# Задача A. Условие задачи

Каждое нажатие на клавишу Enter увеличивает количество частей на 1.

В начале была 1 часть, в конце — $n$ частей, значит было сделано $(n-1)$ нажатий.

# Задача B. Муравейник

Рассмотрим префикс-суммы массива $a$:

- $p_0 = 0$;
- $p_1 = a_1$;
- $p_2 = a_1 + a_2$;
- ...
- $p_n = a_1 + a_2 + \cdots + a_n$.

Очевидно, что $p_i$ означает изменение количества муравьёв внутри муравейника после первых $i$ наблюдений (поэтому $p_0 = 0$).

В муравейнике не могло оказаться отрицательное число муравьёв ни в какой момент времени, что можно записать как:

$ans + p_i \geqslant 0$ для всех $i = 0 \dots n$.

Легко заметить, что на самом деле всё зависит только от минимального значения $p_i$:

$ans + \min(p) \geqslant 0$, откуда следует $ans \geqslant -\min(p)$.

Нам нужно выбрать минимальный ответ, поэтому неравенство превращается в равенство.

Заметим, что для решения задачи даже не нужно было создавать массив $p$ — было достаточно одной переменной «суммарное изменение после текущего наблюдения»:

```
delta = 0;
min_delta = delta;

for (value in a) {
    delta += value;
    min_delta = min(min_delta, delta)
}

ans = -min_delta
```

Также важно было заметить, что максимальный возможный ответ был порядка $10^5 \cdot 10^6 = 10^{11}$, что не помещалось в 32-битный целочисленный тип данных, поэтому было необходимо использовать 64-битный тип (`long long` или `int64_t` в C++, `long` в Java).

# Задача C. Линейный лабиринт

Будем последовательно вычислять ответ для префикса комнат $1 \dots i$.

- Вход в любую комнату увеличивает текущий ответ на 1.
- Если комната является развилкой, то увеличим текущий ответ в 2 раза.

Почему такой способ вычисления работает?

Когда мы попадаем в первый раз в комнату-развилку (обозначим её за $x$), то нам приходится начинать весь путь до неё сначала.

Допустим, что мы прошли второй раз путь от 1 до $x$ (причём он мог быть каким-то отличным от первого раза).

Самое важное, что после выхода из комнаты $x$ во второй раз, её состояние вернётся в изначальное положение.

Соответственно, если в дальнейшем мы попадём в комнату $x$, пытаясь второй раз пройти до комнаты-развилки $y > x$, то мы будем повторять все те же самые шаги, что и ранее.

Откуда следует, что для каждой комнаты-развилки длина пути просто увеличивается в 2 раза.

# Задача D. Группировка

В данной задаче работает «жадный» подход.

Для начала заметим, что изначальный порядок элементов в массиве ни на что не влияет, поэтому отсортируем его по возрастанию.

Утверждается, что существует оптимальный ответ, состоящий только из подотрезков отсортированного массива $a$.

Докажем это конструктивно.

Пусть существует оптимальный ответ такой, что $a_i$ и $a_{i+2}$ принадлежат первому набору, а $a_{i-1}$ и $a_{i+1}$ — второму набору.

Рассмотрим распределение, которое отличается от оптимального только в одном: $a_{i+1}$ и $a_{i+2}$ теперь в первом наборе, а $a_{i-1}$ и $a_i$ — во втором.

Таким образом:

- Количество наборов не изменилось.
- Размер наборов не изменился.
- Разница элементов в первом наборе была не менее $a_{i+2} - a_i$, а стала не менее $a_{i+2} - a_{i+1}$.
- Разница элементов в первом наборе была не менее $a_{i+1} - a_{i-1}$, а стала не менее $a_i - a_{i-1}$.

Так как $a_{i+1} \geqslant a_i$, то обе разницы могли только уменьшиться (либо не измениться), а значит новое распределение тоже корректное.

Таким образом, из любого оптимального распределения можно получить оптимальное, где наборы образуют подотрезки массива.

Для решения задачи было достаточно обрабатывать элементы массива в порядке возрастания, поддерживая позицию и величину первого элемента в текущем отрезке:

```
ans_cnt = 0

first = neg_inf
first_pos = -1

for (i = 0; i < n; i += 1) {
  if (a[i] - first > M or i - first_pos >= k) {
    ans_cnt += 1

    first = a[i]
    first_pos = i
  }
}
```

# Задача E. Горные цепи

Очевидно, что если подходящий $k$ существует, то он лежит в промежутке $[1; \max(h) - \min(h)]$ — обозначим длину промежутка через $H$.

