Турнир Архимеда 2017. Разбор задач.

Представляем вашему вниманию разбор задач Турнира Архимеда по программированию 2017.

Турнир подготовлен под руководством Андрея Станкевича командой школьников Санкт-Петербурга, призёров и победителей Всероссийской олимпиады и международных соревнований по информатике. Турнир готовили: Алексей Трилис, Арсений Кириллов, Влад Епифанов, Даниил Орешников, Егор Ждан, Макар Селиванов, Михаил Анопренко, Михаил Иванов.

Координатор проведения на системе Яндекс.Контест: Лидия Перовская. Задачи подготовлены с помощью системы Polygon, автор Михаил Мирзаянов.

Ниже приведены разборы всех задач турнира. Везде мы старались привести как можно более простое и элегантное решение. Если программа не проходит тесты жюри и приведена как неправильное решение, она обведена в красную рамку.

Разбор задачи «Режем торт»

Автор задачи:	Михаил Анопренко
Подготовка тестов и решений:	Михаил Анопренко
Автор разбора:	Михаил Анопренко

Для начала заметим, что после каждого разреза Крошу выгодно оставлять себе из двух получившихся прямоугольников тот, который имеет большую площадь.

Также заметим, что отрезать от торта всегда выгодно кусок размером $1 \times a$, где a - это одна из сторон прямоугольника, то есть полоску ширины 1. В противном случае можно уменьшить одну из сторон отрезанного куска, увеличив тем самым площадь оставшейся части.

Решение 1. Заметим, что есть всего три способа отрезать два раза полоску ширины 1:

- Сделать оба разреза вдоль стороны длины n. Это можно сделать, только если $m \geqslant 3$. В этом случае мы два раза отрежем прямоугольник $1 \times n$. Суммарная отрезанная площадь будет равна $2n$, а оставшаяся площадь будет равна $nm - 2n$.
- Сделать оба разреза вдоль стороны длины m. Это можно сделать, только если $m \geqslant 3$. В этом случае мы два раза отрежем прямоугольник $1 \times m$. Суммарная отрезанная площадь будет равна $2m$, а оставшаяся площадь будет равна $nm - 2m$.
- Сделать по одному разрезу вдоль каждой из сторон. Это можно сделать, только если $n \geqslant 2$ и $m \geqslant 2$. В этом случае мы либо сначала отрежем прямоугольник $1 \times n$, а потом прямоугольник $1 \times (m-1)$, либо сначала прямоугольник $1 \times m$, а потом прямоугольник $1 \times (n-1)$. В любом случае, суммарная отрезанная площадь будет равна $n + m - 1$, а оставшаяся площадь будет равна $nm - n - m + 1$.

Таким образом, надо просто выбрать из этих трех вариантов тот, который даст максимально возможную оставшуюся площадь.

Пример реализации этого решения на языке Python:

```
n = int(input())
m = int(input())
ans = 0
if m >= 3:
    ans = max(ans, n * m - 2 * n)
if n >= 3:
    ans = max(ans, n * m - 2 * m)
if n >= 2 and m >= 2:
    ans = max(ans, n * m - n - m + 1)
print(ans)
```

Решение 2. Несложно доказать, что выгодно каждый раз отрезать минимальный по площади прямоугольник, то есть прямоугольник $1 \times n$, если $n \leqslant m$, и $1 \times m$ в противном случае. Таким образом, получается еще более простое решение.

Пример реализации этого решения на языке C++:

```
#include <iostream>

using namespace std;

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 0; i < 2; i++) {
        if (n <= m) {
            m--;
        } else {
            n--;
        }
    }

    cout << n * m << endl;
    return 0;
}
```

Разбор задачи «Подборка новостей»

Автор задачи:	Егор Ждан
Подготовка тестов и решений:	Егор Ждан
Автор разбора:	Егор Ждан

Решение за $O(n^2)$

Перебираем индекс начала отрезка i. Для каждого начала переберем конец отрезка j такой, что $i \leqslant j < i+n$, при этом нужно брать индексы по модулю n, т. к. массив зацикленный (или же можно прибавить массив a сам к себе, как это сделано в реализации ниже). В процессе перебора j считаем сумму на отрезке от i до j. Так будут рассмотрены все возможные отрезки, среди них выберем искомый.

