Задача A. Долгое путешествие в Одинцово

Будем двигаться **в обратном порядке** — от последнего рейса к первому.

- Сначала положим $t_n = d$ — последний возможный день, когда Демид может поехать последним автобусом. Но поскольку автобус ходит только в дни, кратные x_n, фактически Демид сможет поехать в день:
$$t_n = \left\lfloor \frac{d}{x_n} \right\rfloor \cdot x_n.$$
- Далее, для $(n-1)$-го рейса, Демид должен поехать в день t_{n-1}, который не позже t_n и кратен x_{n-1}. Следовательно:
$$t_{n-1} = \left\lfloor \frac{t_n}{x_{n-1}} \right\rfloor \cdot x_{n-1}.$$
- Аналогично продолжаем процесс, пока не дойдём до t_1.

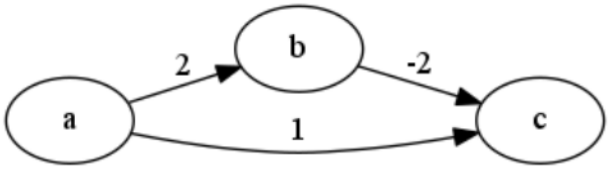
После этого t_1 и будет искомым — самым поздним днём, когда Демид может начать своё путешествие и всё ещё успеть к дню d.

Корректность.

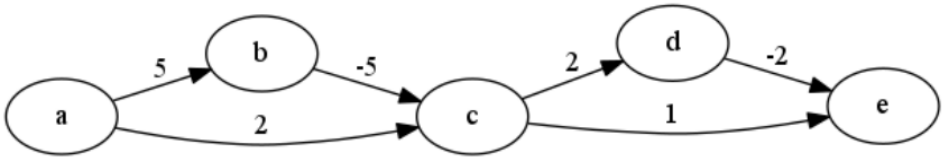
Поскольку на каждом шаге мы выбираем максимально возможный день, кратный x_i, но не превышающий день следующего рейса, полученная последовательность $(t_1, t_2, \ldots, t_n)$ гарантирует завершение не позже d и при этом имеет максимально возможное значение t_1.

Задача B. Ломать Дейкстру

Ниже представлен минимальный пример графа, который заставит алгоритм Дейкстры дважды оценить одну и ту же вершину (в данном случае вершину c). Алгоритм будет обрабатывать вершины в следующем порядке: a на расстоянии 0, c на расстоянии 1, b на расстоянии 2, а затем снова c на расстоянии 0.



Теперь мы можем объединить два таких блока следующим образом:



Такая конфигурация заставит алгоритм пересчитать последнюю вершину 4 раза: на расстоянии 3, затем 2, затем 1 и , наконец, 0. Вышеуказанный шаблон можно легко обобщить до произвольной длины; количество пересчётов последней вершины всегда будет равно 2^t, где t — количество использованных нами треугольников. Достаточно длинная такая конфигурация, подобная этой, достаточна для решения нашей задачи.

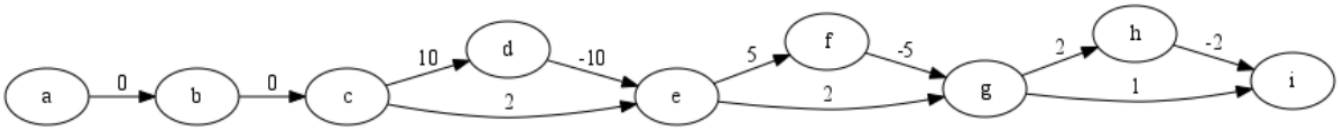
Подправить конфигурацию для решения задачи для произвольного p не так уж сложно. Вот один из вариантов: обратим внимание на самое левое горизонтальное ребро (ребро $a \to c$ на нашем рисунке выше). Уменьшая его длину, мы можем очень плавно уменьшить количество вершин, обрабатываемых алгоритмом. Чтобы построить граф с заданным количеством обработанных вершин, мы можем:

1. Найти наименьшее количество треугольников, которое даёт как минимум необходимое количество обработанных вершин. Другими словами, строим такую конфигурацию, чтобы количество обрабатываемых вершин было хотя бы p.

