Monopoly and Markov Chains

The goal of this project is to use Markov chains to model the game of Monopoly. We aim to answer some simple questions about the game:

- What are the locations visited the most often?
- After how many turns is the game at an equilibrium?
- Is there a better strategy? Should one alway build hotels? Can we stick to houses?

To ease the analysis we will simplify some of the rules by simplifying the rules to get out of jail, and neglect the effect of the "Chance" and the "Community chest" cards. We will also consider players that always stick to the same strategy.

Most of the following analysis were motivated by two articles from Ian stewarts in the Scientific American, (reproductions from the articles are linked here or available on the course page).

Summary of the game

Those short explanations of the game are taken from this website. You can as well check the rules on the wikipedia page (english), german).

Rules of the game

Dice Rolls:

- A player rolls **a pair of dice** and moves his board piece (token) clockwise around the board. The number of board spaces he moves is equal to the sum of the dice.
- If he stops on a "Chance" or "Community Chest" board space, he picks up a card from the indicated stack, and if instructed, moves his token to a new location.
- If he had doubles, he repeats this process.
- However, if he has three doubles in a row, he instead goes directly to Jail.

Going to Jail:

If a player is instructed to go to Jail (at any point during his turn), his turn ends regardless of the doubles status. The player will stay at most 3 turns in Jail. He can go out of jail in one of the following manner:

- Paying is way out (50$). In this case the token goes to "Just visiting" and the player can throw is dice as usual.
- Using a Chance card. This is similar as paying 50$.
- Rolling the dice. If there is a double, the player can go out and moves forwards of the number obtained with the dice. Note that in this case the player cannot roll the dice again (the double was "used" to get out of jail).

After the third turn, if the player does not get a double, he *must* pay 50$ and moves of the number of spaces obtained with the dice.

1. Setting up the Markov Chain

In this first part we will set up the parameters for a Markov chain that models the movements of the token of the board from turn to turn. We make the following simplification to the rules:

- Double with the Dice does not allow a second roll.
- We do not consider the cards in the Chance and Community chest slots.
- When getting into Jail, the player always pays 50$ at the first turn to get out of jail.

We start by constructing the markov chain.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#Sorry the street names are in French!
#you can replace it with the colors as in the table file
squares = [
"Départ",
"Boulevard de Belleville",
"Caisse de Communauté-1",
"Rue Lecourbe",
"Impôts sur le Revenu",
"Gare Montparnasse",
"Rue de Vaugirard",
"Chance-1",
"Rue de Courcelles",
"Avenue de la République",
"Visite de Prison",
"Boulevard de la Villette",
"Compagnie de Distribution d Électricité",
"Avenue de Neuilly",
"Rue de Paradis",
"Gare de Lyon",
"Avenue Mozart",
"Caisse de Communauté-2",
"Boulevard Saint-Michel",
"Place Pigalle",
"Parc Gratuit",
"Avenue Matignon",
"Chance-2",
"Boulevard Malesherbes",
"Avenue Henri-Martin",
"Gare du Nord",
"Faubourg Saint-Honoré",
"Place de la Bourse",
"Compagnie de Distribution des Eaux",
"Rue La Fayette",
"Allez en Prison",
"Avenue de Breteuil",
"Avenue Foch",
"Caisse de communauté-3",
"Boulevard des Capucines",
"Gare Saint-Lazare",
"Chance-3",
"Avenue des Champs-Élysées",
"Taxe de Luxe",
"Rue de la Paix"]
```

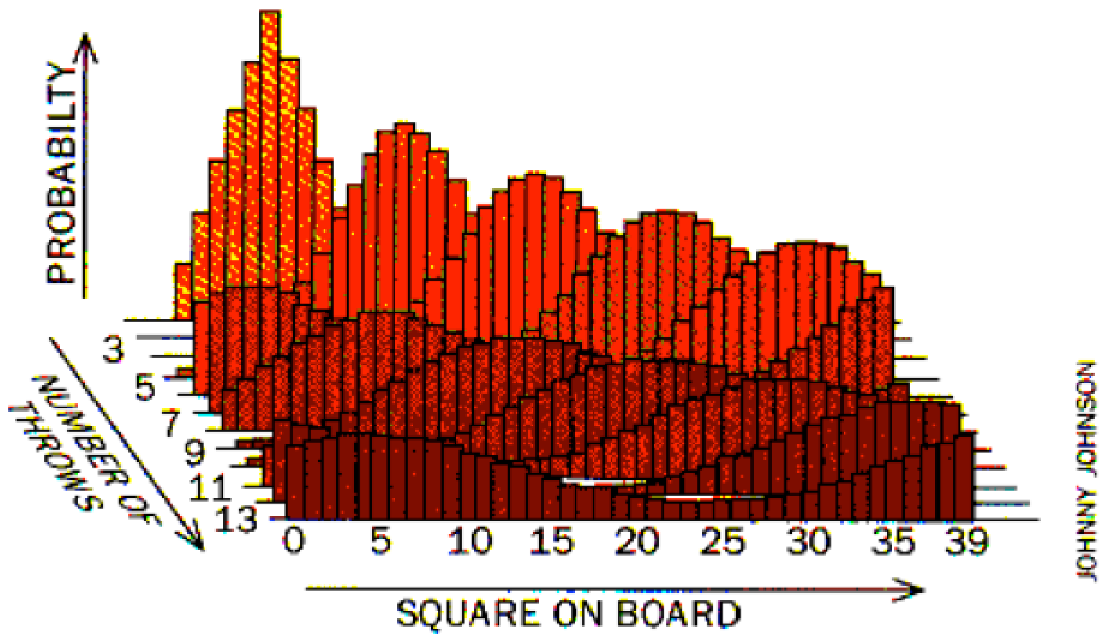
```
squares += ["Prison"]
nsquares = len(squares)

