

Задача G. Прямая кинематическая задача для плоского манипулятора (2 звена, плоскость XZ)

Условие задачи

Рассматривается плоский робот-манипулятор, состоящий из двух жёстких звеньев длиной L_1 и L_2 , соединённых шарнирами. Первое звено закреплено в начале координат $O = (0, 0)$ и может вращаться в плоскости XZ.

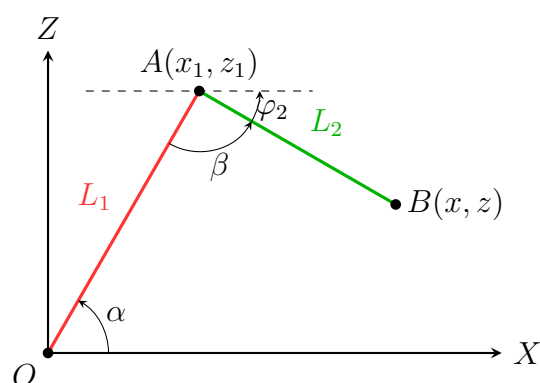
Система координат:

- ось X направлена горизонтально вправо;
- ось Z направлена вертикально вверх.

Углы задаются следующим образом:

- угол α — от горизонтали (оси X) к первому звену L_1 ($0^\circ \leq \alpha \leq 135^\circ$);
- угол β — угол между звеньями L_1 и L_2 ($20^\circ \leq \beta \leq 300^\circ$), как на рисунке.

Эскиз манипулятора в плоскости XZ



Комментарий к углу β

При таком определении:

- $\beta = 0^\circ$ — направлено в точности в сторону, противоположную направлению L_1 ;
- $0^\circ < \beta < 180^\circ$ — L_2 уходит “вниз” относительно линии продолжения L_1 (как на рисунке);
- $\beta = 180^\circ$ — L_2 звенья выпрямлены, L_2 продолжает L_1 ;

- $\beta > 180^\circ$ — L_2 “перебрасывается” на другую сторону и уходит “вверх” относительно линии продолжения L_1 .

Связь β с абсолютным углом второго звена. Угол β в этой задаче задаётся в вершине A между звеном L_1 и звеном L_2 . Поэтому абсолютный угол второго звена относительно оси OX равен

$$\varphi_2 = \alpha + \beta - 180^\circ$$

(в радианах: $\varphi_2 = \alpha + \beta - \pi$).

Интуитивная проверка по крайним положениям:

$$\beta = 180^\circ \Rightarrow \varphi_2 = \alpha \text{ (} L_2 \text{ продолжает } L_1 \text{)}, \quad \beta = 0^\circ \Rightarrow \varphi_2 = \alpha \pm 180^\circ \text{ (} L_2 \text{ сложено назад)}.$$

Ниже, на плоском эскизе, дополнительно показан угол φ_2 как *абсолютный наклон* звена L_2 к горизонтали s .

Требуется найти координаты конца второго звена (x, z) .

Входные данные

Во входной поток подаются четыре вещественных числа:

`L1 L2 alpha beta`

где L_1, L_2 — длины звеньев, α, β — углы в градусах.

Выходные данные

Вывести два числа — координаты (x, z) конца второго звена. Ответ округлить до 3 знаков после запятой.

Пример

Вход:

`3 2 45 75`

Выход:

`3.121 0.389`

Справка по геометрии

Для вывода формул кинематики используются свойства прямоугольного треугольника. Рассмотрим прямоугольный треугольник с гипотенузой c , катетами a и b и острым углом θ (см. рисунок).

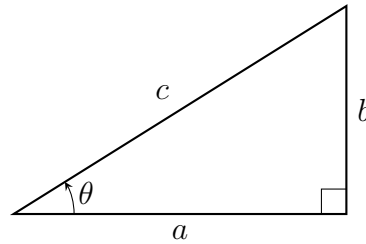


Рис. 1: Прямоугольный треугольник и определение синуса и косинуса.

Синус и косинус угла. Для острого угла θ в прямоугольном треугольнике определены:

$$\sin \theta = \frac{\text{противолежающий катет}}{\text{гипотенуза}} = \frac{b}{c}, \quad \cos \theta = \frac{\text{прилежащий катет}}{\text{гипотенуза}} = \frac{a}{c}.$$

Нахождение катетов по углу и гипотенузе. Если известны гипотенуза c и угол θ , то длины катетов выражаются как

$$a = c \cos \theta, \quad b = c \sin \theta.$$

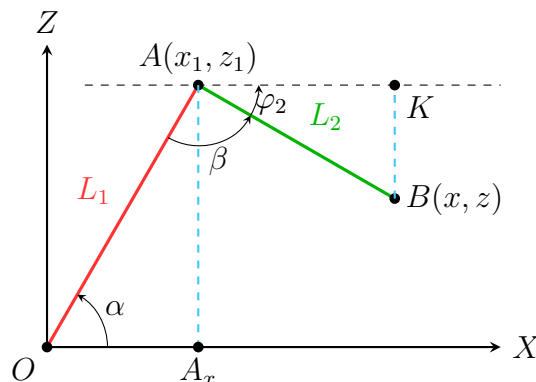
Именно эти соотношения используются при разложении звеньев манипулятора на проекции на координатные оси. Углы в формулах должны быть в радианах. Перевод из градусов:

$$\theta_{\text{рад}} = \theta_{\text{град}} \cdot \frac{\pi}{180}.$$

Теоретическое решение

Работаем в плоскости XZ с координатами (x, z) .

Дополним приложенный эскиз перпендикуляром из конца первого звена (точка A) на ось X (новая точка A_x). Также построим перпендикуляр из конца второго звена на линию проходящую параллельно оси OX через точку A , эту точку обозначим как K . Тогда мы имеем два прямоугольных треугольника: OAA_x и AKB



Определим координаты конца первого звена из прямоугольного треугольника OAA_x :

$$x_1 = L_1 \cos \alpha, \quad z_1 = L_1 \sin \alpha.$$

Абсолютный угол второго звена φ_2 как было сказано в условии и на рисунке равен:

$$\varphi_2 = \alpha + \beta - \pi.$$

Координаты конца второго звена определим из прямоугольного треугольника AKB :

$$x = x_1 + L_2 \cos(\varphi_2),$$

$$z = z_1 + L_2 \sin(\varphi_2).$$

Следует обратить внимание что угол φ_2 имеет отрицательное значение при условиях указанных на рисунке. Косинус такого угла имеет положительное значение, а синус отрицательное, что и дает правильный расчет координат конца второго звена.

Углы в формулах должны быть в радианах. Перевод из градусов:

$$\theta_{\text{рад}} = \theta_{\text{град}} \cdot \frac{\pi}{180}.$$

Пример решения на C++

```
// Прямая кинематическая задача для манипулятора из 2х звеньев на XZ плоскости
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

int main() {
    double L1, L2, alphaDeg, betaDeg;
    cin >> L1 >> L2 >> alphaDeg >> betaDeg;

    // alpha: от оси X до первого звена
    double alpha = alphaDeg * M_PI / 180.0;
    // beta: угол между L1 и L2 (как показано на рисунке)
    double beta = betaDeg * M_PI / 180.0;

    // окончание первого звена
    double x1 = L1 * cos(alpha);
    double z1 = L1 * sin(alpha);

    // абсолютный угол второго звена
    double phi2 = alpha + beta - M_PI;

    // окончание второго звена
    double x = x1 + L2 * cos(phi2);
    double z = z1 + L2 * sin(phi2);

    cout << fixed << setprecision(3)
         << x << " " << z << endl;

    return 0;
}
```

Типичные ошибки

- забыли перевести градусы в радианы;

- неверно вычислили φ_2 (нужно $\alpha + \beta - 180$);
- использовали β как абсолютный угол вместо угла между звеньями;
- неверно интерпретировали ось Z (должна быть вертикальной).

Задача Н. Прямая кинематическая задача с вращающейся базой (3 степени свободы)

Условие задачи

Рассматривается робот-манипулятор, состоящий из:

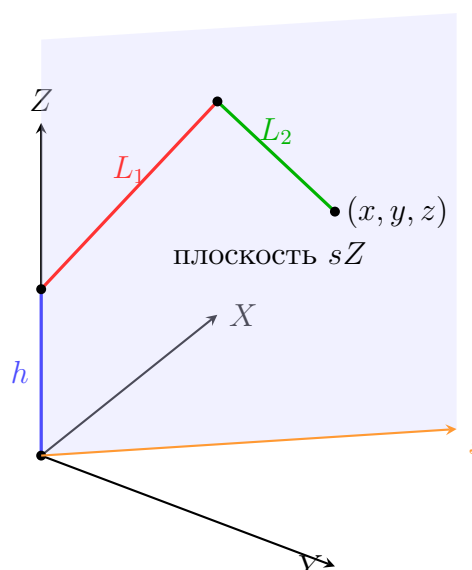
- вертикальной вращающейся базы высотой h ;
- двух звеньев длинами L_1 и L_2 , расположенных в плоскости.

База может вращаться вокруг вертикальной оси OZ на угол γ ($-90^\circ \leq \gamma \leq 90^\circ$). В плоскости, связанной с базой, расположены два звена:

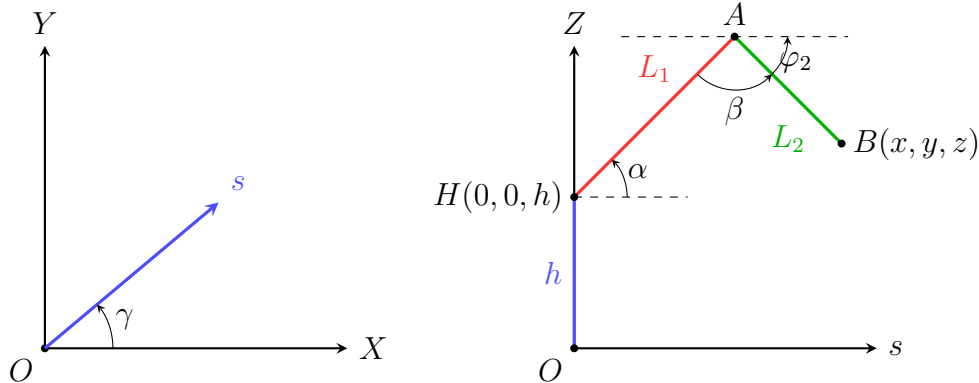
- первое звено поворачивается на угол α от оси OX ($0^\circ \leq \alpha \leq 135^\circ$);
- второе звено отклоняется на угол β от первого ($20^\circ \leq \beta \leq 300^\circ$);
- абсолютный угол второго звена (наклон L_2 к оси s): $\varphi_2 = \alpha + \beta - 180^\circ$.

Требуется найти координаты рабочего органа (x, y, z) в декартовой системе координат по известным размерам манипулятора и углам его звеньев.

3D-вид манипулятора и рабочей плоскости



Углы γ (вид сверху) и α, β (плоскость sZ)



Комментарий к углу β

При таком определении:

- $\beta = 0^\circ$ — направлено в точности в сторону, противоположную направлению L_1 ;
- $0^\circ < \beta < 180^\circ$ — L_2 уходит “вниз” относительно линии продолжения L_1 (как на рисунке);
- $\beta = 180^\circ$ — L_2 звенья выпрямлены, L_2 продолжает L_1 ;
- $\beta > 180^\circ$ — L_2 “перебрасывается” на другую сторону и уходит “вверх” относительно линии продолжения L_1 .

Связь β с абсолютным углом второго звена. Угол β в этой задаче задаётся в вершине A между лучом $A \rightarrow H$ (звено L_1) и лучом $A \rightarrow B$ (звено L_2). Поэтому абсолютный угол второго звена относительно оси s равен

$$\varphi_2 = \alpha + \beta - 180^\circ$$

(в радианах: $\varphi_2 = \alpha + \beta - \pi$).

Интуитивная проверка по крайним положениям:

$$\beta = 180^\circ \Rightarrow \varphi_2 = \alpha \text{ (} L_2 \text{ продолжает } L_1 \text{),} \quad \beta = 0^\circ \Rightarrow \varphi_2 = \alpha \pm 180^\circ \text{ (} L_2 \text{ сложено назад).}$$

Ниже, на плоском эскизе, дополнительно показан угол φ_2 как *абсолютный наклон* звена L_2 к горизонтали s .

Входные данные

Во входной поток подаются две строки.

1. В первой строке три вещественных числа:

L_1 L_2 h

2. Во второй строке три вещественных числа:

alpha beta gamma

где:

- L_1, L_2 — длины звеньев;
- h — высота базы;
- α, β — углы звеньев в плоскости;
- γ — угол поворота базы вокруг оси Z .

Все углы задаются на вход в градусах.

Выходные данные

Вывести три числа — координаты (x, y, z) конца второго звена. Ответ округлить до 3 знаков.

Пример

Вход:

3 2 2
45.000 75.000 45.000

Выход:

2.207 2.207 2.389

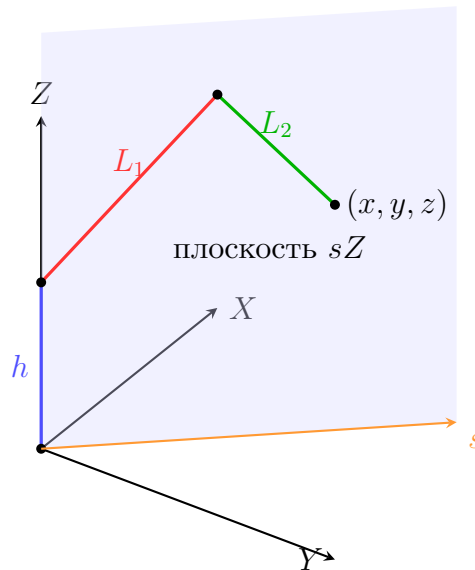
Теоретическое объяснение

Рассматривается манипулятор с двумя звеньями в плоскости sZ и поворотом этой плоскости вокруг оси Z на угол γ .

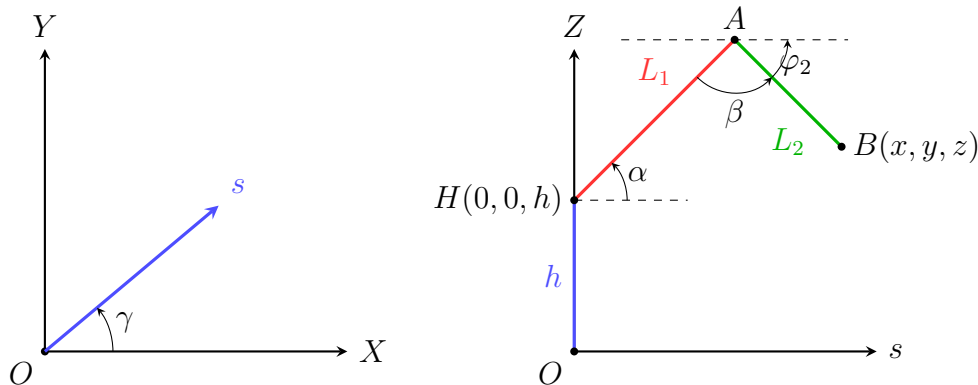
- Ось s — горизонтальная ось в плоскости sZ , полученная как проекция манипулятора на плоскость XY .
- Ось Z направлена вертикально вверх.
- Угол γ — угол в плоскости XY от оси X до оси s .
- Угол α — угол от горизонтали (оси s) к первому звену L_1 в плоскости sZ .
- Угол β — угол между самым звеном L_1 и звеном L_2 , измеренный так, как показано стрелкой на эскизе:
 - при $0^\circ \leq \beta \leq 180^\circ$ звено L_2 отклонено *вниз* относительно линии продолжения L_1 ;
 - при $\beta > 180^\circ$ звено L_2 отклонено *вверх*.

Таким образом, β — это “локальный” угол в суставе, удобный для визуализации и ограничения в задаче, но **не совпадает** с углом наклона второго звена относительно оси Os .

3D-вид манипулятора и рабочей плоскости



Углы γ (вид сверху) и α, β (плоскость sZ)



Формулы прямой кинематики в плоскости sZ

Работаем сначала в плоскости sZ с координатами (s, z) , где

- s — координата вдоль оси s (горизонталь),
- z — высота вдоль оси Z .

Вершина колонны:

$$H = (0, h).$$

Первое звено L_1 имеет угол α от горизонтали, поэтому координаты точки A (конца L_1):

$$s_A = L_1 \cos \alpha, \quad z_A = h + L_1 \sin \alpha.$$

Второе звено L_2 образует с первым звеном угол β как это показано на рисунке. Поскольку β отсчитывается от направления L_1 и полный угол под верхней пунктирной

линией составляет $180^\circ = \alpha + \beta + \varphi_2$, то **абсолютный** угол наклона второго звена относительно оси s :

$$\varphi_2 = \alpha + \beta - 180^\circ.$$

Координаты точки B (конец L_2) в плоскости sZ :

$$s_B = s_A + L_2 \cos \varphi_2,$$

$$z_B = z_A + L_2 \sin \varphi_2.$$

Переход к глобальным координатам (x, y, z)

Ось s — это повёрнутая на угол γ ось X в плоскости XY . Поэтому, зная (s_B, z_B) , получаем:

$$x = s_B \cos \gamma, \quad y = s_B \sin \gamma, \quad z = z_B.$$

Итого:

$$\begin{aligned} s_A &= L_1 \cos \alpha, \\ z_A &= h + L_1 \sin \alpha, \\ s_B &= s_A + L_2 \cos \varphi_2, \quad \text{где } \varphi_2 = \alpha + \beta - 180^\circ, \\ z_B &= z_A + L_2 \sin \varphi_2, \\ x &= s_B \cos \gamma, \\ y &= s_B \sin \gamma, \\ z &= z_B. \end{aligned}$$

Все углы в формулах должны быть переведены в радианы:

$$\theta_{\text{рад}} = \theta_{\text{град}} \cdot \frac{\pi}{180}.$$

Пример реализации на C++

Ниже приведён код прямой кинематики с учётом именно такой геометрии задания углов, как на эскизах.

