,

Задача A. Планировщик для робота

Для решения этой задачи необходимо организовать очередь с приоритетом, то есть кучу (Heap). Куча — структура данных, которая может выдавать минимальный или максимальный элемент за O(1), а удалять его или добавлять новый элемент за O(log(n)). Соответственно, классическая куча должна поддерживать две процедуры: добавление элемента (add) и удаление элемента с наивысшим (или наименьшим) приоритетом (pop).

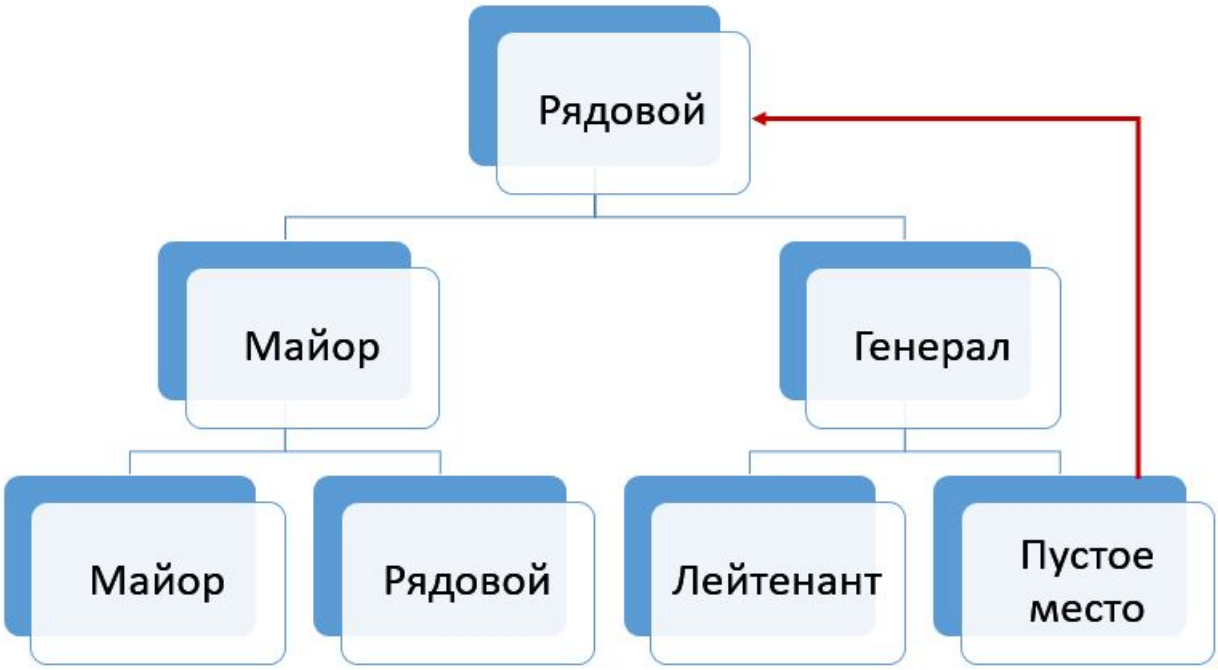
Если использовать обычную очередь, то нам придётся в её начало поместить все элементы с наивысшим приоритетом, за ними — с приоритетом чуть меньше, за ними — со следующим и так далее. Неудобство использования такого подхода состоит в том, что новому элементу придется долго искать своё место среди всех элементов (это будет происходить за O(n)).

Чтобы понять концепцию кучи, представьте себе такую ситуацию (не связанную с нашей задачей напрямую). На приём к военному врачу пришли два генерала, два майора, лейтенант и двое рядовых. Естественно, очередь (вернее, кучу) они организуют так, чтобы человек с более высоким воинским званием прошёл к врачу раньше (даже если по времени он пришёл позже). Выстроятся они, например, вот таким образом:

