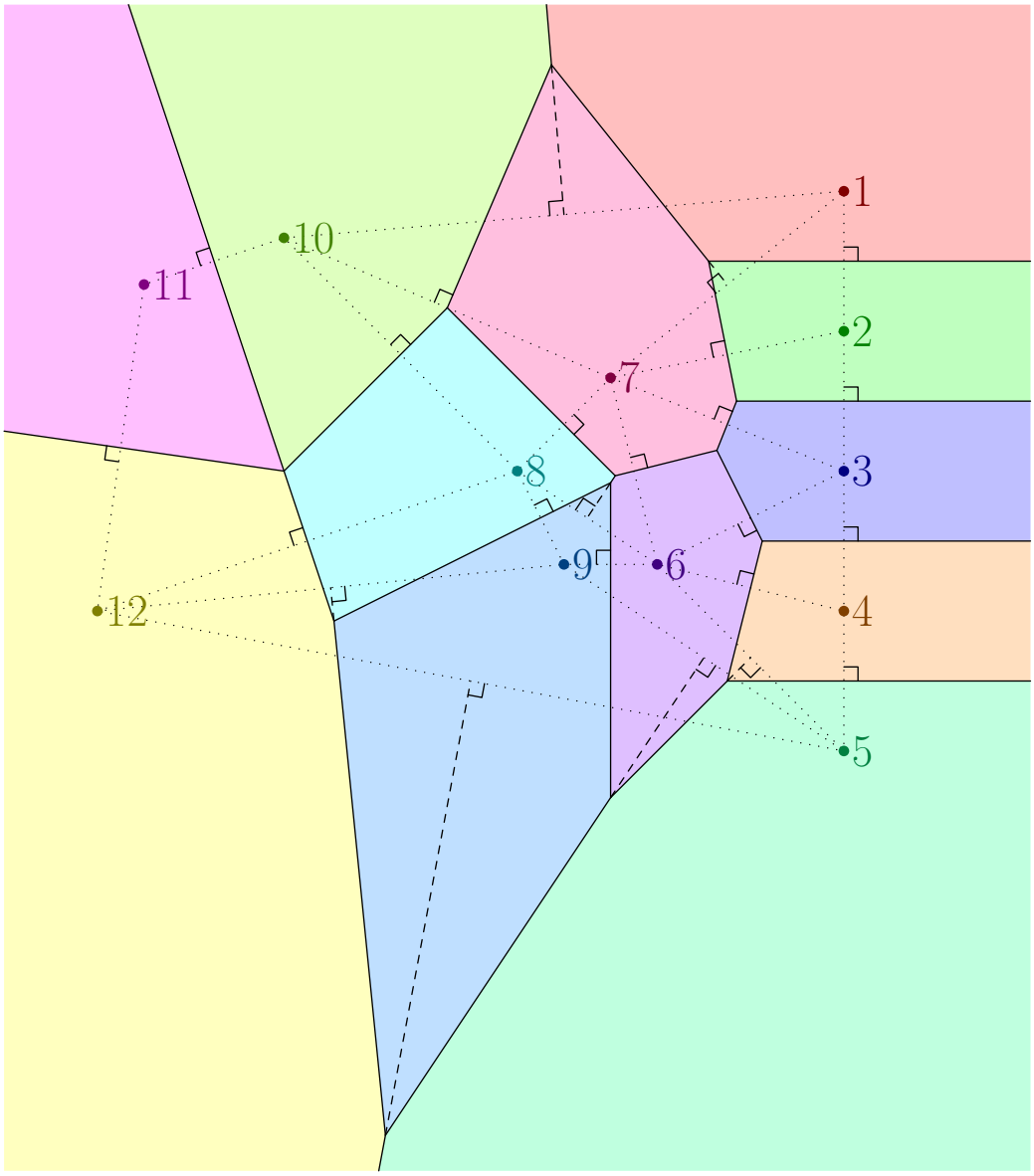
$$h_i^2 \quad ? \quad h_j^2.$$

Так как расстояния h_i и h_j неотрицательны, корень из неравенства можно извлечь, и получатся обычные расстояния.

