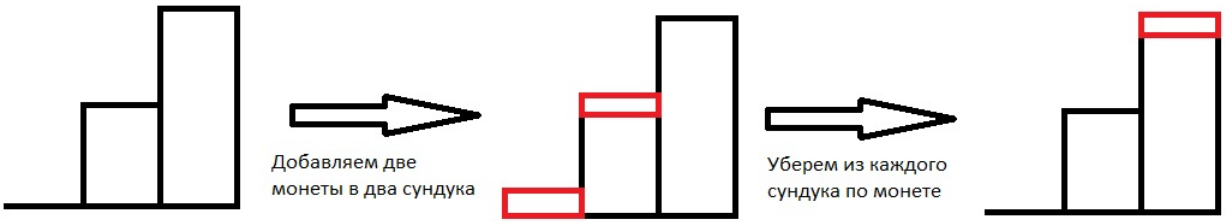
Задача A. Прибытие короля

Отсортируем сундуки по возрастанию так, чтобы в первом было минимальное число монет, а в последнем максимальное. Уменьшим количество монет во всех сундуках на количество монет в первом сундуке. Заметим, что кинуть по монете в два сундука — это тоже самое, что и забрать одну монету из третьего сундука:



Значит, если мы хотим уравнять количество монет в трех сундуках, мы должны убрать лишние монеты (сделать количество монет в сундуках равным нулю). Ответом будет разность между суммой монет в изначальных сундуках и утроенным минимумом количества монет, содержащихся в этих сундуках.

Пример реализации этого решения на языке C++:

```
#include <bits/stdc++.h>

using namespace std;

long long a, b, c;

int main() {
    cin >> a >> b >> c;
    cout << (a + b + c) - 3 * min(a, min(b, c));
    return 0;
}
```

Задача B. Кассовый разрыв

Идея решения: жадно расположим выплаты, уменьшающие счет, как можно раньше, а увеличивающие счет — как можно позже. Докажем, что если в какой-то последовательности выплат мог случиться кассовый разрыв, то в такой последовательности он тоже обязательно случится.

Доказательство: рассмотрим последовательность выплат, в которой случается кассовый разрыв. Рассмотрим самую раннюю операцию, стоящую не на том месте, на котором мы хотим ее расположить.

1. если это операция, уменьшающая деньги на счёте компании, то после ее перемещения в начало диапазона, в каждый момент времени на счёте будет не больше денег, чем в исходной последовательности операций, и кассовый разрыв случится не позже

2. если это операция, увеличивающая деньги на счёте компании, то аналогично при перемещении ее в конец доступного интервала, в каждый момент времени на счёте будет не больше денег, чем в исходной последовательности, и кассовый разрыв произойдет в тот же момент или раньше

Таким образом, чтобы проверить, что кассовый разрыв может произойти, достаточно поставить все операции, уменьшающие счёт, в начало их возможного диапазона, а увеличивающие — в конец.

Если теперь упорядочить все операции по времени, при этом операции внутри одного дня упорядочить так, чтобы уменьшающие шли раньше увеличивающих, то кассовый разрыв равносилен тому, что все операции до какого-то момента времени суммарно уменьшают счёт больше, чем на s. Для этого достаточно просто пройтись по операциям, упорядоченным по времени, и в каждый момент посчитать текущий счёт.

Ограничения задачи позволяли воспользоваться любой сортировкой для упорядочивания событий, и так же считать суммарный счёт в каждый момент времени за $\mathcal{O}(m)$ или $\mathcal{O}(n)$.

Задача C. Реактивные поезда

Давайте представлять себе граф, в котором вершинами являются города, а рёбрами рейсы. Рассмотрим запрос «? v». Для ответа на него достаточно уметь понимать, сколько друзей города v лежат с ним в одной компоненте связности. Когда в граф добавляется новый рейс, какие-то компоненты могут слиться в одну, а когда два города начинают дружить, необходимо будет рассматривать ещё одну вершину в каких-либо запросах.