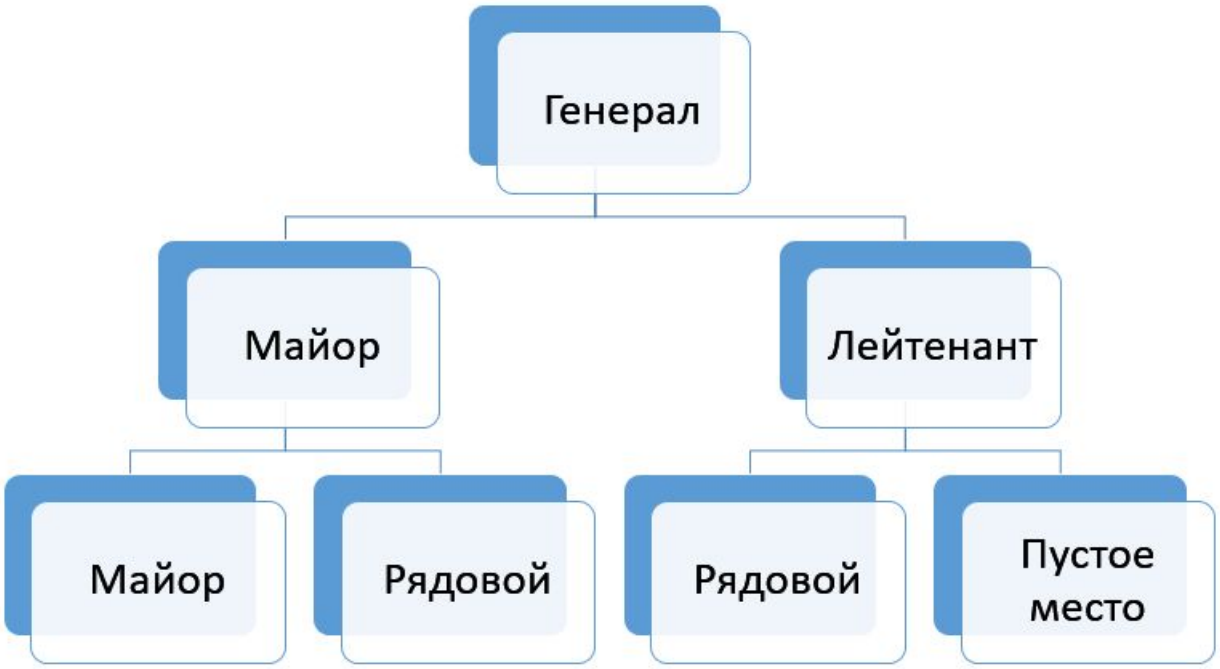
Самым верхним в куче стоит человек с наивысшим званием. Именно он зайдет в кабинет следующим. Непосредственно за ним стоят двое других военнослужащих, которые определят между собой, кто займет ме5сто генерала, когда тот зайдет в кабинет, а кто останется на своём месте. Естественно, из двух людей выше поднимется тот, у кого звание выше (если звания одинаковые — можно выбрать любого). Это правило распространяется на любое освободившееся место в куче.

Удобнее и проще использовать двоичные кучи, это значит, что за каждым человеком стоит не более двух людей с приоритетом не выше, чем у него.

