Impossible Chessboard Escape Puzzle

Solution implementation by Cory Leigh Rahman

GitHub Repo: CoryLR/impossible-chessboard-escape-puzzle

Puzzle Prompt

Prisoner 1 walks in to a room, sees a chessboard where each square has a coin on top, flipped either to heads or tails. The warden places the key under one of the squares, which prisoner 1 sees. Before he leaves, he must turn over one and only one coin. Prisoner 2 then walks in and is supposed to be able to figure out which squares the key is in just by looking at the arrangement of coins. The Prisoners can coordinate a plan ahead of time. What's the plan?

Here's how the scenario will go:

1. The warden will set up the board while the prisoners come up with a plan
2. Prisoner 1 will witness which chessboard square the warden places the key under, flip exactly one coin to try and convey this information, then leave
3. Prisoner 2 will enter and have one chance to pick which square the key is under

Related Links

- Introduction video & walk-through: The almost impossible chessboard puzzle by Stand-up Maths
- Further discussion video: The impossible chessboard puzzle by 3Blue1Brown
- Full breakdown website with interactive examples: Impossible Escape? by DataGenetics

Tools Used

In [1]:

```
import sys
import numpy as np
import pandas as pd
from typing import Sequence

print("Python", sys.version[0:6])
print("Numpy", np.__version__)
print("Pandas", pd.__version__)
```

```
Python 3.8.11
Numpy 1.20.3
Pandas 1.3.2
```

SPOILER ALERT: If you want to solve this puzzle yourself, turn back now!

Set Up the Board

We need to make a representation of a chessboard with a randomly flipped coin in each square

In [2]:

```python
def get_chessboard_of_coins() -> pd.DataFrame:
    """
    Make the chessboard with randomized coins represented by 0 and 1
    * 1 = heads
    * 0 = tails
    """
    binary_8x8_grid = np.random.randint(0, 2, size=(8, 8))
    return pd.DataFrame(binary_8x8_grid)

chessboard = get_chessboard_of_coins()
chessboard
```

Out[2]:

	0	1	2	3	4	5	6	7
0	1	1	0	1	0	0	1	0
1	0	1	1	1	1	0	1	0
2	0	1	0	0	0	0	0	0
3	1	1	0	1	1	1	0	1
4	0	0	1	1	0	1	1	0
5	0	1	1	1	1	0	1	1
6	0	1	0	0	1	1	0	0
7	0	1	0	1	1	0	0	1

The Prisoners Hatch a Plan

First the prisoners decide to give each square a unique ID:

In [3]:

```
def get_chessboard_square_ids() -> pd.DataFrame:
    """Start with 0 and count left to right, top to bottom"""
    squares = np.zeros((8,8))
    for i in range(8):
        squares[i] = np.arange(8*i, 8*(i+1))
    return pd.DataFrame(squares).applymap(lambda x: int(x))

chessboard_square_ids = get_chessboard_square_ids()

print('Each square on the chess board is given a unique ID 0 through 63:')
chessboard_square_ids
```

```
Each square on the chess board is given a unique ID 0 through 63:
```

Out[3]:

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63

Because there are 64 squares each with their own unique ID (0 through 63), we need a minimum of 6 bits of information to represent any one square. For example:

- Square 0 = 0 in binary
- Square 63 = 111111 in binary
- And, for example, Square 37 = 100101 in binary

Let's look at all the chessboard's square IDs in binary:

In [4]:

```
def binary_to_int(binary: str) -> int:
    return int(binary, 2)

def int_to_6bit_binary(num: int) -> str:
    if num >= 64:
        raise ValueError(f'Input to function int_to_6bit_binary must be less 
    return format(num, 'b').zfill(6)

chessboard_square_ids_binary = chessboard_square_ids.applymap(int_to_6bit_bin

print("Each square's unique ID represented in binary:")
chessboard_square_ids_binary
```

```
Each square's unique ID represented in binary:
```

Out[4]:

	0	1	2	3	4	5	6	7
0	000000	000001	000010	000011	000100	000101	000110	000111
1	001000	001001	001010	001011	001100	001101	001110	001111
2	010000	010001	010010	010011	010100	010101	010110	010111
3	011000	011001	011010	011011	011100	011101	011110	011111
4	100000	100001	100010	100011	100100	100101	100110	100111
5	101000	101001	101010	101011	101100	101101	101110	101111
6	110000	110001	110010	110011	110100	110101	110110	110111
7	111000	111001	111010	111011	111100	111101	111110	111111

Now the challenge becomes how to precisely communicate 6 bits of information by flipping just one coin on the board.

The board can be split up in 6 groups and arranged in such a way so that one coin flip can reside in all, some, or none of the the groups. Now let's say we count the number of heads-up coins in each section. An even number of heads-up coins will mean 1, while an odd number of heads-up coins will mean 0. If we associate each of the 6 groups with one of the digits of a 6-bit binary string, then we can then construct a binary string like "100101" by counting the number of heads-up coins in each section.

