Contents

1. Square Game Writeup - L3akCTF

First off, huge thanks to the team behind L3akCTF for some cool challenges.

1.1. Challenge Introduction

We are studying a 2 player game played on an $n \times n$ grid. Before the game starts, Player 1 picks one of the n^2 points, and keeps this point q secret. Each turn, Player 1 picks a new public point $p_1 = (x, y)$ uniformly at random, and communicates it to Player 2. Player 2 replies with a radius r_1. Player 1 checks (privately) if their private point is in the square centered at p_1 with radius r, and sends Player 2 the boolean result. This goes back and forth for 100 turns: Player 1 sending point p_i, Player 2 responding with a radius r_i, and Player 1 checking if q is in the corresponding square.

After the hundredth turn, Player 2 has to guess Player 1's private point p. If they are close (with 1%) to p, they win. To solve this, we are going to apply a binary search, picking radii r_i such that we eliminate approximately halve of the remaining cells each turn. To build up to this solution, we're going to start by walking through an example game, strip the game down to guessing a secret number, and build up our binary search algorithm from there.

1.2. Example Game

For our example, we're going to play on a 6×6 grid for 5 turns.

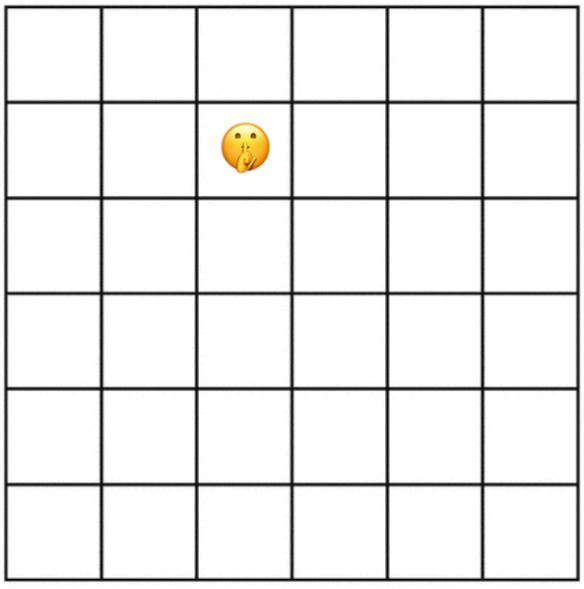

Figure 1: The board state on turn 0, with secret point $(2, 3)$

Player 1 starts by picking the private point $(2, 3)$. On turn 1, Player 1 picks a random point $p_1 = (4, 5)$. Player 2 picks a radius of $r_1 = 2$, and Player 1 responds by saying that q is in that square. From this, Player 2 learns that the point isn't outside this square, so they color cells c inside the square, that is $\|p - c\|_1 \leq r$, blue (meaning those cells are feasible cells for q to be in), and all squares outside the square red.

On turn 2, Player 1 picks a random point $p_2 = (6, 2)$, Player 2 responds with a radius of $r_2 = 2$, and Player 1 responds by saying that q is **not** in that square. From this Player 2 knows that it's in the square centered at p_1 with radius r_1, but not in the square centered at p_2 with radius r_2, leaving a polygonal region of remaining points.