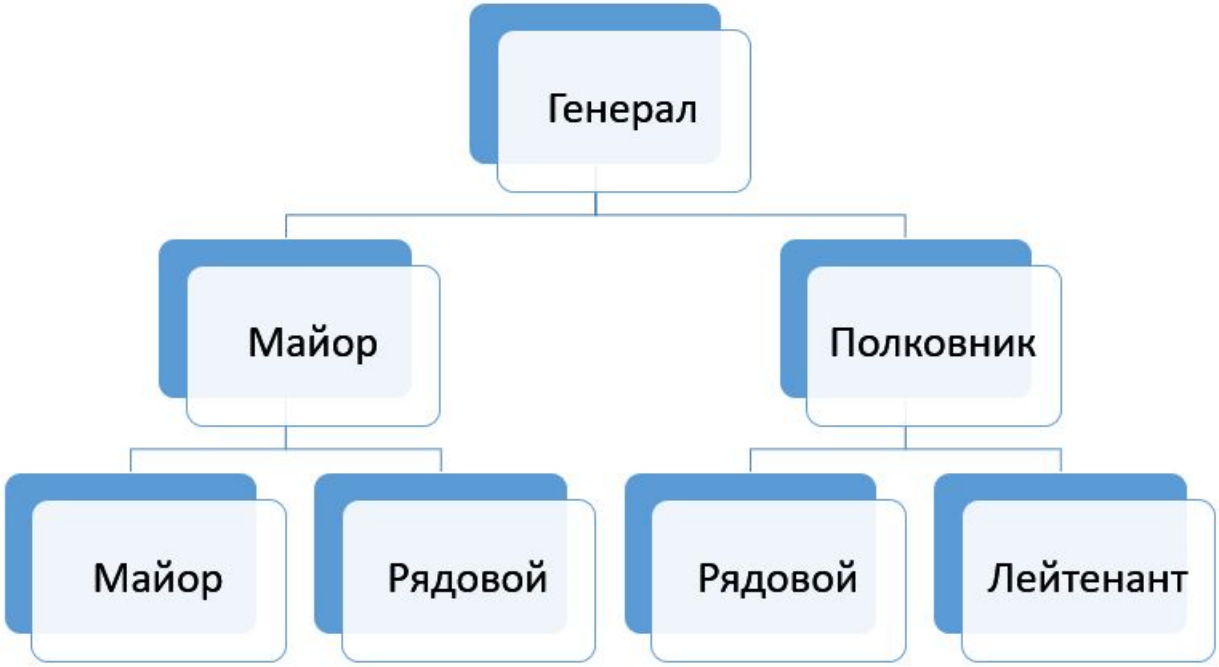
Доктор начал приём и первый генерал зашел в кабинет. На его верхнее в куче место перемещается последний элемент кучи — рядовой. Теперь приоритет нарушен и его нужно восстановить. Этот процесс называется «просеивание».



Из двух людей, которые сейчас стоят за рядовым более высокий приоритет у генерала. Генерал и рядовой меняются местами. Теперь за рядовым стоит лейтенант, что опять неправильно — лейтенант и рядовой меняются местами. Теперь куча выглядит так:



Предположим, теперь к врачу пришел новый пациент в звании полковник. Он занимает свободное пустое место. Увидев, что перед ним стоит лейтенант, полковник меняется с ним местами. Теперь перед полковником стоит генерал, поэтому выше полковник не поднимется, пока это место не освободится.



Обратите внимание, что новый элемент обменивался местами не со всеми элементами, которые имели приоритет ниже его. То же самое касается и удаления элемента из кучи. Поэтому куча работает быстрее (за «высоту» бинарного дерева, которое выражается через логарифмическую функцию от количества элементов в нём). А вот для её хранения достаточно одномерного массива, что очень удобно.

Подробнее о куче и её реализации можно почитать, например, в учебнике Фоксфорда: https://foxford.ru/wiki/informatika/kucha-heap или в Википедии (статья Двоичная куча).

Возвращаясь к задаче «Планировщик для робота», вам останется просто смоделировать описанный в условии процесс. Пока не все задания роботу даны, в кучу будут циклически добавляться по два новых задания и одно удаляться. Когда все задания кончились — только удаляться.

Задача B. Бумажный дом

Центр квадрата будет иметь координаты $(\frac{b+\frac{a}{2}}{2}, \frac{b+\frac{a}{2}}{2})$.

Тогда его верхняя точка будет иметь координаты $(\frac{b+\frac{a}{2}}{2}, \frac{b+\frac{a}{2}}{2} + \frac{a}{2})$.

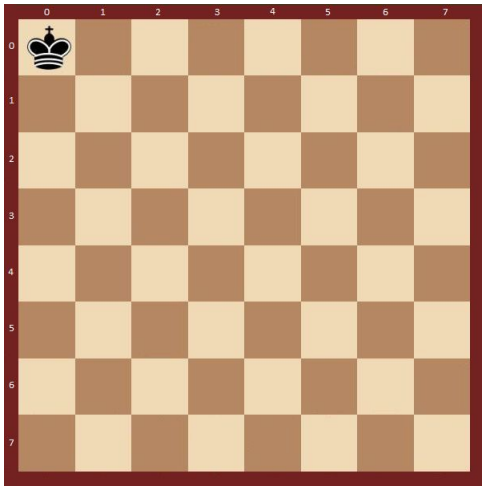
Осталось проверить, выше ли эта точка, чем верхняя точка треугольника с координатами $(0, b)$.