##Some easy links
dict_squares = dict((x, i) for i,x in enumerate(squares))
iprison = dict_squares["Allez en Prison"] ## This square is in fact never visited
ivisite = dict_squares["Visite de Prison"]

#Construct a 12 element array with all dice probabilities
TwoDices = np.zeros(12, dtype=float)
# TwoDices = ...
FirstDices = np.zeros(6, dtype=float)
SecondDices = np.zeros(6, dtype=float)
FirstDices[:] = 1/6
SecondDices[:] = 1/6

dice_combinations = set()
for i in range(len(FirstDices)):
    for j in range(len(SecondDices)):
        dice_combinations.add(','.join(map(str, (i,j))))

for x in range(len(TwoDices)):
    for y, index in enumerate(dice_combinations):
        firstDice_idx = int(index.split(',')[0])
        secondDice_idx = int(index.split(',')[1])
        if firstDice_idx+secondDice_idx+2 == x+1:
            TwoDices[x] = TwoDices[x] + (FirstDices[firstDice_idx]*SecondDices[secondD


## The initial probability. Note that we set a vector to allow matrix product computat
## afterwards.
pi=np.zeros((1, nsquares))
pi[0,0] = 1
##Set up the Transition Matrix
A = np.zeros((nsquares, nsquares), dtype=float)

###YOUR CODE HERE
### A = ...

for i in range(len(A)):
    dice_idx = 0
    for j in range(i+1,len(A)):
        if dice_idx < len(TwoDices):
            A[i,j] = TwoDices[dice_idx]
            dice_idx+=1
        else:
            break

for i in range(len(A)-len(TwoDices),len(A)):
    dice_idx = len(TwoDices)-(len(TwoDices)-(len(A)-i)+1)
    for j in range(len(TwoDices)-(len(A)-i)+1):
        A[i,j] = TwoDices[dice_idx]
        #print('A[i,j] bottom top 12:',i, j, A[i,j])
        dice_idx+=1


#Since "Allez en Prison" is never visited then i can set it to 0 in transition matrix
A[29,:] = 0
A[:,29] = 0
```

In [2]:
```
print(A[38,:])
print(squares)
```