Пример реализации на Python:

```
n, p = map(int, input().split())
a = list(map(int, input().split()))

for i in range(n):
    a.append(a[i])

bestlen = n + 1
besti = -1

if p == 0:
    bestlen = 0
    besti = 0

for i in range(n):
    sm = 0
    for ln in range(1, n + 1):
        sm += a[i + ln - 1]
        if sm >= p:
            if ln < bestlen or (ln == bestlen and i < besti):
                besti = i
                bestlen = ln

if bestlen <= n:
    print(besti + 1, bestlen)
else:
    print(-1)
```

Решение за $O(n * \log(n))$

Будем перебирать индекс начала отрезка i, и для каждого i бинарным поиском искать минимальный конец отрезка j такой, что сумма $a_i + a_{i+1} + ... + a_j \geqslant p$. Чтобы находить сумму на отрезке за $O(1)$, предподсчитаем префиксные суммы для массива a. Среди всех найденных i, j выберем оптимальную.

Решение за $O(n)$

Применим метод двух указателей. Пусть изначально отрезок начинается в первом элементе массива, то есть $i = 0$. Найдем минимальный конец подходящего отрезка j. Сдвинем начало отрезка от i к $i+1$: сумма на отрезке уменьшилась на a_i. Затем увеличим j на такое минимальное $k \geqslant 0$, чтобы сумма на отрезке $[i+1; j+k]$ была не меньше p. При увеличении j будем пересчитывать сумму на текущем отрезке. Среди всех найденных i, j выберем оптимальную.

Разбор задачи «Выбор конфет»

Автор задачи:	Андрей Станкевич
Подготовка тестов и решений:	Михаил Анопренко
Автор разбора:	Михаил Анопренко

В этой задаче участникам была предоставлен большой простор для выбора решения, однако, необходимо было аккуратно реализовать его, учитывая все возможные случаи.

Можно заметить, что решение всегда существует, и поэтому ответа «NO» на самом деле не бывает. Это следует из приведенных ниже решений.

Решение 1. Для начала сведем задачу к случаю, когда s неотрицательно. Действительно, если s отрицательно, то можно решить задачу для числа $-s$, а затем сменить знак у каждого числа в ответе, получив правильный ответ на исходную задачу.

Основная идея решения заключается в том, что можно рассмотреть какой-нибудь набор чисел, не обязательно имеющий сумму s, и либо увеличить в нем максимальный элемент, либо уменьшить минимальный так, чтобы сумма стала равна s.

Например, в качестве изначального набора можно взять набор $a_0 = 0, a_1 = 1, \ldots, a_{n-1} = n - 1$. Сумма этого набора $t = \frac{(n-1)n}{2}$. Далее, возможно три варианта:

- $t = s$. В этом случае исходный набор нам подходит.
- $t < s$. В этом случае увеличим максимальное число в наборе на $s - t$. В таком случае сумма чисел в наборе станет ровно s. Несложно понять, что при этом все числа в наборе по-прежнему будут различны, а также числа в этом наборе не будут превышать ограничение на числа в ответе, так как даже максимальное число не будет превышать s.
- $t > s$. В этом случае уменьшим минимальное число в наборе на $t - s$. Аналогично предыдущему случаю, сумма в наборе будет равна s, все числа будут различны. При этом, так как $s \geqslant 0$, минимальное число уменьшится не более, чем на $\frac{(n-1)n}{2}$, а значит, оно не нарушит ограничение на числа в ответе.

Пример реализации этого решения на языке Python:

```
n = int(input())
s = int(input())

if s < 0:
    sgn = -1
    s = -s
else:
    sgn = 1

a = list(range(n))
t = (n - 1) * n // 2
if s > t:
    a[-1] += s - t
if s < t:
    a[0] -= t - s

for i in range(n):
    a[i] = a[i] * sgn

print("YES")
print(" ".join(map(str, a)))
```

Решение 2. Рассмотрим два принципиально разных случая: $s = 0$ и $s \neq 0$.

Если $s = 0$, то при четном n достаточно составить набор из $\frac{n}{2}$ пар противоположных чисел: $\{-\frac{n}{2}, \ldots, -2, -1, 1, 2, \ldots, \frac{n}{2}\}$, а при нечетном n надо добавить в этот набор 0: $\{-\frac{n}{2}, \ldots, -2, -1, 0, 1, 2, \ldots, \frac{n}{2}\}$.