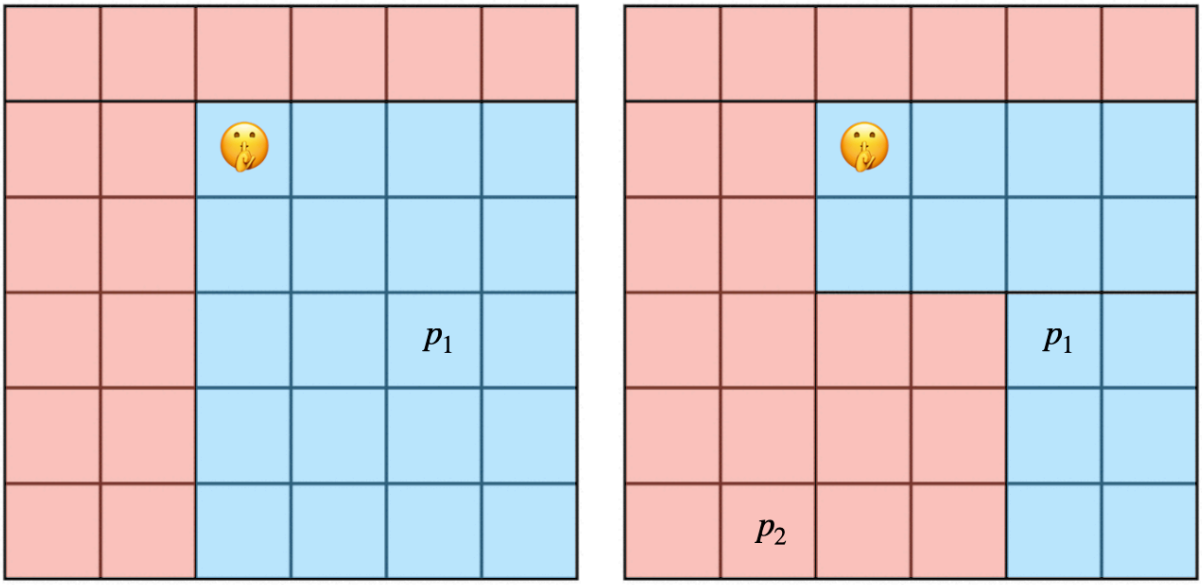

Figure 2: The board state after turns 1 and 2.

This continues back and forth, Player 1 picking a point, Player 2 picking a radius, and Player 1 checking if q is in that square. The five turn case is feasible to play by hand with our nice graphic. In the challenge, $n = 10^9$ and we play for 100 turns, so it's not going to be practical for us to play this way by hand. Therefore, we need some automated approach.

1.3. Developing a Strategy

Each turn, we get to check a square of the grid to see if the secret point q is in that square. This means that on turn i, we can compute the region of remaining points q can be on, the so called *feasible region* of the board, by taking the intersection of the regions from each turn. To develop a strategy, let's back up a bit to study an easier version of square game.

2. Guessing a secret number

Instead of the $n \times n$ grid, what if we have Player 1 pick a random number q from 1 to n. Each turn, Player 2 is going to be allowed to pick an interval $[a, b]$, and Player 2 responds if q is in that interval. What is Player 2's strategy?

The easiest way forward is to just binary search: on turn 1, we check the interval $\left[\frac{n}{2}, n\right]$. If the point isn't in this interval, we know it's in $\left[0, \frac{n}{2}\right]$. At each turn, Player 2 cuts the number of feasible cells in half: turn 1 there are n feasible cells, turn 2 there are $\frac{n}{2}$ feasible cells, then $\frac{n}{4}$ on turn 3, and so on. Extrapolating, we see that there are $\frac{n}{2^{i-1}}$ feasible cells on turn i, meaning that there will only be 1 cell remaining when

$$\frac{n}{2^{i-1}} = 1 \Rightarrow n = 2^{i-1} \Rightarrow i = \log_2(n) + 1.$$

This is a much easier problem than the square game, but we are going to take the same approach to guessing our hidden point. Let's ungeneralize a little bit by giving Player 1 the power to randomly pick the center p_i of our interval, and letting Player 2 pick the radius r_i of the interval. Player 2 is going to take the exact same strategy: each turn, pick a radius r_i such that half of the feasible region is in $I = [p_i - r_i, p_i + r_i]$, and half of it is outside. To convince ourselves that we can do this, observe that when $r_i = 0$, at most 1 cell of the feasible region is in $I = [p_i, p_i]$, and when $r_i = n$, all possible values for q are in $I = [0, n]$. If we increment $r_i \to r_i + 1$, there are at most 2 new feasible cells in our interval for each increment, so there is some radius r_i that gets about half of the remaining feasible cells.

At the end Player 2's final turn, they have to guess some point. A possible strategy is to pick the midpoint of the feasible cells that we are left with, but we are going to see later that we narrow our search space enough that any point we're left with will work (unless we get really unlucky).