2. Уменьшаем длину первого горизонтального ребра, чтобы количество обработанных вершин было либо равно p, либо немного меньше.

3. При необходимости добавляем цепочку вершин, которая будет обработана только один раз, в начале.

Ниже можно увидеть пример построения для $p = 20$. Здесь мы уменьшали вес ребра $c \to e$, а затем добавили две вершины a и b в начало конфигурации.



Задача C. Побег по коридору

Для начала научимся решать задачу без восстановления ответа. Будем рассматривать все y-координаты и поддерживать два множества:

- В первом множестве мы будем хранить тройки чисел (l, r, y) — в координате y мы могли закончить в любой x-координате между l и r. Если мы сейчас находимся в координате y_2, то на самом деле сейчас мы можем закончить в любой x-координате между $l-(y_2-y)$ и $r+(y_2-y)$.

  Мы будем поддерживать эти отрезки таким образом, чтобы они не пересекались между собой.

- Во втором множестве мы будем поддерживать моменты времени, когда ближайшие (соседние) отрезки первого множества начнут пересекаться. Это множество нам потребуется для того, чтобы объединять соседние отрезки в один. Кроме этого момента времени нужно будет также хранить, какие именно отрезки пересекутся в этот момент времени.

Теперь нам нужно научиться обрабатывать отрезки запрещённых точек. Для этого мы можем рассмотреть все отрезки, с которыми мы сейчас пересекаемся, и удалить их из множества. Некоторые из них нужно будет не просто удалить, а ещё и добавить в множество их части, которые не пересекались с удаляемым отрезком, но таких отрезков будет не более двух. Найти эти отрезки можно будет с помощью метода lower_bound или upper_bound (зависит от реализации) и дальнейшего прохода по множеству, начиная с этого найденного итератора. Более того, в зависимости от реализации вам может потребоваться реализовать собственный компаратор для множества.

В процессе пересчёта первого множества мы также должны не забывать пересчитывать второе множество. Если после рассмотрения всех y-координат наше первое множество будет не пустым, то ответ существует, и далее мы обсудим, как его восстановить.

Для этого изучим, как образовывались состояния (l, r, y) во время работы нашего решения:

- Мы могли объединить два отрезка в один. Тогда мы можем любой из них считать предком нашего состояния (l, r, y).

- Мы могли обрезать отрезок, когда учитывали запрещённые точки. Тогда исходный отрезок (который был до) будем считать предком нашего состояния (l, r, y).

Другими словами, во время пересчёта первого и второго множества предлагается дополнительно поддерживать ещё и массив предков, с помощью которого мы сможем восстановить ответ.

Задача D. Побег из общежития

Будем решать задачу конструктивно:

- Если $k = r \cdot c - 1$, то ответа не существует. Иначе он всегда существует.
- Чтобы построить ответ для заданного k, давайте обойдем змейкой все ячейки матрицы. Допустим, мы обходим ячейки в порядке $v_1 \to v_2 \to \ldots \to v_{r \cdot c}$. Двери ориентируем так, чтобы студенты могли пройти вдоль всего найденного нами пути. Другими словами, выберем расположение дверей согласно ориентации найденного пути. Таким образом, мы найдем ответ для $k = r \cdot c$.

  Если $k < r \cdot c$, то нам достаточно изменить расположение двери в ячейке $v_{r \cdot c - k}$, чтобы зациклить маршрут для всех студентов, которые будут стартовать из ячеек $v_1, v_2, \ldots, v_{r \cdot c - k}$.

Решение можно реализовать за $\mathcal{O}(r \cdot c)$.

Задача E. Первомайская

Решим задачу разбором случаев:

- Пусть $d_1 = t_1 \cdot a_1 + t_2 \cdot a_2$ и $d_2 = t_1 \cdot b_1 + t_2 \cdot b_2$. Другими словами, d_1 и d_2 это дистанция, которую пробегают Рома и Вова за $t_1 + t_2$ минуты. Тогда если $d_1 = d_2$, то ответ `infinity`.
- Далее решаем задачу для $d_1 \neq d_2$. Давайте также будем считать, что $d_1 < d_2$, так как иначе можно просто поменять местами значения a_1 и b_1; a_2 и b_2.