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

int main() {
    double L1, L2, h;
    double alphaDeg, betaDeg, gammaDeg;
    cin >> L1 >> L2 >> h ;
    cin >> alphaDeg >> betaDeg >> gammaDeg;

    double alpha = alphaDeg * M_PI / 180.0; // от горизонтали (оси s) к L1
    double beta = betaDeg * M_PI / 180.0; // угол между L1 и L2, как на рисунке
    double gamma = gammaDeg * M_PI / 180.0; // угол между осями X и s
```

```
// Координаты конца первого звена в плоскости sZ
double sA = L1 * cos(alpha);
double zA = h + L1 * sin(alpha);

// Абсолютный угол второго звена в плоскости sZ
double phi2 = alpha + beta - M_PI; // = alpha + beta - 180° (абсолютный угол L2)

// Координаты конца второго звена в плоскости sZ
double sB = sA + L2 * cos(phi2);
double zB = zA + L2 * sin(phi2);

// Переход к глобальным координатам X, Y, Z
double x = sB * cos(gamma);
double y = sB * sin(gamma);
double z = zB;

cout << fixed << setprecision(3)
      << x << " " << y << " " << z << endl;

return 0;
}
```

Пример прямой кинематики

Рассмотрим параметры:

$$L_1 = 3, \quad L_2 = 2, \quad h = 2,$$

$$\alpha = 45^\circ, \quad \beta = 75^\circ, \quad \gamma = 45^\circ.$$

1. Конец первого звена

$$s_A = L_1 \cos \alpha = 3 \cdot 0.7071 = 2.121,$$

$$z_A = h + L_1 \sin \alpha = 2 + 3 \cdot 0.7071 = 4.121.$$

2. Абсолютный угол второго звена

Так как угол β отложен от самого звена L_1 , абсолютный наклон второго звена:

$$\varphi_2 = \alpha + \beta - 180^\circ = 45^\circ + 75^\circ - 180^\circ = -60^\circ.$$

3. Конец второго звена

$$s_B = s_A + L_2 \cos(\varphi_2) = 2.121 + 2 \cdot 0.5 = 3.121,$$

$$z_B = z_A + L_2 \sin(\varphi_2) = 4.121 + 2(-0.8660) = 2.389.$$

4. Переход в глобальные координаты

$$x = s_B \cos \gamma = 3.121 \cdot 0.7071 = 2.207,$$

$$y = s_B \sin \gamma = 3.121 \cdot 0.7071 = 2.207,$$

$$z = z_B = 2.389.$$

Итог

$(x, y, z) = (2.207, 2.207, 2.389)$

Задача I. Обратная кинематическая задача с вращающейся базой (3 степени свободы)

Условие задачи

Рассматривается тот же робот-манипулятор, что и в задаче Н:

- вертикальная вращающаяся база высотой h ;
- два звена длиной L_1 и L_2 , работающих в плоскости sZ ;
- поворот рабочей плоскости вокруг оси Z на угол γ ($-90^\circ \leq \gamma \leq 90^\circ$);
- первое звено L_1 в плоскости sZ наклонено на угол α от оси s ($0^\circ \leq \alpha \leq 135^\circ$);
- второе звено L_2 образует с первым звеном угол β (угол между звеньями, как на эскизе) ($20^\circ \leq \beta \leq 180^\circ$).

Точка O находится в основании колонны. В глобальной системе координат XYZ :

- ось Z направлена вверх;
- оси X и Y лежат в горизонтальной плоскости;
- ось s — это ось X , повернутая на угол γ в плоскости XY .

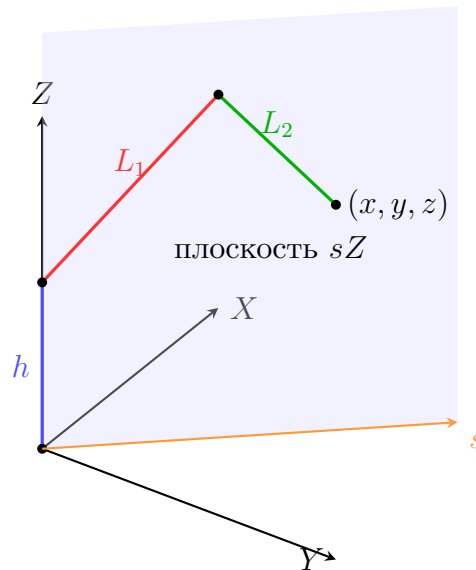
Прямая кинематика (задача Н) по заданным $L_1, L_2, h, \alpha, \beta, \gamma$ позволяет найти координаты конца второго звена (x, y, z) .

В этой задаче требуется решить **обратную** задачу:

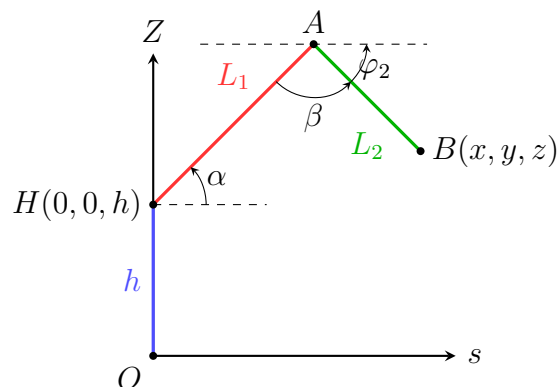
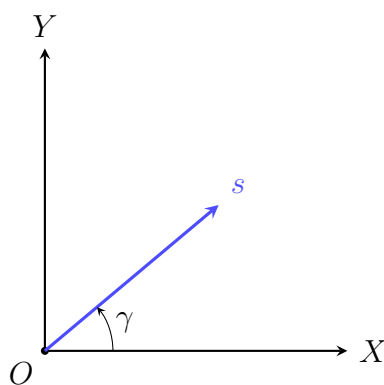
По известным L_1, L_2, h и заданной точке (x, y, z) найти такие углы γ, α и β , чтобы конец второго звена оказался в точке (x, y, z) .

Считаем, что точка достижима и робот работает в конфигурации “локоть вниз”, то есть выбираем решение с $0^\circ \leq \beta \leq 180^\circ$.

3D-вид манипулятора и рабочей плоскости



Углы γ (вид сверху) и α, β (плоскость sZ)



Входные данные

Во входной поток подаются две строки.

1. В первой строке три вещественных числа:

L_1 L_2 h

2. Во второй строке три вещественных числа:

x y z

Выходные данные

Вывести три числа — углы:

- γ — поворот рабочей плоскости вокруг оси Z ;

- α — угол первого звена от горизонтали (оси s) в плоскости sZ ;
- β — суставной угол между звеньями L_1 и L_2 (как на рисунке).

Все углы вывести в **градусах**, с точностью 3 знака после запятой.

Пример

Рассмотрим манипулятор с параметрами

$$L_1 = 3, \quad L_2 = 2, \quad h = 2,$$

и углами

$$\alpha = 45^\circ, \quad \beta = 75^\circ, \quad \gamma = 45^\circ.$$

Сначала найдём абсолютный наклон второго звена по принятому соглашению:

$$\varphi_2 = \alpha + \beta - 180^\circ = 45^\circ + 75^\circ - 180^\circ = -60^\circ.$$

Тогда в плоскости sZ' :

$$s_A = L_1 \cos \alpha = 3 \cos 45^\circ \approx 2,121, \quad z'_A = L_1 \sin \alpha = 3 \sin 45^\circ \approx 2,121,$$

$$s_B = s_A + L_2 \cos \varphi_2 \approx 2,121 + 2 \cos(-60^\circ) \approx 2,121 + 1,000 = 3,121,$$

$$z'_B = z'_A + L_2 \sin \varphi_2 \approx 2,121 + 2 \sin(-60^\circ) \approx 2,121 - 1,732 = 0,389.$$

Переход в пространство даёт:

$$x = s_B \cos \gamma \approx 3,121 \cos 45^\circ \approx 2,207, \quad y = s_B \sin \gamma \approx 3,121 \sin 45^\circ \approx 2,207,$$

$$z = z'_B + h \approx 0,389 + 2 = 2,389.$$

Итак, прямая кинематика даёт координаты конца второго звена:

$$(x, y, z) \approx (2,207, 2,207, 2,389).$$

Тогда в обратной задаче при тех же L_1, L_2, h и точке (x, y, z) восстановленные углы должны совпасть с исходными.

Вход:

3 2 2
2.207 2.207 2.389

Выход:

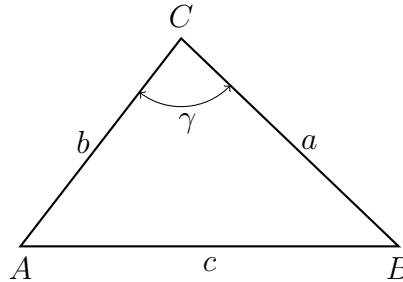
45.000 45.000 75.000

(Порядок: gamma, alpha, beta.)

Справочные материалы из геометрии

1. Треугольник и теорема косинусов (без доказательства)

Рассмотрим произвольный треугольник ABC . Внутренний угол, о котором говорит теорема косинусов, — это угол *при вершине C* . Сторона c лежит *напротив* этого угла.



Теорема косинусов (формулировка без доказательства):

В любом треугольнике со сторонами a, b, c и внутренним углом γ при вершине C (напротив стороны c) выполняется

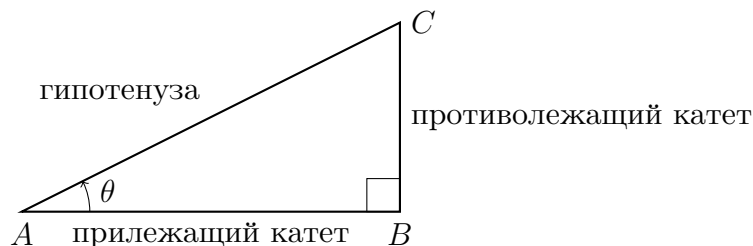
$$c^2 = a^2 + b^2 - 2ab \cos \gamma.$$

Если угол $\gamma = 90^\circ$, то $\cos \gamma = 0$ и получается теорема Пифагора:

$$c^2 = a^2 + b^2.$$

2. Прямоугольный треугольник: тангенс и арктангенс

Рассмотрим прямоугольный треугольник с прямым углом при вершине B и острым углом θ при вершине A .



Определение тангенса:

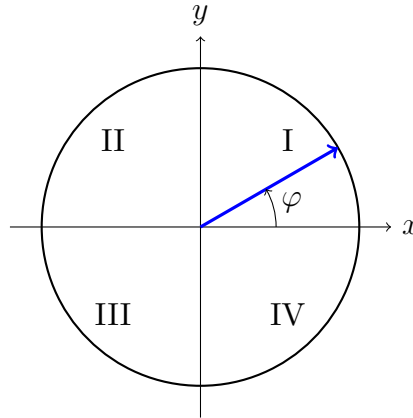
$$\tan \theta = \frac{\text{противолежащий катет}}{\text{прилежащий катет}}.$$

Если известны длины катетов, то угол можно найти с помощью обратной функции:

$$\theta = \arctan\left(\frac{\text{противолежащий катет}}{\text{прилежащий катет}}\right).$$

3. Функция $\text{atan2}(y, x)$ и тригонометрический круг

Функция $\text{atan2}(y, x)$ — более удобная версия арктангенса для работы с координатами точки (x, y) . Она сразу возвращает угол в правильной четверти.



Для точки (x, y) :

- длина вектора: $\sqrt{x^2 + y^2}$;
- угол $\varphi = \text{atan2}(y, x)$ таков, что

$$\cos \varphi = \frac{x}{\sqrt{x^2 + y^2}}, \quad \sin \varphi = \frac{y}{\sqrt{x^2 + y^2}}.$$

В отличие от $\arctan(y/x)$, функция atan2 учитывает знак x и y и возвращает угол сразу в нужной четверти.

Теоретическое объяснение и вывод формул решения

1. Определение угла поворота основания γ

Точка окончания манипулятора в пространстве задана координатами (x, y, z) . Для поворота основания манипулятора рассматриваем проекцию этой точки на плоскость XU (вид сверху):

$$P = (x, y).$$

Угол γ определяется как *абсолютный угол* вектора $\overrightarrow{OP}$, отсчитанный от оси X к оси Y против часовой стрелки. Для корректного выбора четверти используется функция atan2 :

$$\gamma = \text{atan2}(y, x).$$

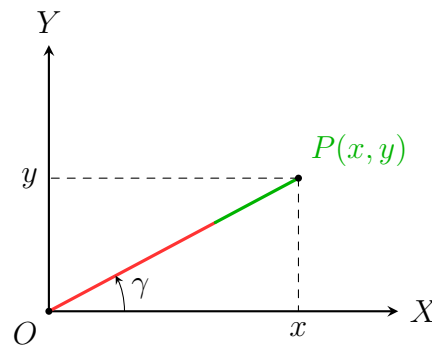


Рис. 1: Определение угла γ в плоскости XY .

Особый случай. Если $x = 0$ и $y = 0$, то точка лежит на оси Z и угол γ не определён однозначно. В этом случае можно положить $\gamma = 0$ (или сохранить предыдущее значение), так как поворот основания не влияет на достижение точки.

2. Треугольник манипулятора и вывод формулы для β

Вернёмся к нашему манипулятору в плоскости sZ . После поворота на угол γ и вычитания высоты базы мы получаем точку

$$(s, z') = (s, z - h),$$

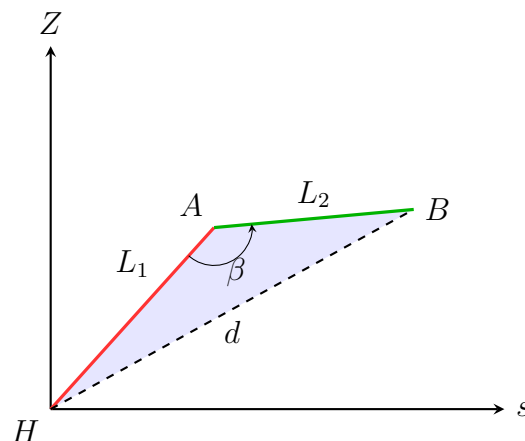
где

$$s = \sqrt{x^2 + y^2}.$$

Обозначим расстояние до рабочей точки в плоскости sZ :

$$d = \sqrt{s^2 + (z')^2}.$$

Звенья L_1 и L_2 и отрезок HV образуют треугольник HAB :



Здесь:

- L_1 — длина первого звена;
- L_2 — длина второго звена;
- d — расстояние от основания до точки $B(s, z')$;

- β — суставной угол между L_1 и L_2 (как на схемах).

Внутренний угол треугольника при A равен

$$\beta.$$

По теореме косинусов для треугольника HAB (внутренний угол при вершине A равен β):

$$d^2 = L_1^2 + L_2^2 - 2L_1L_2 \cos \beta.$$

Отсюда выражаем $\cos \beta$:

$$\cos \beta = \frac{L_1^2 + L_2^2 - d^2}{2L_1L_2}.$$

Далее, учитывая, что мы выбираем конфигурацию “локоть вниз”,

$$\beta = \arccos\left(\frac{L_1^2 + L_2^2 - d^2}{2L_1L_2}\right),$$

где β сначала вычисляется в радианах, а затем переводится в градусы.

3. Треугольник манипулятора и вывод формулы для α

Продолжим работу в плоскости sZ . После поворота на угол γ и вычитания высоты базы мы получаем точку

$$(s, z') = (s, z - h),$$

где

$$s = \sqrt{x^2 + y^2}.$$

Обозначим расстояние до рабочей точки в плоскости sZ :

$$d = \sqrt{s^2 + (z')^2}.$$

Звенья L_1 и L_2 и отрезок OB образуют треугольник OAB :

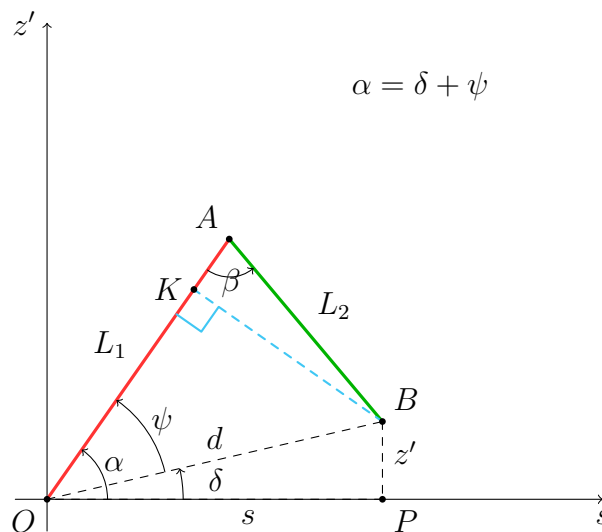


Рис. 2: Геометрия для вывода формулы угла α в плоскости sZ'

Шаг 1. Переход к плоской задаче в координатах (s, z') . Пусть требуется достичь точки с координатами (x, y, z) . Переходим к плоским координатам:

$$s = \sqrt{x^2 + y^2}, \quad z' = z - h,$$

то есть в плоскости sZ' точка

$$B = (s, z').$$

Обозначим P — проекцию точки B на ось s (как на рисунке), тогда $OP = s$, $PB = z'$. Угол наклона стороны OB относительно оси s равен

$$\delta = \text{atan2}(z', s).$$

Шаг 2. Выделяем вспомогательный угол ψ в вершине O . Рассмотрим треугольник OAB со сторонами

$$OA = L_1, \quad AB = L_2,$$

и известным углом при вершине A :

$$\angle OAB = \beta.$$

Введём угол ψ в вершине O между сторонами OB и OA (на рисунке он обозначен ψ). Тогда из геометрии (см. рисунок) угол стороны OA относительно оси s равен сумме:

$$\alpha = \delta + \psi.$$

Шаг 3. Находим ψ . Опустим из точки B перпендикуляр на *прямую*, содержащую сторону OA , и обозначим основание перпендикуляра через K (см. рисунок). Тогда

$$BK \perp OA, \quad O, K, A \text{ находятся на одной прямой}, \quad \angle OKB = 90^\circ.$$

Подчеркнём: точка K может оказаться как на отрезке OA , так и на его продолжении (выше точки A или “за” точкой O). Дальнейшие формулы корректны во всех случаях.

1) Выражаем BK через L_2 и β . В прямоугольном треугольнике ABK угол при вершине A равен β (это угол между сторонами AO и AB , как на рисунке), поэтому

$$BK = L_2 \sin \beta.$$

Эта формула не зависит от того, где расположен K на прямой OA .

2) Выражаем ориентированную величину OK (со знаком). Чтобы избежать неоднозначностей (например, когда K лежит на продолжении OA), вводим *ориентированную* величину OK как проекцию вектора $\overrightarrow{OB}$ на направление $\overrightarrow{OA}$:

$$OK = \text{proj}_{OA}(OB).$$

Из разложения $\overrightarrow{OB} = \overrightarrow{OA} + \overrightarrow{AB}$ получаем:

$$\text{proj}_{OA}(OB) = \text{proj}_{OA}(OA) + \text{proj}_{OA}(AB).$$

Здесь

$$\text{proj}_{OA}(OA) = L_1, \quad \text{proj}_{OA}(AB) = -L_2 \cos \beta,$$

поскольку угол между направлением AB и направлением AO равен β , а направление AO противоположно направлению OA . Поэтому ориентированная проекция равна

$$OK = L_1 - L_2 \cos \beta.$$

Если K оказывается “за” точкой O , то $OK < 0$; если K оказывается выше точки A , то $OK > L_1$. В обоих случаях формула остаётся верной (именно благодаря ориентации).

3) Находим ψ из прямоугольного треугольника OBK . По построению $BK \perp OA$ и $OK \parallel OA$, значит в прямоугольном треугольнике OBK угол ψ (угол между сторонами OB и OA) равен углу между сторонами OB и OK . Тогда корректно во всех четвертях:

$$\psi = \text{atan2}(BK, OK).$$

Подставляя пункты 1)–2), получаем универсальную формулу:

$$\psi = \text{atan2}\left(L_2 \sin \beta, L_1 - L_2 \cos \beta\right).$$

4) Итог для α . Так как (по рисунку) $\alpha = \delta + \psi$, где $\delta = \text{atan2}(z', s)$, окончательно:

$$\alpha = \text{atan2}(z', s) + \text{atan2}\left(L_2 \sin \beta, L_1 - L_2 \cos \beta\right).$$

Шаг 4. Итоговая формула для α . Подставляя выражение для ψ в формулу $\alpha = \delta + \psi$, получаем:

$$\alpha = \text{atan2}(z', s) + \text{atan2}\left(L_2 \sin \beta, L_1 - L_2 \cos \beta\right).$$

Комментарий. В некоторых источниках формулу пишут как $\alpha = \delta \pm \psi$ (две конфигурации). В данной задаче используется именно та конфигурация, которая показана на рисунке, поэтому берётся сумма $\alpha = \delta + \psi$.

Итоговый алгоритм решения обратной задачи

Дано: L_1, L_2, h и точка (x, y, z) .

1. Вычислить угол поворота основания:

$$\gamma = \text{atan2}(y, x).$$

Если $x = 0$ и $y = 0$, можно положить $\gamma = 0$.

2. Перейти к плоскости sZ' :

$$s = \sqrt{x^2 + y^2}, \quad z' = z - h.$$

3. Вычислить расстояние до точки в плоскости:

$$d = \sqrt{s^2 + (z')^2}.$$

Проверить достижимость:

$$|L_1 - L_2| \leq d \leq L_1 + L_2.$$

Если условие не выполняется, решения нет.

4. Найти угол β :

$$\cos \beta = \frac{L_1^2 + L_2^2 - d^2}{2L_1L_2}, \quad \beta = \arccos(\cos \beta),$$

где перед $\arccos$ следует ограничить $\cos \beta$ в диапазон $[-1, 1]$ (из-за погрешностей).

5. Найти углы

$$\delta = \operatorname{atan2}(z', s), \quad \psi = \operatorname{atan2}(L_2 \sin \beta, L_1 - L_2 \cos \beta),$$

и затем (конфигурация как на рисунке):

$$\alpha = \delta + \psi.$$

6. (При необходимости) абсолютный угол второго звена:

$$\varphi_2 = \alpha + \beta - \pi.$$

Все углы в вычислениях получаются в радианах; для вывода в градусах используется множитель $180/\pi$.

Пример решения на C++

Ниже приведён пример решения на C++.