2.1. Algorithm

Player 2's strategy is then as follows:

```
Initialize the feasible region as [1, 2, ..., n]

low, high, r = 0, n, 0
for i in range(num_turns):
  Set r = (high + low) // 2

  Obtain random point p_i from Player 1
  for k in range(log2(n)):  # Binary search has depth log2(n) by above
    compute the intersection of the feasible region and [p_i - r_i, p_i + r_i]
    count the number of cells in the intersection
    if the number of cells is about half of the feasible region:
      break
    if it's less than half:
      low = r - 1
    if it's more than half:
      high = r + 1

  Ask Player 1 if q is in [p_i - r_i, p_i + r_i]
  Update the feasible region

Guess the hidden point q
```

2.2. How to store the feasible region

What remains is to figure out how to store the feasible region. For small enough n, we can create an array of binary values, where 0 means the cell can't contain q and a 1 means it could contain q. However, for large enough n, this is not feasible, as we will run out of memory on our computer. When $n = 10^9$, storing a bit for each integer 1 to n is on the order of 100MB. For the $n \times n$ grid, that's over a hundred PB, which (unless you have plenty of hard drive space to loan me) is not happening.

One possible approach is to use two interval trees, one for intervals that contain q, and the other for intervals that don't contain q. At each iteration, we intersect the intervals in "hit" tree, union and then complement the "miss" tree, and intersect the two results. However, the interval tree approach may encourage a feasible region of many small, disjoint intervals. If these intervals are scattered

across the number line, that's no good for making a guess, since we have no idea which chunk might contain q, and are left to a coinflip. Instead, we'd like a strategy that promotes keeping a connected feasible region at all times, so that we can leverage the 1% tolerance that Player 1 gives us.

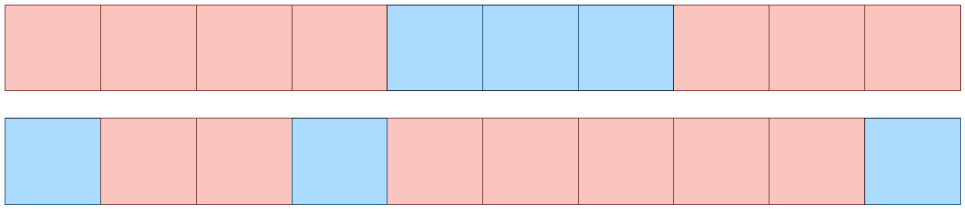

Figure 3: Both game boards have three feasible cells. However, one is connected, where as the other is sparse, meaning if we guess with a tolerance of 1, we are guaranteed to win on the first, but have 1/3 chance on the second.

While a generalized polygon library would get us the most bang for our buck per iteration in terms of minimizing the feasible area as fast as possible, the above example shows that minimizing area is not always best. Thankfully, we are allotted enough turns that applying interval trees or tracking generalized polygons will get us down to a single cell in the interval case, it is yet to be shown that it will work out as cleanly in the grid case.

The strategy we are going to roll with (at least for now) is to only update our feasible region when we get a hit. Our feasible region starts as a connected interval $[1, n]$. At some point, we get our first hit on interval $[p_i - r_i, p_i + r_i]$. Since we know the point is in both intervals, it must be in the intersection. The intersection of two connected intervals is connected, so our update to the feasible region is connected. Inductively, we start our turn with a connected interval, check a connected interval, and either don't update or take the intersection of two connected intervals. Since, on each turn, we cut the feasible region in half, so we expect to get a hit every other turn and (in expectation) it takes $O(\log n)$ turns to find the point p. Sure, the complicated solution may be more "iteration efficient", but if this gets us a solve then it's good enough for our usage.

2.3. Line solve script

line_utils.py

```
def get_endpoints(x, r, n):
    """Gets the endpoints of the interval centered at x with radius r.
    Restricts interval to the n cell line"""
    x1 = max(x-r, 0)
    x2 = min(x+r, n-1)
    return x1, x2


def get_intersection(x1, x2, x3, x4):
    """Given the bottom left and top right corners of two rectangles,
    return the corners of their intersection."""
    x5 = max(x1, x3)
    x6 = min(x2, x4)

    return x5, x6


def compute_length(x1, x2):
    """Computes the length of an interval :)"""
    return x2 - x1
```

```
def check_in_interval(center, radius, point):
    x_min = center - radius
    x_max = center + radius
    return x_min <= point <= x_max
```