  Если выполняется, что $a_1 \cdot t_1 < b_1 \cdot t_1$, то ответ будет равен 0. В этом случае Вова всегда будет опережать Рому.

  Теперь рассмотрим случай, когда $a_1 \cdot t_1 > b_1 \cdot t_1$. В этом случае на большой дистанции Вова в какой-то момент обгонит Рому и никогда не даст ему себя догнать, но до этого момента они могут несколько раз оказаться в одной точке.

  Обозначим за $d = d_2 - d_1$. Другими словами, за каждый период длительностью $t_1 + t_2$ минуты Вова обгоняет Рому на d метров. Обозначим за $p = a_1 \cdot t_1 - b_1 \cdot t_1$. Этим мы обозначаем максимальное количество метров, на которое Рома будет опережать Вову в момент времени t_1.

  Далее хочется сказать следующее:

  - Если p не делится на d, то ответ равен $\lfloor \frac{p}{d} \rfloor \cdot 2 + 1$. Это связано с тем, что за один забег в течение $t_1 + t_2$ минут Рома и Вова пересекутся два раза, и исключением станет лишь последний такой отрезок в течение $t_1 + t_2$ минут.
  - Если p делится на d, то ответ равен $\frac{p}{d} \cdot 2$.

Задача F. Кампусы МФТИ

Будем решать задачу постепенно:

- Для начала давайте рассмотрим граф, в котором оставим только такие рёбра (u, v), что $s_u = s_v$. Выделим в этом графе компоненты связности. Пусть p_v — номер компоненты связности вершины v.
- Как тогда мы сможем обрабатывать запросы? Если $s_a = s_b$, то нам достаточно будет проверить, что вершины a и b лежат в одной компоненте связности, то есть $p_a = p_b$. Далее считаем, что $s_a \neq s_b$.

  В каком случае из вершины a есть путь в вершину b? Давайте теперь будем рассматривать граф между найденными компонентами. Соответственно, если в изначальном графе было ребро (u, v), то если $s_u \neq s_v$, то в новом графе будет ребро между вершинами p_u и p_v. Тогда из вершины a можно добраться до вершины b, если в новом графе есть путь из вершины p_a в вершину p_b, такой что все вершины на пути соответствуют значениям s_a и s_b.

  Так как мы уже выделили все компоненты смежных вершин с одинаковым значением s, то на самом деле на пути между p_a и p_b значения s_a и s_b должны чередоваться. Давайте тогда сделаем следующее:

  - Если в нашем новом графе есть ребро между p_u и p_v, то для пары чисел (s_u, s_v) запомним, что есть ребро в графе между соответствующими вершинами. Здесь нам не важен порядок чисел в паре, поэтому с точки зрения реализации удобнее рассматривать пару $(\min(s_u, s_v), \max(s_u, s_v))$.
  - Тогда между вершинами a и b будет путь, если мы рассмотрим все рёбра для пары $(\min(s_a, s_b), \max(s_a, s_b))$ и окажется, что вершины p_a и p_b связны.

- Чтобы решение работало быстро, предлагается следующее:

  - Для начала для всех пар (s_u, s_v) сложим в словарь (map) все рёбра (p_u, p_v).
  - Далее считаем все запросы и также сложим их все в словарь, распределив по парам (s_a, s_b).
  - Далее будем перебирать какую-либо пару (s_u, s_v) и выделять компоненты связности по рёбрам этой пары. Это можно сделать с помощью поиска в глубину или СНМ, но здесь важно учитывать, что нужно будет рассматривать только те вершины, которым инцидентно хотя бы одно ребро.
  - Выделив компоненты связности для некоторой пары, нужно будет найти ответы на все запросы с такой парой (s_u, s_v).

Решение можно реализовать за $\mathcal{O}((n + m + q) \log n)$.