```
#include <iostream>
#include <cmath>
#include <algorithm>

using namespace std;

int main() {
    double L1, L2, h;
    double x, y, z;
    cin >> L1 >> L2 >> h;
    cin >> x >> y >> z;

    // 1) gamma
    double gamma = 0.0;
    if (fabs(x) > 1e-12 || fabs(y) > 1e-12) {
        gamma = atan2(y, x);
    }

    // 2) s, z'
    double s = hypot(x, y);
    double zp = z - h;

    // 3) d и возможность дойти до координаты
    double d = hypot(s, zp);
    if (d > L1 + L2 + 1e-12 || d < fabs(L1 - L2) - 1e-12) {
        cout << "NO SOLUTION\n";
    }
}
```

```

    return 0;
}

// 4) beta
double cosBeta = (L1*L1 + L2*L2 - d*d) / (2.0 * L1 * L2);
cosBeta = max(-1.0, min(1.0, cosBeta));
double beta = acos(cosBeta);

// 5) alpha = delta + psi (как на рисунке в задаче)
double delta = atan2(zp, s);
double psi    = atan2(L2 * sin(beta), L1 - L2 * cos(beta));
double alpha = delta + psi;

// 6) условия определения абсолютного угла фи_2
double phi2 = alpha + beta - M_PI;

// output in degrees (по условиям задачи)
auto deg = [](double rad){ return rad * 180.0 / M_PI; };

cout.setf(ios::fixed);
cout.precision(3);
cout << "gamma=" << deg(gamma)
    << " alpha=" << deg(alpha)
    << " beta="  << deg(beta)
    // для отладки можно добавить вывод угла фи_2 << " phi2="  << deg(phi2)
    << "\n";

return 0;
}

```

Типичные ошибки

- Забыли вычесть высоту базы h и сразу искали углы по точке (s, z) вместо (s, z') .
- Перепутали порядок углов при выводе (например, сначала α , потом γ).
- Забыли перевести углы из радиан в градусы перед выводом.
- Перепутали знак в формуле теоремы косинусов для β (должно быть $L_1^2 + L_2^2 - d^2$ в числителе).
- Не ограничили $\cos \beta$ в диапазоне $[-1, 1]$ и получили NaN при acos .

Задача J. Достижимость точки и обратная кинематика

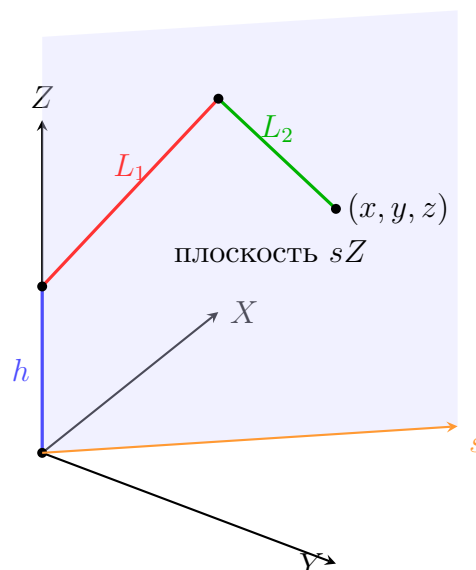
Рассматривается тот же манипулятор, что и в задаче I: основание высотой h , поворот рабочей плоскости вокруг оси Z на угол γ , затем два звена длиной L_1 и L_2 в плоскости sZ .

Углы ограничены так же, как в задаче I:

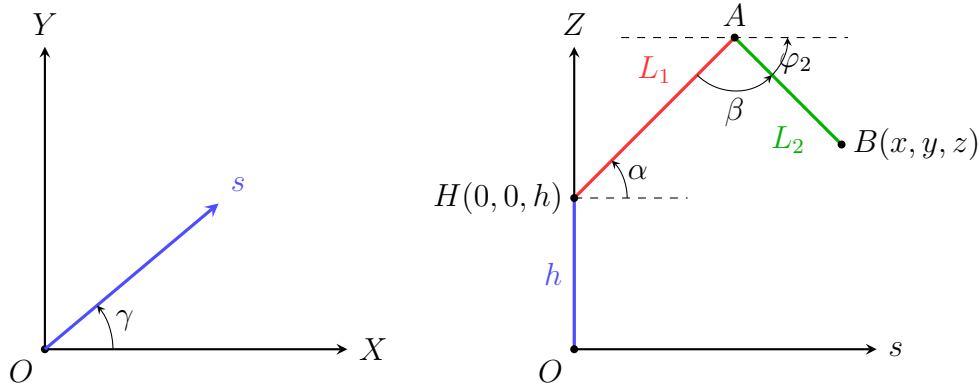
- $-90^\circ \leq \gamma \leq 90^\circ$;
- $0^\circ \leq \alpha \leq 135^\circ$;
- $20^\circ \leq \beta \leq 180^\circ$ (суставной угол между звеньями, как на эскизе).

Отличие от задачи I: теперь во входных данных могут встречаться координаты точки, которые манипулятор *не может* достичь (геометрически или из-за ограничений на углы). Нужно однозначно определить, достижима ли точка, и вывести ответ в требуемом формате.

Иллюстрация манипулятора и обозначений



Углы γ (вид сверху) и α, β (плоскость sZ)



Входные данные

Во входной поток подаются две строки.

1. В первой строке три вещественных числа:

L_1 L_2 h

2. Во второй строке три вещественных числа:

x y z

Выходные данные

Если точка (x, y, z) достижима при заданных ограничениях на углы, вывести:

YES

γ α β

Если точка недостижима, вывести только:

NO

Все углы (если выводятся) вывести в **градусах**, с точностью **3 знака** после запятой.

Решение

Формулы обратной кинематики (как выразить γ, α, β через (x, y, z) при достижимости) выведены в решении задачи I. Здесь мы **не повторяем вывод**, а используем те же обозначения и формулы, и подробно поясняем **проверку достижимости**.

1. Что значит «точка достижима»

Точка достижима тогда и только тогда, когда существует хотя бы один набор углов (γ, α, β) , удовлетворяющий одновременно:

- геометрии манипулятора (звенья длиной L_1 и L_2 в плоскости sZ действительно «дотягиваются» до точки);
- ограничениям на углы:

$$-90^\circ \leq \gamma \leq 90^\circ, \quad 0^\circ \leq \alpha \leq 135^\circ, \quad 20^\circ \leq \beta \leq 180^\circ;$$

- выбранной конфигурации (как в задаче 3, «локоть вниз», то есть берём решение с $\beta \in [0^\circ, 180^\circ]$ и далее проверяем диапазон $[20^\circ, 180^\circ]$).

Важно: поворот основания γ не влияет на *расстояние* до точки, поэтому проверка удобно делится на два независимых шага:

1. сначала проверяем достижимость точки в плоской задаче в координатах (s, z') ;
2. затем проверяем, что вычисленный γ попадает в разрешённый диапазон.

2. Геометрическая проверка достижимости в плоскости sZ'

Как и в задаче 3, переходим к плоской задаче:

$$s = \sqrt{x^2 + y^2}, \quad z' = z - h.$$

В плоскости sZ' нам нужно дотянуться двумя звеньями L_1 и L_2 от начала координат O до точки $B = (s, z')$. Обозначим расстояние:

$$d = \sqrt{s^2 + (z')^2}.$$

Геометрический смысл. Звенья L_1 и L_2 образуют треугольник OAB , где $OA = L_1$, $AB = L_2$, а $OB = d$. Такой треугольник существует **тогда и только тогда**, когда выполнено неравенство треугольника:

$$|L_1 - L_2| \leq d \leq L_1 + L_2.$$

Практическая проверка.

- Если $d > L_1 + L_2$, точка слишком далеко: NO.
- Если $d < |L_1 - L_2|$, точка слишком близко (есть «мертвая зона»): NO.
- Иначе треугольник существует, и *геометрически* точка достижима (без учёта ограничений на углы).

3. Проверка достижимости по ограничениям углов

Даже если треугольник существует, точка может быть запрещена из-за диапазонов углов. Поэтому после вычисления углов нужно проверить их диапазоны.

3.1. Угол γ

Если $(x, y) \neq (0, 0)$, то

$$\gamma = \text{atan2}(y, x).$$

Если $x = y = 0$, можно положить $\gamma = 0$ (поворот вокруг Z не влияет на точку, так как проекция на XY равна нулю).

Далее обязательно проверяем:

$$-90^\circ \leq \gamma \leq 90^\circ.$$

Если нет — NO (нужный поворот основания запрещён).

3.2. Угол β (суставной угол)

Из решения задачи 3:

$$\cos \beta = \frac{L_1^2 + L_2^2 - d^2}{2L_1L_2}, \quad \beta = \arccos(\cos \beta).$$

Если из-за округления получилось $\cos \beta$ чуть за пределами $[-1, 1]$, его можно *аккуратно* ограничить этим диапазоном. Если же треугольник не существует, то $\cos \beta$ выйдет существенно за $[-1, 1]$ — это уже обработано пунктом 2.

После вычисления проверяем ограничение задачи:

$$20^\circ \leq \beta \leq 180^\circ.$$

Если нет — NO.

3.3. Угол α

Из решения задачи 3 используем те же обозначения:

$$\delta = \text{atan2}(z', s), \quad \psi = \text{atan2}(L_2 \sin \beta, L_1 - L_2 \cos \beta), \quad \alpha = \delta + \psi.$$

После вычисления проверяем:

$$0^\circ \leq \alpha \leq 135^\circ.$$

Если нет — NO.

4. Итоговый алгоритм вывода

1. Прочитать L_1, L_2, h, x, y, z .
2. Вычислить $s = \sqrt{x^2 + y^2}$, $z' = z - h$, $d = \sqrt{s^2 + (z')^2}$.

3. Если $d < |L_1 - L_2|$ или $d > L_1 + L_2$, вывести NO.
4. Вычислить $\gamma = \text{atan2}(y, x)$ (или 0 при $x = y = 0$). Если $\gamma \notin [-90^\circ, 90^\circ]$, вывести NO.
5. Вычислить β по формуле выше. Если $\beta \notin [20^\circ, 180^\circ]$, вывести NO.
6. Вычислить $\alpha = \delta + \psi$ по формулам выше. Если $\alpha \notin [0^\circ, 135^\circ]$, вывести NO.
7. Иначе вывести YES и во второй строке γ, α, β (в градусах, 3 знака).

Замечание: если требуется нормировать углы (например, привести α в диапазон $[-180^\circ, 180^\circ]$), это делается до проверки диапазонов, но в данной постановке при корректных вычислениях $\gamma \in [-180^\circ, 180^\circ]$, $\beta \in [0^\circ, 180^\circ]$, а α получается «естественным» из $\alpha = \delta + \psi$; затем остаётся только проверка на разрешённые интервалы.

Пример решения на C++

Ниже приведён пример решения на C++.

```
#include <iostream>
#include <cmath>
#include <algorithm>

using namespace std;

int main() {
    double L1, L2, h;
    double x, y, z;
    cin >> L1 >> L2 >> h;
    cin >> x >> y >> z;

    // 1) gamma
    double gamma = 0.0;
    if (fabs(x) > 1e-12 || fabs(y) > 1e-12) {
        gamma = atan2(y, x);
    }

    // 2) s, z'
    double s = hypot(x, y);
    double zp = z - h;

    // 3) d и возможность дойти до координаты
    double d = hypot(s, zp);
    if (d > L1 + L2 + 1e-12 || d < fabs(L1 - L2) - 1e-12) {
        cout << "NO SOLUTION\n";
        return 0;
    }

    // 4) beta
```

```
double cosBeta = (L1*L1 + L2*L2 - d*d) / (2.0 * L1 * L2);
cosBeta = max(-1.0, min(1.0, cosBeta));
double beta = acos(cosBeta);

// 5) alpha = delta + psi (как на рисунке в задаче)
double delta = atan2(zp, s);
double psi = atan2(L2 * sin(beta), L1 - L2 * cos(beta));
double alpha = delta + psi;

// 6) условия определения абсолютного угла фи_2
double phi2 = alpha + beta - M_PI;

// output in degrees (по условиям задачи)
auto deg = [](double rad){ return rad * 180.0 / M_PI; };

cout.setf(ios::fixed);
cout.precision(3);
cout << "gamma=" << deg(gamma)
    << " alpha=" << deg(alpha)
    << " beta=" << deg(beta)
    // для отладки можно добавить вывод угла фи_2 << " phi2=" << deg(phi2)
    << "\n";

return 0;
}
```

Задача К. Центры цветных фигур

Входные данные

Одна строка — изображение размером **640×480** в формате **base64**. (В строке лежат байты PNG-картинки, закодированные base64.)

Справка: base64 → BGR (OpenCV)

```
1 import base64, cv2, numpy as np
2
3 b64 = input().strip()
4 img_bytes = base64.b64decode(b64)
5 arr = np.frombuffer(img_bytes, dtype=np.uint8)
6 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

Что изображено на картинке

На изображении есть от **3** до **10** геометрических фигур:

- форма: круг, овал или n -угольник ($3 \leq n \leq 8$);
- площадь каждой фигуры не меньше **300** пикселей;
- фигуры одного и того же цвета **не соприкасаются и не пересекаются**;
- фигуры разных цветов **могут пересекаться**;
- цвет фона произвольный и **не попадает целиком** ни в один из цветовых диапазонов ниже.

Цветовые диапазоны (RGB)

Цвет фигуры определяется по попаданию пикселей в диапазон (ширина каждого канала ≤ 20).

Цвет (рус.)	Color (EN)	Диапазон RGB
красный	red	$R \in [235, 255], G \in [0, 20], B \in [0, 20]$
жёлтый	yellow	$R \in [235, 255], G \in [235, 255], B \in [0, 20]$
зелёный	green	$R \in [0, 20], G \in [235, 255], B \in [0, 20]$
синий	blue	$R \in [0, 20], G \in [0, 20], B \in [235, 255]$
чёрный	black	$R \in [0, 20], G \in [0, 20], B \in [0, 20]$
белый	white	$R \in [235, 255], G \in [235, 255], B \in [235, 255]$

Что нужно сделать

Для каждой фигуры:

1. определить её цвет;
2. найти **ограничивающий** прямоугольник фигуры;

3. вывести координаты центра прямоугольника (x_c, y_c) .

Здесь x растёт вправо, y растёт вниз.

Выходные данные

Выведите по одной строке на каждую фигуру:

`<color_en> <x_center> <y_center>`

где `<color_en>` — английское имя цвета (red, yellow, green, blue, black, white).

Строки должны быть отсортированы:

- по `<color_en>` (алфавитно),
- затем по `x_center`,
- затем по `y_center`.

Пример

Ввод. В тестовом примере вход состоит из одной строки base64 (PNG-картинка):

`task_5_example_base64.txt`

Файл содержит одну строку — base64-код PNG-изображения примера.