Задача C. Необычная игра

Задача на двумерное динамическое программирование (подсчёт числа маршрутов).

Вспомним классическую задачу: на шахматной доске в левом верхнем углу находится король. Король может перемещаться только вправо или вниз на одну клетку. Необходимо определить количество различных маршрутов короля, приводящих его в правый нижний угол.

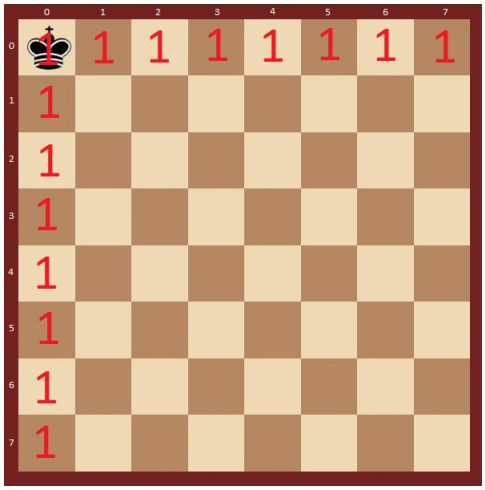
Сопоставим каждой клетке ее координаты (i, j), где i будет обозначать номер строки на доске, j — номер столбца. Нумеровать строки будем сверху вниз, столбцы — слева направо, нумерация начинается с 0. Тогда начальное положение короля будет клетка $(0, 0)$.



Обозначим через $F(i, j)$ количество способов прийти из клетки $(0, 0)$ в клетку (i, j). В клетку (i, j) можно прийти из двух клеток — слева из $(i, j-1)$ и сверху из $(i-1, j)$. Поэтому число маршрутов ведущих в клетку равно числу маршрутов из обоих её предшественников, а именно:

$$F(i, j) = F(i, j-1) + F(i-1, j).$$

Отдельно нужно задать значения для граничных клеток, то есть когда $i = 0$ или $j = 0$. Во все эти клетки король может добраться единственным образом.



В результате получится таблица заполненная следующим образом:

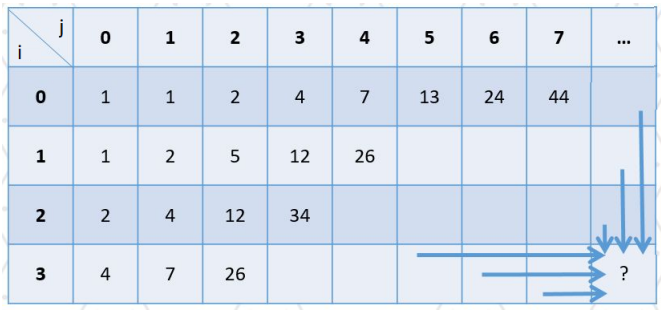
1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792
1	7	28	84	210	462	924	1716
1	8	36	120	330	792	1716	3432

Всего 3432 маршрута.

В нашей задаче счёт игры (и положение в таблице) может меняться не только на 1, а ещё и на 2 или 3. Получается, что опять $F(i, j)$ — количество способов прийти из клетки $(0, 0)$ в клетку (i, j). В клетку (i, j) можно прийти из шести клеток — слева из $(i, j-1)$, $(i, j-2)$ и $(i, j-3)$, а также сверху из $(i-1, j)$, $(i-2, j)$, $(i-3, j)$.

Поэтому число маршрутов ведущих в клетку равно числу маршрутов из шести её предшественников, а именно:

$$F(i,j) = (i,j-1) + (i,j-2) + (i,j-3) + (i-1,j) + (i-2,j) + (i-3,j).$$



i \ j	0	1	2	3	4	5	6	7	...
0	1	1	2	4	7	13	24	44	
1	1	2	5	12	26				
2	2	4	12	34					
3	4	7	26					?	

По прежнему, отдельно нужно задать значения для граничных клеток, то есть когда $i < 3$ или $j < 3$. Для них будет не шесть способов, а меньше.

Осталось посчитать количество способов попасть из клетки $(0,0)$ в клетку (a,b). Отдельно посчитать количество способов попасть из клетки (a,b) в клетку (c,d) и перемножить эти числа.