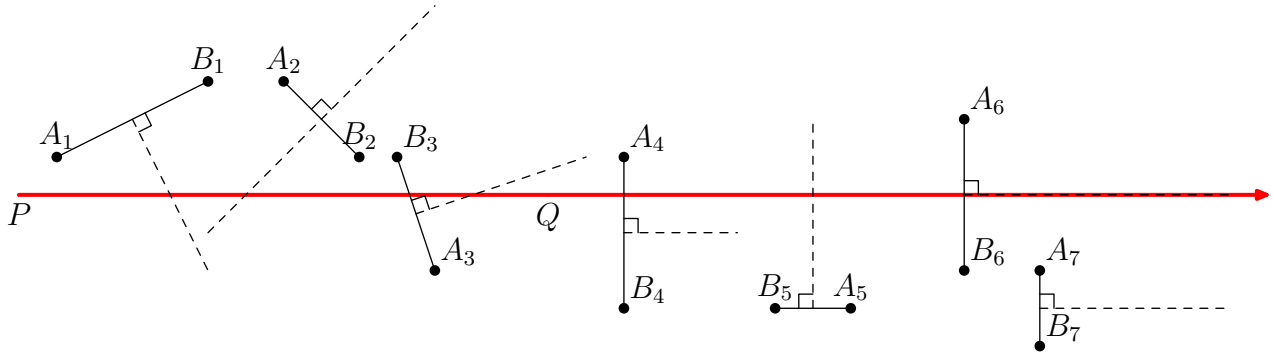
$$h_i \quad ? \quad h_j.$$

Это неравенство уже не зависит от a. Если у нескольких точек одинаковая координата, то та из них, которая имеет меньшее расстояние до прямой PQ, будет иметь меньшее расстояние, чем другая из них, до *каждой* точки луча PQ (и даже прямой PQ). Поэтому надо просто сравнить все расстояния h_i и взять минимальное. Но что, если таких точек несколько? Другими словами, какой должен быть ответ, если нашлось несколько точек, у которых одинаковая координата и расстояние до прямой PQ, а у остальных или координата меньше, или координата такая же, а расстояние до PQ больше? Понятно, что таких точек может быть максимум две — с каждой стороны от прямой PQ (а если $h = 0$, то и вовсе одна). Если их две, то тогда нет ответа: далёкие точки луча не будут покрашены ни в какой цвет, так как луч ляжет ровно по границе между двумя областями, соответствующими этим двум точкам C_i.

Есть и другой, геометрический, способ понять, что надо взять точки с наибольшими координатами, а из них — самую близкую к прямой PQ. Несмотря на то, что совсем строго обосновывать этот способ затруднительно, мы всё равно его обрисуем. Для этого мы будем пользоваться следующим понятием. *Серпер*, или серединный перпендикуляр к отрезку AB — это прямая, которая перпендикулярна отрезку AB и проходит через его середину. Из геометрических соображений можно понять, что серпер образует множество всех точек, которые равноудалены от концов отрезка, и делит плоскость на две половины: те точки, которые находятся в половине с точкой A, ближе к A, чем к B, а те точки, которые находятся в половине с точкой B — наоборот. В диаграмме Вороного границы между областями состоят именно из отрезков серперов к парам точек из контактирующих областей.



Теперь, чтобы сравнить, к какой из двух точек C_i и C_j будет ближе бесконечная часть луча, надо просто разбить плоскость на две половинки серпером к отрезку C_iC_j и посмотреть, в какой из двух половинок окажется почти весь луч. Если C_i и C_j имеют разные проекции и, например, координата у C_i меньше, чем у C_j, то видно, что серпер обязательно оказывается наклонён так относительно луча, что обязательно его пересечёт, причём в одной половине с бесконечной частью луча окажется точка C_j. Если $f(C_i) = f(C_j)$, то серпер параллелен лучу, и луч окажется полностью в одной из двух половинок (в той, в которой расстояние от точки до луча меньше) или просто совпадёт с серпером (если он равноудалён от C_i и C_j). На картинке ниже показано несколько серперов к отрезкам A_iB_i и их взаимодействие с лучом.



Осталось понять, как реализовать поиск нужной точки C_i. Например, среди подгрупп было несколько, в которых было дано, что луч горизонтален. В этом частном случае функция f — это просто x-координата точки, а расстояние до прямой, содержащей луч — модуль y-координаты. Надо было взять все C_i с наибольшей абсциссой и из них выбрать одну или две точки с наименьшим модулем ординаты. Если это две точки, то луч не принадлежит никому, а иначе он принадлежит стране со столицей вот в этой точке.