line.py

```
import numpy as np
from line_utils import *

epsilon = 0.01
n = 1000000

for i in range(1000):
    print(f'Game {i+1}')

    q = np.random.randint(n)
    x1, x2 = 0, n - 1

    for j in range(100):
        print(f'Round {j+1}')

        p_i = np.random.randint(n)
        print(f'Randomly sampled point is {p_i}')
        # Compute the area of the current feasible region to initialize binary search
        original_length = x2 - x1
        new_length = 0
        low, high, r_i = 0, n, 0

        counter, threshold = 0, np.log2(n)
        print(original_length, new_length)
        while np.abs(new_length * 2 - original_length) > n * epsilon:
            r_i = (low + high) // 2

            x3, x4 = get_endpoints(p_i, r_i, n)
            x5, x6 = get_intersection(x1, x2, x3, x4)
            new_length = compute_length(x5, x6)

            # Prints for debugging
            print(f'Endpoints of possible interval: {x3, x4}, r: {r_i}')
            print(f'Endpoints of intersection interval: {x5, x6}')
            print(f'Original length: {original_length}, new length: {new_length}')
            print()

            if new_length * 2 < original_length:
                low = r_i - 1
            else:
                high = r_i + 1

            # See if the process hangs
            counter += 1
            if counter > threshold:
                break

        # Send r
        # Check if the hidden point is in the square we selected
```

```
            print(f'Selected radius {r_i}.')
            print(f'Original length was {original_length}, new length is {new_length}')
            # Get the corners again in case look didn't run
            x3, x4 = get_endpoints(p_i, r_i, n)
            # Check if the point is in our square
            if check_in_interval(p_i, r_i, q):
                print('The point is inside the interval!')
                x1, x2 = get_intersection(x1, x2, x3, x4)
            else:
                print('The point is outside the interval.')

    # Make our guess, sand it, and compare to the actual hidden point
    guess = (x1 + x2) // 2
    print(guess, q)
    distance = np.abs(guess - q)
    print(f'Guessing {guess}, which is {distance} away from the secret point {q}.')
    if distance <= epsilon * n:
        print('Correct!')
    else:
        print('Incorrect.')
        break
```

3. Returning to the Grid

Now that we have solved the guessing game for the line, making a solution for the grid will just mean storing corners of our feasible rectangle rather than endpoints of the feasible interval, exchange length for area, and changing how we compute intersections. The key insight that we carry over from the interval to the grid is that it's best to always store a connected feasible region, as the interval tree approach becomes rather gnarly once we start working with many-sided polygons that might snake through the grid. Storing this additional complexity is ultimately not useful for us, as we can get away with maintaining a single rectangle at all times.

Because a rectangle is the product of two intervals, the intersection of two rectangles is the product of the intersection of the intersections of the intervals:

$$([x_1, x_2] \times [y_1 \times y_2]) \cap ([x_3, x_4] \times [y_3, y_4]) = ([x_1, x_2] \cap [x_3, x_4]) \times ([y_1, y_2] \times [y_3, y_4]).$$

In human language, the intersection of two rectangles is a rectangle. We've effectively reduced the intersection problem on the grid to doing the intersection problem on the interval twice, which is a problem we've already solved. Therefore, a convenient way to store rectangles to make intersection easy is to parameterize them by their bottom left and top right corners, so that the corners of their intersection can be computed as follows

```
def get_intersection(x1, y1, x2, y2, x3, y3, x4, y4):
    """Given the bottom left and top right corners of two rectangles,
    return the corners of their intersection."""
    x5 = max(x1, x3)
    y5 = max(y1, y3)
    x6 = min(x2, x4)
    y6 = min(y2, y4)

    return x5, y5, x6, y6
```