Пусть теперь $s \neq 0$. Тогда рассмотрим набор, содержащий число s, несколько пар противоположных чисел, не равных s, и, если это необходимо, добавим в этот набор 0.

Изначально добавим в набор число s. Далее, будем перебирать натуральные числа, и, если и число, и противоположное ему, не равны s, то добавим в набор это число и противоположное ему. Будем повторять это действие, пока мы можем добавить в набор хотя бы два числа, то есть его размер не больше $n - 2$.

После этого действия мы имеем набор размера $n - 1$ или n, сумма чисел в нем равна s, и в нем нет нуля. Если в наборе $n - 1$ число, то добавим в него 0, и получим искомый набор.

Пример реализации этого решения на языке C++:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n, s;
    cin >> n >> s;

    vector<int> a;

    if (s == 0) {
        for (int i = 1; i <= n / 2; i++) {
            a.push_back(i);
            a.push_back(-i);
        }
        if (n % 2 == 1)
            a.push_back(0);
    } else {
        a.push_back(s);
        int i = 1;
        while (a.size() <= n - 2) {
            if (i != s && i != -s) {
                a.push_back(i);
                a.push_back(-i);
            }
            i++;
        }
        if (a.size() != n)
            a.push_back(0);
    }

    cout << "YES" << endl;
    for (int i = 0; i < a.size(); i++) {
        cout << a[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Разбор задачи «Подстрока-палиндром»

Автор задачи:	Михаил Анопренко
Подготовка тестов и решений:	Михаил Анопренко
Автор разбора:	Михаил Анопренко

Переберем подстроку строки s, которая станет будущим максимально длинным палиндромом. Пусть границы этой подстроки — индексы l и r.

Теперь нам надо решить такую задачу: проверить, что в строке можно изменить не более одного символа так, чтобы она стала палиндромом. Заметим, что символы в строке разбиваются на пары: первый с последним, второй с предпоследним, и так далее. Символ i будет в паре с символом $r-(i-l)$. При нечетной длине строки центральный элемент будет в паре сам с собой.

Чтобы строка была палиндромом, необходимо, чтобы в каждой паре символы были одинаковыми. Значит, если в паре символы и так одинаковые, можно их не менять, а если разные, один из них необходимо изменить так, чтобы они стали одинаковыми.

Значит, минимальное количество символов, которое надо изменить в строке, чтобы она стала палиндромом, равно количеству пар, в которых символы разные. А значит, надо проверить, что в подстроке, которую мы хотим сделать палиндромом, не более одной пары таких символов.

Пример реализации этого решения на языке Python:

```
n = int(input())
s = input()

res = 0
for l in range(n):
    for r in range(l, n):
        cnt = 0
        for i in range(l, r + 1):
            if s[i] != s[r - (i - l)]:
                cnt += 1
        cnt //= 2

        if cnt <= 1:
            res = max(res, r - l + 1)

print(res)
```

В приведенной реализации при проверке подстроки каждая пара символов считается два раза, поэтому переменная *cnt* делится пополам после прохода по подстроке.

Разбор задачи «Метеорит»

Автор задачи:	Михаил Анопренко
Подготовка тестов и решений:	Макар Селиванов и Михаил Анопренко
Автор разбора:	Макар Селиванов

Заметим, что размер города мал, поэтому мы можем перебрать все дома в нем и проверить, входит ли дом в зону поражения. Дом входит в зону поражения тогда и только тогда, когда расстояние от него до точки приземления метеорита не превосходит r.

Чтобы проверить, что расстояние между двумя точками не более r, воспользуемся теоремой Пифагора. По этой теореме, расстояние между двумя точками с координатами $(x; y)$ и $(x_0; y_0)$ равняется $\sqrt{(x - x_0)^2 + (y - y_0)^2}$.

Пример реализации решения на Python:

```
from math import sqrt

n, m = map(int, input().split())
x, y, r = map(int, input().split())

def dist(x1, y1, x2, y2):
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

answer = 0
for house_x in range(n):
    for house_y in range(m):
        if dist(house_x, house_y, x, y) <= r:
            answer += 1
print(answer)
```

Это решение получается за $O(n^2)$, что достаточно, чтобы вложиться в 2 секунды.

