Assignment 3

Jiaqi Guo JGR9647

A Bandit Problem

Part 1

Consider the following situation. You start with a thousand dollars to invest in two different stocks, **Apple** or **Microsoft**, for 100 days. At the beginning of each day, you must pick a stock to invest in. At the end of each day, you observe the return, and decide which stock to invest in the next day. You don't know what the return will be the next day, but you know that the return (as a proportion) of **Apple** is normally distributed with $\mu = 0.05$ and $\sigma^2 = 0.1$. The return of **Microsoft** is normally distributed with $\mu = 0.1$ and $\sigma^2 = 0.3$

Before doing any analyses, what do you expect to be the best strategy Would you prefer to invest in **Apple** or **Microsoft**? Or would you vary your investment based off of your balance? (There is no right answer here, and grading will only be based off of a thoughtful answer that makes a justification by addressing the probability distributions)

Answer:

I would choose Microsoft for 100 days because it has a higher μ value (higher expectation). And, in theory, we have a 57.2 percent chance of making a positive return with Microsoft stock, compared with 56.3 percent with Apple

In [9]:
```
import scipy.stats

# Microsoft
print(scipy.stats.norm.cdf(0,loc=0.1, scale=(0.3**0.5)))
# Apple
print(scipy.stats.norm.cdf(0,loc=0.05, scale=(0.1**0.5)))
```

```
0.42756607029235294
0.4371835305814459
```

Part 2

Using python or a language of your choice, write a program that simulates your strategy (a 100 day simulation starting with 1000 dollars). Repeat the simulation 100 times and report the mean and variance of your results.

In [10]:
```
import numpy as np
# np.random.seed(6)
return_list = []
company_choice = "Microsoft"

def bandit_100_days(company:str, money: float)-> tuple:
    if company =='Microsoft':
        return money*max((1+np.random.normal(loc=0.1, scale=0.3**0.5)),0)
    else:
        return money*max((1+np.random.normal(loc=0.05, scale=0.1**0.5)),0)
for _ in range(100):
    money = 1000
    for day in range(100):
        money= bandit_100_days(company_choice, money)
    return_list.append(money)

print(f' The variance of 100 times simulation is: {np.var(return_list)}')
print(f' The mean of 100 times simulation is: {np.mean(return_list)}')
```

```
The variance of 100 times simulation is: 397479.7292494515
The mean of 100 times simulation is: 63.57723012847625
```

Part 3

One approximation to the problem is known as the $exp3$ algorithm. It initially gives equal weights in deciding between both stocks, and based of results, re-weights the probability of choosing the next stock. An algorithm[1] is shown below, where you may set $\gamma = 1, T = 100$ (the number of days) and $i = 1$ represents Apple and $i = 2$ represents Microsoft.

Implement this algorithm for the problem, and report the final return of the algorithm. How does it compare to your result in Part 2?

In [11]:
```
import math

reward_history = []

def draw(weights):
    choice = np.random.uniform(0, sum(weights))
    choiceIndex = 0

    for weight in weights:
        choice -= weight
        if choice <= 0:
            return choiceIndex
        choiceIndex += 1

def distr(weights, gamma=0.0):
    theSum = float(sum(weights))
    return tuple((1.0 - gamma) * (w / theSum) + (gamma / len(weights)) for w in weights)

def exp3(numActions, reward, gamma=1, rewardMin = 0, rewardMax = 1):
   weights = [1.0] * numActions
```

```
    t = 0
    while True:
        probabilityDistribution = distr(weights, gamma)
        choice = draw(probabilityDistribution)

        theReward = reward(choice, t)
        scaledReward = (theReward - rewardMin) / (rewardMax - rewardMin)

        estimatedReward = 1.0 * scaledReward / probabilityDistribution[choice]
        weights[choice] *= math.exp(estimatedReward * gamma / numActions)

        yield theReward
        t = t + 1

def one_round_invest():
    t = 0
    cumulativeReward = 1000
    numActions = 2
    numRounds = 100

    rewardVector = [[max(np.random.normal(loc=0.1, scale=0.3**0.5),-1),max(np.random.normal(loc=0.05, scale=0.1**0.5),-1)] for _ in rang
    rewards = lambda choice, t: rewardVector[t][choice]

    for reward in exp3(numActions, rewards, rewardMin = max(np.array(rewardVector).flatten()), rewardMax = min(np.array(rewardVector).fla
        cumulativeReward *= (1+reward)
        t += 1
        if t >= numRounds:
            break
    return cumulativeReward

for _ in range(100):
    reward_history.append(one_round_invest())

print(f' The variance of 100 times simulation is: {np.var(reward_history)}')
print(f' The mean of 100 times simulation is: {np.mean(reward_history)}')
```