From here, the grid solution is just a smooth generalization. Substituting Player 1's actions for talking to netcat via `pwntools`, we have the following solution:

utils.py

```
def get_corners(x, y, r, n):
    """Gets the bottom left and top right corners of the square centered at x
    with radius r. Restricts square to the nxn grid"""
    x1 = max(x-r, 0)
    y1 = max(y-r, 0)
    x2 = min(x+r, n-1)
    y2 = min(y+r, n-1)
    return x1, y1, x2, y2


def get_intersection(x1, y1, x2, y2, x3, y3, x4, y4):
    """Given the bottom left and top right corners of two rectangles,
    return the corners of their intersection."""
    x5 = max(x1, x3)
    y5 = max(y1, y3)
    x6 = min(x2, x4)
    y6 = min(y2, y4)

    return x5, y5, x6, y6


def compute_area(x1, y1, x2, y2):
    """Computes the area of a rectangle :)"""
    return (x2 - x1) * (y2 - y1)


def check_in_square(center, radius, point):
    x_min = center[0] - radius
    x_max = center[0] + radius
    y_min = center[1] - radius
    y_max = center[1] + radius
    return x_min <= point[0] <= x_max and y_min <= point[1] <= y_max
```

solve.py

```
import numpy as np
from utils import get_corners, get_intersection, compute_area
from pwn import *

epsilon = 0.01
n = 1000000

conn = remote('34.139.98.117', '6668')

print(conn.recvuntil('100\n\n\n'))
print('\n\n')

for i in range(10):
    print(f'Game {i+1}')
    x1, y1 = 0, 0
    x2, y2 = n - 1, n - 1

    for j in range(100):
        print(f'Round {j+1}')

        # Read in the random point (x, y)
```

```
        conn.recvuntil('(')
        p_i = map(int, conn.recvuntil(')').decode()[:-1].split(', '))
        x, y = p_i
        print(f'Randomly sampled point is ({x}, {y})')
        # Compute the area of the current feasible region to initialize binary search
        original_area = compute_area(x1, y1, x2, y2)
        new_area = 0
        low, high, r_i = 0, n, 0

        counter, threshold = 0, np.log2(n)
        while np.abs(new_area * 2 - original_area) > n * epsilon:
            r_i = (low + high) // 2

            x3, y3, x4, y4 = get_corners(x, y, r_i, n)
            x5, y5, x6, y6 = get_intersection(x1, y1, x2, y2, x3, y3, x4, y4)
            new_area = compute_area(x5, y5, x6, y6)

            # Prints for debugging
            print(f'Corners of possible square: {x3, y3, x4, y4}, r: {r_i}')
            print(f'Corners of intersection rectangle: {x5, y5, x6, y6}')
            print(f'Original area: {original_area} and new area: {new_area}')
            print()

            if new_area * 2 < original_area:
                low = r_i - 1
            else:
                high = r_i + 1

            # See if the process hangs
            counter += 1
            if counter > threshold:
                break

        # Send r
        conn.recv()
        conn.sendline(str(r_i).encode())

        # Check if the hidden point is in the square we selected
        print(f'Selected radius {r_i}.')
        print(f'Original area was {original_area} and new area is {new_area}')
        # Get the corners again in case look didn't run
        x3, y3, x4, y4 = get_corners(x, y, r_i, n)
        # Check if the point is in our square
        membership_query = conn.recvline().decode()
        print(membership_query)
        if 'inside' in membership_query:
            x1, y1, x2, y2 = get_intersection(x1, y1, x2, y2, x3, y3, x4, y4)

    # Make our guess, sand it, and compare to the actual hidden point
    x, y = (x1 + x2) // 2, (y1 + y2) // 2
    print(conn.recv().decode()[2:-2])
    print(f'Guessing ({x}, {y})')
    conn.sendline(f'{x},{y}'.encode())
    # Read the results
    outcome = conn.recvline().decode()
    print(outcome)
```

```
    # Sometimes we get unlucky
    assert 'Incorrect' not in outcome, "Guessed wrong, rerun the script"
# Other times we don't :)
conn.recvuntil("Congratulations, you've done it. ".encode())
print(conn.recvuntil('}'.encode()).decode())
conn.recv()
conn.close()
```

which, when run, nets us the flag:

```
L3AK{5qu4r35_574y_5h4rp}
```

3.1. Takeaways

Fancier solutions ≠ better solutions. Sure you can use some fancy data structure to score the whole polygon of remaining cells at each iteration, but you can write a pretty simple solve from scratch that only stores a rectangle at all times. Your fancy data structure might save iterations, but that may come at a cost of a huge space blow up. This approach is easier to implement and analyze, only ever uses constant space, and is easily extensible in higher dimensions (as shown by how easy our generalization was from the line to the grid).