# Разбор задачи «Бинарная матрица»

Посчитаем количество плохих подматриц и вычтем его из общего количества подматриц — $all = (C_n^2)^2$. Для подсчета плохих подматриц просуммируем следующие значения:

1. $a$ — количество подматриц, у которых побитовое «И» строк равняется 01, или 10;
2. $b$ — количество подматриц, у которых побитовое «ИЛИ» строк равняется 01, или 10;
3. $c$ — количество подматриц, у которых побитовое «И» столбцов равняется 01, или 10;
4. $d$ — количество подматриц, у которых побитовое «ИЛИ» столбцов равняется 01, или 10;

Нетрудно заметить, что в этой сумме каждая плохая подматрица учтется ровно два раза, а хорошие подматрицы не учтутся вообще (т.к. в хороших подматрицах побитовые «И»/«ИЛИ» всегда равны 00, или 11).

Таким образом, ответом на задачу будет $all - \frac{a+b+c+d}{2}$. Осталось понять, как быстро вычислять значения $a$, $b$, , $d$. Опишем, как быстро находить $a$, остальные значения находятся аналогичным способом. Переберем $i$, $j$ $(1 \leqslant i < j \leqslant n)$ — номера нижней и верхней строки нашей подматрицы. Пусть $x_i$ — $i$-я строка нашей матрицы в битовой записи. Рассмотрим битовую строку $y = x_i$ `AND` $x_j$. Посмотрим, какие значения $l$ и $r$ будут подходить в качестве двух столбцов нашей подматрицы. Нетрудно понять, что это все пары $(l, r)$ для которых $y_l \neq y_r$. Таким образом, к ответу нужно прибавить $ones \cdot zeros$, где $ones$ — количество единиц в битовой строке $y$, а $zeros$ — количество нулей. Быстро выполнять битовые операции между двумя битовыми строками и находить количество единичных битов можно с помощью структуры `bitset` за $O(\frac{n}{32})$.

Итоговая сложность — $O(4 \cdot \frac{n^2}{2} \cdot \frac{n}{32}) = O(\frac{n^3}{16})$.

# Разбор задачи «Граф»

Для начала выделим все компоненты связности и для каждой компоненты запомним все числа, которые в ней встречаются. Это можно легко сделать за $O(n + m)$ с помощью $dfs$.

Теперь заметим, что для того, чтобы итоговый `OR` был минимальным, нужно чтобы его старший бит был как можно меньше. При фиксированном старшем бите, следующий бит должен быть как можно меньше, и т.д. Поэтому будем находить биты итогового `OR`-а по одному, начиная от самого старшего.

Будем поддерживать маску $mask$, единичные биты которой означают, что в соответствующей позиции итогового `OR`-а обязательно должен стоять ноль. Тогда на очередной итерации нужно будет проверить, можно ли выбрать в каждой компоненте связности число, удовлетворяющее маске $mask + 2^i$. Если такие числа выбрать возможно, то увеличим значение $mask$ на $2^i$, иначе оставим значение $mask$ таким же. Осталось понять критерий соответствия числа $x$ маске $mask$. Этот критерий очень простой: должно выполняться равенство $x$ `AND` $mask = 0$. Действительно, в таком случае все необходимые биты числа $x$ будут равны нулю, а остальные могут быть какими угодно. Таким образом, последовательно проверяя все числа каждой компоненты связности, мы сможем определить очередной бит нашего числа.

Итоговая сложность — $O(m + n \log x)$, где $x$ — максимальное значение $a_i$.

# Разбор задачи «Два массива»

Перепишем искомую сумму следующим образом: $s = \sum_{i=1}^{n}(a_i \cdot 10^{|b_i|} + b_i) = \sum_{i=1}^{n} a_i \cdot 10^{|b_i|} + \sum_{i=1}^{n} b_i$ (здесь $|x|$ означает длину числа $x$). Введем вспомагательный массива $c_i = 10^{|b_i|}$. Т.к. $\sum_{i=1}^{n} b_i$ — константа, то нам нужно минимизировать значение $\sum_{i=1}^{n} a_i \cdot c_i$. Это уже стандартная задача, которая решается жадным способом: нужно отсортировать значения элементов одного массива по возрастанию, а второго — по убыванию.

Итоговая сложность — $O(n \log n)$.