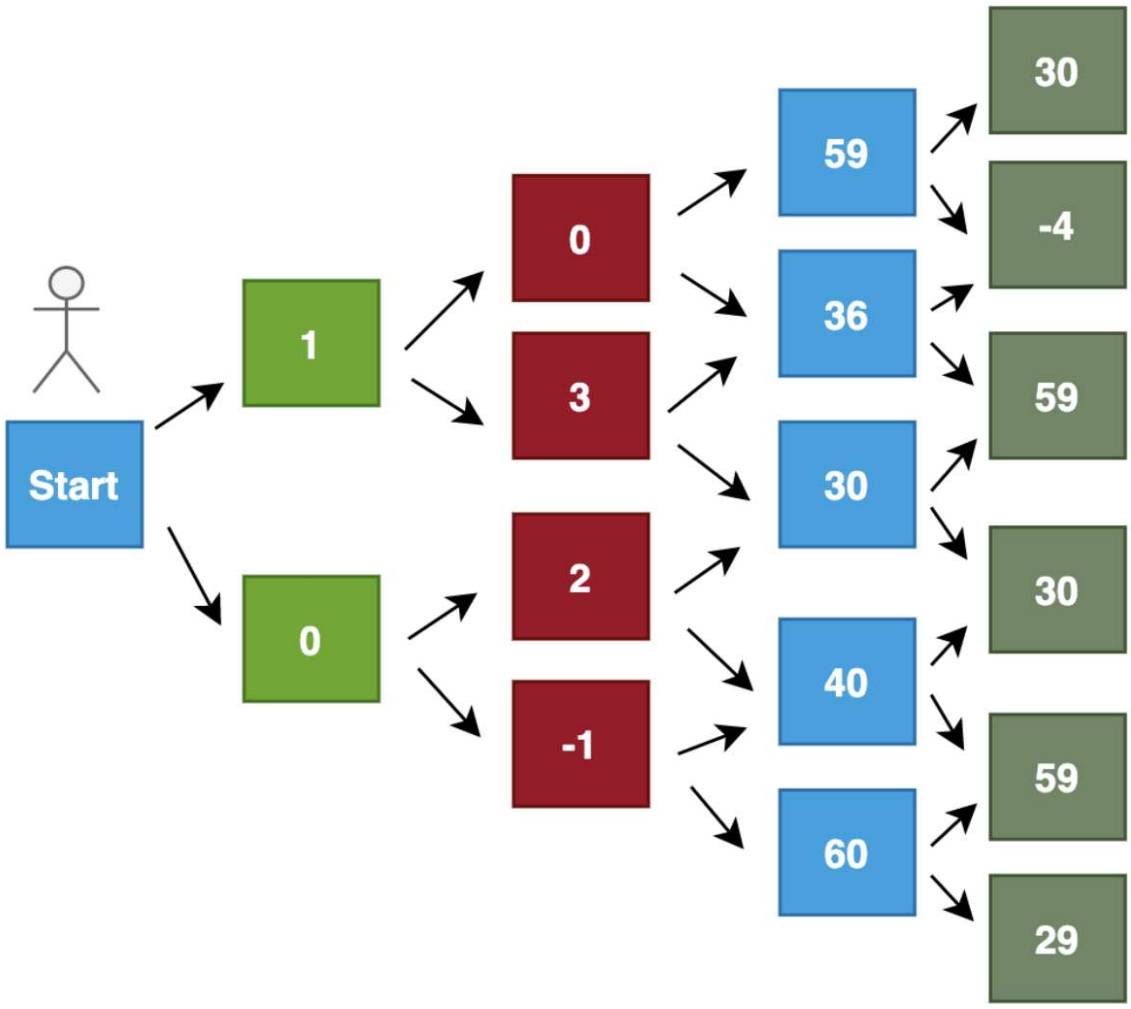
```
The variance of 100 times simulation is: 26483939980.40379
The mean of 100 times simulation is: 27887.14463351453
```