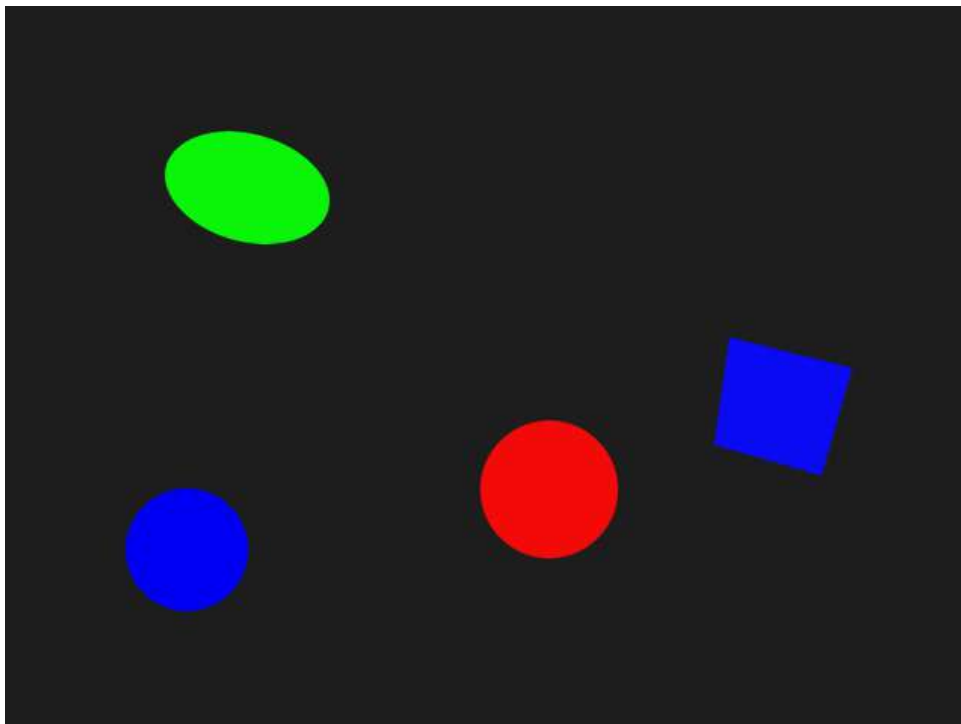


Рис. 1: Пример входного изображения

Пример ответа (для данного рисунка):

```
blue 120 360
blue 515 265
green 160 120
red 360 320
```

```
[12pt]article  
[a4paper,margin=2cm]geometry graphicx float  
fontspec [russian]babel  
minted
```

Разбор решения задачи 5

Идея решения

Основная идея задачи — **искать фигуры по цвету**, а не по яркости. Алгоритм решения состоит из следующих шагов:

1. декодировать изображение из формата base64;
2. получить BGR-изображение OpenCV;
3. для каждого цвета построить бинарную маску;
4. найти контуры фигур;
5. отбросить слишком маленькие области;
6. для каждой фигуры построить **ограничивающий прямоугольник** и найти его центр;
7. отсортировать ответ для однозначной проверки.

Шаг 1. Преобразование base64 в BGR-изображение

```
1 import base64  
2 import cv2  
3 import numpy as np  
4  
5 b64 = input().strip()  
6 img_bytes = base64.b64decode(b64)  
7 arr = np.frombuffer(img_bytes, dtype=np.uint8)  
8 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

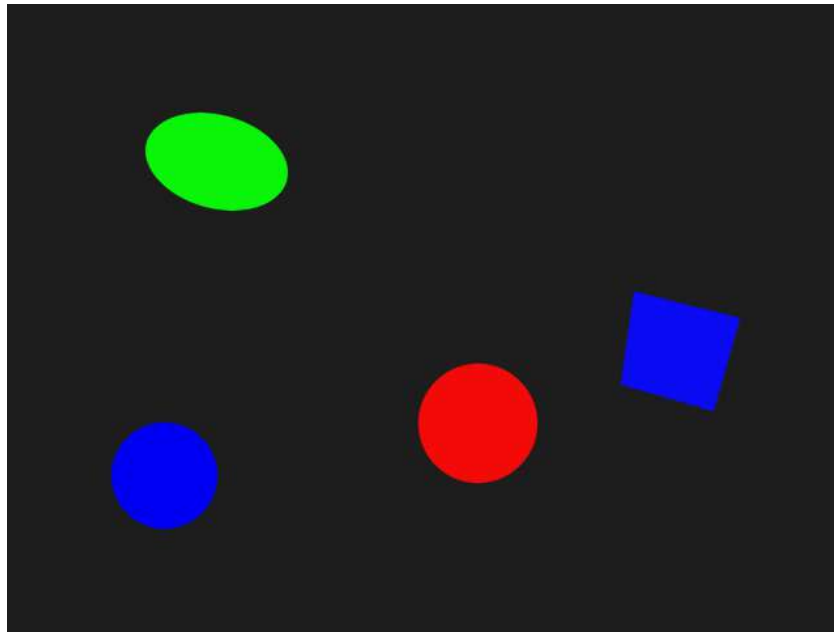


Рис. 2: Рисунок 1. Исходное изображение

Шаг 2. Почему подход grayscale + threshold может дать ошибку

Интуитивно кажется, что можно:

- перевести изображение в оттенки серого;
- применить пороговую бинаризацию;
- искать контуры на бинарном изображении.

Наивный пример кода.

```

1  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
2
3  # Пороговая бинаризация
4  _, bw = cv2.threshold(gray, 60, 255, cv2.THRESH_BINARY)
5
6  # Поиск контуров
7  contours, _ = cv2.findContours(
8      bw,
9      cv2.RETR_EXTERNAL,
10     cv2.CHAIN_APPROX_SIMPLE
11 )
12
13 for cnt in contours:
14     if cv2.contourArea(cnt) < 300:
15         continue
16     x, y, w, h = cv2.boundingRect(cnt)
17     cx = x + w // 2
18     cy = y + h // 2
    
```

Однако в некоторых тестах фон и фигура могут иметь почти одинаковую яркость. В этом случае в grayscale они **сливаются**.

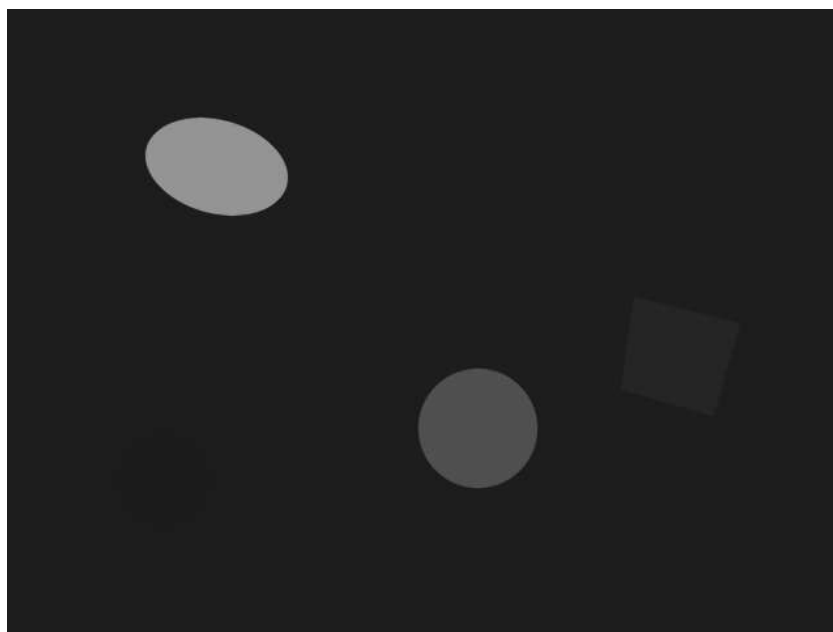


Рис. 3: Рисунок 2. Изображение в оттенках серого

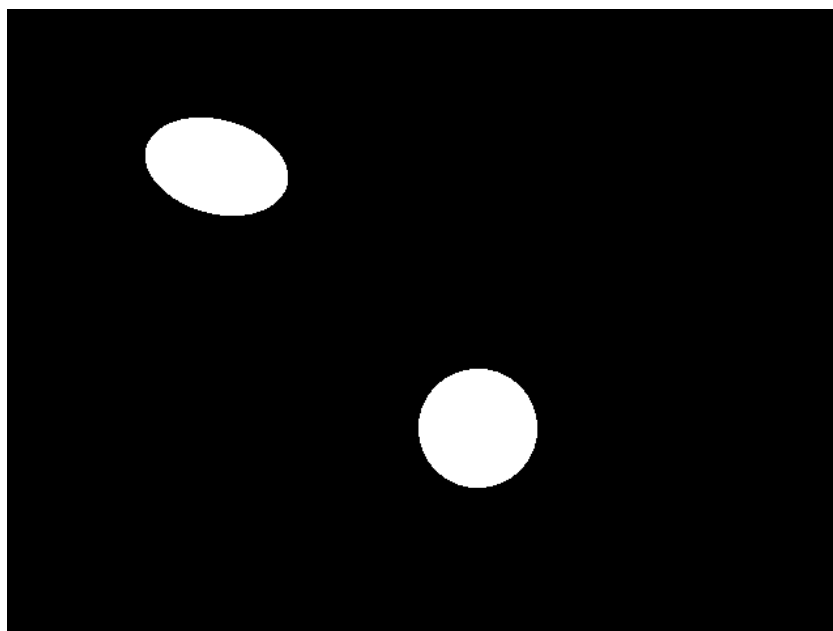


Рис. 4: Рисунок 3. Результат threshold — синяя фигура слева снизу пропала

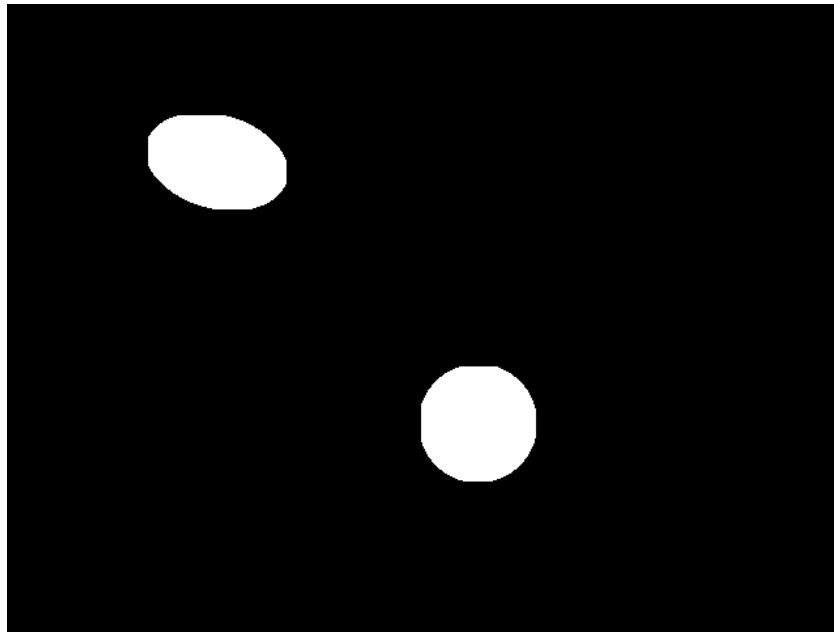


Рис. 5: Рисунок 4. Контуры по threshold — одна фигура не найдена

Шаг 3. Маски по цветам (правильный подход)

Вместо яркости используем **цветовые диапазоны**. Для каждого цвета строится бинарная маска функцией `cv2.inRange`.

Маска для красного (red).

```
1 low = np.array((0, 0, 235), dtype=np.uint8)
2 high = np.array((20, 20, 255), dtype=np.uint8)
3 mask_red = cv2.inRange(img, low, high)
```

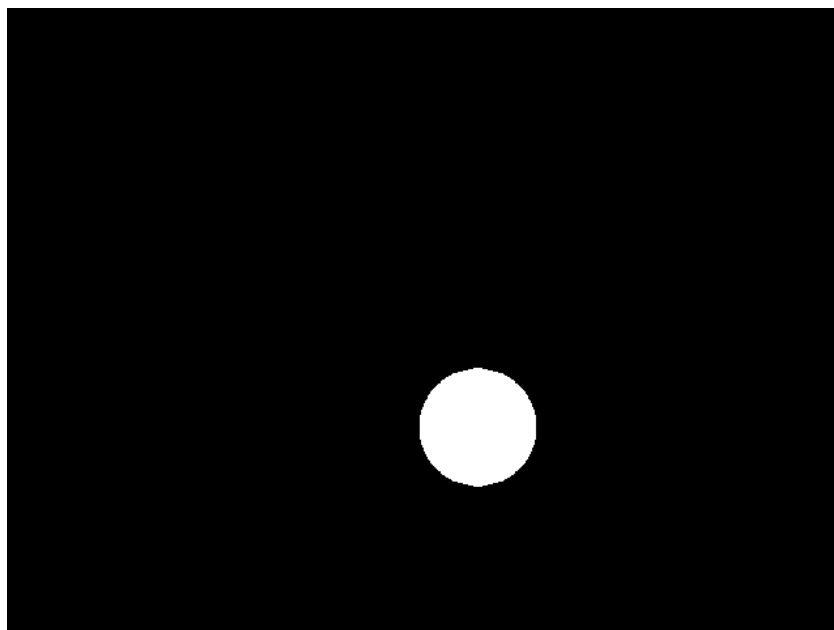


Рис. 6: Рисунок 5. Маска для красного цвета

Маска для зелёного (green).

```
1 low = np.array((0, 235, 0), dtype=np.uint8)
2 high = np.array((20, 255, 20), dtype=np.uint8)
3 mask_green = cv2.inRange(img, low, high)
```

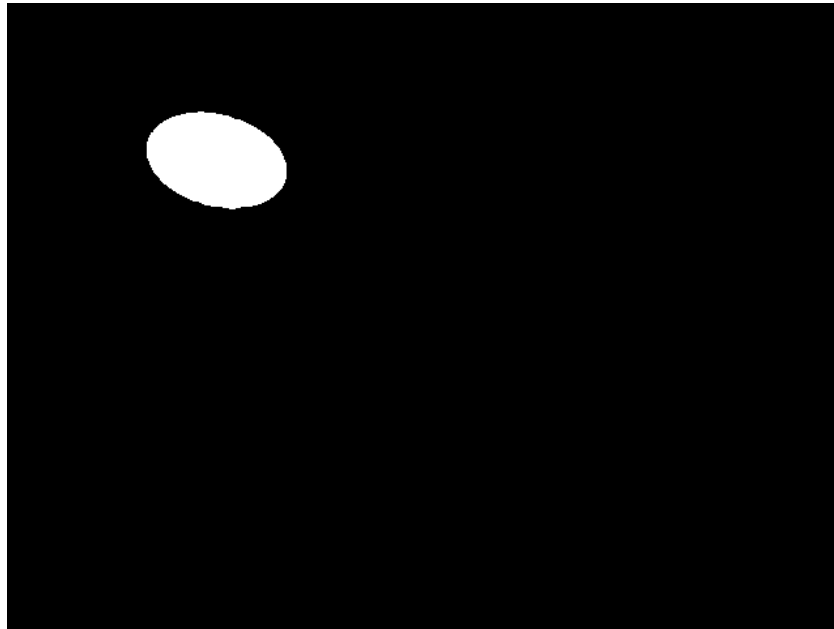


Рис. 7: Рисунок 6. Маска для зелёного цвета

Маска для синего (blue).

```
1 low = np.array((235, 0, 0), dtype=np.uint8)
2 high = np.array((255, 20, 20), dtype=np.uint8)
3 mask_blue = cv2.inRange(img, low, high)
```

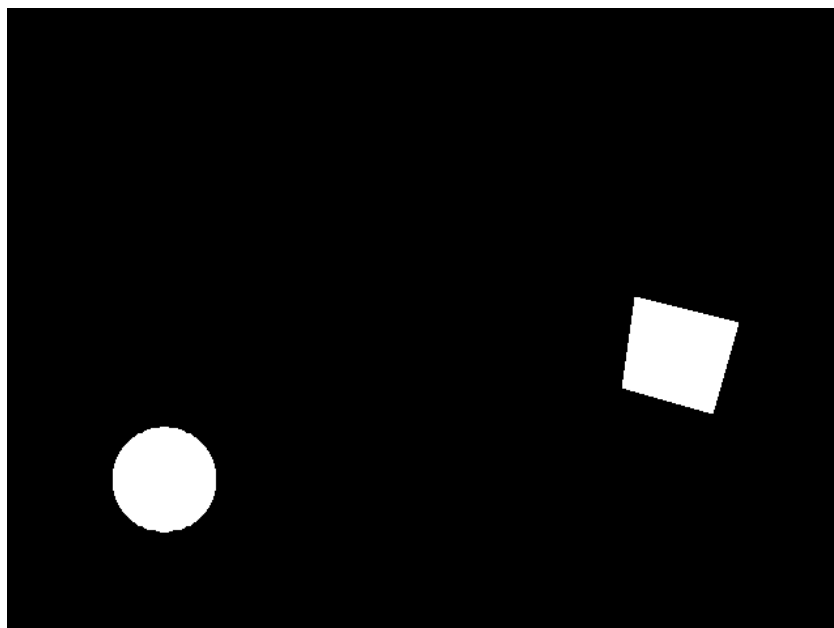


Рис. 8: Рисунок 7. Маска для синего цвета

Шаг 4. Контуры, фильтрация и ограничивающий прямоугольник

Что мы хотим сделать:

- найти отдельные фигуры на маске;
- отбросить шум;
- для каждой фигуры найти центр ограничивающего прямоугольника.

```

1 contours, _ = cv2.findContours(
2     mask,
3     cv2.RETR_EXTERNAL,
4     cv2.CHAIN_APPROX_SIMPLE
5 )
6
7 for cnt in contours:
8     if cv2.contourArea(cnt) < 300:
9         continue
10
11     x, y, w, h = cv2.boundingRect(cnt)
12     cx = x + w // 2
13     cy = y + h // 2

```

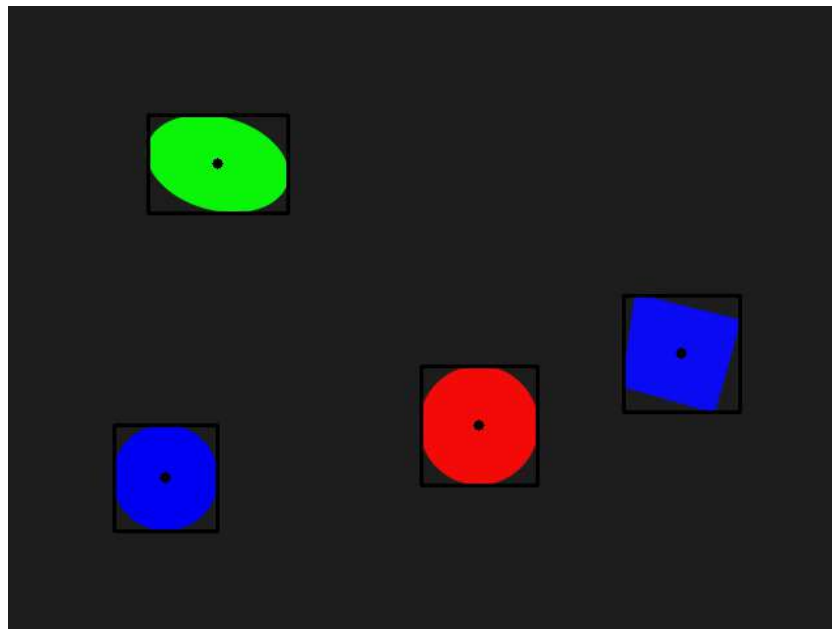


Рис. 9: Рисунок 8. Ограничивающие прямоугольники и их центры

Шаг 5. Сортировка ответа

Чтобы проверка была однозначной, строки сортируются:

- по имени цвета (англ., алфавитно);
- затем по координате x ;

- затем по координате y .