# Разбор задачи «Два числа»

Рассмотрим какой-нибудь оптимальный ответ. Предположим, что в нем мы домножали число $a$ на $p$. Тогда это значит, что количество чисел $p$ в разложении $a$ на простые множители было меньше, чем соответствующее количество в разложении $b$ на простые множители. Действительно, если бы это было не так, то впоследствии нам пришлось бы лишний раз домножать чило $a$ на $p$, а значит ответ не был бы оптимальным.

Поэтому можем перейти немного к другой задаче: вместо того, чтобы домножать число $a$ на $p$, будем делить $b$ на $p$, и наоборот. Заметим, что после применения всех операций в оптимальном ответе, числа $a$ и $b$ станут равны их НОД-у. Поэтому можем изначально поделить $a$ и $b$ на их НОД и решать задачу для новых чисел.

Нетрудно заметить, что теперь каждое число $p$ может встречаться только в разложении одного из чисел $a$, $b$ на простые множители. Поэтому можно применить жадный подход: для каждого заданного числа $p$, проверим, делится ли какое-то из наших чисел на $p$. Если да, то поделим соответствующее число и увеличим количество требуемых операций на 1. Если в конце оба наших числа стали равны 1, то ответ `YES`, иначе — `NO`.

Итоговая сложность — $O(\log(a+b)+n)$.

# Разбор задачи «Дерево и вирус»

Сделаем Эйлеров обход дерева. Теперь у каждой вершины есть время входа $tin[v]$ и время выхода $tout[v]$. Заведем массив $f$. Если какая-то вершина $v$ заблокирована, то в ячейке $f[tin[v]]$ будет записано количество животных, которые находятся в поддереве данной вершины $v$, но при этом между вершиной, в которой находятся данные животные, и вершиной $v$ нет других заблокированных вершин.

С помощью дерева Фенвика, построенного на массиве $f$, мы сможем быстро определить количество животных, которое не сможет заразиться вирусом, если будет заражена вершина $v$. Для этого нужно найти $s = \sum_{i=tin[v]}^{tout[v]} f[i]$ — так мы переберем все заблокированные вершины данного поддерева. А для того, чтобы найти ответ для вершины $v$, нужно предварительно посчитать количество животных в каждом поддереве и вычесть из этого количества значение $s$. Асимптотика данной операции $O(\log n)$ из-за дерева Фенвика.

Посмотрим, как можно делать операции обновления. Заметим, что когда мы изменяем состояние вершины $v$, то в массиве $f$ изменяются всего 2 значения: значение первого заблокированного предка вершины $v$ и значение самой вершины $v$. Оба значения в массиве $f$ изменятся на количество животных, которые заразились бы, если бы в вершину $v$ пустили вирус в то время, когда вершина $v$ не заблокирована — это количество мы уже умеем считать.

Для того, чтобы быстро находить первого заблокированного предка вершины $v$, заведем дерево отрезков (в дальнейшем ДО), в каждой вершине которого будем хранить максимальное значение на подотрезке, за который отвечает вершина ДО. Если вершина $u$ исходного дерева заблокирована, то в позиции $tin[u]$ ДО запишем значение $tout[u]$. Тогда для вершины $v$ найдем на отрезке $[0; tin[v]-1]$ в ДО самое правую позицию, значение в которой больше либо равно $tout[v]$. Сделать это можно спуском по ДО. Нетрудно убедиться, что соответствующая вершина и будет ближайшим заблокированным предком вершины $v$.

Итоговая сложность — $O(m \log n)$.

# Разбор задачи «K нулевых битов»

Существует множество решений данной задачи. Покажем один из способов построить ответ. Изначально присвоим нашему числу значение $2^{k+1}+1 = 100\ldots01_2$ (ровно $k$ нулей). Дальше будем последовательно домножать его на 2, что соответствует дописыванию нуля в конец в двоичной записи. Домножать будем до тех пор, пока число не превысит $10^n$. В первый момент времени, когда число превысит $10^n$, его длина будет не более $n+2$, что всегда не превосходит 2000 при заданных ограничениях.

Единственный нюанс этой задачи — ответ не помещается ни в один из стандартных типов данных. Поэтому нужно было либо написать свою длинную арифметику, либо использовать встроенную

в языках python, java.

Итоговая сложность — $O(n^2)$.