Понятно, что отвечать на запрос «? v» за количество друзей v слишком медленно; требуется обрабатывать нескольких друзей вершины v за одну операцию. Давайте хранить величину $ans[v]$ для каждой вершины v и пересчитвать эту величину после добавления каждого рейса и каждой пары друзей.

Важное замечание: так как ребра только добавляются, а не удаляются, мы можем поддерживать компоненты связности в графе рейсов, используя СНМ. Давайте скажем, что изначально никакой город ни с кем не дружит, а ни один рейс ещё не пущен, а затем добавим m пар друзей и k рейсов по одному; тогда нам не придётся отдельно разбираться с изначальной конфигурацией.

Нужно научиться пересчитывать $ans[v]$ после добавления новых друзей и рейсов.

Если добавляется пара друзей a, b, нам необходимо увеличить $ans[a]$ и $ans[b]$ на единицу, если a и b лежат в одной компоненте СНМ. Давайте говорить, что $comp[v]$ — это компонента связности, в которой лежит вершина v.

Если новый рейс a, b добавляется в граф, нам нужно объединить $comp[a]$ и $comp[b]$, если до этого a была недостижима из b. Давайте выберем наименьшую компоненту из двух (пусть это $comp[a]$). Рассмотрим вершину $v \in comp[a]$ и её друга u. Если $comp[u] = comp[b]$, это обозначает, что после добавления рейса a, b начнёт существовать путь из v в u и нам будет нужно увеличить $ans[v]$ и $ans[u]$ на единицу. Давайте проверим это условие для каждой пары v, u в явном виде, а затем объединим $comp[a]$ и $comp[b]$ в СНМ.

Каждая вершина v будет перебираться в явном виде не более $\mathcal{O}(\log n)$ раз, ведь, если мы рассматриваем v и всех её друзей, то v лежала в меньшей компоненте при объединении. После этого размер $comp[v]$ увеличится не менее, чем вдвое. Из этого следует, что мы переберём всех друзей v не более $\lfloor \log_2 n \rfloor$ раз. Итоговая асимптотика такого решения будет равна $\mathcal{O}(q+m+(k+q)\log(n)*\alpha(n))$, где $\alpha(n)$ — обратная функция Аккермана.

Задача D. Нарезка пиццы

Докажем следующей факт: существует оптимальное решение, в котором в коробке остаётся не более двух кусков, при этом если их два, то они останутся как два угла, прилегающие к одному диаметральному разрезу, будут в разных половинах, относительно этого диаметрального разреза, и не будут соседними.

Рассмотрим только диаметральные разрезы. Назовём секторы, на которые они делят пиццу **большими**(они могут состоять из нескольких секторов, которые в итоге будут отданы людям), а диаметрально противоположные большие углы объединим в пары. Заметим, что в оптимальном ответе можно как угодно менять порядок секторов внутри одного большого сектора, а также можно как угодно менять порядок пар больших секторов и менять местами большие сектора внутри одной пары. Назовём секторы **бесполезными**, если они в итоге не будут отданы никакому человеку.

Заметим, что если есть два бесполезных сектора внутри одного большого, то можно поменять порядок секторов внутри этого большого сектора так, что бы эти два бесполезных сектора оказались рядом, и после этого убрать радиусный разрез, который проходил между ними. Тогда мы можем улучшить ответ, а значит в оптимальном ответе таких ситуаций нет.

Пусть есть два бесполезных сектора, которые находятся в разных парах больших углов. Поменяем местами пары больших углов местами, что бы эти два бесполезных сектора находились в соседних больших. А также поменяем местами сектора внутри одного большого сектора, что бы эти два бесполезных сектора стали соседними. Тогда мы можем из того диаметрального разреза, который отделяет эти два сектора, оставить только половину(то есть, превратим его в радиусный разрез), который находится с другой стороны от бесполезных секторов. Тогда ответ не уменьшится, большие секторы, в которых находились бесполезные, объединятся в один, а количество бесполезных секторов уменьшится. Значит когда-нибудь, мы сможем получить оптимальный ответ, в котором нету двух бесполезных секторов, которые лежат в разных больших углах.