В общем же случае есть разные подходы к поиску точек. Многие из этих подходов кажутся верными, но некоторые часто работают неправильно, а некоторые — только в специально подобранных тестах. Например, можно было бы напрямую с помощью встроенного типа вещественных чисел проецировать точку C_i на прямую, по теореме Пифагора находить расстояние между точкой P и проекцией C_i и сравнивать, а если совпали — сравнивать опять же вещественные расстояния от C_i до проекций. Тут есть две проблемы. Во-первых, если на самом деле какие-то две проекции совпадают, программа их найдёт с погрешностью, которая, хоть и будет довольно маленькой, приведёт к тому, что программа может счесть проекции различными. Тогда она не будет сравнивать расстояния от проекций до PQ, а просто выберет из них ту, которая на величину погрешности дальше от P, чем другая. Обычно эту проблему решают так: выбирают какое-нибудь вещественное $\varepsilon > 0$ и, когда надо сравнить два числа, которые отличаются меньше, чем на ε, они считаются совпадающими. И здесь возникает вторая проблема. Обычно ε берут больше, чем ожидаемая арифметическая погрешность, но меньше, чем точность, с которой необходимо найти ответ. В данной задаче можно придумать тесты, на которых арифметическая погрешность встроенного типа вещественных чисел большинства языков *превосходит* точность, которая требуется, чтобы отличить одинаковые проекции от разных, из-за чего никакой ε не подойдёт. Тем не менее, если написать собственную реализацию структуры, способную хранить и оперировать вещественными числами с точностью до 10^{-10} или 10^{-11}, то должно получиться с помощью неё сдать задачу.

Жюри же считает предпочтительным следующий метод. Вместо функции $f(X)$ введём функцию $g(X) = f(X) \cdot PQ$. Это проекция вектора $\overrightarrow{PX}$ на ось $\overrightarrow{PQ}$, умноженная на расстояние между P и Q. Если α — угол между векторами $\overrightarrow{PQ}$ и $\overrightarrow{PX}$, то $f(X) = PX \cdot \cos\alpha$, и тогда $g(X) = PX \cdot PQ \cdot \cos\alpha$. Функция $g(X)$ находит произведение длин векторов $\overrightarrow{PX}$ и $\overrightarrow{PQ}$ и косинуса угла между ними. Такая величина называется *скалярным произведением* векторов $\overrightarrow{PX}$ и $\overrightarrow{PQ}$ и обозначается $\left\langle \overrightarrow{PX}, \overrightarrow{PQ} \right\rangle$. Одним из понятных свойств функции $g(X)$ является то, что она отличается от $f(X)$ умножением на некоторую положительную константу; это означает, что результат сравнения $f(X_1)$ и $f(X_2)$ такой же, как у $g(X_1)$ и $g(X_2)$. Другим замечательным свойством скалярного произведения двух векторов является то, что оно является целым, если у обоих векторов целые координаты. Более того, если известны координатные представления векторов $\overrightarrow{z_1} = (x_1, y_1)$ и $\overrightarrow{z_2} = (x_2, y_2)$, то их скалярное произведение можно вычислить с помощью очень простой формулы: $\langle \overrightarrow{z_1}, \overrightarrow{z_2} \rangle = x_1x_2 + y_1y_2$. В нашей задаче все координаты по абсолютному значению не превосходили 10^9, что значит, что скалярное произведение двух векторов, соединяющих по паре точек с допустимыми координатами, не больше $8 \cdot 10^{18}$, поэтому его можно вычислять с помощью стандартного 64-битного целочисленного типа. Таким образом, мы сможем найти все точки с наибольшим $g(X)$ (а значит, и с наибольшим $f(X)$), так что осталось найти ближайшие к прямой PQ. Для этого можно повернуть PQ на 90° (в координатах поворот на прямой угол против часовой стрелки осуществляется так: из вектора (x, y) получается $(-y, x)$) и скалярно умножить этот повёрнутый вектор на $\overrightarrow{PC_i}$. Получится величина $\pm h_i \cdot PQ$ (знак будет плюсом, если повёрнутый вектор смотрит в ту же полуплоскость относительно прямой PQ, что и вектор $\overrightarrow{PC_i}$, и минусом иначе), и надо взять наименьший модуль такой величины. По аналогичным причинам достаточно 64-битных целых чисел, чтобы всё это вычислить. Скалярное произведение вектора $\overrightarrow{z_1}$, повёрнутого на 90° против часовой стрелки, и вектора $\overrightarrow{z_2}$ называется *векторным произведением* векторов $\overrightarrow{z_1} = (x_1, y_1)$ и $\overrightarrow{z_2} = (x_2, y_2)$, обозначается $[\overrightarrow{z_1} \times \overrightarrow{z_2}]$ и вычисляется как $x_1y_2 - x_2y_1$. Сравнивать нам в задаче необходимо величины $\left|\left[\overrightarrow{PQ} \times \overrightarrow{PC_i}\right]\right|$ для тех C_i, которые имеют наибольшие значения $g(C_i)$.

