#include <map>

#include <queue>

#include <utility>

#include <iomanip>

#include <string>

#include <set>

#include "raceState.hpp"

const int SEARCH\_DEPTH = 5;

struct PlayerState {

Point position;

IntVec velocity;

bool operator<(const PlayerState &ps) const {

return position != ps.position ?

position < ps.position :

velocity < ps.velocity;

}

PlayerState(Point p, IntVec v) : position(p), velocity(v) {}

};

std::ostream& operator<<(std::ostream& out, const Point& p)

{

return out << "(" << p.x << ", " << p.y << ")";

}

std::ostream& operator<<(std::ostream& out, const LineSegment& ls)

{

return out << ls.p1 << " => " << ls.p2;

}

std::ostream& operator<<(std::ostream& out, const PlayerState& ps)

{

return out << "{" << ps.position << ", " << ps.velocity << "}";

}

map<Point, int> bfsed;

using History = vector<pair<int, int>>;

static pair<long long, long long> decode(const History& hist, const Course& course)

{

long long shift\_y = 1LL;

long long sum\_y = 0LL;

for (auto ite = hist.rbegin(); ite != hist.rend(); ++ite) {

shift\_y \*= (course.length + 1) \* 2;

sum\_y \*= (course.length + 1) \* 2;

sum\_y += ite->first;

sum\_y -= ite->second;

sum\_y += course.length;

}

return make\_pair(sum\_y, shift\_y);

}

static long long cal(const PlayerState& me, const PlayerState& rv, const History& hist, const int depth, const RaceState& rs, const Course& course)

{

// calculating heuristic score here

// HINT: this code use only y-axis, but we can do all kinds of things to win

const auto de = decode(hist, course);

long long val = 0;

val += (SEARCH\_DEPTH - depth) \* de.second;

val += de.first;

return val;

}

const long long INF = 1LL << 60;

**int left\_border;**

using State = pair<int, pair<PlayerState, PlayerState>>;

map<State, pair<long long, IntVec>> memo;

// this function(pseudo alpha-beta algorithm) attempt to prevent enemy's move

static pair<long long, IntVec> alpha\_beta(const RaceState& rs, const Course& course, const PlayerState& me, const PlayerState& rv,

History &hist, int depth = 0, long long alpha = -INF, long long beta = INF)

{

if (memo.count({ depth, {me, rv} })) {

return memo[{depth, { me, rv }}];

}

IntVec myBestAction = { 0, 0 };

if (depth == SEARCH\_DEPTH || me.position.y >= course.length) {

if (rs.position == me.position) {

// stucked

return { -INF, myBestAction };

}

return { cal(me, rv, hist, depth, rs, course), myBestAction };

}

**int priority=1;**

**if(me.position.x<=left\_border)priority=-1**;

for (int my = 1; -1 <= my; --my) {

// limit velocity

if (me.velocity.y + my > course.vision / 2) {

continue;

}

**int roop\_count=0;**

**for (int mx = -1 \* priority; roop\_count < 3; mx += priority) {**

**roop\_count++;**

long long gamma = beta;

**int rv\_priority = 1;**

**if(rv.position.x <= left\_border)rv\_priority=-1;**

for (int ey = 1; -1 <= ey; --ey) {

**int rv\_roop\_count=0;**

**for (int ex = -1 \* rv\_priority; rv\_roop\_count < 3; ex += rv\_priority) {**

**rv\_roop\_count++;**

PlayerState nextMe = me;

nextMe.velocity.x += mx;

nextMe.velocity.y += my;

nextMe.position.x += nextMe.velocity.x;

nextMe.position.y += nextMe.velocity.y;

const LineSegment myMove(me.position, nextMe.position);

PlayerState nextRv = rv;

nextRv.velocity.x += ex;

nextRv.velocity.y += ey;

nextRv.position.x += nextRv.velocity.x;

nextRv.position.y += nextRv.velocity.y;

const LineSegment enMove(rv.position, nextRv.position);

bool stopped = false;

if (course.obstacled(me.position, nextMe.position)

|| myMove.goesThru(rv.position)) {

nextMe.position = me.position;

stopped |= true;

}

if (rv.position.y >= course.length

|| course.obstacled(rv.position, nextRv.position)

|| enMove.goesThru(me.position)) {

nextRv.position = rv.position;

stopped |= true;