Разбор задачи «Мне только справочку!»

Автор задачи:	Арсений Кириллов
Подготовка тестов и решений:	Алексей Трилис
Автор разбора:	Алексей Трилис

Решение 1. Заметим, что чем позже из очереди вышел человек, тем правее в итоговой последовательности он стоит. Следовательно, всех людей, которые выходили из очереди хотя бы один раз, нужно отсортировать по возрастанию времени их последнего выхода. Людей, которые не выходили из очереди ни разу, нужно отсортировать по их номеру, так как они стоят в том же порядке, как и в начале.

Асимптотика решения — $O(n \log n + t)$.

Пример реализации этого решения на языке C++:

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

using namespace std;

const int MAX_N = 1e5 + 10;

int n, t;
int a[MAX_N];
int mxpos;
pair<int, int> pos[MAX_N];

int main() {
    cin >> n >> t;
    for (int i = 0; i < n; i++) {
        pos[i].first = i;
        pos[i].second = i;
    }
    mxpos = n;
    for (int i = 0; i < t; i++) {
        cin >> a[i];
        a[i]--;
        pos[a[i]].first = mxpos;
        mxpos++;
    }
    sort(pos, pos + n);
    for (int i = 0; i < n; i++) {
        cout << pos[i].second + 1 << " ";
    }
    cout << "\n";
}
```

Решение 2. Рассмотрим события в обратном порядке.

Будем обрабатывать выходящих людей в обратном порядке. Каждое событие выхода обработаем следующим образом: если выходящий человек выходил позже (то есть событие его выхода уже обработано), то его выход не оказывает никакого влияния на итоговую перестановку, пропустим его. Иначе добавим выходящего человека в конец последовательности. После обработки всех событий добавим в начало последовательности людей, которые ни разу не выходили из очереди в том порядке, в котором они стояли изначально.

После выполнения этого алгоритма люди, которые выходили из очереди хотя бы один раз, будут отсортированы по возрастанию времени их выхода, как и в предыдущем решении.

Асимптотика решения — $O(n+t)$.

Пример реализации этого решения на языке C++:

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

using namespace std;

const int MAX_N = 1e5 + 10;

int n, t;
int a[MAX_N];
bool used[MAX_N];
vector<int> ans;

int main() {
    cin >> n >> t;
    for (int i = 0; i < t; i++) {
        cin >> a[i];
        a[i]--;
    }
    for (int i = t - 1; i >= 0; i--) {
        if (!used[a[i]]) {
            used[a[i]] = 1;
            ans.push_back(a[i]);
        }
    }
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            cout << i + 1 << " ";
        }
    }
    for (int i = ans.size() - 1; i >= 0; i--) {
        cout << ans[i] + 1 << " ";
    }
    cout << "\n";
}
```

Решение 3. Также эту задачу можно решить с помощью двусвязного списка.

Будем хранить для каждого человека номер человека слева и справа в очереди от него, а также номер первого и последнего человека в очереди.

Заметим, что при каждом событии меняются соседи только у двух людей: того, который выходит из очереди, и и последнего в очереди, а также, возможно, номера людей в начале и конце очереди. Достаточно аккуратно изменять все необходимые значения при каждом выходе человека.

Асимптотика решения — $O(n+t)$.

Пример реализации этого решения на языке C++:

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

using namespace std;

const int MAX_N = 1e5 + 10;
int n, t;
int a[MAX_N];
int st, fi;
int nx[MAX_N], pr[MAX_N];