Задача G. Стена из полимино

Каждое полимино можно рассматривать как вершину графа. Если полимино A имеет хотя бы одну клетку, под которой находится клетка другого полимино B, то при построении стены B должно быть добавлено раньше A. Иными словами, существует направленное ребро $A \to B$.

Таким образом, задача сводится к проверке, существует ли **топологический порядок** вершин (полимино) в этом графе. Если он существует — стена устойчива, и этот порядок можно вывести. Если нет (в графе есть цикл) — устойчивость невозможна.

Построение графа.
Для каждой клетки (i, j) с буквой x:

- если под ней находится клетка $(i + 1, j)$ с буквой $y \neq x$, добавляем ребро $x \to y$ (чтобы y появилось раньше x);

После обработки всех клеток получаем ориентированный граф, в котором вершины — это различные буквы, а рёбра отражают зависимости между полимино.

Решение.
Нужно проверить, можно ли выполнить **топологическую сортировку** построенного графа. Если можно — выводим любой порядок букв. Если нельзя (обнаружен цикл) — выводим `-1`.

Задача H. Торт

Для начала изучим необычное условие, а именно то, что мы за один раз откусываем не окружность, а окружность и то, что выше неё. Это на самом деле нам гарантирует, что для каждой x-координаты мы в каждый момент времени ещё не откусили какую-то часть торта $[0, y_x]$ для некоторого y. Наша задача научиться поддерживать эти y_x для всех $0 \leqslant x \leqslant w$. Изначально все $y_x = h$.

Далее мы сможем решить задачу, используя дерево Ли Чао:

- Когда мы откусываем окружность с центром в точке (x, y), мы накладываем ограничение на все x-координаты из диапазона $[x - r, x + r]$. Более того, в дальнейшем будет полезно рассматривать не окружности, а только нижние их половины.

  В дереве Ли Чао предлагается для подотрезков $[l, r]$ хранить все окружности, которые накладывают ограничения на этот отрезок, а среди всех таких — хранить наиболее оптимальную. Изначально можно считать, что в дереве Ли Чао ничего нет.

- Как тогда изменяется дерево Ли Чао при добавлении новой окружности? Для начала мы доходим до всех вершин нашего дерева, которые целиком лежат внутри диапазона $[l, r]$. Далее из каждой из них мы начинаем спускаться вниз и учитывать нашу новую окружность. Нам нужно будет рассмотреть следующие случаи (научиться сравнивать нашу текущую найденную окружность в вершине дерева Ли Чао и нашу новую; если в вершине ещё нет окружности, то мы сможем просто положить туда нашу окружность и выйти из рассмотрения этого случая):

  - Окружности могут не пересекаться на заданном подотрезке x-координат. Тогда одна из них лучше, и нужно оставить будет лучшую из двух окружностей в вершине и закончить алгоритм. Для этого достаточно найти точки пересечения окружностей, проверить, что они не попадают в заданный диапазон, и если это так, то просто посмотреть, какая окружность ниже в точке $x = l$. В этом случае нам дальше не нужно будет спускаться в дереве Ли Чао.
  - Окружности пересекаются. Тогда мы будем делать всё как в обычном дереве Ли Чао и определим, в какого сына нам следует идти дальше.

    Другими словами, в обычном дереве Ли Чао мы бы поддерживали выпуклую оболочку прямых, а здесь нам нужно поддерживать выпуклую оболочку полуокружностей одинакового радиуса. Это разные объекты, но поддерживать дерево Ли Чао мы будем аналогично. Отличие в том, что здесь мы учитываем тот факт, что окружности действуют не на все x-координаты, а только на некоторый подотрезок, и пересекать окружности немного сложнее, чем пересекать прямые.

    Но можно заметить, что полуокружности необязательно нужно пересекать. На самом деле достаточно будет посмотреть на их значение y-координаты в x-координатах l, r и $m = \frac{l+r}{2}$ для некоторой вершины дерева Ли Чао. Этой информации будет достаточно, чтобы определить случай, в котором вы находитесь.