Существует также другое решение, которое гораздо лучше следует методике «сделай ровно то, что просят в условии». Надо найти ту из C_i, которая является самой близкой ко всем далёким

точкам луча. Зачем же проверять все далёкие точки луча? Давайте выберем одну такую точку A на луче PQ, для которой расстояние PA велико, и найдём из всех C_i ближайшую к A. Чтобы работать только с целыми числами (в этом решении точности обычной встроенной вещественной арифметики недостаточно), сделаем точку A целочисленной так: возьмём точку P и отложим от неё вектор $k\overrightarrow{PQ}$, где k — достаточно большое натуральное число (k может зависеть от того, какие координаты у вектора $\overrightarrow{PQ}$). Чтобы сравнивать расстояния AC_i и AC_j, будем сравнивать их квадраты — целые числа. Ну и дальше всё, как просят в условии: найдём наименьший квадрат величины AC_i и выведем i, для которого достигается минимум, а если таких хотя бы два, то выведем -1. Остаётся вопрос: насколько далеко на луче надо брать точку A, в каких пределах будут все вычисляемые значения? Неприятная новость заключается в том, что (как понятно из решения) точка на луче, которую мы берём, должна быть дальше, чем все точки пересечения луча с серперами к парам точек C_i, иначе (как ясно из геометрических рассуждений ранее) ответ может быть неправильным: например, если правильный ответ 1, а мы возьмём точку A, у которой координата будет меньше, чем у точки пересечения PQ с серпером к отрезку C_1C_2, то программа точно не выведет 1.

Насколько же далеко от начала координат может быть точка пересечения луча с серпером? Довольно сложно это оценить точно, однако это расстояние может быть порядка $C \cdot M^3$, где M — максимальный разрешённый модуль координат, а C — какая-то положительная константа. С запасом можно сказать, что самая далёкая точка пересечения находится на расстоянии, меньшем 10^{29}, от начала координат, поэтому достаточно выбрать такую точку A, у которой хотя бы одна из координат не меньше 10^{29} по модулю. Нахождение квадратов и операции сложения говорят нам о том, что нужно уметь работать с числами до 10^{58}, поэтому в этом решении потребуется реализация длинной арифметики (встроенная или вручную написанная). Ограничения в задаче позволяли такому решению уложиться в ограничение по времени, если длинная арифметика не работает совсем уж медленно. Обратите внимание, что в подгруппах, где координаты маленькие по абсолютному значению, можно было реализовать это решение и при этом обойтись без длинной арифметики.

Разбор задачи «Пончики»

Решение на 7 баллов

Заметим, что существует всего 2^n вариантов выпечки пончиков. Рассмотрим их все и промоделируем процесс за линейное время. Общее время работы в таком случае составит $\mathcal{O}(n \cdot 2^n)$.

Решение на 24 балла

Скажем, что каждый день приносит нам один *заряд* пончиков, который можно использовать либо в этот день, либо позднее (иными словами, один свободный заряд можно в любой момент сконвертировать в k пончиков любого цвета). Рассмотрим обычное жадное решение и попытаемся его улучшить.

Пусть в начале i-го дня у нас осталось x неиспользованных пончиков красного цвета, y неиспользованных пончиков синего цвета (имеются в виду пончики, которые были выпечены в предыдущие дни) и c свободных зарядов. Представим, что в i-й день r покупателей пришли за красными и b за синими пончиками. Вначале выгодно отдать им как можно больше пончиков из x и y. Теперь заметим, что если мы можем прямо сейчас отдать хотя бы k пончиков одного цвета, то выгодно использовать заряд и выпечь k пончиков. Также стоит обратить внимание, что пока можно продавать по k пончиков за раз, нам стоит это делать, потому что цвет не имеет значения. Скажем, что после всех трансформаций количество пончиков превращается в r' и b' соответственно, количество свободных зарядов равняется c', а остатки равняются x' и y'. Теперь возможны два варианта:

1. $max(r', b') = 0$. Тогда можно перейти к следующему дню и ни о чем не думать.
2. $max(r', b') > 0$. Это именно тот случай, когда жадный алгоритм ломается, потому что совершенно непонятно, стоит ли использовать заряд сейчас, и если да, то какой цвет предпочесть.