# Разбор задачи «Количество битов»

Нетрудно заметить, что искомое количество равняется $1 + cnt(n)$, где $cnt(n)$ — количество битов в числе $n$. Это следует из того, что $n > \lfloor \log_2 n \rfloor$, а значит единственный бит слагаемого $2^n$ никогда не пересечется с битами числа $n$. Находить $cnt(n)$ можно за $O(\log n)$ последовательным делением пополам.

Итоговая сложность — $O(t \log n)$.

# Разбор задачи «Класс»

Во-первых докажем, что если хотя бы одно из чисел $n$, $m$ кратно двум, то ответ «`ТАК`». Без ограничения общности, будем считать, что $n \vdots 2$. Действительно, в таком случае, студенты в каждом столбце могут разбиться по парам и поменяться друг с другом: первый со вторым, третий с четвертым, и т.д.

Теперь докажем, что если каждое из чисел $n$ и $m$ не кратно двум, то ответ «`НІ`». Для этого раскрасим парты в шахматную раскраску. Заметим, что количество черных клеток будет не равно количеству белых клеток, а т.к. каждый студент хочет пересесть на клетку противоположного цвета, то хотя бы у одного студента сделать это не получится.

Итоговая сложность — $O(t)$.

# Разбор задачи «Крутые подстроки»

Посчитаем количество «некрутых» подстрок и вычтем его из общего количества $\frac{n \cdot (n+1)}{2}$. Заметим, что в каждой «некрутой» подстроке ровно один символ будет встречаться более $\frac{l}{2}$ раз. Переберем этот символ $c$ и посчитаем количество «некрутых» подстрок, у которых доминирующий символ равен $c$.

Построим вспомогательный массив $a$, в котором $a_i = 1$ в случае, когда $s_i = c$, и $a_i = -1$ в противном случае. Теперь можно заметить что подстрока $s[l \dots r]$ будет «некрутой» с доминирующим символом $c$ тогда и только тогда, когда $\sum_{i=l}^{r} a_i > 0$. Посчитаем массив префиксных сумм $sum_i = \sum_{j=1}^{i} a_i$. Теперь задача свелась к тому, чтобы посчитать количество пар $(l, r)$, для которых выполняется условие $sum_r - sum_{l-1} > 0$, или $sum_r > sum_{l-1}$. Нетрудно заметить, что это количество не что иное, как количество инверсий в перевернутом массиве $sum$. Искать это количество можно различным способами, самым простым из которых является проход по массиву с поддержкой дерева Фенвика.

Итоговая сложность — $O(|\sigma| n \log n)$, где $|\sigma|$ — размер алфавита (в нашем случае равен 26).

# Разбор задачи «Покрытие»

Нетрудно заметить, что в данной задаче работает жадный подход. А именно, каждым новым числом $x$ выгоднее всего покрыть максимальное число $a$ из еще непокрытых. Находить такое число проще всего с помощью структуры `multiset`. Перед обработкой всех запросов добавим все числа $a_i$ в `multiset`. Дальше, для каждого запроса $x$ найдем минимальное число меньше или равное $x$ с помощью функции `--upper_bound(x)`. Если такое число существует, то удалим его из `multiset`-а и увеличим ответ на 1.

Итоговая сложность — $O((n + m) \log n)$.

# Разбор задачи «Пончик»

Будем перебирать границу $l$ нашего подмассива справа налево и для каждой такой границы прибавлять к ответу количество подходящих значений $r$. Рассмотрим первые позиции $l_x \geqslant l$ вхождения в массив каждого числа $x$. Заметим, что начиная с этой позиции, число $x$ всегда будет присутствовать в нашем подмассиве, а значит все значения $r$, где $l_x \leqslant r < r_x$, не подойдут (здесь $r_x$ — позиция $m$-го вхождения числа $x$, или $n + 1$, если такой позиции нет). Все значения $(l_x, r_x)$ можно довольно легко поддерживать.

Будем называть позицию $r$ заблокированной тогда, когда существует такое $x$, что $l_x \leqslant r < r_x$. Осталось научиться быстро находить количество разблокированных позиций $r$. Сделать это можно с помощью обычного дерева отрезков, поддерживающего операцию увеличения чисел на отрезке. Каждая вершина этого дерева будет хранить два значения: минимум на отрезке, и количество раз, которое этот минимум встречается на отрезке. Тогда при изменении какой-то пары $(l_{1x}, r_{1x})$ на $(l_{2x}, r_{2x})$ ($l_{2x} \leqslant l_{1x}$, $r_{2x} \leqslant r_{1x}$), сделаем два запроса прибавления:

1. увеличить значения $[r_{2x}+1 \dots r_{1x}]$ на $-1$;
2. увеличить значения $[l_{2x} \dots l_{1x}-1]$ на 1.