**Первый** подход к решению

Построим функцию $F(k)$ — количество получающихся горных цепей для фиксированного параметра схожести $k$.

Сама функция реализуется за $O(n)$ достаточно очевидным способом:

- Поддерживаем счетчик горных цепей, изначально он равен 1.

- Обрабатываем горы слева направо, начиная со второй горы.
- Если разница между горами выше параметра схожести, то увеличиваем счетчик на 1.

Но решение за $O(H \cdot n)$ слишком неэффективное.

Для дальнейшей оптимизации заметим, что при увеличении параметра $k$ значение функции $F(k)$ может либо остаться таким же, либо уменьшиться: $F(1) \geqslant \ldots \geqslant F(H-1) \geqslant F(H)$.

Соответственно, если $m < F(H)$ или $F(1) < m$ — ответа в принципе не существует.

Иначе введём дополнительную функцию $G(k) = (F(k) \leqslant m)$.

- Если $G(k)$ истинно, то и $G(k+1)$ истинно.
- Если $G(k)$ ложно, то и $G(k-1)$ ложно.
- $G(0)$ примем за ложь, $G(H+1)$ за истину.

Явно видно, что функция $G(k)$ монотонно убывает, поэтому на ней можно выполнить **двоичный (бинарный)** поиск по ответу.

Заметим, что после выполнения двоичного поиска по ответу надо проверить найденное значение $k_{opt}$ — может быть ситуация, когда $F(k_{opt}) < m < F(k_{opt}+1)$. В таком случае ответа также не существует.

Итоговая асимптотика будет равна $O(\log H \cdot n)$.

**Второй** подход к решению

Вычислим массив $D_i = |H_{i+1} - H_i|$ и отсортируем его по неубыванию.

Также дополним массив $D_0 = -1$ и $D_n = H + 1$.

Пусть для параметра схожести $k$ выполняется неравенство $D_i \leqslant k < D_{i+1}$.

В таком случае можно уверенно утверждать, что для заданного параметра $k$ будет получена ровно $n - i$ горная цепь.

Почему так? Давайте назовём условие «соседние горы лежат в одной цепи» **связью**. В таком случае новая горная цепь образуется каждый раз, когда связь разрушается.

- При $k = (H+1)$ все горы в одной цепи — то есть в наличии $(n-1)$ связь.
- Если $k < D_{i+1}$, то все связи с $i+1$ до $(n-1)$ разрушены.
- Всего получается разрушено $n-(i+1)$ связей, а значит количество горных цепей стало равным $1 + n - (i+1) = n - i$.

Нам необходимо получить ровно $m$ горных цепей, а значит $k$ должно быть в полуинтервале $[D_{n-m}; D_{n-m+1})$.

Если данный полуинтервал пустой — то ответа не существует. Иначе необходимо взять минимальное значение — то есть $D_{n-m}$.

Итоговая асимптотика получается равной $O(n \cdot \log n)$ из-за сортировки.

# Задача F. Светофоры

Данная задача решалась аккуратной эмуляцией описанного процесса.

Будем поддерживать текущее время с начала поездки в переменной $time$.

Когда мы подъезжаем к $i$-му перекрестку — увеличим $time$ на $a_i$.

Теперь осталось понять, какой цвет в данный момент горит на $i$-м светофоре.

Мы знаем, что за $d_i$ секунд до начала поездки загорался зеленый цвет. С того момента прошло суммарно $total_i = (time + d_i)$ секунд.

Процесс смены цветов цикличен с периодом $period_i = (g_i + r_i)$.

Вычислим остаток от деления $rem_i = total_i \mod period_i$. Это будет равно времени, прошедшему со старта последнего периода горения.

Если $rem_i < g_i$, то автобус проедет перекрёсток сразу; иначе нам необходимо прибавить ко времени величину $period_i - rem_i$ — ровно столько времени нам необходимо ждать на красном светофоре.

# Задача G. Нужно больше золота

Для начала покажем, что основная идея решения данной задачи — жадность.

Пусть мы нашли какой-то оптимальный ответ, где мы подряд победили сначала монстра $j$, а сразу после — монстра $i$, причём $g_j < g_i$.

Покажем, в каком случае мы можем поменять местами этих монстров (и только этих), чтобы ответ остался оптимальным.

Обозначим количество золота до сражения с монстром $j$ через $cw$.

В таком случае мы сразу можем зафиксировать два утверждения:

- $b_j \leqslant cw$;
- $b_i \leqslant cw + g_j$.