Задача D. Никита и очки

Задача на рекурсию (вызов функцией самой себя с другими параметрами). Для того, чтобы реализовать рекурсию нужно ответить на следующие вопросы:

1) Какой случай (для какого набора параметров) будет крайним (простым) и что функция возвращает в этом случае?

2) Как свести задачу для какого-то набора параметров (за исключением крайнего случая) к задаче, для другого набора параметров (как правило, с меньшими значениями)?

Определим крайний случай — когда заготовка сразу идет в отходы. Это случится, когда любая из сторон меньше трёх или когда обе стороны меньше пяти.

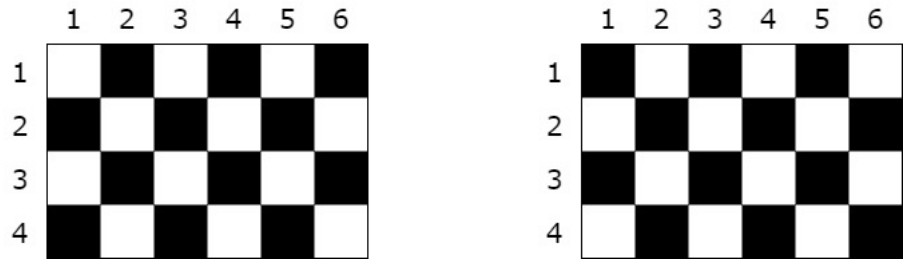
В противном случае пытаемся расположить новую заготовку вдоль и поперёк внутри текущей, как это сделано в примере 3. Иногда новою заготовку можно расположить единственным образом внутри текущей.

При этом полезно отслеживать, на какую глубину рекурсии мы ушли (чтобы определить количество одинаковых маленьких кусочков, которые у нас в конце-концов останутся). При каждом погружении в рекурсию на один уровень количество одинаковых деталек (из которых в конце-концов ничего не получится сделать) удваивается. В случае наступлении крайнего случая определяем размер заготовки, из которой уже ничего получить нельзя и узнаём общее количество таких заготовок (их площадь). Если новая площадь меньше, чем определённая к текущему моменту — обновляем это значение.

Так, в примере 3 при эффективном построении мы ушли на глубину рекурсии 3 и получили в крайнем случае заготовку размером 1 на 1. Ответ для этого варианта изготовления равен $2^3 \times a \times b = 8$. При неэффективном построении мы ушли на глубину рекурсии 2 и получили в крайнем случае заготовку размером 2 на 8. Ответ для этого варианта изготовления равен $2^2 \times a \times b = 64$.

Задача E. Странные шашки

Заметим, что существует всего два способа раскрасить доску для шашек так, чтобы любые две черные клетки и любые две белые клетки не имели общей стороны.



Кроме того, способ явно определяется цветом левой верхней клетки:

- если она белая, то и все клетки с **четной суммой номера строки и столбца** должны быть белыми, а **нечетной** — черными
- иначе, белыми должны быть клетки с **нечетной суммой**, а черными — с **четной**.

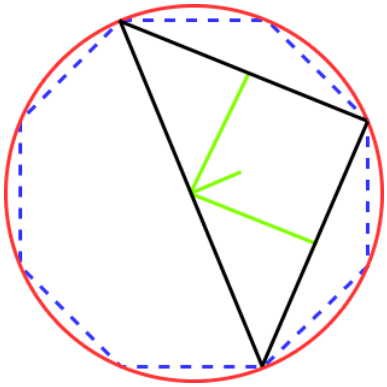
Давайте рассмотрим оба способа и для каждого посчитаем количество клеток, у которых цвет не совпадает с цветом на Петиной доске. Выберем способ, в котором таких клеток меньше, выведем количество таких клеток. Этом можно сделать за два прохода по всем клеткам доски. Далее, за еще один проход выведем клетки, которые нужно будет перекрасить.

Время работы такого решения — $\mathcal{O}(N \cdot M)$.

Задача F. Правильный боулинг

Давайте разобьем задачу на две подзадачи: найти центр правильного многоугольника и найти число углов.

Центр правильного многоугольника должен быть на одинаковом расстоянии от трех вершин. Рассмотрим треугольник, образованный данными точками.