Легко понять, что теперь позиция $r$ будет разблокирована тогда и только тогда, когда соответствующее значение в позиции $r$ будет равно нулю. Более того, нетрудно заметить, что в каждой позиции значение будет неотрицательным, а значит для подсчета количества разблокированных позиций достаточно узнать минимум на суффиксе $[l..n]$ и его количество.

Итоговая сложность — $O(n \log n)$.

# Разбор задачи «Четверки на отрезке»

Основной идеей в этой задаче являлось то, что для каждого бита сумму можно было считать независимо. Формально, искомую сумму можно переписать следующим образом:

$$\sum_{l \leqslant i \leqslant j \leqslant k \leqslant w \leqslant r} a_i \oplus a_j \oplus a_k \oplus a_w = \sum_{b=0}^{29} 2^b \sum_{l \leqslant i \leqslant j \leqslant k \leqslant w \leqslant r} x_{i,b} \oplus x_{j,b} \oplus x_{k,b} \oplus x_{w,b}$$

Здесь $x_{i,b} \in \{0,1\}$ — $b$-й бит числа $a_i$. Посмомотрим, как можно находить эту сумму для последовательности из нулей и единиц. Для этого нужно заметить, что очередное слагаемое будет равняться единице в случае, когда среди чисел $x_{i,b}, x_{j,b}, x_{k,b}, x_{w,b}$ нечетное количество единиц, и нулю в противном случае. Поэтому нужно знать всего лишь количество единиц $ones$ на отрезке $l..r$ и количество нулей $zeros = r-l+1-ones$. Зная эти значения, искомую сумму можно очень легко вычислить: она равна $C^1_{ones}C^3_{zeros} + C^3_{ones}C^1_{zeros}$. Осталось научиться быстро находить количество единиц на отрезке от $l$ до $r$. Сделать это можно с помощью дерева отрезков, поддерживающего операции присвоения на отрезке и нахождения суммы на отрезке.

Итоговая сложность — $O(\log x \cdot m \log n)$, где $x$ — максимальное значение $a_i$ в какой-либо момент времени.

# Разбор задачи «Разделите последовательность»

Первым делом нужно было заметить, что вне зависимости от порядка проведения разрезов, итоговая сумма всегда будет одинаковой: $sum = \sum_{i=1}^{k+1} \sum_{j=i+1}^{k+1} s_i s_j$, где $s_i$ — сумма $i$-й части после всех разрезов. В частности, это означает, что разрезы можно проводить последовательно, в порядке увеличения позиций.

Будем считать следующую динамику: $dp_{i,j}$ — максимальная сумма, которую можно получить на префиксе длины $i$, используя $j$ разрезов. Для подсчета значений этой динамики, можно воспользоваться нехитрой рекуррентной формулой: $dp_{i,j} = \max_{q=1}^{i-1}(dp_{q,j-1} + sum(1,q) \cdot sum(q+1,i))$, где $sum(l,r)$ — сумма чисел на отрезке от $l$ до $r$. Однако «в лоб» такую динамику считать не получится, т.к. такое решение работало бы за $O(n^2 k)$. Посмотрим, как можно его ускорить.

Для этого перепишем рекуррентную формулу, используя префиксные суммы $s_r = \sum_{i=1}^{r} a_i$.

$$sum(l,r) = s_r - s_{l-1}$$

$$dp_{i,j} = \max_{q=1}^{i-1}(dp_{q,j-1} + sum(1,q) \cdot sum(q+1,i))$$

$$dp_{i,j} = \max_{q=1}^{i-1}(dp_{q,j-1} + s_q \cdot (s_i - s_q))$$

$$dp_{i,j} = \max_{q=1}^{i-1}(dp_{q,j-1} - s_q^2 + s_q \cdot s_i)$$

$$k_q = s_q$$

$$b_q = dp_{q,j-1} - s_q^2$$

$$dp_{i,j} = \max_{q=1}^{i-1}(k_q \cdot s_i + b_q)$$

Из последней формулы отлично видно, что можно воспользоваться оптимизацией «`convex hull trick`». Более того, т.к. значения массива $s$ только возрастают, то обе операции добавления новой прямой в множество и нахождения оптимальной прямой можно выполнять амортизированно за $O(1)$ с помощью поддержки указателя на текущую оптимальную прямую.