- Как и в стандартном дереве Ли Чао, чтобы найти ответ для x-координаты, нам достаточно рассмотреть все окружности на пути от корня до листа, соответствующего данной x-координате.

При реализации нужно не забыть аккуратно учесть одинаковые окружности. Решение можно реализовать за $\mathcal{O}(q \log^2 w)$.

Задача I. Нечетный подграф

Для начала заметим, что для каждой компоненты связности мы решаем задачу отдельно. Далее нам нужно рассмотреть ряд случаев:

- Если размер компоненты связности нечётный, то ответа не существует. Это связано с тем, что если мы для каждой вершины выберем нечётное количество инцидентных её рёбер, то суммарная степень всех вершин тоже будет нечётной (мы нечётное количество раз сложим нечётные числа). А суммарная степень всех вершин должна быть чётной, так как каждое ребро мы учитываем ровно 2 раза.

- С другой стороны, если размер компоненты связности чётный, то ответ существует и его можно найти жадно. Для этого достаточно запустить обход в глубину из некоторой вершины компоненты и рассмотреть дерево обхода. Из дерева обхода возьмём в ответ следующие рёбра:

  - Для всех вершин v, что её поддерево нечётного размера, возьмём в ответ ребро между вершиной v и её предком в дереве обхода. Так, например, мы возьмём в ответ все рёбра, которые ведут в листовые вершины.

Решение можно реализовать за $\mathcal{O}(n)$.

Задача J. Игра на буквенных строках

Обозначим за $T = \max(|t_1|, |t_2|, \ldots, |t_n|)$. Другими словами, T — максимальная длина среди всех строк t_i. В нашей задаче $T \leqslant 50$.

Заметим, что так как в каждый момент времени текущее состояние игры должно быть подстрокой некоторой строки t_i, то у нас есть не более $\mathcal{O}(n \cdot T^2)$ различных состояний игры. Так как n тоже достаточно маленькое, то далее мы можем решать задачу наивно:

- Пусть s — текущее состояние игры. Изначально строка s пустая. Реализуем функцию $win(s)$, которая будет возвращать единицу только в том случае, если первый игрок, начав игру с начальным состоянием s, выиграет.

  Заметим, что $win(s)$ равно 1, если существует символ c из нашего алфавита (а в нашей задаче мы рассматриваем символы ASCII с кодами от 33 до 126), что состояние $c + s$ или состояние $s + c$ являются проигрышными. Другими словами, либо $win(c + s) \neq 1$, либо $win(s + c) \neq 1$.

  Мы должны будем рассмотреть несколько крайних случаев, чтобы наше решение работало правильно:

  - Если строка s есть в наборе t, то $win(s) = 1$. Это связано с тем, что в такое состояние переходить нельзя и игрок не должен уметь выигрывать, переходя в такое состояние.
  - Если строка s не является подстрокой какой-либо строки t_i, то $win(s) = 1$. Аналогичные рассуждения предыдущему пункту.

- Чтобы наше решение работало быстро, будем делать следующее:

  - Вместо строки s будем передавать в функцию её хеш. Дополнительно нам также нужно будет знать длину строки, поэтму состояние мы можем хранить как пару (хеш, длина).
  - Чтобы не считать дважды значение функции win для одной и той же строки, будем запоминать ответы (делаем ленивую динамику).

- Соответственно, выигрывает первый игрок, если win от пустой строки возвращает единицу. Чтобы определить, с каких символов первый игрок может начать игру, можно их перебрать и проверить, что $win(c)$ возвращает не единицу (то есть второй игрок окажется в проигрышном состоянии).

Обозначим за $C = 126 - 33 + 1 = 94$ — размер алфавита. Тогда у нас есть $\mathcal{O}(n \cdot T^2)$ состояний и $2 \cdot C$ переходов из каждого из них. Если использовать хеш-таблицу, то решение можно реализовать за $\mathcal{O}(n \cdot T^2 \cdot C)$.

В этой задаче есть много разных альтернативных решений — некоторые работают быстрее и используют другие структуры данных.