Значит, бесполезные сектора лежат в не более одной паре больших секторов, и не более по одному в каждом их них, значит их всего не более двух. Ну и если их два, то внутри этих больших секторов опять можно поменять порядок секторов так, что бы бесполезные сектора прилегали к одному диаметральному разрезу.

После того, как мы доказали этот факт, решение становится понятным. Бывает случай, в котором в ответе не будет диаметральных разрезов. Тогда ответом будет либо n разрезов, если сумма углов, которые хотят получить люди, равна 360, либо $n + 1$, если будет сектор, который останется в коробке. Если же будет хотя бы один диаметральный разрез, то проведём его с углом 0, а после этого посчитаем следующую динамику — $dp[mask][sum]$ это минимальное количество разрезов, которое надо провести, чтобы были секторы, которые можно отдать людям из маски `mask`, причём эти секторы расположены так, чтобы занимать `sum` градусов по часовой стрелке от 0 градусов, а оставшиеся секторы тоже лежат подряд по часовой стрелке от 180. Чтобы её посчитать, переберём следующего человека, и отведённый ему сектор положим дальше в одну из двух половин. Тогда, если углы, которые мы отложили от исходного диаметрального разреза, совпадают, то новый разрез проводить не надо, достаточно старый радиусный разрез продлить на другую половину, а иначе надо провести один новый радиусный разрез.

Всего в этой динамике $360 \cdot 2^n$ состояний, из каждого $2n$ переходов. Суммарное время работы $\mathcal{O}(2^n n)$.

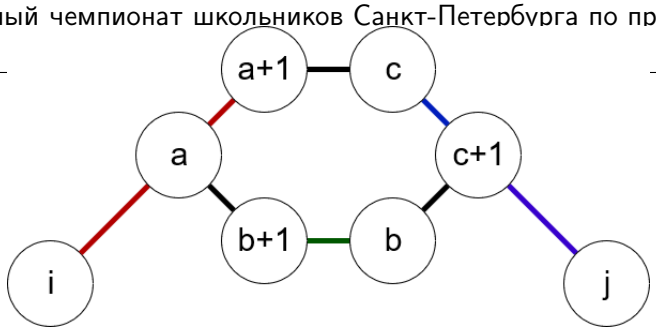
Задача E. Единственное решение

Будем решать задачу рекурсивно. Если в массиве a нет нулей, положим $x_i = a_i \cdot 2^i$, а $m = 2^n - 1$. Заметим, что сумма не может превосходить m, и она достигает m только при данном наборе a. Если в массиве a есть ноль, решим задачу без него, домножим все x_i и m на 2, а x_i при нуле сделаем равным 1. Тогда сумма должна быть четной, значит a_i при единице должно быть равно нулю.

Задача F. Метро 2345

У данной задачи есть множество решений, рассмотрим два различных подхода:

1. **Горе от ума и графы**. Можно заметить, что в данной задаче есть ровно восемь «интересных» станций (стартовая и конечная станции, а также двойные станции пересадок).



Можно построить граф на в котором вершины соответствуют станциям и есть следующие ребра:

- Ребра веса d, между станциями пересадок
- Ребра веса t_i, между соседними станциями одной линии
- Ребра между стартовой вершиной и вершинами пересадок на ее линии (вес считается, как разность номеров вершин умножить на соответствующее t_i)
- Аналогично ребра между конечной вершиной и вершинами пересадок на ее линии
- Ребро между стартовой и конечной вершинами, если они располагаются на одной линии

Затем на построенном графе можно запустить любой из алгоритмов поиска кратчайших путей, например алгоритм Флойда, чтобы найти кратчайшее расстояние между вершинами, соответствующими начальной и конечной станциям.

2. **Просто, но аккуратно**. Можно же было по аналогичному рисунку понять, что существует всего четыре способа движения от стартовой станции до конечной:

- Стартовая и конечная станция на одной линии:
  - Ноль пересадок
  - Три пересадки
- Стартовая и конечная станция на разных линиях:
  - Одна пересадка
  - Две пересадки

После осознания набора этих случаев, остается лишь аккуратно рассмотреть данные случаи и выбрать минимальный.