```
[0.05555556 0.08333333 0.11111111 0.13888889 0.16666667 0.13888889
 0.11111111 0.08333333 0.05555556 0.02777778 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.02777778]
['Départ', 'Boulevard de Belleville', 'Caisse de Communauté-1', 'Rue Lecourbe', 'Impô
ts sur le Revenu', 'Gare Montparnasse', 'Rue de Vaugirard', 'Chance-1', 'Rue de Courc
elles', 'Avenue de la République', 'Visite de Prison', 'Boulevard de la Villette', 'C
ompagnie de Distribution d Électricité', 'Avenue de Neuilly', 'Rue de Paradis', 'Gare
de Lyon', 'Avenue Mozart', 'Caisse de Communauté-2', 'Boulevard Saint-Michel', 'Place
Pigalle', 'Parc Gratuit', 'Avenue Matignon', 'Chance-2', 'Boulevard Malesherbes', 'Av
enue Henri-Martin', 'Gare du Nord', 'Faubourg Saint-Honoré', 'Place de la Bourse', 'C
ompagnie de Distribution des Eaux', 'Rue La Fayette', 'Allez en Prison', 'Avenue de B
reteuil', 'Avenue Foch', 'Caisse de communauté-3', 'Boulevard des Capucines', 'Gare S
aint-Lazare', 'Chance-3', 'Avenue des Champs-Élysées', 'Taxe de Luxe', 'Rue de la Pai
x', 'Prison']
```

2. After a few game turns.

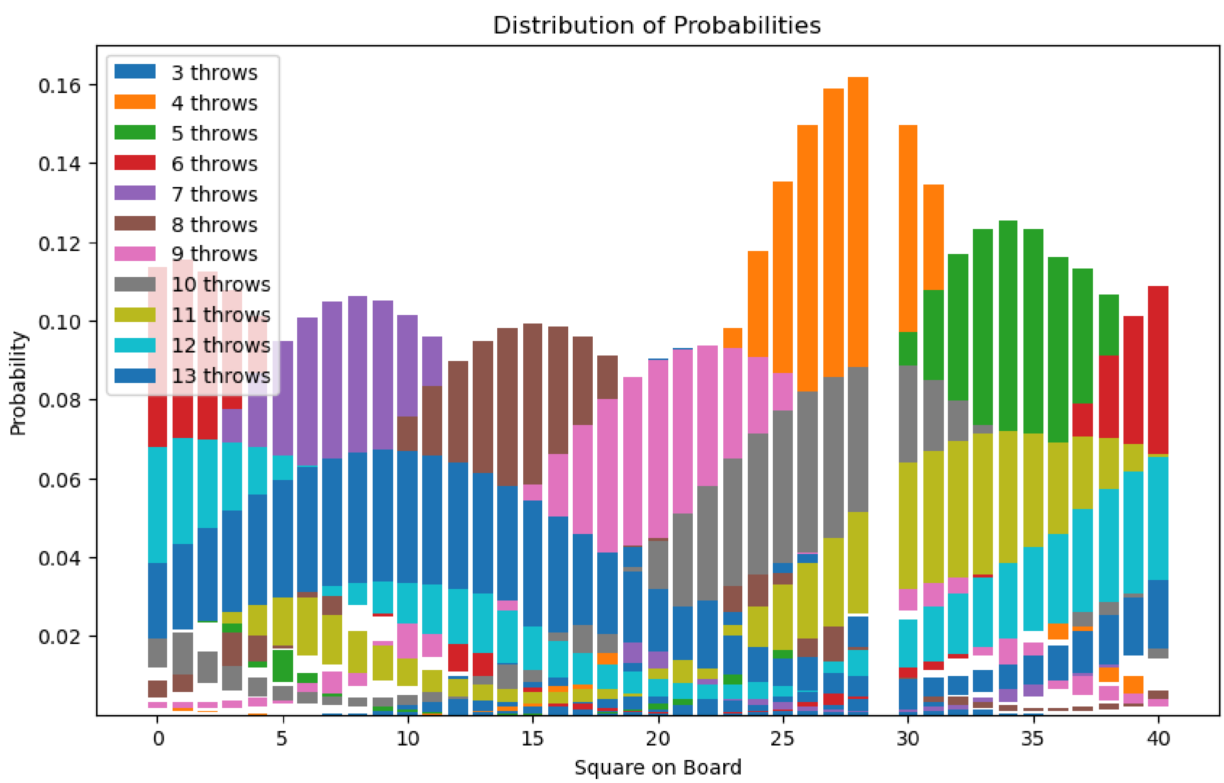
Compute the probability of occupation on the board after a few turns of games. You will produce a set of histograms in the like of the one below



DISTRIBUTION OF PROBABILITIES *over the 40 squares is shown above, as well as how it changes at each throw of a die. The height of each bar gives the probability of landing on the corresponding square. The graphs for throws 2 through 13 are stacked from back to front.*

In [3]:
```
def probability(throw):
    prob_dist = np.zeros(len(A),dtype = float)
    for i in range(throw):
        if i == 0:
            prob_dist = np.matmul(pi, A)
        else:
            prob_dist = np.matmul(prob_dist, A)
    return prob_dist