Он может иметь произвольный вид, однако нетрудно заметить, что он вписан в ту же окружность, что и правильный многоугольник. Центр описанной окружности треугольника — точка пересечения срединных перпендикуляров. Рассмотрим отрезки $(x_1, y_1)-(x_2, y_2)$, $(x_2, y_2)-(x_3, y_3)$, найдем их середину, построим перпендикуляры, пересечение их даст искомый центр. Для этого нужно представить каждый из отрезков в виде векторов, найти перпендикулярные из прямые и их пересечение.

Зная центр многоугольника, найдем число углов в нем. Поскольку, при равном расстоянии от центра до вершины площадь правильного многоугольника возрастает монотонно с ростом N (пусть N — число углов), можно просто перебирать все N, начиная с 3, и при нахождении первого подходящего количества вершин печатать ответ и прекращать работу.

Для этого нужно для определенного N уметь определять, могут ли данные три точки быть вершинами N–угольника. На самом деле, это не сложно проверить: углы между отрезками от центра до вершин должны быть кратны $\frac{2\pi}{N}$.

Пусть $a_i = atan2(y_i - y_{center}, x_i - x_{center})$ — угол между осью координат и прямой от центра до вершины i. Воспользуемся свойством синуса: $\sin(x) = 0$, если аргумент кратен π. Условие кратности угла между двумя отрезками заменяется при этом на следующее: $\sin\left(\frac{N \cdot (a_i - a_j)}{2}\right) = 0$. Так как мы имеем дело с арифметикой конечной точности, то сравнение нужно проводить с точностью, до некоторого ε.

Время работы решение линейно зависит от максимального числа вершин в правильном многоугольнике, которое ограничено 100.

Задача G. Эрудит

Задачу можно переформулировать так: можно ли из символов первых двух строк составить слово в третьей строке?

Для решения этой задачи нужно уметь по символу латинского алфавита получать его порядковый номер. Например, «A» $\to 1$, «B» $\to 2$, ...

Давайте заведем два массива a и b по 26 элементов — количество вхождений буквы с номером i в первых двух строчках и в третьей соответственно. Почитать количество вхождений можно за один проход по каждой строчке, увеличивая элемент массива, соответствующий текущему символу, на один.

Заметим, что если массив a поэлементно равен массиву b, то из символов первых двух строк можно составить слово в третьей строке. Значит, ответ в таком случае «YES». Если массивы не равны, то какие-то символы останутся лишними или каких-то не хватит, то есть ответ «NO».

Время работы такого решения асимптотически равно времени, необходимому на чтение трех строчек.

Задача H. Красная шапочка и волк

В этой задаче достаточно было в точности воспроизвести последовательность событий, происходящих на прямой от норы волка до дома бабушки.

В качестве точек отчёта можно выбрать события:

- волк находится в норе и собирается начать бежать, красная шапочка где-то на прямой,
- волк догнал красную шапочку.

Тогда достаточно отслеживать время и координаты красной шапочки. Впервые событие первого типа произойдет в момент t, когда красная шапка отбежит на $t \cdot v_p$ от начала леса. Если при этом она уже добралась до дома бабушки, пирожки вообще не понадобятся.

Иначе можно начинать цикл: промежуток времени, через которое произойдет событие второго типа, равно расстоянию от норы волка до девочки, разделенному на их скорость сближения (разность скоростей волка и девочки).

Сдвинем положение красной шапочки на расстояние, которое она успеет пробежать, и проверим, не попался ли по дороге бабушкин дом. Если нет, то волк успеет ее догнать, значит понадобится еще один пирожок. Второе действие в цикле — найти промежуток времени, через который произойдет событие первого типа, то есть волк вернется в нору. Этот промежуток равен новой координате девочки, деленной на скорость волка + время на поедание пирожка (f). Передвигаем девочку расстояние, которое она успеет пробежать, пока волк возвращается и ест пирожок.

Будем повторять пока красная шапочка, наконец, не добежит до дома бабушки.

Сложность алгоритма линейно зависит от количества итераций цикла, описанного выше. Из интуитивных соображений, больше всего итераций понадобится при максимальных скорости волка, расстоянии до дома бабушки и минимальной скорости девочки. Таким образом, время работы решения можно оценить, как $\mathcal{O}(c)$.

