In [1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("HW 2.ipynb")
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Stat 198: Introduction to Quantitative Finance DeCal

Stat 198 is hosted by Traders at Berkeley.

Course Website: https://traders.berkeley.edu/education

Homework #2

In this homework, we'll cover a range of asset classes and how to deal with their pricing models. Most notably, we will focus on how options are priced in relation to their underlying asset, hitting home at why options are a type of **derivative**.

Learning Objectives:

- Bond Pricing
- Options Pricing
- Monte Carlo Simulation

Question 1

We will start off by filling in a function that will help us price any type of bond, given the parameters:

- face: face value of the bond
- coupon_rate: annual coupon rate as percentage
- coupons_per_year: number of coupon payments per year
- years_to_maturity: time to maturity in years
- yield_to_maturity: yield to maturity as a percentage

Calculate the price of a bond based on its face value, coupon rate, years to maturity, coupons per year, and yield to maturity.

Input:

- face: The face value of the bond.
- coupon_rate: The annual coupon rate of the bond.
- years_to_maturity: The number of years until the bond matures.
- coupons_per_year: The number of coupon payments per year.
- yield_to_maturity: The yield to maturity of the bond.

Output:

- bond_price: The present value of the bond.

As a refresher, to calculate the price of a bond, you'll want to:

- calculate each coupon payment
- calculate the number of coupon payments
- calculate the discount rate and cash flow per payment
- calculate the present value of each payment by discounting the future cash flow
- make sure to include the face value for the final payment at maturity
- calculate the sum of all the present values of the bond's future cash flows

See also: https://www.omnicalculator.com/finance/bond-price#how-to-calculate-the-bond-price-the-bond-price-formula

In [3]:
```
def bond_price(face, coupon_rate, coupons_per_year, years_to_maturity, yielc
    present_value = 0

    for x in range(1, years_to_maturity + 1):
        for y in range(1, coupons_per_year + 1):
            cash_flow = face * coupon_rate / coupons_per_year
            if x == years_to_maturity and y == coupons_per_year:
                cash_flow += face
#             yearly_value_divisor = (1 + yield_to_maturity) ** x
            yearly_value_divisor = (1 + yield_to_maturity / coupons_per_year
            present_value += cash_flow/yearly_value_divisor
    return present_value

bond_price(face = 700, coupon_rate = 0.09, yield_to_maturity = 0.12, years_t
```

Out[3]: 663.050658413677

In [4]:
```
grader.check("q1")
```

Out[4]:

q1 passed! 🚀

Question 2

Next, we will price options by going through a game.

In this game, you are trying to predict the sum of 10 cards that are drawn randomly from a standard deck of 52 cards. In this problem, we will determine the EV of the game at any point in time, along with fair options prices for both call and put options for a set strike price at any point in the game state.

Question 2.1: Expectation

We will start by determing the expected value of the game at any point in the game state. Below are a couple examples of game states.

For the purpose of this game, we will be considering an Ace as a 1, a Jack as a 11, a Queen as an 12, and a King as a 13.

Example 1

- Starting game state: []
- Game state after 3 cards have been revealed: [1, 4, 9]
- Game state after 7 cards have been revealed: [1, 4, 9, 5, 13, 9, 10]
- Ending game state: [1, 4, 9, 5, 13, 9, 10, 1, 3, 11]
  - Final Score: 1 + 4 + 9 + 5 + 13 + 9 + 10 + 1 + 3 + 11 = 66

Example 2

- Starting game state: []
- Game state after 2 cards have been revealed: [12, 12]
- Game state after 8 cards have been revealed: [12, 12, 12, 12, 1, 1, 1, 1]
- Ending game state: [12, 12, 12, 12, 1, 1, 1, 1, 7, 8]
  - Final Score: 12 + 12 + 12 + 12 + 1 + 1 + 1 + 1 + 7 + 8 = 67

Implement the function below to determine the EV of the final score of a game state. As a starting point, the EV of an empty game state should be 70.