int main() {
    cin >> n >> t;
    for (int i = 0; i < t; i++) {
        cin >> a[i];
        a[i]--;
    }
    fi = n - 1;
    for (int i = 0; i < n; i++) {
        pr[i] = i - 1;
        nx[i] = i + 1;
    }
    nx[fi] = -1;
    for (int i = 0; i < t; i++) {
        if (a[i] == fi)
            continue;
        if (nx[a[i]] != -1)
            pr[nx[a[i]]] = pr[a[i]];
        if (pr[a[i]] != -1)
            nx[pr[a[i]]] = nx[a[i]];
        if (a[i] == st && n > 1)
            st = nx[a[i]];
        nx[fi] = a[i];
        pr[a[i]] = fi;
        nx[a[i]] = -1;
        fi = a[i];
    }
    int i = st;
    while (i != -1) {
        cout << i + 1 << " ";
        i = nx[i];
    }
    cout << "\n";
}
```

Разбор задачи «Ежедневные награды»

Автор задачи: Михаил Иванов
Подготовка тестов и решений: Егор Ждан
Автор разбора: Михаил Иванов

Будем решать задачу методом динамического программирования. Для удобства введём $a_0 = 0$. Обозначим через $\texttt{dp}[k][\ell]$ наибольшую суммарную награду, которую мы сможем получить за k дней, если в последний из них была получена награда a_ℓ ($k \in [0; m]$, $\ell \in [0; n]$). Если получить в k-й день награду a_ℓ невозможно (то есть $k < \ell$), то вместо этого $\texttt{dp}[k][\ell] = -\infty$.

Зададим начальные значения: $\texttt{dp}[0][0] = 0$, $\texttt{dp}[0][\ell] = -\infty$ для $\ell \neq 0$ (за 0 дней мы не получили ни одной награды, суммарная прибыль равна нулю). Пересчёт $\texttt{dp}[k][\ell]$ для $k > 0$ осуществляется в зависимости от ℓ. Если $\ell = 0$, то мы можем получать награды лишь в первые $k-1$ дней. Максимальная сумма для них — наибольшее значение $\texttt{dp}[k-1][\ell]$ по всем ℓ. Если $\ell = 1$, то есть мы в k-й день получили награду a_1, то за день до этого мы могли получить либо a_n, либо 0. Сумма в этих случаях получится равной $\texttt{dp}[k-1][0] + a_1$ и $\texttt{dp}[k-1][n] + a_1$ соответственно, а нам необходимо выбрать из этих величин максимум. Наконец, если $\ell > 1$, то в день $k-1$ гарантированно была получена награда $a_{\ell-1}$. Поэтому максимальная возможная суммарная награда будет равна $\texttt{dp}[k-1][\ell-1] + a_\ell$.

Вычислим значения этой динамики при всех $k \in [0; m]$, $\ell \in [0; n]$. Очевидно, тогда ответом на задачу будет максимум из $\texttt{dp}[m][\ell]$ по всем $\ell \in [0; n]$ (нужное ℓ окажется равным номеру награды, которую нужно получить в последний день при оптимальном алгоритме действий).

Пример реализации этого решения на языке C++:

```
#include <iostream>
#include <vector>

using namespace std;

const int MAXN = 1002;
const int INF = 1000000000;

int dp[MAXN][MAXN];

int main() {
    int n, m;
    scanf("%d%d", &m, &n);
    vector<int> a(m);
    for (int i = 0; i < m; i++) {
        scanf("%d", &a[i]);
    }
    for (int i = 0; i <= n; i++) {
        fill(dp[i], dp[i] + m + 1, -INF);
    }
    dp[0][0] = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (dp[i][j] == -INF) {
                continue;
            }
            dp[i + 1][0] = max(dp[i + 1][0], dp[i][j]);
            int rew = a[j % m];
            dp[i + 1][(j + 1) % m] =
```

```
                max(dp[i + 1][(j + 1) % m], dp[i][j] + rew);
            }
        }

        int mx = 0;
        for (int i = 0; i < m; i++) {
            mx = max(mx, dp[n][i]);
        }

        cout << mx << endl;