Попробуем пофиксить этот недостаток жадного решения с помощью динамики. Пусть $dp_{i,x,y}$ означает минимальное количество свободных зарядов в начале i-го дня (в частности, это означает, что все остальные заряды использованы). Попробуем понять, как динамика помогает сделать выбор при $max(r', b') > 0$. Здесь тоже возможны варианты:

1. $c' = 0$. В таком случае переход происходит в $dp_{i+1,x',y'}c'$.
2. $c' = 1$. В таком случае совершаются дополнительные переходы вида "использовать один заряд нужного цвета"в состояния $dp_{i+1,x'+k-r',y'}dp_{i+1,x',y'+k-b'}$ со значением $c' - 1$.
3. $c' > 1$. В таком случае к трем предыдущим переходам добавляется переход с одновременным использованием двух зарядов разных цветов, т.е $dp_{i+1,x'+k-r',y'+k-b'}$ со значением $c' - 2$.

После всего ответом будет $max_{i,j}(k \cdot (n - dp_{n,i,j}) - i - j)$ по всем посещенным состояниям динамики. Такое решение работает за $\mathcal{O}(n^3 \cdot k^2)$.

Решение на 50 баллов

Несложно заметить, что остаток пончиков в начале дня по каждому из цветов не превосходит $k - 1$ (иначе k пончиков можно было бы превратить в свободный заряд, что априори выгоднее). Это наблюдение помогает сократить количество состояний и время работы до $\mathcal{O}(n \cdot k^2)$.

Решение на 72 балла

Попробуем улучшить решение из предыдущего пункта. Зафиксируем некоторый x и рассмотрим все $dp_{i,x,y}$ при $y > 0$. Пусть $min_{y>0}(dp_{i,x,y}) = c$ достигается в точке $y = y_0$. Несложно видеть, что значения $dp_{i,x,y>0} > c$ не являются оптимальными (чтобы это понять, достаточно посмотреть на формулу подсчета ответа, т.к один заряд позволяет получить k покупателей, в то время как разность между y и y' всегда строго меньше k). Значение $y = 0$ все равно приходится рассматривать отдельно, так как может случиться ситуация, когда мы никак не сможем продать оставшиеся y, т.е когда выгоднее было сохранить заряд на одном из предыдущих шагов. В остальном переходы являются практически такими же. Количество состояний и сложность уменьшится до $\mathcal{O}(n \cdot k)$.

Альтернативное решение на 72 балла

Чтобы лучше понять решение на 100 баллов, сперва опишем его более простую версию с асимптотическим временем работы $O(n \cdot k)$, которое, тем не менее, содержит почти все идейно важные детали полного решения. Это решение пытается минимизировать количество *потерянных* пончиков. А именно, мы обязаны произвести ровно nk пончиков, но какое-то количество мы не сможем продать — эти пончики мы *потеряли*. Ответ на задачу равен разности $n \cdot k$ и минимального количества потерянных пончиков.

Пусть R_i — общее количество покупателей красных пончиков в последние i дней ($0 \leqslant i \leqslant n$), B_i — аналогичное количество для синих пончиков. В частности, $R_0 = B_0 = 0$.

Зафиксируем какой-то способ выпекать пончики, а именно последовательность целых неотрицательных чисел c_i — сколько раз мы выпекали красные пончики в последние i дней. В силу определения c_i должны удовлетворять условиям $c_0 = 0$ и $c_{i+1} \in \{c_i, c_i + 1\}$ для $0 \leqslant i \leqslant n - 1$. Если c_i удовлетворяют этим условиям, то существует способ выпекать пончики, чтобы c_i было именно таким. В этом случае мы выпекали синие пончики $i - c_i$ раз за последние i дней.

Наше первое наблюдение состоит в том, что в таком случае мы точно потеряли хотя бы $\max\limits_{i=0}^{n} c_i \cdot k - R_i$ красных пончиков и хотя бы $\max\limits_{i=0}^{n}(i - c_i) \cdot k - B_i$ синих пончиков. Действительно, в последние i дней мы испекли $c_i \cdot k$ красных пончиков, а продать могли только R_i, даже если до этого у нас не было пончиков, то мы точно потеряли хотя бы $c_i \cdot k - R_i$ красных пончиков. Аналогично с синими.