}

if (myMove.intersects(enMove) && !stopped) {

if (me.position.y != rv.position.y) {

if (me.position.y < rv.position.y) {

nextRv.position = rv.position;

}

else {

nextMe.position = me.position;

}

}

else if (me.position.x != rv.position.x) {

if (me.position.x < rv.position.x) {

nextRv.position = rv.position;

}

else {

nextMe.position = me.position;

}

}

}

hist[depth] = make\_pair(

min(course.length, nextMe.position.y),

min(course.length, nextRv.position.y)

);

auto res = alpha\_beta(rs, course, nextMe, nextRv, hist, depth + 1, alpha, gamma);

hist[depth] = make\_pair(0, 0);

if (res.first < gamma) {

gamma = res.first;

}

if (alpha >= gamma) {

gamma = alpha;

goto END\_ENEMY\_TURN;

}

}

}

END\_ENEMY\_TURN:

if (alpha < gamma) {

alpha = gamma;

myBestAction = { mx, my };

}

if (alpha >= beta) {

memo[{depth, { me, rv }}] = { beta, myBestAction };

return { beta, myBestAction };

}

}

}

memo[{depth, { me, rv }}] = { alpha, myBestAction };

return { alpha, myBestAction };

}

pair<int, IntVec> dls(const Point& p, const IntVec v, const Point rvp, const Course& course, int depth, set<IntVec> done = {})

{

if (depth == 0) {

return { 1 << 28, {} };

}

int best = 1 << 28;

IntVec bestAction = {};

for (int dy = -1; dy <= 1; ++dy) {

for (int dx = -1; dx <= 1; ++dx) {

int nvx = v.x + dx;

int nvy = v.y + dy;

int npx = p.x + nvx;

int npy = p.y + nvy;

const IntVec nv(nvx, nvy);

const Point np(npx, npy);

if (done.count(nv)) {

continue;

}

done.insert(nv);

const LineSegment move(p, np);

if (course.obstacled(p, np)

|| move.goesThru(rvp)) {

const auto& ret = dls(p, nv, rvp, course, depth - 1, done);

if (ret.first < best) {

best = ret.first;

bestAction = { dx, dy };

}

}

else {

if (np.y >= course.length) {

best = -1;

bestAction = { dx, dy };

}

if (bfsed.count(np) && bfsed[np] < best) {

best = bfsed[np];

bestAction = { dx, dy };

}

}

}

}

return { best, bestAction };

}

static IntVec find\_movable(const RaceState& rs, const Course& course)

{

// IDDFS

for (int d = 1; ; ++d) {

const auto ret = dls(rs.position, rs.velocity, rs.oppPosition, course, d);

if (ret.first == 1 << 28) {

continue;

}

return ret.second;

}

}

static void bfs(const RaceState& rs, const Course& course)

{

bfsed.clear();

queue<Point> queue;

const int ymax = rs.position.y + course.vision;

for (int x = 0; x < course.width; ++x) {

if (course.obstacle[ymax][x] == ObstState::OBSTACLE) {

continue;

}

queue.push(Point(x, ymax));

bfsed[Point(x, ymax)] = 0;

}

const vector<pair<int, int>> ofs = {

{0, 1},

{0, -1},

{1, 0},

{-1, 0}

};

while (queue.size()) {

Point p = queue.front();

queue.pop();

for (const auto& d : ofs) {

int nx = d.first + p.x;

int ny = d.second + p.y;

Point np(nx, ny);

if (ny > course.length + course.vision || course.obstacle[ny][nx] != ObstState::NONE) {

continue;

}

if (bfsed.count(np)) {

continue;

}

bfsed[np] = bfsed[p] + 1;

queue.push(np);

}

}

}

static IntVec play(const RaceState& rs, const Course& course) {

memo.clear();

bfs(rs, course);

History hist(SEARCH\_DEPTH);

auto p = alpha\_beta(rs, course, { rs.position, rs.velocity }, { rs.oppPosition, rs.oppVelocity }, hist);

if (p.first == -INF) {

// If my player will be stuck, use greedy.

return find\_movable(rs, course);

}

return p.second;

}

int main() {

Course course(cin);

cout << 0 << endl;

cout.flush();

**left\_border=(int)((course.width-1)/3);**

while (true) {

RaceState rs(cin, course);

IntVec accel = play(rs, course);

cout << accel.x << ' ' << accel.y << endl;

}

}