board_turn_1 = probability(1).flatten()
board_turn_2 = probability(2).flatten()
board_turn_3 = probability(3).flatten()
board_turn_4 = probability(4).flatten()
board_turn_5 = probability(5).flatten()
board_turn_6 = probability(6).flatten()
board_turn_7 = probability(7).flatten()
board_turn_8 = probability(8).flatten()
```

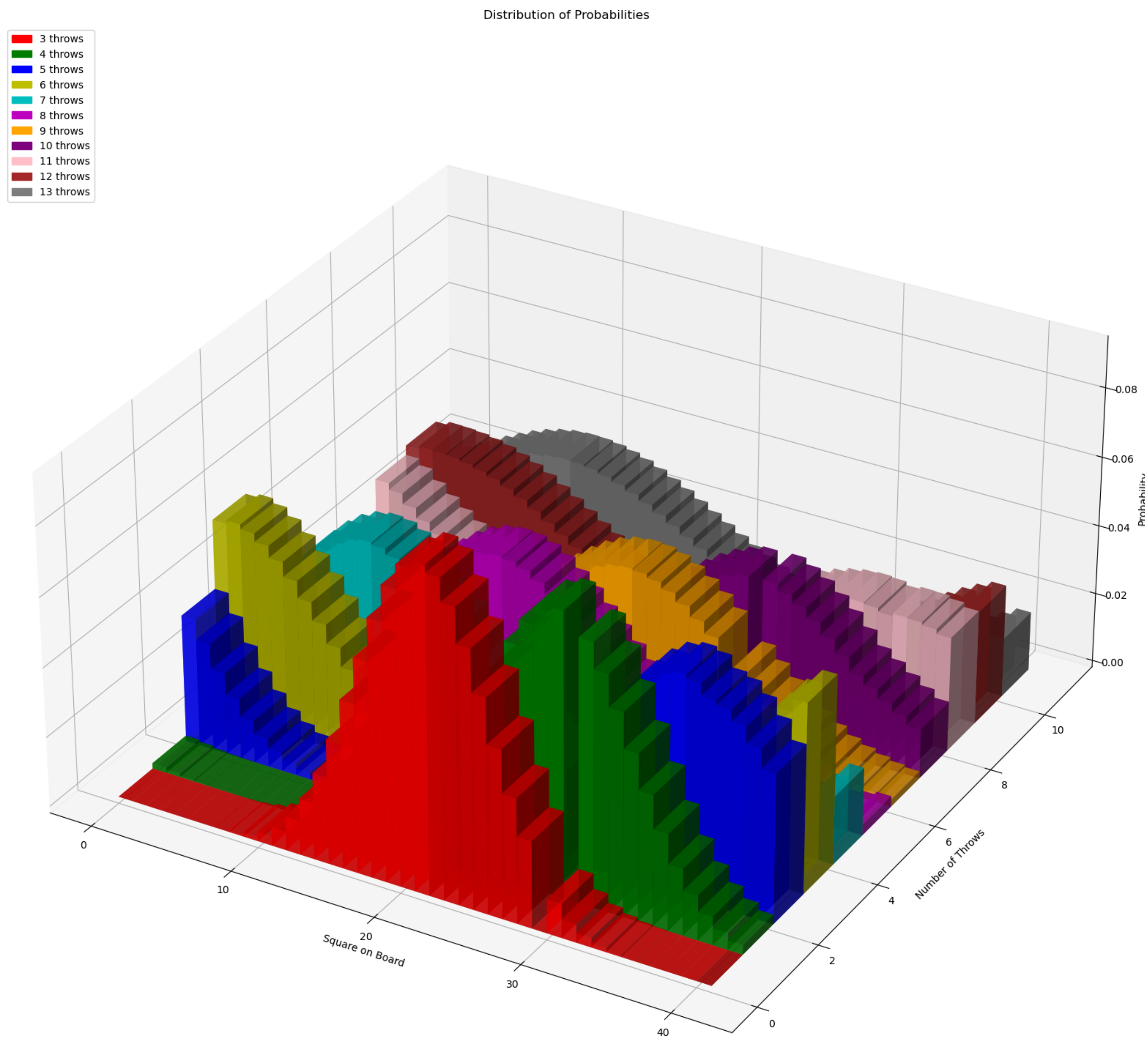
```python
board_turn_9 = probability(9).flatten()
board_turn_10 = probability(10).flatten()
board_turn_11 = probability(11).flatten()
board_turn_12 = probability(12).flatten()
board_turn_13 = probability(13).flatten()