In [5]:
```
# assume that all inputs will be valid game states
```

In [6]:
```
def expected_final_score(game_state):
    exp_val_x4 = ((1+2+3+4+5+6+7+8+9+10+11+12+13)) * 4
    curr_total = exp_val_x4 - sum(game_state)
    expected_card = curr_total / (52 - len(game_state))
    exp_future = (10 - len(game_state)) * expected_card
    score = sum(game_state) + exp_future
    return score

expected_final_score([12, 12, 12, 12, 1, 1, 1, 1])
```

Out[6]: 66.18181818181819

In [7]:
```
grader.check("q2.1")
```

Out[7]:

q2.1 passed! 🙌

Question 2.2: Call Options Pricing

Next we'll dive into options pricing using monte carlo simulation. We'll go about this by pricing options with a set strike price of 70 to simplify the calculations.

We'll start by creating a call option for this game. We'll try to create a function, that given a specific game state, can return the fair price of a call option with strike price 70 on the final score of the game.

We can determine the options price by using the concept of EV. We start by making a distribution of the final scores of a game through simulation. For example, if we start with an empty game state, we want to run at least n = 1,000 sample games and determine the distribution of final scores to create a probability distribution. We can do this by finding the frequency of a specific score and dividing it by our n. While you should use at least n = 1,000, I would heavily recommend at least n = 10,000 samples in your monte carlo simulation.

This concept extrapolates to a given game state. If you have k cards drawn, you only need to simulate the final score given the 10 - k extra draws needed per simulation.

After you have a probability distribution of final scores, you can determine the EV of a call option with strike price of 70. As a reminder, if the final score is 71, your payoff is max(71 - 70, 0) = 1. If the final score is 80, your payoff is max(80 - 70, 0) = 10. If the final score is 60, your payoff is max(60 - 70, 0) = max(-10, 0) = 0. Thus, you should calculate the EV of the final payoff of the option given your probability distribution of final scores and the payoffs of each score. Given this, you can ignore final scores below 70 since the payoff is 0 and this sets those terms in the EV calculation to 0.

In general, the form of the price is max(final_score - strike_price, 0).

hints and reminders

- make sure you are sampling **with** replacement from all possible cards in the deck
- when you are working with an existing game state, make sure to remove the cards that were already drawn from the possible space of cards that can be drawn in your simulation
- In theory, if you have a larger game state (more cards drawn), your simulation should run faster since you draw less cards per simulation
- make sure to determine the EV over possible payoffs, not final scores in a game

In [8]:
```
def call_option_game(game_state, strike_price=70):
    np.random.seed(1)
    payoff = 0
    base_deck = []
    for x in range(1, 14):
        for y in range(0 , 4):
            base_deck.append(x)

    for z in game_state:
        base_deck.remove(z)

#     my_deck = []
#     temp_val = 1
#     #ranges from 1 to 14
#     for x in range(1, len(base_deck)):
#         for y in range(base_deck[x]):
#             my_deck.append(temp_val)
#         temp_val += 1
    base_deck = np.array(base_deck)
#     print(base_deck)
    for trial in range(1000):
        rand_choice = np.random.choice(base_deck, 10 - len(game_state), repl
#         print(rand_choice)
        simulated_sum = sum(game_state) + sum(rand_choice)
        payoff += max(0, simulated_sum - strike_price)

    score = payoff/1000
    return score

call_option_game([12, 12, 12, 12, 11, 11, 11, 11, 10, 10], strike_price=100)
```

Out[8]: 12.0

In [9]:
```
grader.check("q2.2")
```

Out[9]:

q2.2 passed! 🎉

Question 2.3: Put Options Pricing

Now that we've created a pricing function for a call option of strike 70, we'll do the next logical step and create a pricing function for a put option of strike 70.