Во-первых, заметим, что для монстров, сражённых после этой пары, ничего не изменится — суммарное количество золота после обеих побед будет $cw + g_i + g_j = c_w + g_j + g_i$.

Во-вторых, рассмотрим два случая:

- $cw < b_i$ — в таком случае на момент сражения с монстром $j$ мы никак не могли победить монстра $i$.
- $b_i \leqslant cw$ — в таком случае мы могли поменять местами сражения с $j$-м и $i$-м монстрами:
  - $i$-го монстра мы можем победить по нашему предположению;
  - $j$-го монстра мы можем победить, так как $b_j \leqslant cw \leqslant cw + g_i$ ($g_i \geqslant 0$ по условию).

Таким образом мы показали следующее: если в данный момент мы **можем победить** монстров $m_1, m_2, \ldots, m_c$, то для получения оптимального ответа всегда можно выбирать монстра с наибольшим значением $g_{m_i}$.

В данный момент мы уже можем реализовать решение, в котором будем каждый раз выбирать оптимального монстра за $O(n)$, откуда общая асимптотика будет равна $O(n^2)$ — корректно, но недостаточно эффективно.

Для дальнейшей оптимизации нам потребуется структура данных (СД), которая умеет выполнять эффективно следующие операции:

- добавить значение;
- вернуть экстремум (минимум или максимум);
- извлечь соответствующий экстремум.

Известно, что структуры данных «бинарная куча» и «дерево поиска» способны выполнять все эти операции за $O(\log N)$ или эффективнее:

- C++: `std::priority_queue` и `std::set`;
- Java: `PriorityQueue` и `TreeSet`;
- Python: пакет `heapq`.

Для начала добавим всех монстров в СД *alive*, которая будет искать минимум по значению $b_i$. В данной структуре будут храниться монстры, которые ещё живы, но мы пока не можем их победить.

Также будем поддерживать СД *ready*, которая будет искать максимум по значению $g_i$. В данной структуре будут храниться монстры, которые ещё живы, но мы теперь способны их победить.

Каждый шаг будет состоять из нескольких частей:

- Если $cw \geqslant w$, то обработка закончена.

- Извлекаем из *alive* монстров с минимальным значением $b_i$, пока $b_i \leqslant cw$.
- Всех извлечённых таким образом монстров добавляем в *ready*.
- Если *ready* пуст, то обработка закончена.
- Иначе извлекаем из *ready* монстра с максимальным значением $g_i$, после чего увеличиваем $cw$ на $g_i$ и счётчик побеждённых монстров на 1.

# Задача H. Иерархия

Структура каждой компании образует собой **дерево** (а все компании вместе образуют **лес**).

В таком случае генеральный директор является корнем соответствующего дерева, а количество сотрудников в его компании — количеством вершин в данном дереве.

Искать количество вершин в каждом дереве заданного леса можно двумя способами:

- полный обход каждого дерева с помощью обхода в глубину / ширину;
- использование системы непересекающихся множеств (disjoint set union).

Заметьте, что при решении через обходы рекомендуется использовать **списки смежности** для хранения структуры графа.

Итоговая асимптотика решения будет равна $O(n)$ для решения через обходы и $O(n \cdot D)$ для решения через СНМ, где $D$ — сложность операций СНМ (зависит от выбранной реализации).

# Задача I. Учебный день

Задача может быть решена методом динамического программирования.

Пусть $dpO_i$ — максимальное суммарное знание, которое вы получите, прослушав лекции до $i$ включительно, если вы не медитировали перед $i$-й лекцией.

Аналогично $dpM_i$ — максимальное суммарное знание, которое вы получите, прослушав лекции до $i$ включительно, если вы медитировали перед $i$-й лекцией.

Переходы будут следующими:

- $dpO_0 = dpM_0 = 0$ — до прослушивания лекций знаний нет;
- $dpO_i = \max(dpO_{i-1}, dpM_{i-1}) + a_i$ — слушать лекцию без медитации можно без ограничений;
- $dpM_i = dpO_{i-1} + 2 \cdot a_i$ — нельзя слушать две лекции подряд с медитацией.

Ответ будет равен $\max(dpO_n, dpM_n)$.

Чтобы восстановить оптимальный порядок действий, можно перебрать лекции в обратном порядке, начиная с оптимального состояния (O или M):

- Если мы находимся в M-состоянии, то предыдущее действие будет обязательно O;
- Если мы находимся в O-состоянии, то предыдущее состояние O, если $dpO_{i-1} > dpM_{i-1}$, иначе — M.

В итоге мы получим последовательность действий в обратном порядке.