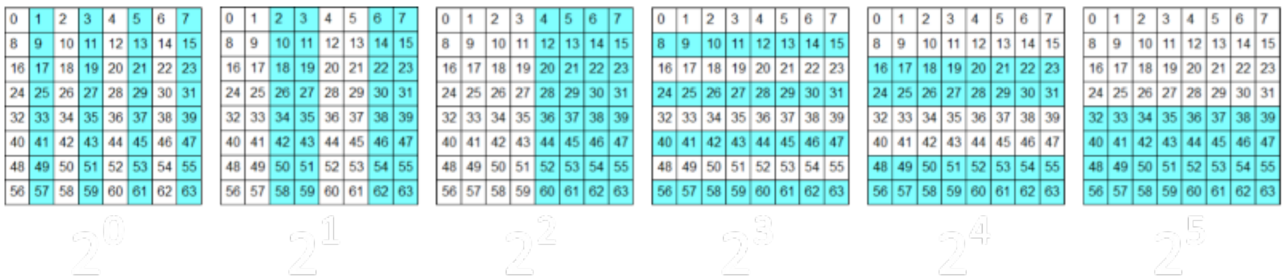

Image Source: https://datagenetics.com/blog/december12014/index.html

This system enables Prisoner 1 to have total control over the board's inherent randomly encoded binary string. Prisoner 1 can change it to any number 0 to 63, allowing Prisoner 2 to decode this message and discover the location of the square containing the key.

For example, if you flip Square ID 0 then the chess board's initial, random encoded number will not change at all. If you flip the coin in Square ID 1 then only the first digit of the binary will change. If you flip the coin in Square ID 5, then exactly the first and third digits of the binary will change, and none others. This works with all combinations such that there is exactly one square on the board for every possible adjustment needed to correctly convey the ID of the square containing the warden's key.

The prisoners agree on the following section locations on the chessboard:

In [5]:

```
class SectionDefinition:
    """
    We have defined 6 sections in such a way that allows one coin flip
    to have an effect on any combination of the 6 binary digits

    Sections (6 total starting with "0"):
    * 0 = Columns 1, 3, 5, 7
    * 1 = Columns 2, 3, 6, 7
    * 2 = Columns 4, 5, 6, 7
    * 3 = Rows 1, 3, 5, 7
    * 4 = Rows 2, 3, 6, 7
    * 5 = Rows 4, 5, 6, 7
    """

    def __init__(self):
        self._get_section_switcher = [
            self._get_section_0, self._get_section_1, self._get_section_2,
            self._get_section_3, self._get_section_4, self._get_section_5
        ]

    def get(self, section_number: int, data_frame: pd.DataFrame,
                in_section: bool = True) -> pd.DataFrame:
        """
        Get a subsection of the provided chessboard based on the section numb

        Parameters:
        * section_number - the section
        * data_frame - the chessboard
        * in_section (default:True) - set False to get outside the section nu

        Example:
        `section1 = sections.get(1, chessboard)`
        """
        return self._get_section_switcher[section_number](data_frame, in_sect

    def _get_section_0(self, data_frame: pd.DataFrame, in_section: bool) -> p
        return data_frame.iloc[:,[1, 3, 5, 7]] if in_section else data_frame.
    def _get_section_1(self, data_frame: pd.DataFrame, in_section: bool) -> p
        return data_frame.iloc[:,[2, 3, 6, 7]] if in_section else data_frame.
    def _get_section_2(self, data_frame: pd.DataFrame, in_section: bool) -> p
        return data_frame.iloc[:,[4, 5, 6, 7]] if in_section else data_frame.
    def _get_section_3(self, data_frame: pd.DataFrame, in_section: bool) -> p
        return data_frame.iloc[[1, 3, 5, 7]] if in_section else data_frame.il
    def _get_section_4(self, data_frame: pd.DataFrame, in_section: bool) -> p
        return data_frame.iloc[[2, 3, 6, 7]] if in_section else data_frame.il
    def _get_section_5(self, data_frame: pd.DataFrame, in_section: bool) -> p
        return data_frame.iloc[[4, 5, 6, 7]] if in_section else data_frame.il


sections = SectionDefinition()

print("For example, here is Section # 0 of the chessboard (columns 1, 3, 5, 7
sections.get(0, chessboard)
```

```
For example, here is Section # 0 of the chessboard (columns 1, 3, 5, 7):
```

Out[5]:

	1	3	5	7
0	1	1	0	0
1	1	1	0	0
2	1	0	0	0
3	1	1	1	1
4	0	1	1	0
5	1	1	0	1
6	1	0	1	0
7	1	1	0	1

The prisoners then agree on the system to decode the chessboard into a 6-bit binary string:

In [6]:

```
# Functions to read a 6-bit binary string from the board using 6 sections

def one_if_even_zero_if_odd(num: int) -> int:
    if (num % 2) == 0:
        return 1
    else:
        return 0

def count_heads_up_coins_in_section(section: pd.DataFrame) -> int:
    return section.to_numpy().sum()

def read_binary_from_chessboard(board: pd.DataFrame, sections: SectionDefinit
    digits = []
    for i in range(6):
        section_of_chessboard = sections.get(i, board)
        number_of_heads_up_in_section = count_heads_up_coins_in_section(secti
        one_or_zero = one_if_even_zero_if_odd(number_of_heads_up_in_section)
        digits.append(one_or_zero)
    binary_string_based_on_sections = "".join([str(int) for int in digits])
    return binary_string_based_on_sections
```

Prisoner 1 is Shown Which Square Holds the Key

In [7]:

```
# The warden places the key under one of the squares on the chess board

def get_warden_key_placement() -> int:
    key_location_row = np.random.randint(0, 8)
    key_location_col = np.random.randint(0, 8)
    return key_location_row * 8 + key_location_col

square_where_warden_put_key = get_warden_key_placement()

print(f'The prison warden has placed the key under square # {square_where_war
```

```
The prison warden has placed the key under square # 19
```

Now that we have our target key location, Prisoner 1 needs to determine:

1. The initial, random binary string read from the 6 agreed-upon sections
2. The target binary to change it to: the ID of the square containing the warden's key

In [8]:

```
initial_binary_value = read_binary_from_chessboard(chessboard, sections)
target_binary_value = int_to_6bit_binary(square_where_warden_put_key)
print(f'Initial chessboard binary: {initial_binary_value}')
print(f'Target chessboard binary:  {target_binary_value} (this represents "{s
```

```
Initial chessboard binary: 010010
Target chessboard binary:  010011 (this represents "19", the location of the
key)
```

Next we need to figure out which coin on the board will, when flipped, **turn the board's initial binary value into the target binary value**. This will allow Prisoner # 2 to read the read the location of the key from the board.

```python
In [9]: def get_squares_in_section(section_number: int, in_section: bool) -> pd.DataF
            return sections.get(section_number, chessboard_square_ids, in_section).to_

        def find_which_coin_to_flip(board: pd.DataFrame, key_location_id: int,
                                    sections: pd.DataFrame) -> int:
            initial_binary = read_binary_from_chessboard(board, sections)
            target_binary = int_to_6bit_binary(key_location_id)

            square_options = []

            # For each digit in the 6-bit binary string,
            # decide if we need to flip that digit or not
            for i in range(6):
                if initial_binary[i] != target_binary[i]:
                    # If the digits are not the same, then we need to flip
                    # a coin in this section, so get all squares in the section
                    square_options.append(get_squares_in_section(i, True))
                else:
                    # If the digits are the same then we need to keep this
                    # digit the same, so get all squares *not* in the section
                    square_options.append(get_squares_in_section(i, False))

            # These are all the squares which have a correct impact on
            # at least one digit of the binary
            squares_with_correct_impact = np.array(square_options).flatten()

            # Only one square will appear in `squares_with_correct_impact`
            # six times, that's the square we need to flip to change all six
            # digits to our target binary
            coin_to_flip_square_id = np.bincount(squares_with_correct_impact).argmax(

            return coin_to_flip_square_id


        coin_to_flip_square_id = find_which_coin_to_flip(chessboard, square_where_ward

        print(f"Square ID of the coin to flip: {coin_to_flip_square_id}")
```

```
Square ID of the coin to flip: 32
```

Prisoner 1 has determined that if they flip the coin in square above, then when Prisoner 2 decodes the binary value of the board, he will read the number of the square containing the key.

Next, Prisoner 1 will go ahead and flip the coin.

In [10]:
```
def flip_coin_for_new_chessboard_state(board: pd.DataFrame, square_id: int):

    # Figure out where the target square_id is on the board
    row_location = int(square_id/8)
    column_location = square_id % 8
    coin = board.iloc[row_location, column_location]

    # Flip the coin and return a new chessboard state
    new_board = board.copy()
    if (coin == 1):
        # Coin is heads
        new_board.iloc[row_location, column_location] = 0
    else:
        # Coin is tails
        new_board.iloc[row_location, column_location] = 1

    return new_board

chessboard_after_coin_flip = flip_coin_for_new_chessboard_state(chessboard, c

print(f"Prisoner 1 flips the coin in square {coin_to_flip_square_id}, then le
```

```
Prisoner 1 flips the coin in square 32, then leaves.
```

Prisoner 1 Leaves, Prisoner 2 Enters

Prisoner 2 walks in to see the following chessboard

In [11]:
```
chessboard_after_coin_flip
```

Out[11]:

	0	1	2	3	4	5	6	7
0	1	1	0	1	0	0	1	0
1	0	1	1	1	1	0	1	0
2	0	1	0	0	0	0	0	0
3	1	1	0	1	1	1	0	1
4	1	0	1	1	0	1	1	0
5	0	1	1	1	1	0	1	1
6	0	1	0	0	1	1	0	0
7	0	1	0	1	1	0	0	1

Prisoner 2 decodes the binary value from the chessboard. Using the agreed-upon sections, they count the number of coins which are heads-up in each section. Depending on if each number was even or odd, they write either 1 or 0 as the next digit in the 6-digit binary number.