As a reminder, put options work as follows:

- if the final score is 69, the value of the option is max(70 - 69, 0) = 1
- if the final score is 60, the value of the option is max(70 - 60, 0) = 10
- if the final score is 80, the value of the option is max(70 - 80, 0) = 0

Thus, the general form of the price is max(strike_price - final_score, 0).

In [10]:
```
def put_option_game(game_state, strike_price=70):
    np.random.seed(1)
    payoff = 0
    base_deck = []
    for x in range(1, 14):
        for y in range(0 , 4):
            base_deck.append(x)

    for z in game_state:
        base_deck.remove(z)

#     my_deck = []
#     temp_val = 1
#     #ranges from 1 to 14
#     for x in range(1, len(base_deck)):
#         for y in range(base_deck[x]):
#             my_deck.append(temp_val)
#         temp_val += 1
    base_deck = np.array(base_deck)
#     print(base_deck)
    for trial in range(1000):
        rand_choice = np.random.choice(base_deck, 10 - len(game_state), repl
#         print(rand_choice)
        simulated_sum = sum(game_state) + sum(rand_choice)
        payoff += max(0, strike_price - simulated_sum)

    score = payoff/1000
    return score

put_option_game([1, 1, 1, 1, 2, 2, 2, 2, 3, 3], strike_price=60)
```

Out[10]: 42.0

In [11]:
```
grader.check("q2.3")
```

Out[11]: **q2.3** passed! 🙌

Question 3: Option Greeks Intuition

Now that we've priced options within a simple game setting, we'll build on this idea by generating some fake stock data and introducing some intuition behind option greeks along the way

Question 3.1: Geometric Brownian Motion

The famous **Black-Scholes** Options Pricing model uses **Geometric Brownian Motion** as the base for how asset prices behave.

We use GBM because:

- it creates a continuous model (for ease of calculations)
- uses log-normally distributed returns (as seen in empirical observations)
- is memoryless (for computational efficiency)
- and has a drift and volatility component (helps better simulate specific asset price paths using historical data)

A GBM applies a volatility scalar **sigma** to a **Wiener Process**, and adds a mean **mu**

The wiener process, W_t, is commonly used in physics to study brownian motion.

W_t starts at 0, has independent gaussian increments, and is continuous in the space t

$$W_0 = 0$$

$$W_{t+u} - W_t \sim N(0, u)$$

Formally, a stochastic process S_t follows a GBM if it satisfies

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t$$

In this formula, mu is our mean, and sigma is our volatility or variance

We can solve for S_t given some arbitrary initial S_0 using

$$S_t = S_0 e^{(\mu - \frac{\sigma^2}{2})t + \sigma W_t}$$

$$X = \ln \frac{S_t}{S_0} = (\mu - \frac{\sigma^2}{2})t + \sigma W_t$$

$$S_t = S_0 e^X$$

In english, we can generate an asset price using GBM by:

- finding the time values that we will simulate the asset price at
- generating n+1 standard iid standard normal random numbers (the base for the wiener process)
- using the random numbers to simulate a brownian motion process by taking the cumulative sum and scaling by our time increment dt
- computing GBM by multiplying our time increment, drift rate, and volatility with our brownian motion process
- computing the final prices by taking the exponential of our log prices X and scaling them by our starting asset price s0

Define a function named geometric_brownian_motion that takes the following parameters as input:

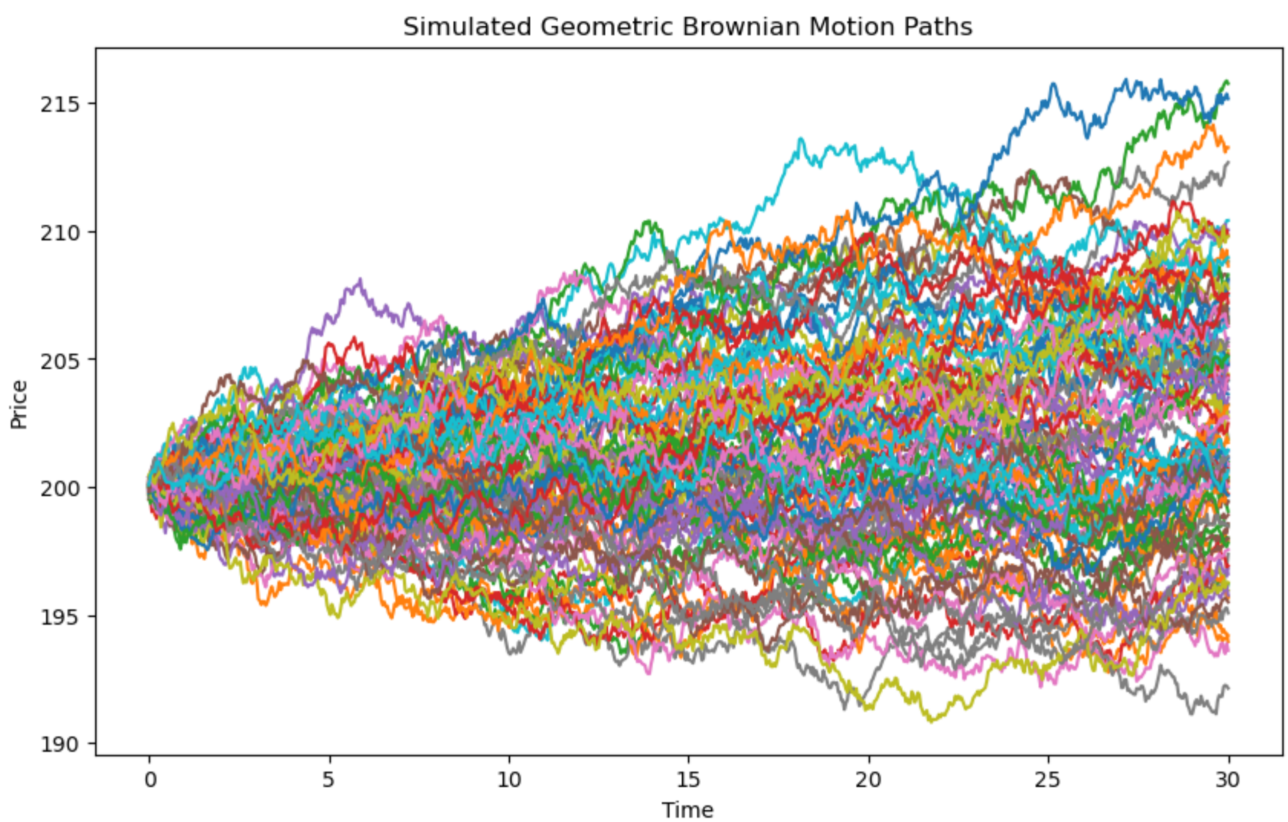
- s0 (initial_value): the initial value of the process (start price for the asset)
- mu (drift): the drift rate (stocks tend to trend upwards over time)
- sigma (volatility): the size of fluctuations (stocks move in wiggly lines)
- n (time steps): amount of time steps per T (for example: hours per day)
- T (total time): total amount of time to simulate (for example: days to simulate)
- num_paths: amount of paths to simulate
- plot: true or false based on if the function should plot or not

The function should return the simulated stock price paths as an array of arrays

In [12]:
```
def simulate_gbm_paths(s0, mu, sigma, n=24, T=30, num_paths=1000, plot=False
    res = []
    for x in range(num_paths):
        dt = 1/n
        iters = n * T
        t = np.linspace(0, T, iters+1)
        dW = np.random.normal(size = iters+1) * np.sqrt(dt)
        W = np.cumsum(dW)
        X = (mu - sigma**2/2) * t + sigma * W
        S = s0*np.exp(X)
        res.append(S)
    S = np.array(res)
    if plot:
        plt.figure(figsize=(10,6))
        for i in range(num_paths):
            plt.plot(t, S[i,:])

        plt.xlabel('Time')
        plt.ylabel('Price')
        plt.title('Simulated Geometric Brownian Motion Paths')
        plt.show()
    return S

simulate_gbm_paths(s0=200, mu=0.0005, sigma=0.005, n=24, T=30, num_paths=100
```