Задача K. Равномерное распределение задач

Обозначим за s — сумму чисел от 1 до n. Другими словами, $s = \frac{n \cdot (n+1)}{2}$. Теперь обсудим случаи, когда ответа не существует:

- Если s не делится на k, то ответа не существует.
- Обозначим как $p = \frac{s}{k}$. Если $p < n$, то ответа не существует, так как нам некуда будет определить задачу со сложностью n.

Иначе ответ существует. Далее есть два варианта реализации — можно найти ответ конструктивно, сводя задачу к меньшим значениям n и k, а можно решить задачу жадно. Рассмотрим жадное решение:

- Сложим все числа от 1 до n в мультимножество. Будем набирать каждый контест жадно и независимо.

- Пусть мы хотим набрать задачи для очередного контеста. Тогда давайте возьмём в этот контест задачу максимальной сложности, которая не больше p. Обозначим сложность этой задачи как x. Далее, если $x \neq p$, возьмём следующую задачу в контест, а именно задачу максимальной сложности не более $p - x$. И так далее.

  Все использованные задачи мы будем удалять из мультимножества. Найти задачу максимальной сложности, не больше чем заданное число, можно с помощью встроенного метода upper_bound, дополнительно вычитая единицу из найденного итератора.

Решение можно реализовать за $\mathcal{O}(n \log n)$.

Доказательство корректности жадного алгоритма

Инвариант. Перед формированием любой очередной группы пусть остаётся m групп (включая текущую). Обозначим множества ещё неразданных чисел через R. Тогда

$$\sum_{y \in R} y = mp,$$

и наибольший элемент множества R не превосходит p. (Последнее верно в начале, так как $n \leqslant p$; ниже мы покажем, что сохраняется.)

Достаточно доказать, что при выполнении инварианта можно выделить подмножество $S_x \subseteq R$ для текущего максимума $x = \max R$ с суммой

$$\sum_{y \in S_x} y = p \qquad \text{и} \qquad x \in S_x.$$

Тогда после удаления S_x из R инвариант будет выполнен для $m - 1$ групп, и процесс можно повторить.

Лемма о представимости сумм префиксом. Пусть $1 \leqslant r \leqslant n$. Множество $\{1, 2, \ldots, r\}$ может представить (в виде суммы некоторых своих элементов) любое целое число v в диапазоне $0 \leqslant v \leqslant \frac{r(r+1)}{2}$. Более точно: для любого v из этого диапазона существует подмножество $\{1, \ldots, r\}$, сумма элементов которого равна v.

Доказательство леммы. Достаточно наблюдения, что числа $1, 2, \ldots, r$ образуют плотный набор, позволяющий жадно представлять любое $v \leqslant \sum_{i=1}^{r} i$: выбирая на каждом шаге наибольшее число не превосходящее оставшийся остаток, мы получим разложение (аналогично ряду представлений в смешанной системе счисления). Формальная индукция по r легко завершает доказательство.

Доказательство существования дополнения. Пусть перед формированием текущей группы множество оставшихся чисел R содержит в себе некоторый префикс $\{1, 2, \ldots, r\}$ (это верно, поскольку из первоначального набора $1, \ldots, n$ на предыдущих шагах удалялись только большие элементы). Рассмотрим $x = \max R$ и необходимую для дополнения величину

$$D = p - x \geqslant 0.$$

Если $\frac{r(r+1)}{2} \geqslant D$, то по лемме набор $\{1, \ldots, r\} \subseteq R$ уже позволяет набрать сумму D, и, взяв эти элементы вместе с x, мы получим группу суммы p. Если $\frac{r(r+1)}{2} < D$, то суммарная сумма всех элементов, меньших либо равных r, недостаточна, однако суммарный ресурс множества $R \setminus \{x\}$ равен $mp - x$ и, поскольку $m \geqslant 1$, он не меньше D. В этом случае можно дополнить x некоторыми более крупными оставшимися элементами вместе с всем префиксом $\{1, \ldots, r\}$ так, чтобы в сумме получить D (т.е. взять префикс и доподобрать несколько больших элементов); опять же, за счёт плотности префикса и наличия достаточной общей суммы такое точное дополнение существует. Следовательно, существует подмножество $S_x \subseteq R$ с суммой p, содержащие x.