In [12]:

```
new_binary_value = read_binary_from_chessboard(chessboard_after_coin_flip, se
print(f'New chessboard binary: "{new_binary_value}"')
```

```
New chessboard binary: "010011"
```

Next, Prisoner 2 finds which square the binary number represents:

In [13]:

```
square_picked_by_prisoner_2 = binary_to_int(new_binary_value)
print(f'Prisoner 2 selects square number {square_picked_by_prisoner_2}')
```

```
Prisoner 2 selects square number 19
```

In [14]:

```
if square_picked_by_prisoner_2 == square_where_warden_put_key:
    #
    #   _____  _    _   _____   _____   ______   _____   _____   _
    #  / ____|| |  | | / ____| / ____| |  ____| / ____| / ____| | |
    # | (___  | |  | || |     | |     | |__   | (___  | (___  | |
    #  \___ \ | |  | || |     | |     |  __|   \___ \  \___ \ | |
    #  ____) || |__| || |____ | |____ | |____  ____) | ____) ||_|
    # |_____/  \____/  \_____| \_____||______||_____/ |_____/ (_)
    #
    print(f"Success! Prisoner 2 finds the key in square number {square_picked_
else:
    print("Whoops, wrong square... The warden locks both prisoners back up.")
```

```
Success! Prisoner 2 finds the key in square number 19 and both prisoners go
free!
```

Here's a recap using all the calculated values:

In [15]:

```
print("RECAP:")
print(f'- The warden hides the key in sqaure number "{square_where_warden_put_
print(f'- Prisoner 1 reads the starting chessboard binary and sees "{initial_
print(f'- Prisoner 1 flips the coin in square number "{coin_to_flip_square_id
print(f'- This coin flip caused the chessboard binary to change from "{initia
print(f'- Prisoner 2 enters and converts the chessboard binary "{new_binary_v
print(f'- Prisoner 2 selects square "{square_picked_by_prisoner_2}" and finds
```

```
RECAP:
- The warden hides the key in sqaure number "19" while Prisoner 1 watches
- Prisoner 1 reads the starting chessboard binary and sees "010010"
- Prisoner 1 flips the coin in square number "32"
- This coin flip caused the chessboard binary to change from "010010" to
"010011"
- Prisoner 2 enters and converts the chessboard binary "010011" to "19"
- Prisoner 2 selects square "19" and finds the key!
```

Testing the Solution

In [16]:

```
def test_puzzle_solution(
        get_chessboard_of_coins, get_chessboard_square_ids,
        binary_to_int, int_to_6bit_binary,
        sections, read_binary_from_chessboard, get_warden_key_placement,
        find_which_coin_to_flip, flip_coin_for_new_chessboard_state
    ) -> bool:
    chessboard = get_chessboard_of_coins()
    chessboard_square_ids = get_chessboard_square_ids()
    initial_binaryStringValue = read_binary_from_chessboard(chessboard, secti

    square_where_warden_put_key = get_warden_key_placement()
    target_binaryStringValue = int_to_6bit_binary(square_where_warden_put_key

    coin_to_flip_square_id = find_which_coin_to_flip(chessboard, square_where
    chessboard_after_coin_flip = flip_coin_for_new_chessboard_state(chessboar
    newBinaryStringValue = read_binary_from_chessboard(chessboard_after_coin_
    square_picked_by_prisoner_2 = binary_to_int(newBinaryStringValue)

    return square_picked_by_prisoner_2 == square_where_warden_put_key

def batchtest_puzzle_solution(count: int):
    success = 0
    failure = 0
    for i in range(count):
        solution_successful = test_puzzle_solution(
            get_chessboard_of_coins, get_chessboard_square_ids,
            binary_to_int, int_to_6bit_binary,
            sections, read_binary_from_chessboard, get_warden_key_placement,
            find_which_coin_to_flip, flip_coin_for_new_chessboard_state
        )
        if (solution_successful):
            success += 1
        else:
            failure += 1
    return [success, failure]

success, failure = batchtest_puzzle_solution(100)
print(f"Result of {success + failure} random test runs:")
print(f"Success: {success}, failure: {failure}")
```

```
Result of 100 random test runs:
Success: 100, failure: 0
```