##Your plotting command here
x = np.arange(A.shape[0])
plt.figure(figsize=(10,6))
plt.bar(x, board_turn_3, label = '3 throws')
plt.bar(x, board_turn_4, bottom=board_turn_4, label = '4 throws')
plt.bar(x, board_turn_5, bottom=board_turn_5, label = '5 throws')
plt.bar(x, board_turn_6, bottom=board_turn_6, label = '6 throws')
plt.bar(x, board_turn_7, bottom=board_turn_7, label = '7 throws')
plt.bar(x, board_turn_8, bottom=board_turn_8, label = '8 throws')
plt.bar(x, board_turn_9, bottom=board_turn_9, label = '9 throws')
plt.bar(x, board_turn_10, bottom=board_turn_10, label = '10 throws')
plt.bar(x, board_turn_11, bottom=board_turn_11, label = '11 throws')
plt.bar(x, board_turn_12, bottom=board_turn_12, label = '12 throws')
plt.bar(x, board_turn_13, bottom=board_turn_13, label = '13 throws')


plt.xlabel('Square on Board')
plt.ylabel('Probability')
plt.legend()
plt.title('Distribution of Probabilities')
plt.show()
```



In [4]:

```python
x = np.arange(len(A))
board_turns = [
    board_turn_3,
    board_turn_4,
    board_turn_5,
    board_turn_6,
    board_turn_7,
```

```python
    board_turn_8,
    board_turn_9,
    board_turn_10,
    board_turn_11,
    board_turn_12,
    board_turn_13
]

fig = plt.figure(figsize=(50,20))
ax = fig.add_subplot(111, projection='3d')

colors = ['r', 'g', 'b', 'y', 'c', 'm', 'orange', 'purple', 'pink', 'brown', 'gray']
bottom = np.zeros(41)

# Plot each round
for i, (dz, color) in enumerate(zip(board_turns, colors)):
    ax.bar3d(x, i * np.ones_like(x), bottom, 1, 1, dz, color=color, alpha=0.7)

ax.set_xlabel('Square on Board')
ax.set_ylabel('Number of Throws')
ax.set_zlabel('Probability')
ax.set_title('Distribution of Probabilities')


handles = [plt.Rectangle((0, 0), 1, 1, color=color) for color in colors]
labels = [f'{i+3} throws' for i in range(len(board_turns))]
ax.legend(handles, labels, loc='upper left')

ax.set_box_aspect([2, 2, 1])

plt.show()
```




```
In [ ]: import matplotlib.pyplot as plt
        # Line plot of state transitions
        plt.figure(figsize=(12, 6))
        plt.plot(states, drawstyle='steps-pre')
        plt.xlabel('Step')
        plt.ylabel('State')
        plt.title('State Transitions in the Markov Chain')
        plt.show()