```
1 results.sort(key=lambda t: (t[0], t[1], t[2]))
```

Полный код решения

Ниже приведён полный эталонный код решения задачи:

```
1 import sys
2 import base64
3 import cv2
4 import numpy as np
5
6 # 1) Читаем base64-строку
7 b64 = sys.stdin.read().strip()
8
9 # 2) base64 -> байты PNG -> BGR изображение OpenCV
10 img_bytes = base64.b64decode(b64)
11 arr = np.frombuffer(img_bytes, dtype=np.uint8)
12 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
13
14 # 3) Диапазоны цветов (BGR!), соответствуют диапазонам RGB из условия
15 COLOR_RANGES_BGR = {
16     'red': ((0, 0, 235), (20, 20, 255)),
17     'yellow': ((0, 235, 235), (20, 255, 255)),
18     'green': ((0, 235, 0), (20, 255, 20)),
19     'blue': ((235, 0, 0), (255, 20, 20)),
20     'black': ((0, 0, 0), (20, 20, 20)),
21     'white': ((235, 235, 235), (255, 255, 255)),
22 }
23
24 results = [] # (color_en, x_center, y_center)
25
26 for color_name, (low, high) in COLOR_RANGES_BGR.items():
27     # 4) Маска цвета
28     mask = cv2.inRange(img, np.array(low, np.uint8), np.array(high, np.uint8))
29
30     # 5) Очистка мелкого шума
31     kernel = np.ones((3, 3), np.uint8)
32     mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel, iterations=1)
33
34     # 6) Контуры объектов данного цвета
35     contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
36     ↪ cv2.CHAIN_APPROX_SIMPLE)
37
38     for cnt in contours:
39         if cv2.contourArea(cnt) < 300:
40             continue
41
42         # 7) Ограничивающий прямоугольник и центр
43         x, y, w, h = cv2.boundingRect(cnt)
44         cx = x + w // 2
```

```
44         cy = y + h // 2
45         results.append((color_name, cx, cy))
46
47     # 8) Сортировка: color -> x -> y
48     results.sort(key=lambda t: (t[0], t[1], t[2]))
49
50     # 9) Вывод
51     for color, x, y in results:
52         print(color, x, y)
```

Задача L. Отбор фигур по форме и площади

краткое описание задачи

В данной задаче вам предстоит научиться отличать фигуры по форме и размеру друг от друга. На изображении которое вы будете получать на вход есть цветные **выпуклые** фигуры разных типов:

- возможные формы: треугольники, квадраты, прямоугольники, круги, овалы и выпуклые многоугольники (5...8 сторон);
- фигуры могут быть повернуты вокруг центра (квадрат не обязан быть параллелен краям картинки);
- фигуры одного цвета не соприкасаются и не пересекаются; фигуры разных цветов могут пересекаться;
- площадь каждой фигуры не меньше 300 пикселей;
- фон произвольный и не попадает целиком ни в один из цветовых диапазонов (как в задаче 5).

Допуски на размеры фигур

Для распознавания форм вводится допуск 2%:

- если у “круга” полуоси (после аппроксимации эллипсом) отличаются больше чем на 2%, то это считается **овал**;
- если у “квадрата” стороны (по минимальному ограничивающему прямоугольнику) отличаются больше чем на 2%, то это считается **прямоугольник** и не подходит под запрос `square`.

Цветовые диапазоны (RGB)

Цвет фигуры определяется по попаданию пикселей в диапазон (ширина каждого канала ≤ 20):

- red: $R \in [235, 255]$, $G \in [0, 20]$, $B \in [0, 20]$
- yellow: $R \in [235, 255]$, $G \in [235, 255]$, $B \in [0, 20]$
- green: $R \in [0, 20]$, $G \in [235, 255]$, $B \in [0, 20]$
- blue: $R \in [0, 20]$, $G \in [0, 20]$, $B \in [235, 255]$
- black: $R \in [0, 20]$, $G \in [0, 20]$, $B \in [0, 20]$
- white: $R \in [235, 255]$, $G \in [235, 255]$, $B \in [235, 255]$

Что нужно сделать

Найти на изображении все фигуры требуемой формы, у которых площадь находится в диапазоне $A_{\min}..A_{\max}$. Для каждой подходящей фигуры:

- определить её цвет (в выводе использовать английское имя цвета);
- построить **ограничивающий прямоугольник**;
- вывести координаты центра этого прямоугольника (x_c, y_c) .

Входные данные

Вход состоит из трёх строк:

1. тип фигуры: `triangle`, `oval`, `circle`, `square` или `polygon n` (где $5 \leq n \leq 8$);
2. два целых числа $A_{\min}$ $A_{\max}$ — диапазон площади в пикселях;
3. строка `base64` — изображение 640×480 (PNG), закодированное `base64`.

Справка: `base64` → BGR (OpenCV)

```
1 import base64, cv2, numpy as np
2
3 b64 = input().strip()
4 img_bytes = base64.b64decode(b64)
5 arr = np.frombuffer(img_bytes, dtype=np.uint8)
6 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

Выходные данные

В первой строке выведите число K — количество подходящих фигур.
Далее выведите K строк:

`<color_en> <x_center> <y_center>`

Строки должны быть отсортированы:

- по `<color_en>` (алфавитно),
- затем по `x_center`,
- затем по `y_center`.

Пример

Ввод. В тестовом примере входные данные находятся в файлах:

`circle`
`300 1000`

тестовое изображение в формате `base64`. файл `task_6_example_base64.txt` содержит одну строку `base64`.

т.е. необходимо найти круги площадью от 4500 до 6500 пикселей

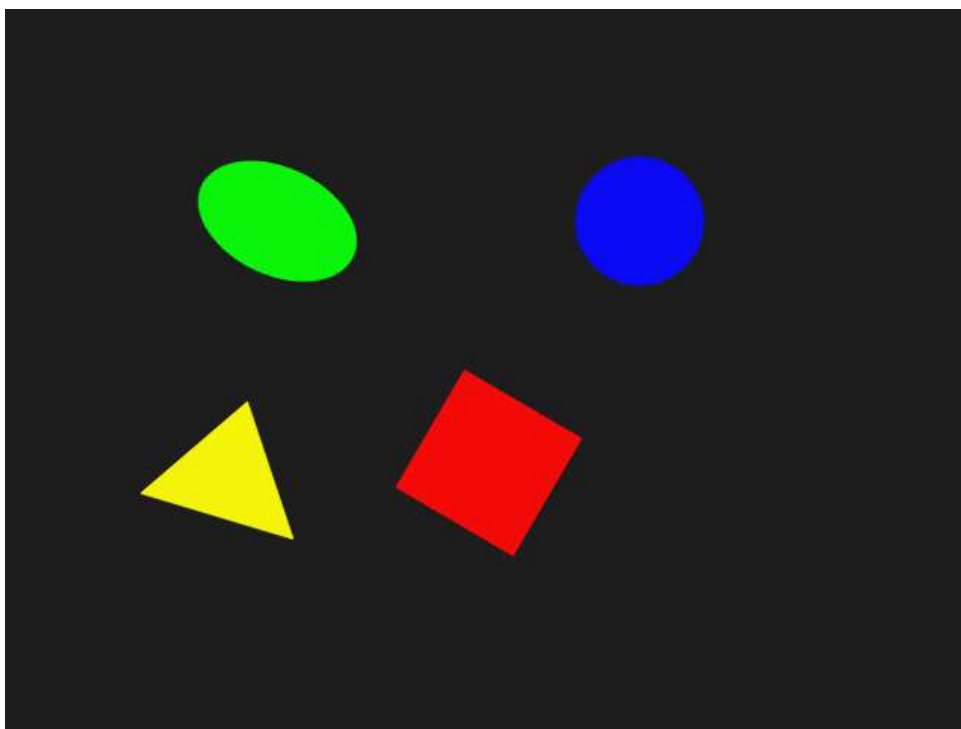


Рис. 1: Пример входного изображения

Вывод. Для данного примера корректный ответ:

1
2 blue 420 140
3

Разбор решения задачи L

Идея

Мы должны выбрать фигуры:

- заданной формы (triangle / circle / oval / square / polygon n),
- с площадью в диапазоне $[A_{min}, A_{max}]$,

а затем для каждой подходящей фигуры вывести её цвет и центр **ограничивающего** прямоугольника.

Шаг 1. Чтение входа и декодирование base64

```

1 shape_line = input().strip()
2 a_min, a_max = map(float, input().split())
3 b64 = input().strip()
4
5 img_bytes = base64.b64decode(b64)
6 arr = np.frombuffer(img_bytes, dtype=np.uint8)
7 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR

```

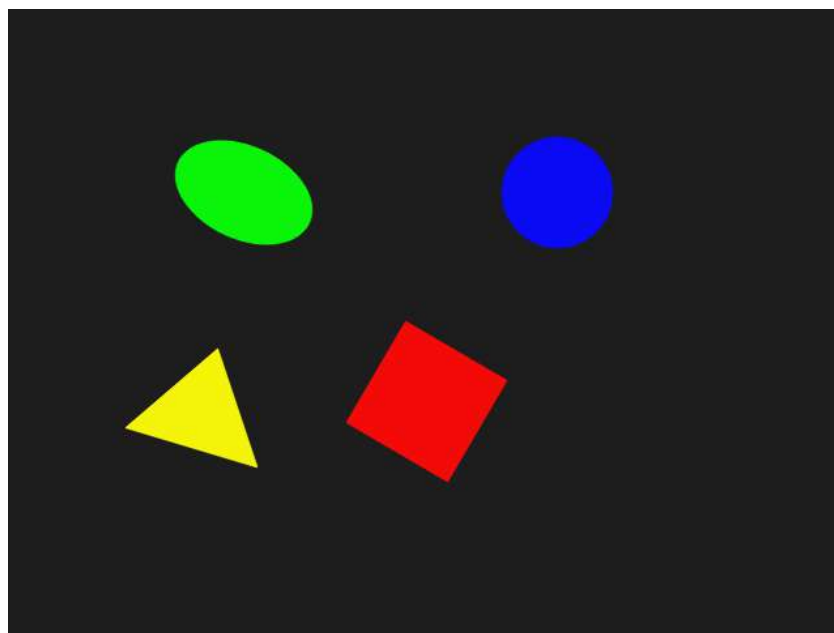


Рис. 2: Исходное изображение

Шаг 2. Почему grayscale + threshold может ошибаться

Фон может быть подобран так, что в grayscale он близок по яркости к части фигур. Поэтому решение “в сером + threshold” может пропускать объекты.

```

1 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
2 _, bw = cv2.threshold(gray, 60, 255, cv2.THRESH_BINARY)
3 contours, _ = cv2.findContours(bw, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

```

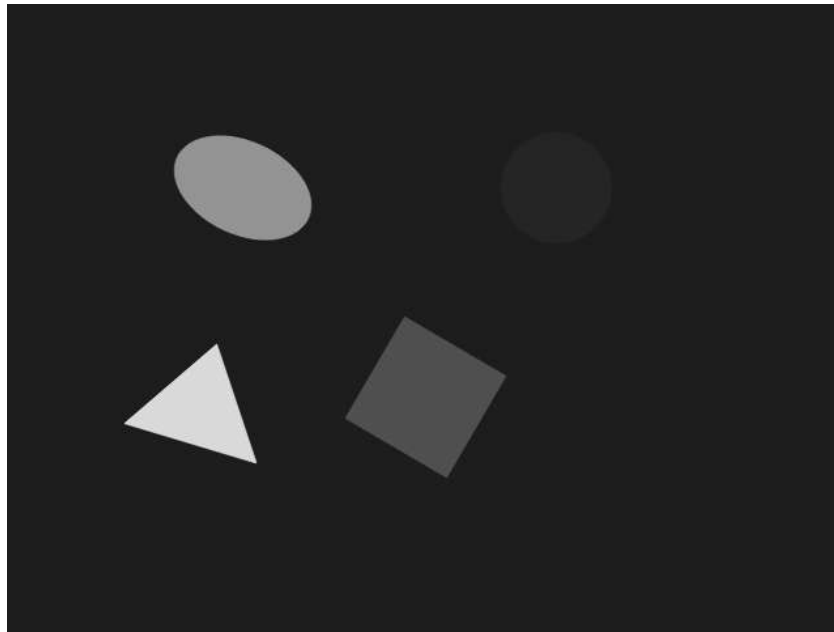


Рис. 3: Grayscale (пример коварной яркости)

Шаг 3. Маска по цвету и контуры

Перебираем все цвета, строим маску `inRange` и ищем контуры.

```
1 mask = cv2.inRange(img, low_bgr, high_bgr)
2 contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

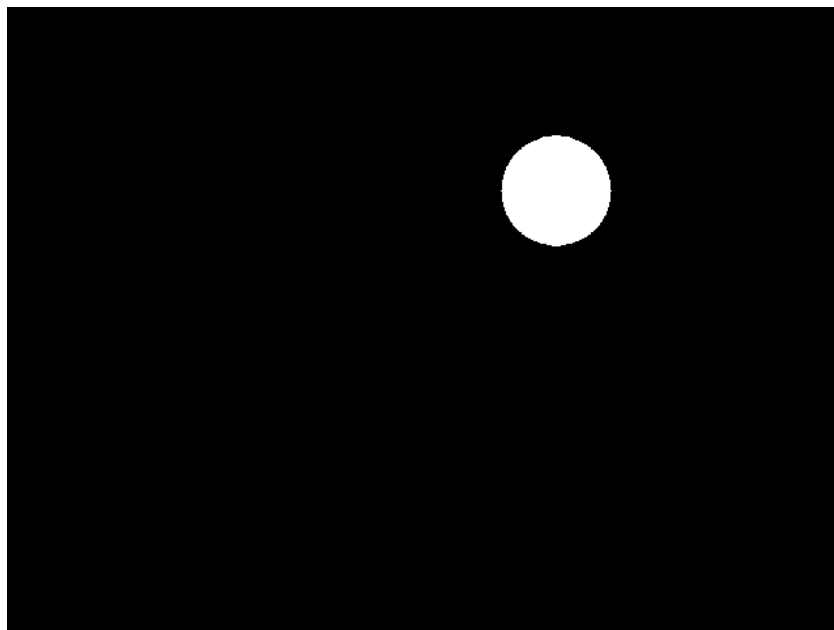


Рис. 4: Пример маски для синего цвета

Шаг 4. Форма + площадь + центр прямоугольника

Что делаем: для каждого найденного контура:

- считаем площадь и отбрасываем контуры вне диапазона $[A_{min}, A_{max}]$;
- определяем тип фигуры в **строго определённом порядке**:
 - **сначала** проверяем, является ли фигура кругом или овалом;
 - **только если фигура не круглая**, определяем многоугольную форму по числу вершин `approxPolyDP`;
 - квадрат и прямоугольник различаем по `minAreaRect` с допустимым отклонением 2%.
- если форма и площадь подходят, берём центр **ограничивающего прямоугольника**.

Почему важен такой порядок. Если сначала аппроксимировать контур многоугольником, то даже круг часто аппроксимируется 6–10 вершинами (из-за пиксельной границы и сглаживания) и ошибочно распознаётся как многоугольник. Поэтому в решении **сначала** проверяется, является ли фигура достаточно круглой, и только потом анализируется число вершин.

Коэффициент округлости (circularity). Для предварительной проверки “круглости” используется коэффициент

$$\text{circularity} = \frac{4\pi S}{P^2},$$

где S — площадь контура, а P — его периметр.

- для идеального круга значение равно 1;
- из-за растеризации и дискретной границы даже настоящий круг на изображении даёт заметно меньшее значение;
- поэтому `circularity` используется **не как критерий точности**, а только как предварительный фильтр “фигура достаточно круглая”.

В задаче используется порог **0.85**. Он позволяет надёжно отделить круги и овалы от угловатых фигур (треугольников, квадратов и многоугольников), но не связан напрямую с допуском 2%.

Важно про допуск 2%. Указанный в условии допуск 2% применяется **только**:

- при различении `circle` и `oval` (по разнице полуосей эллипса);
- при различении `square` и `rectangle` (по разнице сторон минимального прямоугольника).

Он не относится к коэффициенту округлости.

```

1 area = cv2.contourArea(cnt)
2 if area < A_min or area > A_max:
3     continue
4 
```

```

5  peri = cv2.arcLength(cnt, True)
6  if peri <= 1e-6:
7      continue
8
9  # 1) Предварительная проверка на круг / овал
10 circularity = 4.0 * math.pi * area / (peri * peri)
11
12 if circularity >= 0.85 and len(cnt) >= 5:
13     (_, _), (MA, ma), _ = cv2.fitEllipse(cnt)
14     ratio = abs(MA - ma) / max(MA, ma)
15     shape = "circle" if ratio <= 0.02 else "oval"
16 else:
17     # 2) Угловатые фигуры: по числу вершин
18     approx = cv2.approxPolyDP(cnt, 0.02 * peri, True)
19     k = len(approx)
20
21     if k == 3:
22         shape = "triangle"
23     elif k == 4:
24         rect = cv2.minAreaRect(cnt)
25         (w, h) = rect[1]
26         ratio = abs(w - h) / max(w, h)
27         shape = "square" if ratio <= 0.02 else "rectangle"
28     else:
29         shape = f"polygon_{k}"
30
31 # Центр ограничивающего прямоугольника
32 x, y, w, h = cv2.boundingRect(cnt)
33 cx = x + w // 2
34 cy = y + h // 2

```

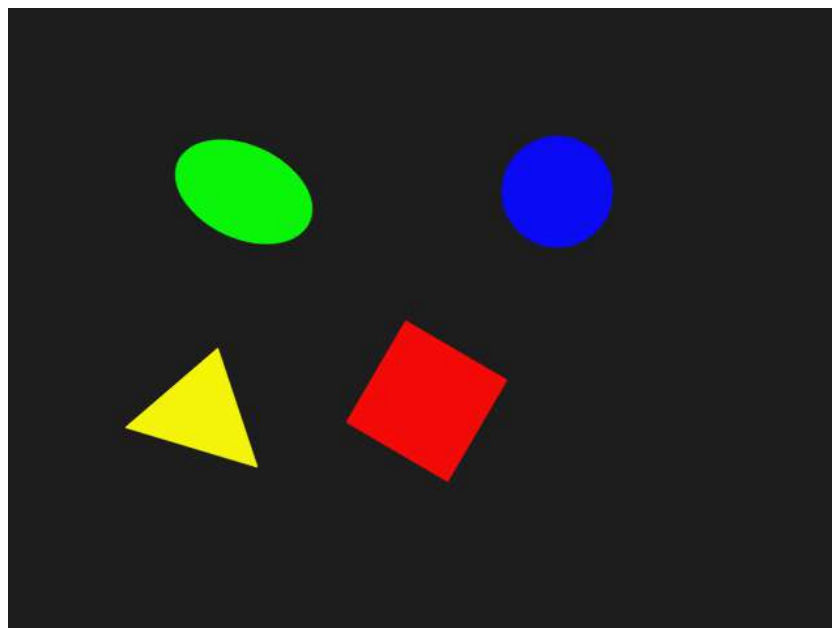


Рис. 5: Подходящие фигуры и их ограничивающие прямоугольники

Шаг 5. Сортировка и вывод

Сортируем по (color, x, y), затем печатаем количество и строки.