Оказывается, верно и обратное! А именно, если мы зафиксируем возможную последовательное c_i, то мы потеряем *ровно* $\max\limits_{i=0}^{n} c_i \cdot k - R_i$ и *ровно* $\max\limits_{i=0}^{n}(i-c_i)\cdot k - B_i$ синих пончиков. Поскольку мы уже показали, что мы потеряем *хотя бы столько* пончиков каждого цвета, то нам теперь достаточно показать, что мы потеряем *не больше*, чем столько пончиков.

Посмотрим на то, как мы продавали красные пончики. Перед каждым днём у нас есть какое-то количество ещё не проданных красных пончиков. Иногда оно равно 0, иногда нет; перед первым днём оно точно равно 0. Рассмотрим последний момент, когда это количество равнялось нулю; пусть после этого мы работали ещё ℓ дней (как мы уже поняли выше, $\ell \leqslant n$; возможно, $\ell = 0$, в таком случае мы вообще не потеряли ни одного красного пончика, а $\max\limits_{i=0}^{n} c_i \cdot k - R_i \geqslant c_0 \cdot k - R_0 = 0 \cdot k - 0 = 0$, то есть в этом случае мы действительно потеряли не больше, чем $\max\limits_{i=0}^{n} c_i \cdot k - R_i$ красных пончиков). Поскольку в последние ℓ дней работы магазина количество оставшихся красных пончиков ни разу не обнулялось, в каждый из этих дней мы смогли обслужить всех пришедших за красными пончиками покупателей. Поэтому в конце у нас осталось $c_\ell \cdot k - R_\ell$ красных пончиков — мы испекли $c_\ell \cdot k$ красных пончиков в последние ℓ дней, продали R_ℓ, а непроданных красных пончиков в запасе у нас не было. Поэтому мы потеряли $c_\ell \cdot k - R_\ell \leqslant \max\limits_{i=0}^{n} c_i \cdot k - R_i$ красных пончиков, что и хотели показать (несложно убедиться напрямую, что в последнем неравенстве на самом деле выполняется равенство; оно и должно это делать, так как иначе $\max\limits_{i=0}^{n} c_i \cdot k - R_i \leqslant$ потерянные красные пончики $= c_\ell \cdot k - R_\ell < \max\limits_{i=0}^{n} c_i \cdot k - R_i$). С синими пончиками аналогично.

Какое отношение это имеет к решению? Самое прямое! Переберём r — сколько красных пончиков мы готовы потерять и для каждого r найдём минимальное количество потерянных синих пончиков, при условии, что мы потеряли не больше r красных. Важное замечание состоит в том, что r имеет смысл перебирать только в отрезке $[0, k-1]$. Действительно, чтобы это понять, вспомним старое решение с динамикой и *зарядами*. В нём оказывалось, что в оптимальном ответе после n дней у нас осталось какое-то количество непроданных красных пончиков от 0 до $k-1$, какое-то количество непроданных синих пончиков от 0 до $k-1$ и какое-то количество неиспользованных зарядов. Но неиспользованные заряды мы можем красить как угодно, в частности мы можем покрасить их все в синий цвет. При таком подходе мы получим оптимальный ответ, в котором может быть много синих потерянных пончиков, но точно не больше, чем $k-1$ красный потерянный пончик.

Положим $d_i = \lfloor \frac{r+R_i}{k} \rfloor$. Несложно видеть, что условие $r \geqslant \max\limits_{i=0}^{n} c_i \cdot k - R_i$ в точности означает, что $c_i \leqslant d_i$ для каждого $i \in [0, n]$. Утверждается, что при фиксированном r мы потеряем в точности $\max\limits_{i=0}^{n}(i-d_i)\cdot k - B_i$ синих пончиков.