        # Histogram of state frequencies
        unique, counts = np.unique(states, return_counts=True)
        plt.bar(unique, counts / num_steps, align='center')
        plt.xlabel('State')
        plt.ylabel('Frequency')
        plt.title('State Frequencies in the Markov Chain')
        plt.xticks(unique)
        plt.show()
```

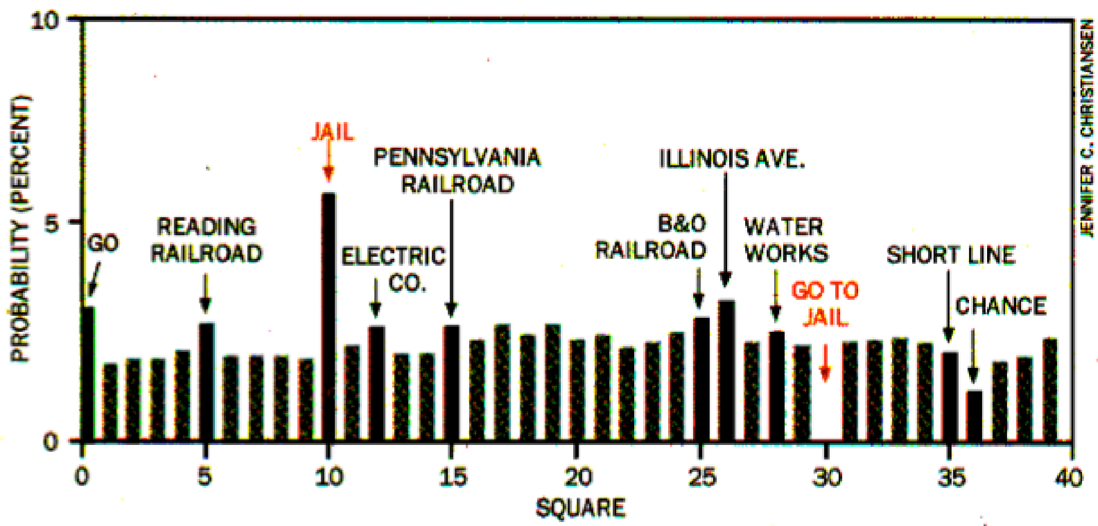
3.a Computing the stationary distribution

Compute the stationary distribution μ of the Markov Chain, by using either one of the following properties:

- $\lim_{n \to \infty} A^n = \mu$
- $\mu \cdot A = \mu$. In other word μ is the eigenvector associated with the eigenvalue 1.

Both can be obtained with the linear algebra functions of numpy.

Verify that you get something similar to this image (this will not be exactly the same as it was obtained from a more detailled model):

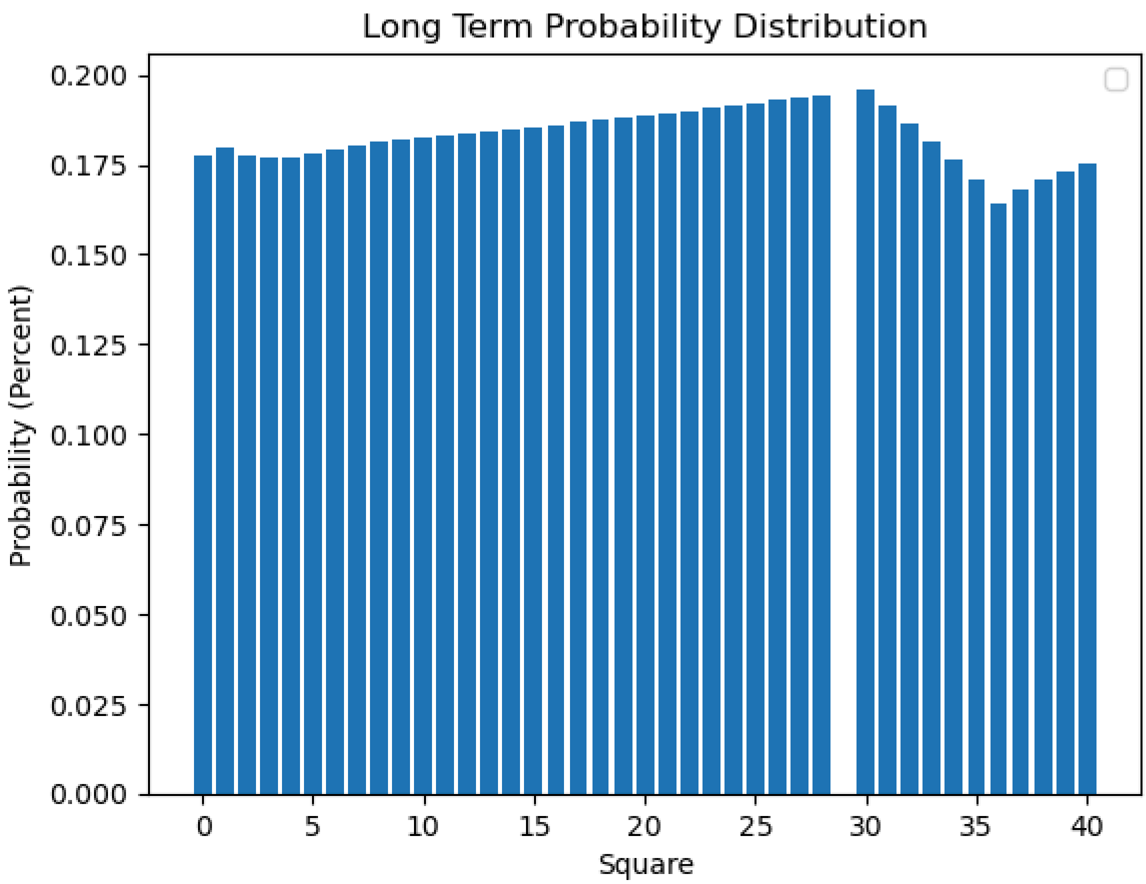


LONG-TERM PROBABILITY DISTRIBUTION
shows that the Jail square is most likely to be occupied.

In [5]:
```
#np.linalg.matrix_power is one function you can use