Answer:

Through the introduction of EXP3 algorithm, the average value of our results is usually higher, but there is also a larger variance

Implementing Value Iteration

The diagram below represents a four period problem, where at each state (a box), you receive reward the number on the box. At each state, you must decide to move upwards or downwards to the next box



Part 1

Consider the diagram above. What is the path(s) that maximizes your total reward (the sum of the values in the boxes)? In your notation, let U denote moving upwards and D denote moving downwards. Then the path U, U, U, D, for example gives reward r = 1 + 0 + 59 − 4 = 56.

Answer:

$r = 0 + (-1) + 60 + 59$, which correspond to D, D, D, U.

Part 2

Implement the value iteration algorithm for the problem above (by writing a program) with no more than ten iterations, and report your result. Are you able to achieve the optimal path?

In [37]:

```
env={0:[0],1:[1,0],2:[0,3,2,-1],3:[59,36,30,40,60],4:[30,-4,59,30,59,29]}

def state_value(env, period:int, idx:int):
    next_state = []
    if period == 1:
        for act1 in range(1,3):
            next_state.append(env[period+1][2*(idx+1)-act1])
    else:
        for act2 in range(2):
            next_state.append(env[period+1][(idx+1)-act2])
    return max(next_state)


def one_iter(env):
    converge = False
    current_state = []
    updated_state = []
    for i in range(0,4):
        current_state +=env[i]

    for i in range(4):
        period = i
        for idx in range(len(env[period])):
            env[period][idx]=0.5*(0.5*state_value(env,period,idx)+env[period][idx])

    for i in range(0,4):
        updated_state +=env[i]
    loss = np.sum(np.fabs(np.array(current_state)-np.array(updated_state)))
    if np.sum(np.fabs(np.array(current_state)-np.array(updated_state)))<=1:
        converge = True
    return env, converge, loss
```

```
def select_path(env):
    policy = []
    idx = 0
    for i in range(1,5):
        if i > 2:
            p = np.argmax(env[i][idx:idx+2])
            idx = idx + p
            policy.append(p)
        else:
            p = np.argmax(env[i][(idx+1)*2-2:(idx+1)*2])
            targe = env[i][(idx+1)*2-2:(idx+1)*2]
            idx = int(np.argwhere(np.array(env[i])==targe[p]))
            policy.append(p)
    return policy

def convert(policy:list):
    Policy = []
    for P in policy:
        if P==1:
            Policy.append('D')
        else:
            Policy.append('U')
    return Policy



iter = 0
while True:
    iter+=1
    env, conv, loss = one_iter(env)
    if conv:
        print(f'Model converges at iteration: {iter}')
        break

policy = select_path(env)

print(f"The optimal policy is {convert(policy)}")
```

```
Model converges at iteration: 9
The optimal policy is ['D', 'D', 'D', 'U']
```

Answer:

We can find the optimal path at the 10^th iteration, which is D, D, D, U. That is $Start \rightarrow 0 \rightarrow -1 \rightarrow 60 \rightarrow 59$