Сохранение инварианта и окончание. После удаления из R подмножества S_x сумма оставшихся элементов становится $(m-1)p$, а максимальный оставшийся элемент не превышает p (так как мы удалили текущий максимум x и другие элементы $\leqslant x$). Таким образом, инвариант сохраняется, и тот же аргумент применим к следующему шагу. Процесс повторяется конечное число раз (каждый шаг уменьшает количество неразданных групп на 1), в конце используются все числа и получаются k групп с суммой p.

Вывод. Итак, при выполнении необходимых условий (S кратно k и $p \geqslant n$) жадный алгоритм корректно находит разбиение чисел $1, 2, \ldots, n$ на k непересекающихся подмножеств с одинаковой суммой p. В противном случае разбиение не существует.

Задача L. Джекпот

Эту задачу можно решать наивно, рассмотрев один случай отдельно:

- Будем моделировать процесс. Каждый раз будем считать среднее арифметическое всех m_i и выдавать нужное количество рублей самому бедному другу.
- Если моделировать этот процесс полностью, то решение будет работать долго. Давайте обработаем крайний случай отдельно. Заметим, что если $m_1 = m_2 = \ldots m_n$, то в дальнейшем мы будем раздавать всем друзьям по одному рублю.

  Поэтому когда при моделировании мы оказываемся в ситуации, что $m_1 = m_2 = \ldots m_n$, мы перестаём моделировать процесс и добавляем ко всем m_i число $\lfloor \frac{j}{n} \rfloor$, где j — количество рублей, которое мы ещё не раздали. И дополнительно нам ещё нужно будет раздать по рублю $j \bmod n$ друзьям.
- Почему это решение работает быстро? Для этого задумаемся о том, когда мы окажемся в ситуации $m_1 = m_2 = \ldots m_n$. Пусть x — максимальное значение в изначальном массиве m. Тогда мы окажемся в ситуации $m_1 = m_2 = \ldots m_n$, когда все эти значения будут равны x.

  Обозначим за s — сумму элементов в массиве m. Давайте тогда заметим, что за одну операцию моделирования сумма в массиве m увеличивается хотя бы на $\lceil \frac{(x \cdot n - s)}{n} \rceil = \lceil x - \frac{s}{n} \rceil$.

  Из всего этого можно сделать вывод, что нам достаточно будет сделать не более $\mathcal{O}(\log_{\frac{n}{n-1}} x)$ операций моделирования процесса.

Задача M. Шкафы

Для начала отсортируем все шкафы по возрастанию их позиции. Далее считаем, что $p_1 < p_2 < \ldots < p_n$. Теперь наша задача для каждого отрезка позиции $[p_i, p_{i+1}]$ определить, сможем ли мы выбраться из него или нет.

Давайте задумаемся о том, когда мы в процессе нашего движения можем не смочь выбраться? Для этого должны существовать два шкафа $1 \leqslant l < r \leqslant n$, что $p_r - p_l \leqslant \min(s_l, s_r)$. Для всех таких пар шкафов мы можем сказать, что из всех позиций от p_l до p_r мы не сможем выбраться.

Но мы не можем рассмотреть все пары шкафов, поэтому будем рассматривать только особые пары шкафов. Для этого достаточно будет рассмотреть только $\mathcal{O}(n)$ таких пар.

- Для каждого шкафа i найдём ближаший слева шкаф, размер которого не меньше. Обозначим номер найденного шкафа как lef_i.
- Для каждого шкафа i найдём ближаший справа шкаф, размер которого не меньше. Обозначим номер найденного шкафа как $righ_i$.
- Утверждение: мы можем рассмотреть только пары шкафов (lef_i, i) и $(i, righ_i)$.

Если пара шкафов l и r говорит нам о том, что мы не сможем выбраться из этого подотрезка, то мы можем добавить единицу на отрезке $[l, r-1]$, сказав, что для всех $i \in [l, r-1]$ мы не сможем выбраться из отрезка позиций $[p_i, p_{i+1}]$.