#
```

```python
A_pow = np.linalg.matrix_power(A, 100)
A_longterm = np.matmul(pi, A_pow).flatten()

x = np.arange(len(A_longterm))
plt.bar(x, A_longterm*100)
plt.xlabel('Square')
plt.ylabel('Probability (Percent)')
plt.legend()
plt.title('Long Term Probability Distribution')
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start w
ith an underscore are ignored when legend() is called with no argument.
```



Can you explain why some locations are more visited than others?

your answer here

because the roll of the dice. certain number has higher probabilities to show up and it cause certain tails have higher probabilities also. the traffice around jail is the highest because after 4 times rolling the dice, it has higher probabilities landing around these areas

3b. Convergence to the stationnary distribution

Use the successive powers of the matrix to determine the number of turns after which the chain converged to the stationary distribution (*e.g.* when the difference between the distribution of the rows of the matrix and the stationnary distribution is below 1%)

In [6]:
```
# Your code here

A_conv = np.eye(A.shape[0])

max_iter = 100
tolerance = 1/100

for n in range(1, max_iter + 1):
    A_pow = np.linalg.matrix_power(A, n)

    if np.max(np.abs(A_pow - A_conv)) < tolerance:
        print(f'Convergence achieved after {n} iterations.')
        break

    A_conv = A_pow

print("Limit matrix:")
print(A_conv)
```

```
Convergence achieved after 20 iterations.
Limit matrix:
[[0.01504841 0.01640058 0.01751057 ... 0.01125565 0.0124781  0.01374152]
 [0.01375677 0.01509023 0.01623391 ... 0.01018235 0.01130381 0.01249472]
 [0.01251151 0.01379999 0.01494502 ... 0.00922052 0.01022221 0.01131874]
 ...
 [0.0189067  0.02014782 0.02096556 ... 0.01491459 0.01629997 0.01762773]
 [0.01765391 0.01896021 0.01990611 ... 0.01364687 0.0150038  0.01633747]
 [0.01635752 0.01770086 0.01874469 ... 0.01241851 0.01372034 0.01503203]]
```

3c. Evaluating investments

We provide in attachment a table with all the incomes associated to the various properties of the board. Let's see how we can use that information to understand how to evaluate the various real estate investments on the board.

For each square k of the monopoly, we thus know the cost c_k of the property, the price h_k of an house and the rent $r_{k,h}$ with h house. The total cost of a property with h house is $c_{k,h} = c_k + h \cdot h_k$

1. Verify that the expected income per opponent turn on square k is $i_{k,h} = \mu_k \cdot r_{k,h}$ and compute this value for all square and all number of houses. For instance a property where an opponent as 5% chance of landing with a rent of 100$ provides an expected income per turn of 5\$.

2. Now an interesting quantity is the expected number of opponents turns before we are able to return on our investment with buying the property. For instance if the property mentionned in q.1 costed 50$, given that we expect to earn 5\$ per turn with no house, our

average waiting time before returning on our investment will be after $\frac{50}{5} = 10$ opponent turns. Verify that the general formula for this value is:

$$e_{k,h} = \frac{c_{k,h}}{i_{k,h}}$$

3. compute $e_{k,h}$ for all square and all number of houses and check the most rentable properties, as a function of the number of houses.

4. (optional). Compute the expected number of turns to get a return of investment on a given color.

1. A classical recommendation for Monopoly is to buy orange properties, can you back up this claim?

In [7]:
```
print(A_conv)
print(np.sum(A_conv[:,0]))
```

```
[[0.01504841 0.01640058 0.01751057 ... 0.01125565 0.0124781  0.01374152]
 [0.01375677 0.01509023 0.01623391 ... 0.01018235 0.01130381 0.01249472]
 [0.01251151 0.01379999 0.01494502 ... 0.00922052 0.01022221 0.01131874]
 ...
 [0.0189067  0.02014782 0.02096556 ... 0.01491459 0.01629997 0.01762773]
 [0.01765391 0.01896021 0.01990611 ... 0.01364687 0.0150038  0.01633747]
 [0.01635752 0.01770086 0.01874469 ... 0.01241851 0.01372034 0.01503203]]
0.5853179596537863
```

In [8]:
```
# read monopoly square data in Monopoly_square_data.csv
import pandas as pd

df = pd.read_csv('Data/Monopoly_squares_data.csv',header=0)