Почему? Понятно, что это оценка снизу: так как $c_i \leqslant d_i$, то $\max\limits_{i=0}^{n}(i-d_i)\cdot k - B_i \leqslant \max\limits_{i=0}^{n}(i-c_i)\cdot k - B_i$ для любого выбора c_i. Покажем, как выбрать c_i таким образом, что они будут образовывать корректную последовательность c-шек и максимумы выражений $(i-d_i)\cdot k - B_i$ и $(i-c_i)\cdot k - B_i$ достигались при одном и том же i и совпадали. А именно, положим $c_i := \min(d_i, c_{i-1}+1)$. Грубо говоря, c_i пытается угнаться за иногда прыгающим семимильными шагами d_i (так как $R_{i+1} \geqslant R_i$, то $d_{i+1} \geqslant d_i$; однако разность $d_{i+1} - d_i$ может быть сколь угодно большой и не ограничена сверху числом 1) настолько быстро, насколько может (c_i может делать шаги только длины 1 или 0).

Пусть максимум $(i-c_i)\cdot k - B_i$ достигается в первый раз при $i = \ell$ (то есть ℓ — наименьшее число i, для которого достигается максимум). Тогда либо $\ell = 0$ и $(\ell - c_\ell)\cdot k - B_\ell = 0 = (\ell - d_\ell)\cdot k - B_\ell$ (тут важно, что $d_0 = \lfloor \frac{r}{k} \rfloor = 0$, так как $r < k$), либо $c_\ell = c_{\ell-1}$.

Действительно, есть всего два варианта: $c_\ell = c_{\ell-1}$ или $c_\ell = c_{\ell-1} + 1$. Во втором случае оказывается, что $((\ell-1) - c_{\ell-1})\cdot k - B_{\ell-1} = (\ell - c_\ell)\cdot k - B_{\ell-1} \geqslant (\ell - c_\ell)\cdot k - B_\ell$, то есть максимум выражения достигался и при $i = \ell - 1$ тоже.

Раз так оказалось, что $c_\ell = c_{\ell-1}$, то $d_\ell = c_\ell$: иначе (если $d_\ell \geqslant c_\ell + 1$, $d_\ell < c_\ell$ невозможно) мы бы положили $c_\ell = \min(d_\ell, c_{\ell-1}+1) = c_{\ell-1}+1$. Поэтому $\max\limits_{i=0}^{n}(i-d_i)\cdot k - B_i \geqslant (\ell - d_\ell)\cdot k - B_\ell \geqslant (\ell - c_\ell)\cdot k - B_\ell = \max\limits_{i=0}^{n}(i-c_i)\cdot k - B_i$. Получили оценку в другую сторону, как и хотели. Вместе обе оценки дают $\max\limits_{i=0}^{n}(i-d_i)\cdot k - B_i = \max\limits_{i=0}^{n}(i-c_i)\cdot k - B_i$, что мы и хотели получить.

Из всего вышесказанного решение следует легко: перебираем r от 0 до $k-1$, для каждого r

считаем $d_i := \lfloor \frac{r+R_i}{k} \rfloor$ и $b_r := \max\limits_{i=0}^{n}(i - d_i) \cdot k - B_i$. Выбираем r с минимальным $r + b_r$. Ответ на задачу будет равен $n \cdot k - (r + b_r)$. В такой реализации нам потребуется $O(n \cdot k)$ времени.

Решение на 100 баллов

Уже имея альтернативное решение на 72 балла, довести его до решения на 100 баллов несложно. А именно: будем постепенно увеличивать r от 0 до $k - 1$. Заметим, что каждое d_i (а, соответственно, и b_r) меняет своё значение при таком изменение не больше, чем один раз: это изменение происходит в точности при таком r, что $r + R_i$ делится на k. Поскольку ответ есть $\min_{r=0}^{k-1} r + b_r$, то в качестве кандидатов на r достаточно рассматривать только такие r, в которых происходят изменения и $r = 0$ (остальные r точно не являются оптимальными, так как уступают $b_{r-1} = b_r$, откуда $(r - 1) + b_{r-1} < r + b_r$).

Более того, пока мы увеличиваем r от 0 до $k - 1$ происходит всего не больше, чем $n + 1$ (по одному для каждого i от 0 до n) событий вида "d_i изменилось". Поэтому можно поддерживать массив из чисел $(i - d_i) \cdot k - B_i$ в дереве отрезков и обрабатывать события вида "d_i изменилось" в порядке возрастания k, то нам нужно будет обработать $O(n)$ запросов вида "поменять число в данной позиции" и "спросить минимум во всём массиве".

Такое решение работает за $O(n \log n)$ и с большим запасом по времени проходит последнюю подгруппу.

Также возможна реализация, не использующая продвинутых структур данных вообще и работающая за $O(n)$ после сортировки чисел $(k - R_i) \bmod k$, но она чуть идейно сложнее.