Все добавления можно обработать в оффлайне с помощью массива отложенных добавлений: add_i — величина, которую нужно добавить всем элементам на суффиксе, начиная с i-го элемента. Таким образом, добавление единицы на отрезке $[l, r-1]$ изменяет массив add следующим образом: add_l увеличивается на единицу, а add_r уменьшается на единицу. Значение i-го элемента определяется как сумма на префиксе, то есть $add_1 + add_2 + \ldots + add_i$.

Решение можно реализовать за $\mathcal{O}(n \log n)$.

Задача N. Фрукты

На первый взгляд задача кажется подходящей для жадного решения: сажать всё, что можно, чтобы получить экспоненциальный рост. Однако иногда выгоднее продать немного фруктов, чтобы купить экзотический фрукт, который даёт больше прибыли. Жадность в таких случаях не работает.

Поэтому используется динамическое программирование. Однако прямое состояние вида «сколько рублей, фруктов, деревьев и экзотических деревьев» быстро становится необозримым: количество деревьев и фруктов может расти экспоненциально. Ключевое наблюдение: когда какой-либо параметр становится достаточно большим, его точное значение перестаёт быть важным — достаточно знать, что его «много».

Рассмотрим, сколько фруктов реально нужно хранить. Чтобы купить экзотический фрукт, нужно 400 рублей, то есть 4 обычных фрукта. За 3 дня можно купить максимум 3 экзотических фрукта, то есть потратить 1200 рублей $\rightarrow$ 12 фруктов. Следовательно, хранить больше 12 фруктов нет смысла: всё, что сверх этого, можно сразу продать — оно не повлияет на возможность покупки экзотики в ближайшие дни.

Аналогично, обычных деревьев достаточно хранить до 4 штук на каждый «слот» (урожай сегодня/завтра/послезавтра), так как больше не нужно для покрытия будущих потребностей. Экзотических деревьев достаточно хранить по 1 на слот — они дают много фруктов, и больше не требуется. Эти ограничения позволяют свести пространство состояний к разумному размеру (менее 10^6 состояний).

Состояние и переходы

Состояние DP описывается следующими параметрами:

- d — оставшиеся дни,
- b — количество «сотен» рублей (остаток рублей хранится отдельно),
- f — количество обычных фруктов (ограничено),
- `trees`[3] — количество обычных деревьев, дающих урожай через 0, 1, 2 дня,
- `exotic`[3] — аналогично для экзотических деревьев.

На каждом шаге сначала собирается урожай: `trees`[0] дают $3 \cdot$ `trees`[0] фруктов, аналогично для экзотики. Затем перебирается, сколько фруктов и экзотических фруктов продать (`sell1`, `sell2`), и вычисляются полученные деньги. После этого рассматриваются три варианта действий:

1. Ничего не покупать.
2. Купить экзотический фрукт и сразу продать.
3. Купить экзотический фрукт и посадить.

После этого массивы деревьев сдвигаются: `trees`$[i] \leftarrow$ `trees`$[i+1]$, и добавляются новые посаженные деревья в `trees`[2].

Оптимизация: вынос «лишнего» в прибыль

Если в состоянии фруктов или деревьев больше порогового значения, то «лишнее» сразу конвертируется в деньги с учётом их будущей ценности. Для этого используется можно использовать функцию, которая оценивает, сколько денег можно получить с одного фрукта за d дней (учитывая экспоненциальный рост при посадке). Это позволяет не хранить большие числа в состоянии, а сразу учитывать их вклад в итоговую прибыль.

Не забываем хешировать и кешировать состояния. Количество дней $d \leqslant 40$. Благодаря ограничениям на параметры ($b \leqslant 12$, $f \leqslant 12$, $\texttt{trees}[i] \leqslant 4$, $\texttt{exotic}[i] \leqslant 1$), общее число состояний не превышает 10^6. Для каждого состояния перебирается небольшое число вариантов продажи и покупки, что делает решение эффективным.