```
1 results.sort(key=lambda t: (t[0], t[1], t[2]))
2 print(len(results))
3 for color, x, y in results:
4     print(color, x, y)
```

Полный код решения

```
1 import sys
2 import base64
3 import cv2
4 import numpy as np
5 import math
6
7 # ----- Ввод -----
8 shape_line = sys.stdin.readline().strip()
9 a_min, a_max = map(float, sys.stdin.readline().split())
10 b64 = sys.stdin.readline().strip()
11
12 # ----- base64 -> BGR -----
13 img_bytes = base64.b64decode(b64)
14 arr = np.frombuffer(img_bytes, dtype=np.uint8)
15 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
16
17 # ----- Цветовые диапазоны (BGR) -----
18 COLOR_RANGES_BGR = {
19     'red': ((0, 0, 235), (20, 20, 255)),
20     'yellow': ((0, 235, 235), (20, 255, 255)),
21     'green': ((0, 235, 0), (20, 255, 20)),
22     'blue': ((235, 0, 0), (255, 20, 20)),
23     'black': ((0, 0, 0), (20, 20, 20)),
24     'white': ((235, 235, 235), (255, 255, 255)),
25 }
26
27 def target_name_from_input(s: str) -> str:
28     s = s.strip()
29     if s.startswith('polygon'):
30         n = int(s.split()[1])
31         return f'polygon_{n}'
32     return s # triangle / square / circle / oval
33
34 def classify_shape(cnt, tol=0.02):
35     # Классификация формы по контуру.
36     #
37     # ВАЖНО: сначала проверяем "круглые" фигуры (круг/овал), чтобы круги не
38     # превращались в polygon_k из-за approxPolyDP.
39     #
40     # tol=0.02 означает допуск 2%:
41     # - круг: оси эллипса (fitEllipse) почти равны
```

```

42     # - квадрат: стороны minAreaRect почти равны
43     area = cv2.contourArea(cnt)
44     if area <= 1:
45         return None
46
47     peri = cv2.arcLength(cnt, True)
48     if peri <= 1e-6:
49         return None
50
51     # 1) Проверка на "круглость" по circularity
52     # circularity = 1 у идеального круга, меньше у многоугольников/угловатых фигур
53     circularity = 4.0 * math.pi * area / (peri * peri)
54
55     # Порог подобран так, чтобы круги/овалы почти всегда попадали сюда,
56     # а треугольники/квадраты/полигоны - нет.
57     if circularity >= 0.82 and len(cnt) >= 5:
58         (_, _), (MA, ma), _ = cv2.fitEllipse(cnt)
59         if MA == 0 or ma == 0:
60             return 'oval'
61         ratio = abs(MA - ma) / max(MA, ma)
62         return 'circle' if ratio <= tol else 'oval'
63
64     # 2) Угловатые фигуры: по числу вершин аппроксимации
65     approx = cv2.approxPolyDP(cnt, 0.02 * peri, True)
66     k = len(approx)
67
68     if k == 3:
69         return 'triangle'
70     if k == 4:
71         rect = cv2.minAreaRect(cnt)
72         (w, h) = rect[1]
73         if w == 0 or h == 0:
74             return 'rectangle'
75         ratio = abs(w - h) / max(w, h)
76         return 'square' if ratio <= tol else 'rectangle'
77
78     if 5 <= k <= 8:
79         return f'polygon_{k}'
80
81     return f'polygon_{k}'
82
83
84     peri = cv2.arcLength(cnt, True)
85     approx = cv2.approxPolyDP(cnt, 0.02 * peri, True)
86     k = len(approx)
87
88     if 3 <= k <= 10:
89         if k == 3:
90             return 'triangle'
91         if k == 4:
92             rect = cv2.minAreaRect(cnt)
93             (w, h) = rect[1]
94             if w == 0 or h == 0:

```

```

95         return 'rectangle'
96         ratio = abs(w - h) / max(w, h)
97         return 'square' if ratio <= tol else 'rectangle'
98     if 5 <= k <= 8:
99         return f'polygon_{k}'
100    return f'polygon_{k}'
101
102    if len(cnt) >= 5:
103        (_, _), (MA, ma), _ = cv2.fitEllipse(cnt)
104        if MA == 0 or ma == 0:
105            return 'oval'
106        ratio = abs(MA - ma) / max(MA, ma)
107        return 'circle' if ratio <= tol else 'oval'
108
109    return 'oval'
110
111    target = target_name_from_input(shape_line)
112
113    results = []
114    for color_name, (low, high) in COLOR_RANGES_BGR.items():
115        mask = cv2.inRange(img, np.array(low, np.uint8), np.array(high, np.uint8))
116        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((3, 3), np.uint8),
117                               ↪ iterations=1)
118
119        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
120                                     ↪ cv2.CHAIN_APPROX_SIMPLE)
121
122        for cnt in contours:
123            area = cv2.contourArea(cnt)
124            if area < 300:
125                continue
126
127            shape_name = classify_shape(cnt, tol=0.02)
128            if shape_name != target:
129                continue
130
131            if not (a_min <= area <= a_max):
132                continue
133
134            x, y, w, h = cv2.boundingRect(cnt)
135            cx = x + w // 2
136            cy = y + h // 2
137            results.append((color_name, cx, cy))
138
139    results.sort(key=lambda t: (t[0], t[1], t[2]))
140
141    print(len(results))
142    for color, x, y in results:
143        print(color, x, y)

```

Задача М. Перспективное преобразование: восстановить квадрат

Что видит робот?

На изображении, которое получает робот в камеру в перспективе виден квадрат, заданный четырьмя кругами одинакового цвета:

- круги находятся в вершинах квадрата;
- из-за перспективы в кадре видна трапеция;
- верхняя и нижняя стороны квадрата **параллельны нижней границе изображения**. Для определения размера квадрата считайте что нижняя сторона квадрата не искажена перспективой.

Дополнительные условия. В некоторых тестах между кругами могут присутствовать соединяющие линии того же цвета. Эти линии:

- не ближе одного диаметра круга к его краю;
- не влияют на правильный ответ;
- не обязательны для использования в решении.

Также могут присутствовать 2/3/5 кругов отличного от трапеции цвета.

Параметры

- диаметр кругов: от 10 до 40 пикселей;
- цвет кругов — один из: `red`, `yellow`, `green`, `blue`, `black`, `white`.

Цветовые диапазоны (RGB)

- red: $R \in [235, 255]$, $G \in [0, 20]$, $B \in [0, 20]$
- yellow: $R \in [235, 255]$, $G \in [235, 255]$, $B \in [0, 20]$
- green: $R \in [0, 20]$, $G \in [235, 255]$, $B \in [0, 20]$
- blue: $R \in [0, 20]$, $G \in [0, 20]$, $B \in [235, 255]$
- black: $R \in [0, 20]$, $G \in [0, 20]$, $B \in [0, 20]$
- white: $R \in [235, 255]$, $G \in [235, 255]$, $B \in [235, 255]$

Что нужно сделать

Нужно восстановить изображение квадрата (выполнить перспективное преобразование) и вывести **длину стороны квадрата** в пикселях.

Входные данные

Во входных данных одна строка — изображение 640×480 (PNG), закодированное base64.

Справка: base64 → BGR (OpenCV)

```
1 import base64, cv2, numpy as np
2
3 b64 = input().strip()
4 img_bytes = base64.b64decode(b64)
5 arr = np.frombuffer(img_bytes, dtype=np.uint8)
6 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

Выходные данные

Одно целое число — длина стороны квадрата в пикселях.

Пример

Ввод. В тестовом примере base64 находится в файле:

task_7_example_base64.txt

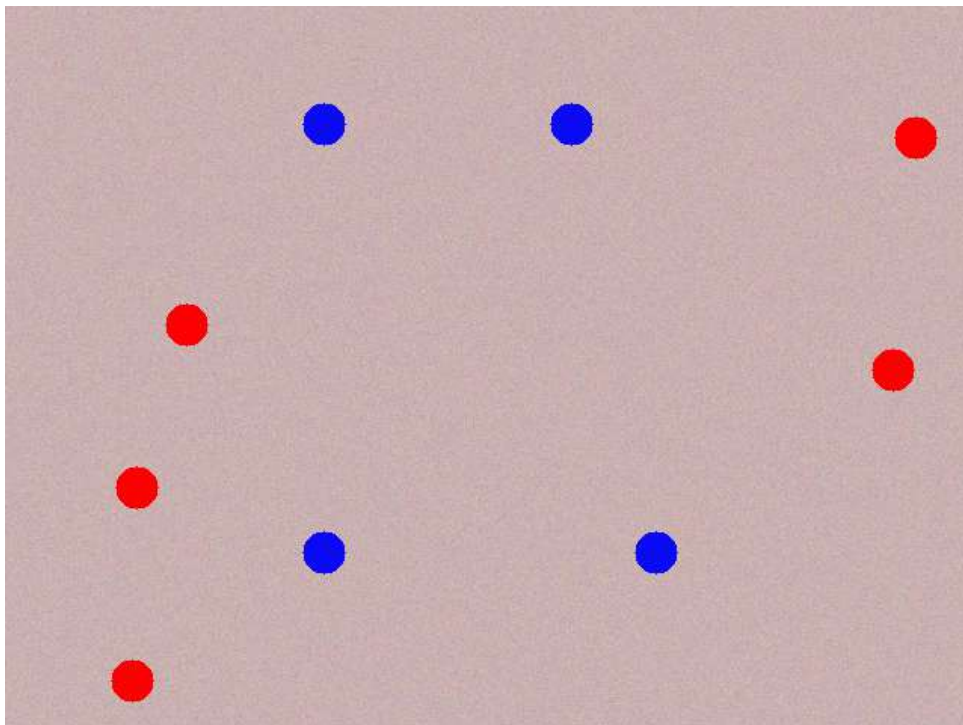


Рис. 1: Пример входного изображения

Вывод. Для данного примера:

```
1 220
```

Разбор решения задачи М (перспективное преобразование)

Идея

На изображении есть ровно **4 одинаковых круга одного цвета** — это углы квадрата, который снят **в перспективе**. В кадре квадрат выглядит как **трапеция**. Наша цель:

1. найти цвет, для которого на маске получается **ровно 4 круга**;
2. взять их центры и упорядочить как TL, TR, BR, BL;
3. восстановить квадрат (перспективой) и вывести длину его стороны.

Шаг 1. Преобразуем base64 в изображение OpenCV (BGR)

В тестах изображение передаётся одной строкой base64. Декодируем её и читаем PNG через `cv2.imdecode`.

```
1 import base64, cv2, numpy as np
2
3 b64 = input().strip()
4 img_bytes = base64.b64decode(b64)
5 arr = np.frombuffer(img_bytes, dtype=np.uint8)
6 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

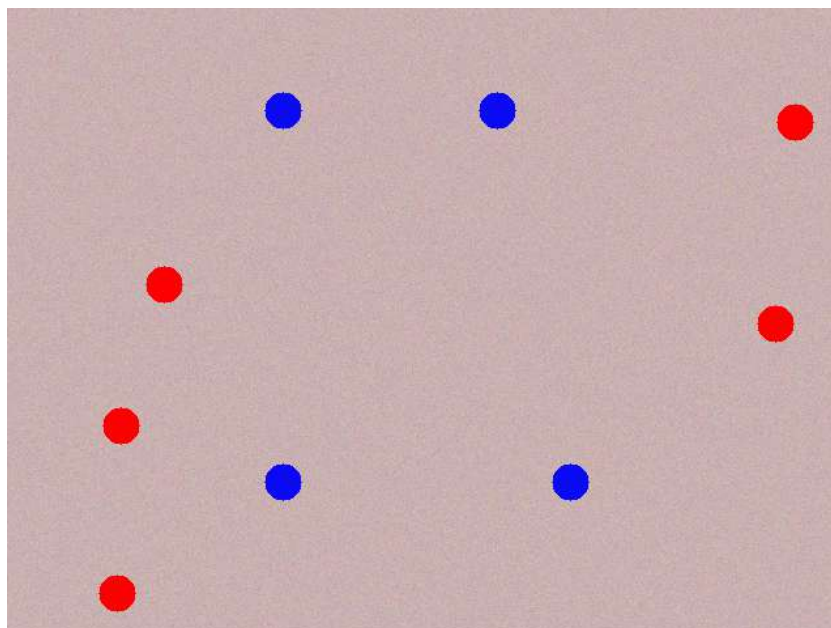


Рис. 2: Пример входного изображения

Шаг 2. Перебор цветов и выбор “правильной” маски

На картинке мы **не знаем заранее**, какого цвета четыре угловых круга. Зато цветов всего 6 (red/yellow/green/blue/black/white), и для каждого цвета нам заранее дан **узкий диапазон BGR** (ширина по каждому каналу не больше 20).

Идея очень простая:

1. Перебираем все цвета из таблицы.
2. Для каждого цвета строим **маску**: белое (255) — пиксели нужного цвета, чёрное (0) — всё остальное.
3. На маске пытаемся найти **ровно 4 угловых круга** функцией `find_four_circles(mask)`.
4. Если получилось — значит, мы нашли нужный цвет, дальше работаем только с ним.

Почему важно требование “ровно 4”. В усложнённых тестах могут быть **дополнительные круги другого цвета** (например, 5 красных “приманок”). Если брать “первый попавшийся цвет, где есть круги”, то легко ошибиться. Поэтому мы принимаем цвет только если на его маске функция возвращает **именно 4 угловых круга** (или 4 больших + мелкие артефакты, см. Шаг 3).

```
1 chosen = None
2 for color_name, (lo, hi) in COLOR_RANGES_BGR.items():
3     mask = cv2.inRange(img, np.array(lo, np.uint8), np.array(hi, np.uint8))
4
5     pts = find_four_circles(mask)    # либо None, либо 4 точки (x, y)
6
7     if pts is not None:
8         chosen = pts
9         chosen_color = color_name
10        break
11
12 if chosen is None:
13     print(0)    # на всякий случай (в корректных тестах не должно случаться)
14     sys.exit(0)
```

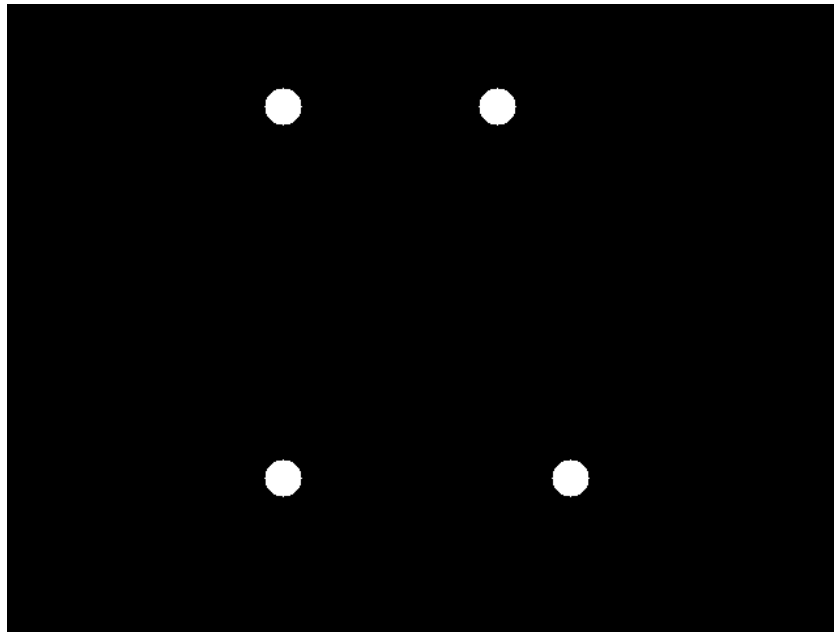


Рис. 3: Маска для **правильного** цвета (blue): видно 4 белых круга

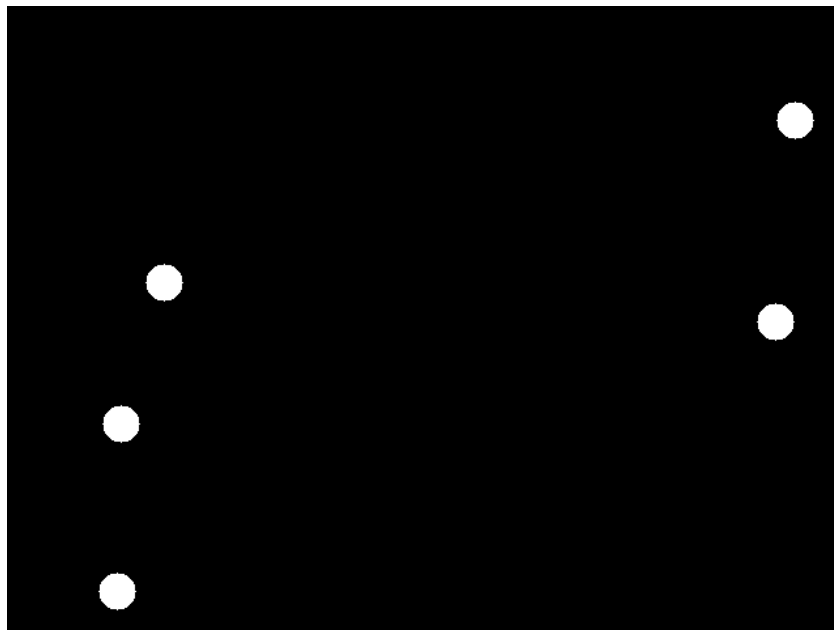


Рис. 4: Маска для **неверного** цвета (red): видно **5** кругов, поэтому этот цвет отбрасываем

Шаг 3. Функция `find_four_circles`: как найти именно 4 угловых круга

На вход функция получает **маску** выбранного цвета (чёрно-белое изображение). Далее мы делаем два действия:

1. Находим **все контуры** на маске: `findContours`.
2. Отбираем среди них те, которые действительно похожи на круги.

Фильтры “похоже на круг”. Для каждого контура считаем:

- **Площадь** `area`. Маленькие контуры отбрасываем как шум.
- **Округлость** (`circularity`):

$$\text{circularity} = \frac{4\pi S}{P^2},$$

где S — площадь, P — периметр. Для идеального круга значение равно 1. Если значение слишком маленькое, контур слишком “угловатый” — это не круг.

- **Диаметр** круга через `minEnclosingCircle`: в задаче диаметр должен быть в диапазоне 10..40 пикселей.
- **Заполненность** (`fill`): если контур похож на “ободок” или тонкую линию, площадь будет слишком маленькой по сравнению с площадью описанного круга.