        return 0;
}
```

Разбор задачи «Сломанный робот»

Автор задачи:	Арсений Кириллов
Подготовка тестов и решений:	Арсений Кириллов
Авторы разбора:	Арсений Кириллов, Михаил Иванов

Так как в этой задаче не требовалось находить кратчайший путь до точки, можно было действовать многими способами.

Например, давайте сначала сделаем путь, в котором можно два раза подряд идти в одном направлении. Такой путь можно получить, если прийти из точки к оси OY шагами влево или вправо, а потом, уже по этой оси идя вниз или вверх, попасть в точку $(0, 0)$. После этого надо между каждыми двумя одинаковыми символами вставить последовательность, которая вернёт точку к исходному месту, но не содержит этих символов. Например, между горизонтальными передвижениями можно вставить строку `UD`, а между вертикальными — строку `LR`. После этого у нас получится последовательность действий, которая приводит в точку $(0, 0)$, длины не более чем 10000.

Приведём также рассуждение, позволяющее найти маршрут, содержащий минимальное число шагов. Первая идея такова: при любой операции чётность суммы координат клетки, в которой находится робот, меняется (так как чётность одной из координат остаётся прежней, а другой — заменяется на противоположную). Так как конечная клетка маршрута $(0, 0)$ имеет чётную сумму координат, количество операций должно совпадать по чётности с суммой координат стартовой клетки. Значит, если x и y одной чётности, то количество операций кратно двум, а в противном случае не кратно.

Вторая идея заключается в рассмотрении одной координаты отдельно от другой. Пусть $x \geqslant 0$. Тогда за две соседние операции x не может уменьшиться сильнее, чем на 1 (так как иначе были подряд применены операции `LL`, что запрещено условием). Из этого следует, что уменьшить x до нуля невозможно последовательностью операций, длина которой не превосходит $2x - 2$ (так как иначе все операции удастся поделить на $x - 1$ или менее групп, в каждой из которых либо одна операция, либо две соседних; тогда в каждой группе не более одной операции `L`, вследствие чего абсцисса не уменьшится сильнее, чем на $x - 1$). Итак, операций не может быть меньше $2x - 1$. Аналогично рассмотрев случай $x < 0$, выясним, что операций не меньше $2|x| - 1$. Так как про координату y можно сказать то же самое, получаем оценку на $2\max(|x|, |y|) - 1$ операций.

Объединим эти две идеи: если $x+y$ чётно, то операций не может быть ровно $2\max(|x|, |y|)-1$, так как это число нечётно. Значит, их должно быть хотя бы $2\max(|x|, |y|)$. В общем случае количество операций не может быть меньше $2\max(|x|, |y|) - (x + y) \mod 2$.

Чтобы доказать, что это оценка точная, достаточно привести алгоритм, действуя согласно которому, робот доберётся за базы именно за это число операций. Для начала научимся оптимально перемещать робота так, чтобы его абсцисса изменилась на x, а ордината — на y, причём $|x| = |y|$. Легко видеть, что робот сможет добраться до начала координат, чередуя горизонтальные и вертикальные шаги, причём он может начать как с вертикального, так и с горизонтального (например, если необходимо переместиться на $(-3, 3)$, то нас удовлетворят обе последовательности `LULULU` и `ULULUL`). Значит, если до этого робот уже совершил некоторое количество операций, он сможет выбрать ту последовательность, которая доведёт до базы за $2|x|$ шагов и не конфликтует с предыдущими операциями.

Теперь мы можем оптимально добраться до базы. Действительно, если x и y одной чётности, то применим последовательность из предыдущего абзаца, сначала чтобы переместиться из (x, y) в $\left(\frac{x+y}{2}, \frac{x+y}{2}\right)$, а потом оттуда в начало координат — легко видеть, что модули изменения абсциссы и ординаты при обоих перемещениях совпадают. Мы потратим $2|x - \frac{x+y}{2}| + 2|\frac{x+y}{2}| = |x - y| + |x + y| = 2\max(|x|, |y|)$.

Наконец, если x и y разной чётности, то $|x| \neq |y|$. Тогда первым делом совершим ту операцию, которая уменьшит на единицу величину $\max(|x|, |y|)$ (такая есть ровно одна). После этого доберёмся до базы, как сказано в предыдущем абзаце, за $2(\max(|x|, |y|) - 1)$ операций, что в совокупности с первым шагом даёт нужные $2\max(|x|, |y|) - 1$.

Разбор задачи «Нужно больше конфет!»

Автор задачи:	Михаил Анопренко
Подготовка тестов и решений:	Михаил Анопренко
Автор разбора:	Михаил Анопренко

По условию, в конце количества конфет в банках могло принимать не более двух различных значений. Пусть эти значения равны x и y, причем $x \leqslant y$.

Заметим сначала, что x и y — это исходные количества конфет в каких-то банках, в противном случае одно из них можно уменьшить, и ответ улучшится. Также, $y \geqslant \max a_i$, ведь в банке, в которой изначально было максимальное количество конфет, тоже стало x или y конфет. А тогда, $y = \max a_i$. Значит, все, что мы можем выбирать в этой задаче — это значение x.

Пусть мы выбрали значение x. Тогда заметим, что во всех банках, в которых было не более x конфет, надо сделать ровно x конфет, а в остальных — y конфет.

Пример реализации этого решения на языке C++:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++)
        cin >> a[i];

    int y = a[0];
    for (int i = 1; i < n; i++)
        y = max(y, a[i]);

    long long best = ((long long) y) * n;
    for (int x : a) {
        long long cur = 0;
        for (int i = 0; i < n; i++) {
            if (a[i] <= x) {
                cur += x - a[i];
            } else {
                cur += y - a[i];
            }
        }
        best = min(best, cur);
    }

    cout << best << endl;

    return 0;
}
```

При наивной реализации этого решения, приведенной выше, время работы программы составляет $O(n^2)$, и она не укладывается в ограничение по времени.