P.S. Нужно не забыть про 64-битный тип данных независимо от способа решения.

Задача G. Переполнение

Первое приходящее в голову решение — для каждой троичной маски исходных чисел перебрать набор, который мы выбираем, посчитать сумму и проверить, что она делится на m. Общее время работы — $\mathcal{O}(3^n n)$.

Такое решение не проходит по времени, поэтому применим несколько оптимизаций.

Первая идея: будем перебирать двоичные маски, соответствующие множествам чисел, которые мы возьмем со знаком «+». Запомним для каждого остатка по модулю m маску, сумма чисел в которой равна этому остатку. Заметим, что если у двух масок совпали остатки, то теперь можно взять набор, соответствующий первой, с плюсом, а второй — с минусом (а все общие элементы просто выкинуть). Получаем время работы $\mathcal{O}(2^n n + m)$. В наивной реализации это тоже не проходит по времени, поэтому оптимизируем еще немного.

Вторая идея: научимся считать сумму набора по маске за $\mathcal{O}(1)$. Для этого получим из маски другую, выкинув из нее последнюю единицу: $\texttt{prev} = \texttt{mask}\&(\texttt{mask} - 1)$. Теперь можно пересчитать сумму, соответствующую `mask`, через сумму `prev` и $\texttt{mask}\hat{}\texttt{prev}$. Таким образом, мы получили алгоритм за $\mathcal{O}(2^n + m)$, который укладывается во временные ограничения.

Также можно было придумать решение с использованием *meet in the middle*, разбив массив на две половины, и посчитав все возможные остатки в каждой половине за $\mathcal{O}(3^{\frac{n}{2}} + m)$. Теперь, если можно

получить два остатка, дающие в сумме m, то просто объединяем две соответствующие маски. Такое решение работает быстро по времени, но чтобы оно прошло по памяти, нельзя заводить два массива размера m, что, например, можно было сделать, обработав вторую половину чисел, используя уже заполненный массив остатков первой половины.

Задача H. Проверка теста

Нужно для каждой пары работ проверить, являются ли они похожими. Чтобы это сделать, сначала посчитаем количество правильных ответов в каждой работе. Потом для каждой пары работ посчитаем количество одинаковых правильных ответов и количество одинаковых неправильных ответов. Теперь можно проверить условие того, что эти работы похожи.

Посчитаем асимптотику этого решения: решение рассматривает $\frac{m\cdot(m-1)}{2} = O(m^2)$ пар работ, для каждой пары решение просмотривает ответы в этих работах на все n вопросов, поэтому асимптотика суммарного времени работы равна $O(m^2 \cdot n)$.

Задача I. Магический фокус

Выберем произвольную тройку и переберём все варианты её упорядочить.

Будем поэлементно восстанавливать последовательность. Рассмотрим все тройки, содержащие два последних элемента последовательности. Так как входные данные корректны, то их не более двух — тройка из последних трёх элементов или тройка из последних двух и следующего элемента.

Если можем, выписываем следующий элемент. Иначе рассматриваем другой способ упорядочить изначальную тройку.

Как по паре элементов (a, b) сохранить подходящие третьи элементы из троек? Воспользуемся `std::map<pair<int, int>, vector<int>>`, или её аналогами.

Следует разобрать отдельно крайними случаи $n \leqslant 4$, в таком случае ответом является произвольная перестановка.

Задача J. Суперперестановки

Пусть мы хотим найти позицию перестановки p длины $n+1$ в *суперперестановке* s_{n+1}.

Заметим, что циклические сдвиги перестановки p находятся рядом в суперперестановке. Тогда циклически сдвинем перестановку, чтобы максимум стоял в начале, а после этого удалим максимум.

Мы получили перестановку q размера n. Если знать позицию pos_n перестановки q в суперперестановке порядка n, то можно посчитать позицию p в S по следующей формуле:

$$pos_{n+1} = pos_n + (n+1) \cdot cnt_{n+1} + n - \texttt{ind}[n+1]$$

, где $\texttt{ind}[n+1]$ — позиция числа $n+1$ в перестановке p, cnt_{n+1} — количество вхождений числа $n+1$ в s_{n+1} перед вхождением p. Считается, что pos_n — ответ, если отсчитывать индексы с нуля.