Out[12]:
```
array([[199.94669806, 200.08694564, 200.2573493 , ..., 204.29682175,
         204.11175502, 204.15586606],
        [199.92629357, 199.76210939, 199.38728335, ..., 198.19115217,
         197.87286505, 197.87498344],
        [199.8417896 , 200.09568129, 199.85882271, ..., 200.87424187,
         200.91061661, 200.98011914],
        ...,
        [200.38878686, 200.55514882, 200.50393573, ..., 198.60264766,
         198.49134764, 198.47355442],
        [200.44737863, 200.57509644, 200.44013111, ..., 209.55076217,
         209.76447318, 209.78586887],
        [200.08216839, 200.45685424, 200.55615215, ..., 200.50030347,
         200.36422763, 200.64612574]])
```

In [13]:
```
grader.check("q3.1")
```

Out[13]:

q3.1 passed! 🌈

Question 3.2: Call Option Pricing

Type your answer here, replacing this text.

In [14]:
```
def call_option_asset(asset, strike_price = 250, plot=True, hist=True, num_p
    S = simulate_gbm_paths(s0 = asset['s0'], mu = asset['mu'], sigma = asset
    prices = S[:, -1]
    payoffs = prices > strike_price
#     print(payoffs)

    if hist:
        prices_below = prices[prices < strike_price]
        prices_above = prices[prices >= strike_price]
        meanlen = (len(prices_below) + len(prices_above))/2
        if (len(prices_below) > 0):
            plt.hist(prices_below, color='red', alpha=0.5, bins=max(1, int((
        if (len(prices_above) > 0):
            plt.hist(prices_above, color='green', alpha=0.5, bins=max(1,int(


        # add labels and legend
        plt.xlabel('Price')
        plt.ylabel('Frequency')
        plt.title('Asset Price at Expiry')
        plt.legend(loc='upper right')

        # display the histogram
        plt.show()

        plt.figure()
        plt.hist(payoff, color='green', bins=int((len(payoff))/20))
        plt.title('All Payoffs')
        plt.figure()
        plt.hist(payoff[payoff > 0], color='green', bins=int((len(payoff[pay
        plt.title('Nonzero Payoffs')

    return np.sum(prices[payoffs] - strike_price) / len(prices)

asset = {'s0': 200, 'mu': 0.0005, 'sigma': 0.005, 'n': 24, 'T': 30}
call_option_asset(asset = asset, strike_price=200, hist=False, plot=False)
```

Out[14]: 4.012897402565865

In [15]:
```
grader.check("q3.2")
```

Out[15]:

q3.2 passed! 🍀

Assuming that the strike price is constant ...

- What happens to the distribution when you increase or decrease T?
- What happens to the distribution when you increase or decrease s0?
- What happens to the distribution when you increase or decrease sigma?
- How does the shape and center of the distribution impact the price of the option?
- How does changing the option's strike impact the price of the option

Submission

Submit your code to the gradescope assignment for `[Coding HW] Homework 2`. The due date is March 23rd at midnight, pacific time.

Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

In [16]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export(pdf=False, run_tests=True)
```

```
Running your submission against local test cases...



Your submission received the following results when run against available tes
t cases:

    q1 results: All test cases passed!

    q2.1 results: All test cases passed!

    q2.2 results: All test cases passed!

    q2.3 results: All test cases passed!

    q3.1 results: All test cases passed!

    q3.2 results: All test cases passed!
```

Your submission has been exported. Click here to download the zip file.