Итоговая сложность — $O(nk)$.

# Разбор задачи «В поисках медианы»

Для начала заметим, что за одну операцию медиана массива может либо не измениться, либо увеличиться на 1. Пусть на шаге $i$ медиана была равна $m$. Попробуем узнать, как изменилась медиана на шаге $i+1$. Посчитаем значение $cnt\_more$ — количество чисел строго больших $m$ в получившемся массиве. Если $cnt\_more > \frac{n}{2}$, то $m$ по определению не может быть медианой, тогда медианой становится значение $m+1$. Если же $cnt\_more < \frac{n}{2}$ то медиана не изменяется, т.е. остается равной $m$. Таким образом можно поддерживать медиану массива после каждой операции. Единственная возникшая проблема — нам нужно каким-то образом быстро узнавать значение $cnt\_more$ после каждого запроса.

Перед первым запросом посчитаем значение $m$ и $cnt\_more$ любым возможным способом (например, отсортируем массив и пройдемся по нему). Далее после каждого запроса будем пересчитывать значение $cnt\_more$. Для этого создадим некоторую структуру (сама структура будет описана ниже), которая может выполнять 2 типа запросов:

1. `add`$(l, r)$ — добавить 1 к элементам массива от $l$ до $r$;
2. `count`$(l, r, x)$ — посчитать количество элементов массива от $l$ до $r$, равных $x$.

Посмотрим, как пересчитывать значение $cnt\_more$ после выполнения запроса. Заметим, что после добавления 1 к подотрезку массива, элементы, которые были больше $m$, и останутся больше $m$ (и не изменят значение $cnt\_more$). Те, которые были меньше $m$, станут меньше либо равными $m$ (и тоже не изменят $cnt\_more$). А все элементы, которые были равными $m$, станут равными $(m+1)$, и должны быть учтены в $cnt\_more$. Поэтому добавим к $cnt\_more$ результат операции `count`$(l, r, m)$. Если же мы увеличиваем медиану на 1, то, аналогично, из значения $cnt\_more$ нужно вычесть `count`$(1, n, m)$, где $m$ — новое значение медианы.

Теперь перейдем к самой структуре, которая умеет быстро обрабатывать вышеуказанные запросы. Разобьем массив на $\sqrt{n}$ блоков. В каждом блоке будем хранить массив $cnt[]$, где $cnt[i]$ — количество чисел $i$ в текущем блоке. Рассмотрим, как выполнить операцию `add`$(l, r)$. Для блоков, которые не полностью попадают в отрезок $[l, r]$, просто пройдем и обновим каждый элемент и соответствующие значения массива $cnt[]$. Для тех блоков, которые целиком входят в отрезок $[l, r]$, заметим следующий факт: если мы увеличим все числа в блоке на 1, то каждое число $i$ перейдет в число $(i+1)$, т.е. массив $cnt[]$ просто «сдвинется» на 1 вправо. Вместо того, чтобы сдвигать весь массив, будем для каждого блока хранить счетчик $shift$, означающий, на сколько мы сдвинули массив в этом блоке. Тогда для того, чтобы узнать количество чисел $x$ (операция `count`), нужно просто обратиться к значению $cnt[x - shift]$ в каждом блоке.

Отдельно стоит обратить внимание на то, что элементы исходного массива достаточно большие (до $10^9$), из-за чего мы не можем завести массивы $cnt[]$ такого размера. Однако, из вышеуказанных утверждений следует, что за $10^5$ операций медиана изменится не более чем на $10^5$, поэтому мы можем работать только с числами от $(m_0 - 10^5)$ до $(m_0 + 10^5)$, где $m_0$ — начальное значение медианы.

Итоговая сложность по времени — $O(n + m\sqrt{n})$.

Итоговая сложность по памяти — $O(n\sqrt{n})$.

# Разбор задачи «Футбол»

Заметим, что после каждого забитого гола, сумма голов в очередном записанном счете увеличится ровно на один. Это значит, что при суммарном количестве забитых голов обеими командами равном $n$, общая сумма всех счетов будет равна $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$. Поэтому можно было перебрать все возможные значения $n$ и для каждого из них проверить, выполняется ли равенство $\frac{n \cdot (n+1)}{2} = x$.

Итоговая сложность — $O(\sqrt{x})$.