Отсортируем массив a_i так, чтобы количества конфет в банках располагались по неубыванию: $a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n-1}$. Пусть мы выбрали число $x = a_j$. Тогда заметим, что для любого $k \leqslant j$, в банку с индексом k будет добавлено $x - a_k$ конфет, в противном случае в нее будет добавлено $y - a_k$ конфет.

Тогда всего будет добавлено конфет:

$(x - a_0) + (x - a_1) + \cdots + (x - a_j) + (y - a_{j+1}) + \cdots + (y - a_{n-1}) = x(j+1) + y(n-1-j) - \sum_{i=0}^{n-1} a_i$

Заметим, что эту сумму можно вычислять для каждого j за $O(1)$, посчитав заранее сумму всех элементов в массиве a. Получаем решение задачи: отсортировать массив a, посчитать сумму всех элементов в нем и найти минимальное значение функции, описанной выше.

Пример реализации этого решения на языке Python:

```
n = int(input())
a = list(map(int, input().split()))
a.sort()
s = sum(a)

res = a[n - 1] * n
y = max(a)
for j in range(n):
        x = a[j]
        cur = x * (j + 1) + y * (n - 1 - j) - s
        res = min(res, cur)

print(res)
```

Разбор задачи «Проверка автомата»

Автор задачи:	Михаил Анопренко
Подготовка тестов и решений:	Михаил Анопренко
Автор разбора:	Михаил Анопренко

Для решения этой задачи нужно было понять, какое действие является композицией действий, совершаемых минимизаторами и максимизаторами.

Определим операцию «*Загнать число x в отрезок $[L; R]$*». Результат этой операции равен

- L, если $x \leqslant L$
- R, если $x \geqslant R$
- x, если $L < x < R$

Иными словами, загнать число x в отрезок — это взять ближайшее к x на числовой прямой число, находящееся в этом отрезке.

Докажем, что после каждого добавления блоков автомат загоняет данное ему на вход число в какой-то отрезок. Изначально, когда в автомате нет блоков, можно считать, что его действие — это загнать число в отрезок $[1; 10^9]$.

Пусть после добавления некоторого количества блоков автомат загоняет числа в отрезок $[L; R]$. Рассмотрим возможные варианты добавляемого блока:

- Добавился максимизизатор с числом x. Тогда если $x \leqslant L$, действие автомата не изменится. Если $x \geqslant R$, то любое число под действием автомата будет обращаться в x, то есть автомат будет загонять числа в отрезок $[x; x]$. Если же $L < x < R$, то автомат будет загонять числа в отрезок $[x; R]$.
- Добавился минимизатор с числом x. Тогда если $x \geqslant R$, действие автомата не изменится. Если $x \leqslant L$, то любое число под действием автомата будет обращаться в x, то есть автомат будет загонять числа в отрезок $[x; x]$. Если же $L < x < R$, то автомат будет загонять числа в отрезок $[L; x]$.

Таким образом, чтобы решить задачу, необходимо лишь хранить текущий отрезок, в который автомат загоняет данные ему числа, и изменять его в соответствии с описанными выше правилами.

Пример реализации этого решения на языке C++:

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    int L = 1, R = 1000000000;
    for (int i = 0; i < n; i++) {
        int t, x;
        cin >> t >> x;
        if (t == 1) {
            if (x >= R)
                L = R = x;
            else if (x >= L)
                L = x;
        } else if (t == 2) {
            if (x <= L)
                L = R = x;
            else if (x <= R)
                R = x;
        } else {
            if (x <= L)
                cout << L << '\n';
            else if (x >= R)
                cout << R << '\n';
            else
                cout << x << '\n';
        }
    }

    return 0;
}
```