```

1 cand = [] # сюда соберём кандидатов: (r, x, y)
2
3 mask2 = cv2.morphologyEx(mask, cv2.MORPH_OPEN,
4                             np.ones((3, 3), np.uint8), iterations=1)
5 cnts, _ = cv2.findContours(mask2, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
6
7 for c in cnts:
8     area = cv2.contourArea(c)
9     if area < 50:
10         continue
11
12     peri = cv2.arcLength(c, True)
13     if peri <= 1e-6:
14         continue
15
16     circularity = 4.0 * math.pi * area / (peri * peri)
17     if circularity < 0.82:
18         continue
19
20     (x, y), r = cv2.minEnclosingCircle(c)
21     d = 2 * r
22     if d < 10 or d > 40:
23         continue
24
25     fill = area / (math.pi * r * r + 1e-6)
26     if fill < 0.60:
27         continue
28
29     cand.append((r, x, y))

```

Почему “ровно 4” иногда не хватает. Если включён режим с линиями, на концах линий могут появляться **маленькие круглые артефакты** на маске (похожие на круги, но заметно меньше). Тогда кандидатов может стать больше 4. С другой стороны, бывают и **приманки** — например, 5 одинаковых кругов неверного цвета. Нам нужно:

- **принять** случай “4 больших + несколько маленьких”;
- **отбросить** случай “5 одинаковых кругов”.

Поэтому финальная логика такая:

1. Если кандидатов меньше 4 — это не наш цвет.
2. Если кандидатов ровно 4 — отлично, возвращаем их центры.
3. Если кандидатов больше 4 — берём 4 самых больших и проверяем:
 - четыре больших должны быть **примерно одного размера**;
 - пятый по размеру должен быть **заметно меньше** (иначе это “5 одинаковых”, и цвет неверный).

```

1  if len(cand) < 4:
2      return None
3
4  cand.sort(key=lambda t: t[0], reverse=True)  # по радиусу, большие впереди
5
6  if len(cand) == 4:
7      chosen = cand
8  else:
9      top4 = cand[:4]
10     r_max = top4[0][0]
11     r_min = top4[-1][0]
12     r5 = cand[4][0]
13
14     # 4 круга должны быть похожи по размеру
15     if (r_max / max(r_min, 1e-6)) > 1.25:
16         return None
17
18     # 5-й должен быть значительно меньше (иначе это "приманка" из 5 кругов)
19     if r5 >= 0.80 * r_min:
20         return None
21
22     chosen = top4
23
24 pts = np.array([[x, y] for (r, x, y) in chosen], dtype=np.float32)
25 return pts

```

Мини-проверка от “вырождения”. Если четыре точки почти лежат на одной линии (по X или по Y), то перспективное преобразование становится плохим. Поэтому в коде есть короткая проверка, что разброс по X и по Y не слишком мал.

Шаг 4. Упорядочиваем углы трапеции

После `find_four_circles` у нас есть 4 точки-центра, но в произвольном порядке. Так как по условию **верхняя и нижняя стороны параллельны низу изображения**, можно просто:

- отсортировать точки по y (сначала верхние, потом нижние);
- внутри каждой пары отсортировать по x (левее/правее).

```

1 def order_points_trapezoid(pts):
2     # TL, TR, BR, BL
3     pts = pts.tolist()
4     pts.sort(key=lambda p: (p[1], p[0]))    # по y, потом по x
5     top = sorted(pts[:2], key=lambda p: p[0])
6     bot = sorted(pts[2:], key=lambda p: p[0])
7     tl, tr = top[0], top[1]
8     bl, br = bot[0], bot[1]
9     return np.array([tl, tr, br, bl], dtype=np.float32)

```

Шаг 5. Восстанавливаем квадрат и считаем сторону

Что хотим сделать: взять трапецию (углы TL, TR, BR, BL) и “распрямить” её в квадрат. Это делает перспективное преобразование: `getPerspectiveTransform + warpPerspective`.

Почему на warped видны только части кругов. Мы строим квадратного размера выходное изображение $L \times L$. Часть кругов может оказаться **за границей** этого квадрата и “обрежется” — это нормально. Нам важны только **центры** кругов, которые уже найдены на исходном изображении.

Что такое сторона квадрата в ответе. В ответ мы выводим длину стороны исходного квадрата в пикселях. После упорядочивания точек это просто расстояние между нижними углами BL и BR:

$$L = \sqrt{(x_{BR} - x_{BL})^2 + (y_{BR} - y_{BL})^2}.$$

```

1 quad = order_points_trapezoid(chosen)    # TL, TR, BR, BL
2
3 # сторона: расстояние между нижними маркерами
4 br = quad[2]
5 bl = quad[3]
6 side = math.hypot(br[0] - bl[0], br[1] - bl[1])
7
8 print(int(round(side)))

```

```

1 # (для иллюстрации в генераторе)
2 dst = np.array([[0,0],[L,0],[L,L],[0,L]], np.float32)
3 M = cv2.getPerspectiveTransform(quad, dst)
4 warped = cv2.warpPerspective(img, M, (L, L))

```

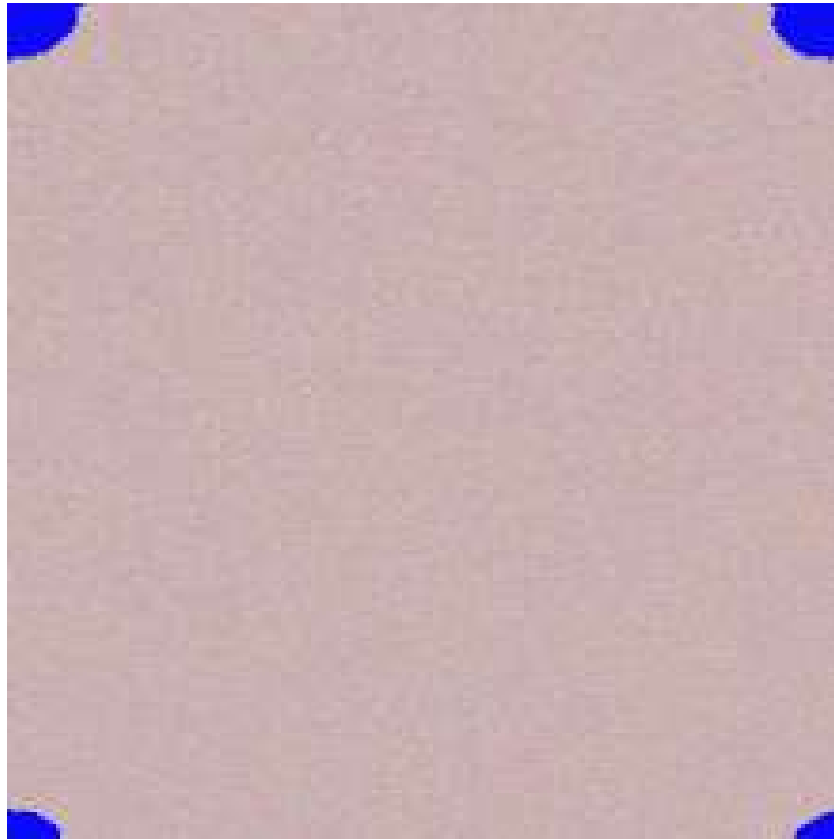


Рис. 5: Картинка после распрямления (части кругов могут обрезаться)

Полный код эталонного решения

```

1  import sys
2  import base64
3  import cv2
4  import numpy as np
5  import math
6
7  # Ввод: одна строка base64
8  b64 = sys.stdin.readline().strip()
9
10 # base64 -> BGR
11 img_bytes = base64.b64decode(b64)
12 arr = np.frombuffer(img_bytes, dtype=np.uint8)
13 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
14
15 # Диапазоны цвета (BGR)
16 COLOR_RANGES_BGR = {
17     'red': ((0, 0, 235), (20, 20, 255)),
18     'yellow': ((0, 235, 235), (20, 255, 255)),
19     'green': ((0, 235, 0), (20, 255, 20)),
20     'blue': ((235, 0, 0), (255, 20, 20)),
21     'black': ((0, 0, 0), (20, 20, 20)),
22     'white': ((235, 235, 235), (255, 255, 255)),
23 }
24 def find_four_circles(mask):

```

"""

Находит 4 угловых круга нужного цвета.

В тестах могут встречаться:

- соединительные линии (могут давать маленькие "кружочки"-артефакты на концах);
- дополнительные кружки других цветов (2/3/5 шт.) как усложнение.

Поэтому логика такая:

- 1) ищем все "похожие на круг" контуры в маске;
- 2) если нашли ровно 4 - отлично;
- 3) если нашли больше 4, то принимаем ТОЛЬКО если 4 самых больших заметно больше остальных (т.е. остальные - это мелкие артефакты).
Это защищает от случая "5 одинаковых кружков" (дистрактор).

"""

Убираем одиночный шум

```
mask2 = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((3, 3), np.uint8),
    iterations=1)
```

```
cnts, _ = cv2.findContours(mask2, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

```
cand = [] # (r, x, y)
```

```
for c in cnts:
```

```
    area = cv2.contourArea(c)
```

```
    if area < 50:
```

```
        continue
```

```
    peri = cv2.arcLength(c, True)
```

```
    if peri <= 1e-6:
```

```
        continue
```

```
    circularity = 4.0 * math.pi * area / (peri * peri)
```

```
    # 0.82 устойчиво для толстых линий/антиалиасинга
```

```
    if circularity < 0.82:
```

```
        continue
```

```
    (x, y), r = cv2.minEnclosingCircle(c)
```

```
    d = 2.0 * r
```

```
    if d < 10 or d > 40:
```

```
        continue
```

```
    fill = area / (math.pi * r * r + 1e-6)
```

```
    if fill < 0.60:
```

```
        continue
```

```
    cand.append((r, x, y))
```

```
if len(cand) < 4:
```

```
    return None
```

Сортируем по радиусу (сначала большие)

```
cand.sort(key=lambda t: t[0], reverse=True)
```

```

76     if len(cand) == 4:
77         chosen = cand
78     else:
79         # Берём 4 самых больших и проверяем, что они заметно больше остальных
80         top4 = cand[:4]
81         r4 = top4[-1][0]
82         r5 = cand[4][0]
83
84         # 1) четыре больших должны быть примерно одного размера
85         r_max = top4[0][0]
86         r_min = top4[-1][0]
87         if r_min <= 1e-6 or (r_max / r_min) > 1.25:
88             return None
89
90         # 2) пятый (и дальше) должны быть существенно меньше (артефакты)
91         # Если 5-й почти такого же размера, скорее всего это дистрактор (5
92         → одинаковых кружков).
93         if r5 >= 0.80 * r4:
94             return None
95
96         chosen = top4
97
98     pts = np.array([[x, y] for (_, x, y) in chosen], dtype=np.float32)
99
100    # Проверка на "не выродилось ли в линию"
101    if float(pts[:, 0].max() - pts[:, 0].min()) < 5:
102        return None
103    if float(pts[:, 1].max() - pts[:, 1].min()) < 5:
104        return None
105
106    return pts
107
108    def order_points_trapezoid(pts):
109        """TL, TR, BR, BL. Так как верх/низ горизонтальны, сортируем по y, потом по
110        → x."""
111        pts = pts.tolist()
112        pts.sort(key=lambda p: (p[1], p[0]))
113        top = sorted(pts[:2], key=lambda p: p[0])
114        bot = sorted(pts[2:], key=lambda p: p[0])
115        tl, tr = top[0], top[1]
116        bl, br = bot[0], bot[1]
117        return np.array([tl, tr, br, bl], dtype=np.float32)
118
119    chosen = None
120    for _, (lo, hi) in COLOR_RANGES_BGR.items():
121        mask = cv2.inRange(img, np.array(lo, np.uint8), np.array(hi, np.uint8))
122        pts = find_four_circles(mask)
123        if pts is not None:
124            chosen = pts
125            break
126
127    if chosen is None:
128        print(0)

```

```
127     sys.exit(0)
128
129 quad = order_points_trapezoid(chosen)
130
131 # Ответ: расстояние между нижними маркерами (BR и BL).
132 br = quad[2]
133 bl = quad[3]
134 side = math.hypot(br[0] - bl[0], br[1] - bl[1])
135
136 print(int(round(side)))
```

Задача N. Ходы коня на шахматной доске

Условие

На изображении показана шахматная доска 8×8 в перспективе. Позиция коня отмечена **красным кругом** внутри одной клетки.

Нужно определить, **в какие клетки конь может попасть за один ход**, и вывести эти клетки в шахматной нотации.

Как устроено изображение

- Доска состоит из **жёлтых** и **чёрных** клеток (шахматный порядок).
- Вокруг доски есть **чёрная рамка** толщиной **20%** от размера клетки.
- Доска видна **в перспективе**, при этом **нижняя сторона доски параллельна** нижней стороне изображения.
- Размер доски на кадре: от **25%** до **85%** ширины кадра.
- Фон почти белый: каждый канал RGB в диапазоне $[235, 255]$.

Система координат клеток

Клетки обозначаются как в шахматах: **буква + цифра**.

- Буквы **A..H** идут **слева направо** вдоль **нижней** стороны доски.
- Цифры **1..8** идут **снизу вверх**.

То есть **A1** — нижняя левая клетка, **H8** — верхняя правая.

Формат входных данных

Одна строка: изображение PNG, закодированное в Base64.

Как прочитать Base64 и получить OpenCV-изображение (BGR):

```
1 import base64
2 import numpy as np
3 import cv2
4
5 b64 = input().strip()
6 data = base64.b64decode(b64)
7 arr = np.frombuffer(data, dtype=np.uint8)
8 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

Формат выходных данных

- В первой строке: число K — сколько клеток доступно за один ход.
- Далее K строк: названия клеток (например, A3, H6).

Порядок вывода: отсортировать по букве (A..H), затем по цифре (1..8).

Пример

Ввод. Base64-строка для примера находится в файле `task_8_example_base64.txt`.

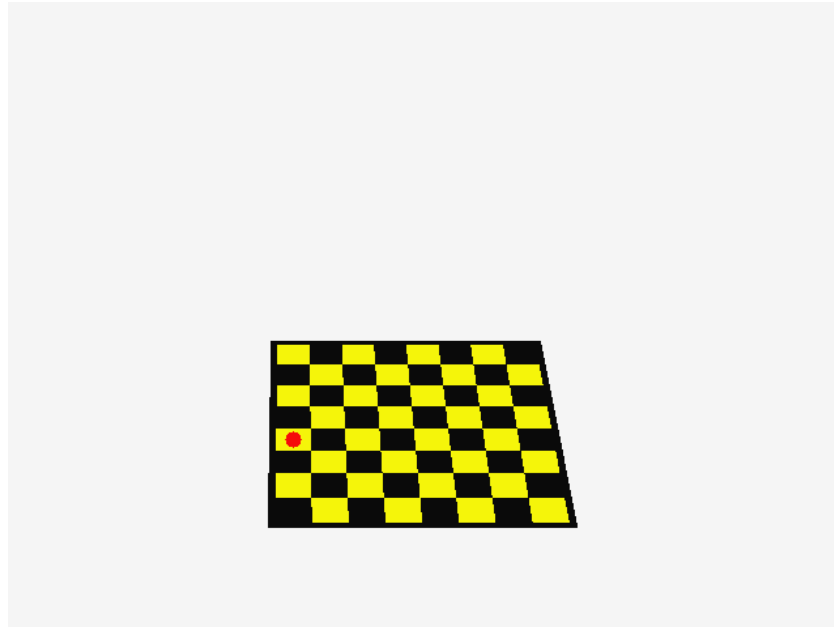


Рис. 1: Пример входного изображения

Вывод.