Задача I. Надежный код

Разделим код на две части по n цифр в каждой. Запишем в массив a первые n цифр, а в массив b — последние n цифр. Отсортируем элементы этих массивов по возрастанию. Для этого можно использовать сортировку из стандартной библиотеки.

Нетрудно понять, что если для всех i выполняется $a[i] < b[i]$ или для всех i выполняется $a[i] > b[i]$, то код точно не надёжный. Действительно, каждой цифре из первой половины мы взаимно однозначно сопоставили цифру из второй, и такое соответствие удовлетворяет условию ненадёжности. Также заметим, что для любого ненадежного замка выполняется или первое условие, или второе. Таким образом, мы нашли критерий, по которому можно проверить замок на надежность.

Итак, считаем цифры замка в два массива, отсортируем их, проверим выполняется ли условие ненадежности и выведем ответ. Время работы такого решения — $\mathcal{O}(n \log n)$.

Задача J. Разбиение на команды

Нам нужно разбить всех одноклассников на две равные команды и соблюсти все пожелания.

Бывают ли такие комбинации пожеланий, которые не позволяют разбить на две, пусть даже не равные по количеству участников, команды? Действительно, пусть есть тройка ребят A, B, C, и

пары (A, B), (B, C), (A, C) не могут быть в одних командах, тогда мы не можем распределить их по двум командам, чтобы выполнить все пожелания.

Давайте представим все пожелания в виде графа, где вершины — ребята. Будем проводить ребро между вершинами a и b, если одноклассники a и b не хотят быть в одной команде. Тогда ситуация, описанная выше, будет представляться треугольником в таком графе. Давайте попробуем раскрасить этот граф в два цвета. Если получится, то в графе нет треугольников и одноклассников можно распределить на две команды. Для раскраски графа в два цвета достаточно запустить обычный DFS (алгоритм поиска «в глубину»). Граф пожеланий может быть не связным, поэтому DFS придется запускать из всех вершин, которые до этого не были покрашены.

Предположим, граф получилось раскрасить в два цвета. Давайте выберем одноклассников, которые попадут, например, в первую командую.

Поскольку граф может состоять из нескольких компонент связности, в каждой такой компоненте мы должны выбрать один цвет и всех ребят с таким цветом взять в команду. То есть, в каждой компоненте связности надо выбрать один цвет из двух так, чтобы суммарное количество ребят в выбранных цветах компонент было равно n, тогда распределение будет равным.

Явный перебор здесь не подойдет, так как компонент может быть много, поэтому задачу нужно решать при помощи динамического программирования. Давайте последовательно рассматривать все компоненты связности и поддерживать массив флажков $dp[0 \dots 2n]$: можно ли получить команду со столькими участниками из ребят, входящих в ранее рассмотренные компоненты связности. Рассматривая следующую компоненту с $blue$ вершинами, покрашенными в синий, и red — в красный, пометим $newdp[i]$ истинным, если $dp[i - red]$ или $dp[i - blue]$ истинно. Если после рассмотрения всех компонент $dp[n]$ истинно, то разделить одноклассников на равные команды можно, иначе выведем `IMPOSSIBLE`.

Чтобы вывести само распределение, давайте дополнительно в динамическом программировании для каждого пересчета dp запоминать то, как мы получили текущее значение (какой цвет из двух мы взяли). Тогда, рассматривая выбор цветов в компонентах с конца, можно однозначно восстановить способ, которым мы получили n человек в команде.

Время работы такого решения — $\mathcal{O}(n^2)$.

Задача K. Сигнал из космоса

В условиях было дано неявное описание прямоугольника. Действительно, это выпуклый многоугольник, у которого четыре стороны, между которыми прямые углы.

Таким образом, задача сводится к вопросу: можно ли из отрезков с длинами a, b, c и d составить прямоугольник? Для ответа на этот вопрос нужно рассмотреть три соотношения между длинами сторон:

- $a = b$ и $c = d$;
- $a = c$ и $b = d$;
- $a = d$ и $b = c$.

Если хотя бы одно из этих соотношений выполняется, нужно вывести «`YES`», и иначе ответ — «`NO`».