Значение cnt_n можем также пересчитывать по следующей формуле:

$$cnt_{n+1} = cnt_n \times (n+1) + n - 1 - \texttt{ind}[n]$$

Таким образом, нужно найти позицию максимального элемента, циклически сдвинуть перестановку так, чтобы максимальный элемент стал первым, а затем удалить максимальный элемент. Это можно реализовать явно с помощью Декартова дерева, либо посчитать $\texttt{ind}[n]$, используя запросы изменения элемента и суммы на отрезке.

Итоговое время работы $\mathcal{O}(n \log n)$.

Задача K. Подготовка тестов

Для удобства, будем рассматривать все массивы в 0-индексации. Рассмотрим элемент массива, с которого начинается подотрезок с тестом. Пусть это элемент на позиции i. Тогда ребра $(a_{i+1}, a_{i+2}), (a_{i+3}, a_{i+4}), \dots (a_{i+a_i\cdot 2-1}, a_{i+a_i\cdot 2})$ должны образовывать ацикличный граф. Сделаем два массива ребер: $e_{0,i} = (a_{i\cdot 2}, a_{i\cdot 2+1})$, и $e_{1,i} = (a_{i\cdot 2+1}, a_{i\cdot 2+2})$. Тогда для четных i нужно проверить, что $e_{1,\frac{i}{2}}, \dots e_{1,(\frac{i}{2}+a_i-1)}$ образуют ацикличный граф, а для нечетных i нужно проверить, что $e_{0,\frac{i+1}{2}}, \dots e_{0,(\frac{i+1}{2}+a_i-1)}$ образуют ацикличный граф. В обоих случаях нас интересует, является ли отрезок ребер ацикличным. Допустим, мы научились для каждой позиции проверять, может ли она

быть началом теста. Тогда для того, чтобы найти ответ, нужно посчитать количество отрезков, разбивающихся на найденные тесты. Причем, в каждой позиции может начинаться максимум один тест. Поэтому, количество отрезков в ответе, начинающихся на позиции i, можно найти следующим образом:

$$dp[i] = \begin{cases} 0, \text{если позиция } i \text{ не является началом корректного теста} \\ dp[i + a_i + 1] + 1, \text{иначе} \end{cases}$$

Осталось научиться решать следующую задачу: дан массив ребер и много запросов l_i, r_i. Для каждого отрезка нужно проверить, образуют ли ребра на отрезке от l_i до r_i ацикличный граф. Это достаточно стандартная задача, которую можно решать разными методами. Например, используя алгоритм Мо вместе с СНМ, или Euler tour tree, или Link-cut tree.

Рассмотрим решение с использованием Мо. Нужно реализовать стандартный алгоритм Мо, но, поскольку СНМ не поддерживает удаление произвольного ребра, для удаления ребер мы всегда будем использовать откаты. Рассмотрим отрезки, левые границы которых попадут в один блок в алгоритме Мо. Если отрезок целиком попал в один блок, найдем для него ответ наивным алгоритмом. Правые границы всех остальных отрезков, левые границы которых попали в один блок, лежат правее конца блока. Обработаем эти запросы по увеличению правой границы. Для того, чтобы найти ответ на очередной отрезок, добавим в СНМ все ещё не добавленные ребра от конца блока до правой границы отрезка, после этого добавим ребра от конца блока до левой границы отрезка. Найдем ответ для текущего отрезка ребер, и откатим добавленные ребра, находящиеся левее правой границы блока.

Финальное время работы: $\mathcal{O}(n\sqrt{n}\log n)$ с использованием алгоритма Мо и СНМ с откатами. Если использовать решение с Euler tour tree или Link-cut tree и двумя указателями (для каждой левой границы нам интересна максимальная правая, что граф на этих ребрах ацикличен), можно достичь времени работы $\mathcal{O}(n\log^2 n)$ или $\mathcal{O}(n\log n)$.