```
1 4
2 B2
3 B6
4 C3
5 C5
```

Задача N. Разбор решения

Идея решения

Доска в перспективе выглядит как трапеция. Мы превратим её в «вид сверху», чтобы клетки стали ровной сеткой 8×8 :

1. найдём внешнюю чёрную рамку доски;
2. выровняем доску через `warpPerspective`;
3. найдём красный круг и определим клетку;
4. посчитаем ходы коня и выведем клетки A1..H8.

Шаг 1. Декодируем Base64

```
1 import base64, numpy as np, cv2
2
3 b64 = input().strip()
4 data = base64.b64decode(b64)
5 arr = np.frombuffer(data, dtype=np.uint8)
6 img = cv2.imdecode(arr, cv2.IMREAD_COLOR) # BGR
```

Шаг 2. Находим чёрную рамку

Что такое маска и зачем она нужна

Маска — это чёрно-белая картинка, где:

- **белый** (значение 255) означает “это пиксели нужного цвета”;
- **чёрный** (значение 0) означает “это всё остальное”.

Рамка чёрная и толстая (20% от клетки), поэтому выделяем «почти чёрные» пиксели:

```
1 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
2 mask = cv2.inRange(gray, 0, 70)
3 mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE,
4                          np.ones((5,5), np.uint8), iterations=2)
```

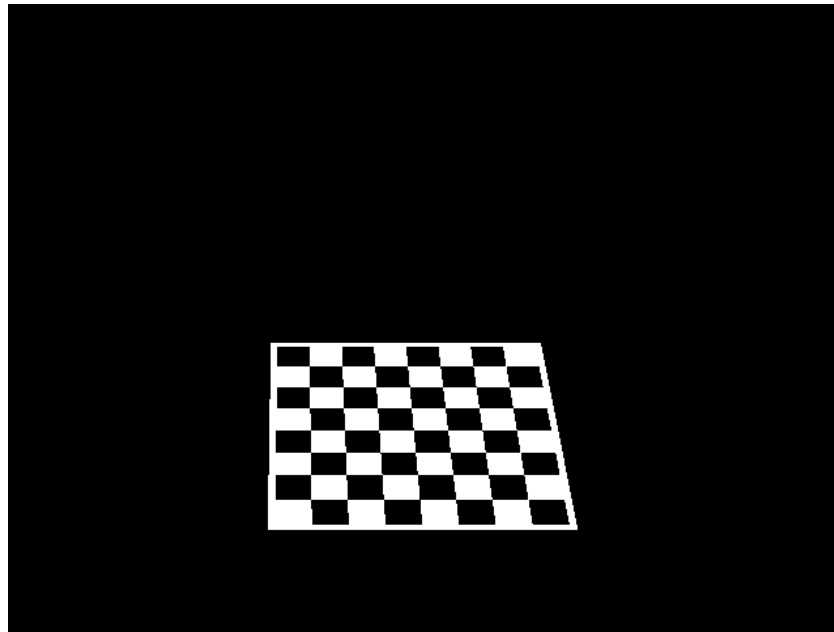


Рис. 2: Маска рамки

Шаг 3. Ищем 4 угла и выравниваем доску

Ищем большой контур, аппроксимируем его многоугольником и берём вариант с 4 вершинами. После этого преобразуем трапецию в квадрат $L \times L$, размер L выберем удобный для нас. Поскольку сказано что доска может занимать от **25%** до **85%** ширины кадра то выберем L таким образом чтобы она была около **60%** кадра и при этом была кратна размерам клеток и рамки вокруг доски:

```

1 cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
2 cnts = sorted(cnts, key=cv2.contourArea, reverse=True)
3
4 for c in cnts[:10]:
5     peri = cv2.arcLength(c, True)
6     approx = cv2.approxPolyDP(c, 0.02 * peri, True)
7     if len(approx) == 4:
8         quad = approx.reshape(4, 2)
9         break
10
11 dst = np.array([[0,0],[L-1,0],[L-1,L-1],[0,L-1]], np.float32)
12 M = cv2.getPerspectiveTransform(quad, dst)
13 warped = cv2.warpPerspective(img, M, (L, L), flags=cv2.INTER_NEAREST)

```

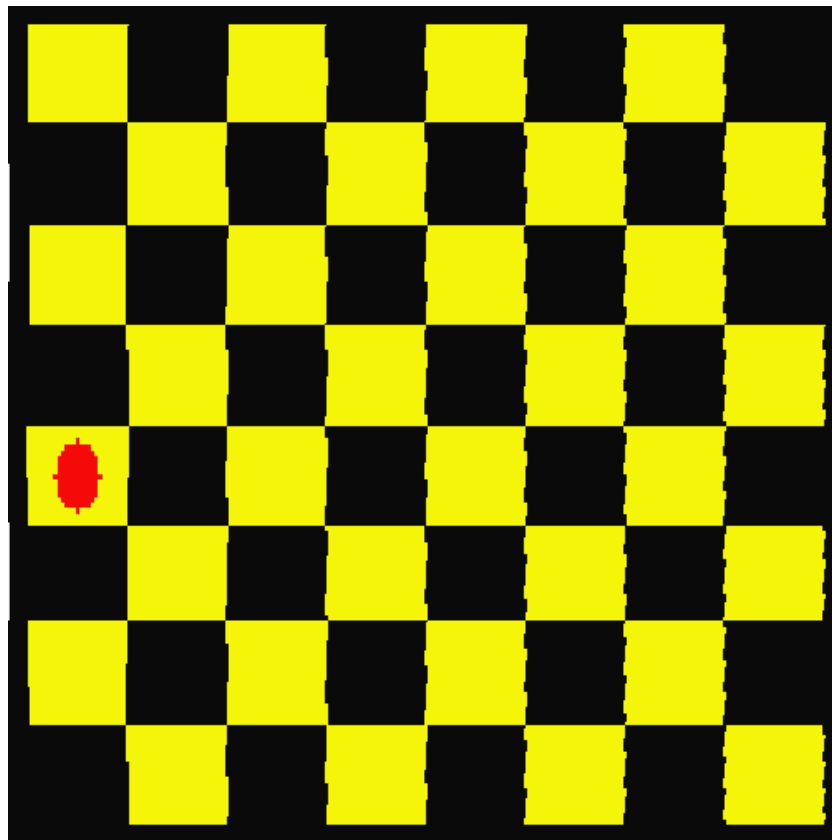


Рис. 3: Выровненная доска

Шаг 4. Находим красный круг (коня) на выровненной доске

После шага 3 у нас есть изображение `warped`: доска уже “как сверху”, клетки ровные, и красный круг тоже ищется намного проще.

На этом шаге мы сделаем маску красного цвета: оставим только те пиксели, которые похожи на красный цвет нашего маркера.

4.1. Выделяем красный цвет через `inRange`

В OpenCV цвет хранится как **BGR** (синий, зелёный, красный), а не RGB. Наш красный круг в генераторе имеет цвет примерно $(B, G, R) = (10, 10, 245)$. Поэтому мы задаём диапазон:

$$B \in [0, 20], \quad G \in [0, 20], \quad R \in [235, 255].$$

Команда `cv2.inRange` проверяет каждый пиксель: если он попал в диапазон — в маске будет 255, иначе 0.

```
1 # warped - выровненная картинка доски
2 mask_red = cv2.inRange(warped, (0,0,235), (20,20,255))
```

4.2. Зачем нужна морфология (и что такое OPEN)

Иногда на маске появляются маленькие “шумы”: одиночные белые точки или крошечные пятнышки. Например, из-за сжатия PNG, переходов на границах или небольших артефактов.

Чтобы убрать такие мелкие мусорные точки, используют морфологические операции. Операция **OPEN** = **эрозия** + **дилатация**:

- эрозия “съедает” маленькие белые точки;
- дилатация возвращает размер большой фигуры обратно.

То есть OPEN оставляет большую фигуру (наш круг), но убирает мелкий мусор.

```
1 kernel = np.ones((3,3), np.uint8)
2 mask_red = cv2.morphologyEx(mask_red, cv2.MORPH_OPEN, kernel, iterations=1)
```

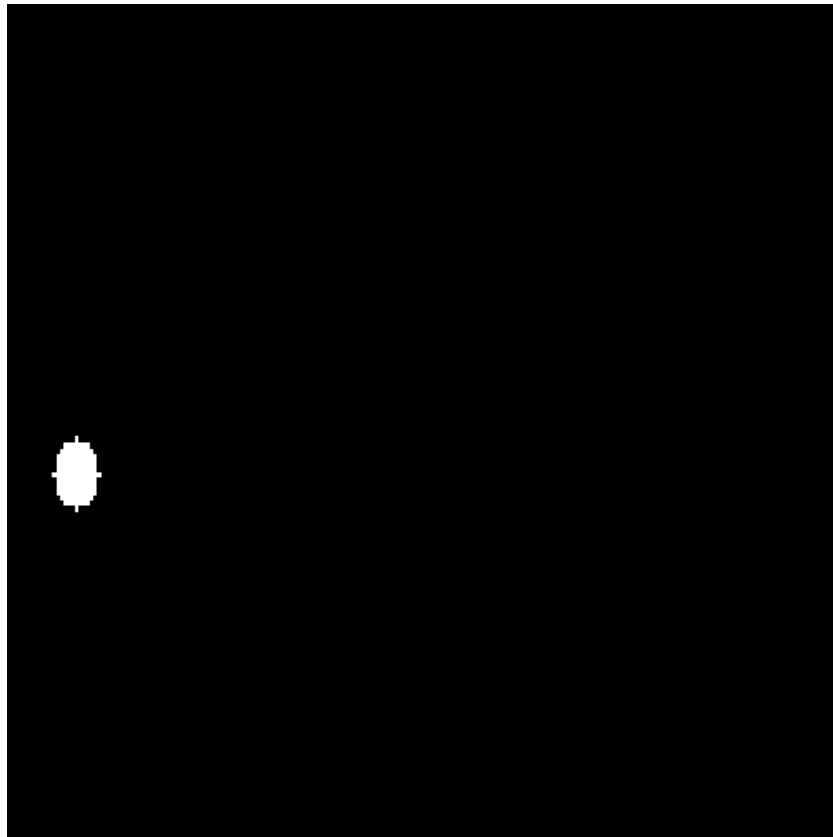


Рис. 4: Маска красного маркера на выровненной доске

4.3. Находим контур красного круга

Когда есть маска, мы ищем на ней **контур** — границы белых областей. Функция `findContours` возвращает список контуров.

Почему мы берём **самый большой**? Потому что красный круг — заметная фигура, а случайные шумы (если вдруг остались) обычно имеют очень маленькую площадь.

```
1 cnts, _ = cv2.findContours(mask_red, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
2 c = max(cnts, key=cv2.contourArea)
```

4.4. Находим центр круга через моменты

Нам нужен не радиус и не идеальная окружность — нам важно понять, **в какой клетке стоит конь**. Для этого достаточно центра.

OpenCV умеет считать **моменты** контура (`cv2.moments`). Из них можно получить координаты центра масс (центроида):

$$cx = \frac{m_{10}}{m_{00}}, \quad cy = \frac{m_{01}}{m_{00}}.$$

Интуитивно: `m00` связано с “площадью”, а `m10`, `m01` — с “площадью с учётом x и y ”. Их отношение даёт среднее положение.

```
1 M = cv2.moments(c)
2 cx = M['m10'] / M['m00']
3 cy = M['m01'] / M['m00']
```

4.5. А если круг частично обрезан?

Иногда красный круг может быть близко к краю выровненного изображения `warped`. Тогда часть круга может “обрезаться”.

Это **не проблема**, потому что:

- маска всё равно содержит большую часть круга;
- контур находится;
- моменты дают центр **по оставшейся части**.

А дальше в шаге 5 мы всё равно переводим (cx, cy) в номер клетки, и небольшой сдвиг обычно не меняет клетку, потому что маркер лежит **внутри клетки**, а не на границе.

Шаг 5. Определяем стартовую позицию и считаем ходы

Зная координаты центра круга в системе координат кадра и учитывая что рамка составляет 0.2 от размера клетки, найдем размер клетки и размер рамки:

$$L = 8 \cdot cell + 2 \cdot 0.2 \cdot cell = 8.4 \cdot cell \Rightarrow cell \approx \frac{L}{8.4}, \quad border \approx 0.2 \cdot cell.$$

Далее перейдем из системы координат кадра, в систему координат шахматной доски (размерность от 1 до 8 по каждой оси):

$$col = \left\lfloor \frac{cx - border}{cell} \right\rfloor, \quad row = \left\lfloor \frac{cy - border}{cell} \right\rfloor.$$

Но номера клеток (ранги) в шахматах нумеруются снизу вверх, поэтому:

$$rank = 7 - row.$$

Переходим от клетки к ответу: список ходов коня

Когда мы уже нашли клетку коня как пару индексов:

$$file_idx \in \{0..7\}, \quad rank_idx \in \{0..7\},$$

можно считать ходы коня чисто “по математике” — без обработки изображения.

Для этого подготовим Функцию `knight_moves` которая будет возвращать список всех клеток, куда конь может попасть за один ход. Конь ходит буквой “Г”: на 2 клетки в одном направлении и на 1 клетку перпендикулярно. Всего таких вариантов 8:

$$(\pm 1, \pm 2), (\pm 2, \pm 1).$$

Но часть ходов может выходить за границы доски, например, если конь стоит у края. Поэтому внутри функции мы:

- перебираем все 8 смещений (dx, dy) ;
- считаем новую клетку $(nf, nr) = (file_idx + dx, rank_idx + dy)$;
- оставляем только те, где $0 \leq nf < 8$ и $0 \leq nr < 8$.

```

1 def knight_moves(f, r):
2     deltas = [(1,2),(2,1),(-1,2),(-2,1),(1,-2),(2,-1),(-1,-2),(-2,-1)]
3     res = []
4     for dx, dy in deltas:
5         nf, nr = f + dx, r + dy
6         if 0 <= nf < 8 and 0 <= nr < 8:
7             res.append((nf, nr))
8     # сортируем, чтобы ответ всегда был в одном и том же порядке
9     res.sort(key=lambda t: (t[0], t[1]))
10    return res

```

Мы сортируем клетки в `knight_moves` по правилу: сначала по букве (А..Н), затем по цифре (1..8). Тогда:

- ответ получается однозначным и будет соответствовать требованиям условия задачи;

После `knight_moves` у нас список пар чисел, например:

$$[(0, 2), (2, 2), (3, 1), (3, 3)].$$

Однако в ответе нужно печатать шахматные обозначения: А3, С3, D2, D4. Для этого каждая пара (f, r) должна превратиться в строку функцией `sq_name`:

- $f = 0..7$ переводим в букву А..Н;
- $r = 0..7$ переводим в цифру 1..8 (по формуле $r + 1$).

```

1 def sq_name(f, r):
2     # f=0 -> 'A', f=7 -> 'H'
3     letter = chr(ord('A') + f)
4     # r=0 -> '1', r=7 -> '8'
5     digit = str(r + 1)
6     return letter + digit
7
8 moves = knight_moves(file_idx, rank_idx)
9 ans = [sq_name(f, r) for (f, r) in moves]

```

Полный код решения

```

1  """
2  Эталонное решение задачи 8 (OpenCV, Python).
3
4  1) base64 -> BGR.
5  2) Находим внешнюю чёрную рамку доски (почти чёрный большой четырёхугольник).
6  3) Warp в квадрат LxL (вид сверху).
7  4) Находим красный круг и определяем клетку.
8  5) Печатаем клетки, куда конь может попасть за 1 ход (A1..H8), отсортировано.
9
10 Координаты:
11 - A..H: слева направо вдоль НИЖНЕЙ стороны.
12 - 1..8: снизу вверх.
13 """
14
15 import sys
16 import base64
17 import numpy as np
18 import cv2
19
20 R_LO = np.array((0, 0, 235), np.uint8)
21 R_HI = np.array((20, 20, 255), np.uint8)
22
23 def decode_base64_image(b64_str: str) -> np.ndarray:
24     data = base64.b64decode(b64_str)
25     arr = np.frombuffer(data, dtype=np.uint8)
26     img = cv2.imdecode(arr, cv2.IMREAD_COLOR)
27     if img is None:
28         raise ValueError('Не удалось декодировать изображение')
29     return img
30
31 def order_points_tltrbrbl(pts: np.ndarray) -> np.ndarray:
32     """
33     Возвращает точки в порядке TL, TR, BR, BL.
34     Так как нижняя сторона доски параллельна низу кадра,
35     верхние точки имеют меньший y, нижние - больший y.
36     """
37     pts = pts.astype(np.float32)
38     p = pts.tolist()
39     p.sort(key=lambda t: (t[1], t[0]))
40     top = sorted(p[:2], key=lambda t: t[0])
41     bot = sorted(p[2:], key=lambda t: t[0])
42     tl, tr = top[0], top[1]
43     bl, br = bot[0], bot[1]
44     return np.array([tl, tr, br, bl], dtype=np.float32)
45
46 def find_board_quad(img: np.ndarray) -> np.ndarray:
47     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
48     mask = cv2.inRange(gray, 0, 70)
49     mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, np.ones((5, 5), np.uint8),
50     ↪ iterations=2)

```

```

51 cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
52 if not cnts:
53     raise ValueError('Контур рамки не найден')
54
55 cnts = sorted(cnts, key=cv2.contourArea, reverse=True)
56 for c in cnts[:10]:
57     peri = cv2.arcLength(c, True)
58     approx = cv2.approxPolyDP(c, 0.02 * peri, True)
59     if len(approx) == 4 and cv2.contourArea(approx) > 5000:
60         return order_points_tltrbrbl(approx.reshape(4, 2))
61
62     raise ValueError('Не удалось найти четырёхугольник доски')
63
64 def warp_to_canonical(img: np.ndarray, quad: np.ndarray, L: int) -> np.ndarray:
65     dst = np.array([[0, 0], [L - 1, 0], [L - 1, L - 1], [0, L - 1]],
66                   ↪ dtype=np.float32)
67     M = cv2.getPerspectiveTransform(quad.astype(np.float32), dst)
68     return cv2.warpPerspective(img, M, (L, L), flags=cv2.INTER_NEAREST)
69
70 def find_red_center(warped: np.ndarray):
71     mask = cv2.inRange(warped, R_LO, R_HI)
72     mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((3, 3), np.uint8),
73                       ↪ iterations=1)
74
75     cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
76     if not cnts:
77         raise ValueError('Красный маркер не найден')
78
79     c = max(cnts, key=cv2.contourArea)
80     M = cv2.moments(c)
81     if abs(M['m00']) < 1e-6:
82         raise ValueError('Плохой контур красного маркера')
83     return float(M['m10'] / M['m00']), float(M['m01'] / M['m00'])
84
85 def knight_moves(f: int, r: int):
86     deltas = [(1,2),(2,1),(-1,2),(-2,1),(1,-2),(2,-1),(-1,-2),(-2,-1)]
87     res = []
88     for dx, dy in deltas:
89         nf, nr = f + dx, r + dy
90         if 0 <= nf < 8 and 0 <= nr < 8:
91             res.append((nf, nr))
92     res.sort(key=lambda t: (t[0], t[1]))
93     return res
94
95 def sq_name(f: int, r: int) -> str:
96     return f'{chr(ord('A') + f)}{r + 1}'
97
98 def main():
99     b64 = sys.stdin.readline().strip()
100     img = decode_base64_image(b64)
101     quad = find_board_quad(img)

```

```
102     # фиксированный размер для удобства
103     L = 520
104     warped = warp_to_canonical(img, quad, L=L)
105
106     cx, cy = find_red_center(warped)
107
108     # В генераторе рамка = 0.2*cell => L = 8*cell + 2*0.2*cell = 8.4*cell
109     cell = L / 8.4
110     border = 0.2 * cell
111
112     col = int((cx - border) / cell)
113     row = int((cy - border) / cell)
114     col = max(0, min(7, col))
115     row = max(0, min(7, row))
116
117     file_idx = col
118     rank_idx = 7 - row # row 0 сверху => rank 8; row 7 => rank 1
119
120     moves = knight_moves(file_idx, rank_idx)
121     ans = [sq_name(f, r) for (f, r) in moves]
122
123     print(len(ans))
124     for s in ans:
125         print(s)
126
127 if __name__ == '__main__':
128     main()
```