# Compute i_k_h
#Compute expected income per opponent turn on square k for all square and all number c
rent = np.zeros((len(A)-1,4), dtype = float)
for i in range(1,len(A_conv)-1):
    for j in range(4):
        rent[i,j] = np.sum(A_conv[:,i])*df.iloc[i,j+3]


# Compute e_k_h
num_columns = min(rent.shape[1], 4)
opponent_turns = np.zeros((len(rent), num_columns), dtype = float)
for i in range(num_columns):
    opponent_turns[:,i] = df['Cost'] / rent[:,i]

print('Number of turns:',opponent_turns)
```

```
Number of turns: [[        nan         nan         nan         nan]
 [25.21629775 10.0865191   3.36217303  1.12072434]
 [        nan         nan         nan         nan]
 [12.75990109  5.10396044  1.70132015  0.56710672]
 [        nan         nan         nan         nan]
 [        nan 13.48687367  6.74343684  3.37171842]
 [13.93818722  5.57527489  1.85842496  0.61947499]
 [        nan         nan         nan         nan]
 [13.72538734  5.49015493  1.83005164  0.61001721]
 [12.29741153  4.91896461  1.96758585  0.65586195]
 [        nan         nan         nan         nan]
 [11.40686407  4.56274563  1.52091521  0.50697174]
 [ 3.48202688         nan         nan         nan]
 [11.34154982  4.53661993  1.51220664  0.50406888]
 [10.76861916  4.30744767  1.43581589  0.51689372]
 [        nan 12.87986114  6.43993057  3.21996528]
 [10.31888399  4.1275536   1.44464376  0.525325  ]
 [        nan         nan         nan         nan]
 [10.27116638  4.10846655  1.43796329  0.52289574]
 [ 9.96802543  3.98721017  1.44989461  0.53162802]
 [        nan         nan         nan         nan]
 [ 9.71476181  3.88590473  1.3989257   0.49961632]
 [        nan         nan         nan         nan]
 [ 9.68018352  3.87207341  1.39394643  0.49783801]
 [ 9.48581809  3.79432724  1.26477575  0.5059103 ]
 [        nan 12.622093    6.3110465   3.15552325]
 [ 9.30311972  3.72124789  1.24041596  0.51167158]
 [ 9.28169765  3.71267906  1.23755969  0.51049337]
 [ 3.35745461         nan         nan         nan]
 [        inf         inf         inf         inf]
 [        nan         nan         nan         nan]
 [ 9.21778596  3.68711439  1.22903813  0.53258319]
 [ 9.45764444  3.78305778  1.26101926  0.54644168]
 [        nan         nan         nan         nan]
 [ 9.90691581  3.6985819   1.23286063  0.55478729]
 [        nan 14.32722627  7.16361313  3.58180657]
 [        nan         nan         nan         nan]
 [ 9.06604014  3.62641606  1.26924562  0.57692983]
 [        nan         nan         nan         nan]
 [ 7.01806332  4.0103219   1.40361266  0.63800576]]
```

4. Improving the model

By using the article provided on the moodle (also reproduced at this adress), design a more realistic transition matrix that would take into account double on the dice as well as the traditional getting out of jail.

You will integrate the following ingredients:

- Double allow the player to do a second roll.
- After three double the player goes to jail.
- To simplify, we also hypothesize that the player cannot buy properties on the intermediate squares in a serie of doubles.
- Player always try to get out of jail by throwing the dice up to three times

- (optional) Account for the cards in Chance and community chests that redirect the token to other cases (more details are given in the article and you can look online what are the statistics for the cards)

To take this information into account in the Markov chain you can increase the state space by consider for each square the number of doubles you did when you landed on it. So now each square will correspond to 3 states in the Markov chain, the ID of the square, together with an additional field that stores the information about the number of doubles.

For instance imagine you start from (0, "Départ"):

```
- if you throw two 3, that's a double and you will land on square 6
("Rue de Vaugirard") while keeping the information that you did a
double so you will be in state (1, "Rue de Vaugirard").
- if you throw a 2 and a 4, that's not a double and you will land on
state (0, "Rue de Vaugirard").
```

In [9]:
```
## Design the improved transition matrix here
```

Recompute the values of question 2 and 3a with this new matrix and comment on possible difference.

In [10]:
```
## Compute powers of the matrix and the stationary distribution
```