In [1]:
```
import random
import numpy as np
import statistics as s
import scipy.stats
from collections import defaultdict
```

In [2]:
```
CONF = 0.95
RUNS = 100000
TRIALRUNS = 100
```

In [3]:
```
statdict = defaultdict(int)
```

In [4]:
```
CHOICE = ['A', 'B']
```

In [5]:
```
def winloss(obs1, obs2, p):
    if obs1 == 'A':
        if obs2 < p:
            return 1
        else:
            return -1
    else:
        if obs2 < 0.5:
            return 1
        else:
            return -1
```

In [6]:
```
def final_choice(stats, conf):

    stats.sort(key = lambda x:x[1][0], reverse = True)

    # Then tests whether the second, third and so forth values contain 0 within their joi
    inconclusive_list = [stats[0][0]]
    inconclusive = False
    for i in range(1, len(stats)):
        if in_range(construct_CI(stats[0], stats[i], conf)):
            inconclusive_list.append(stats[i][0])
            inconclusive = True
        else: # Because all values are sorted, if the current choice's joint Confidence In
            break

    if inconclusive: # If inconclusive, return statement with the list of 'drawed' choices
        return f'Inconclusive: the following came to a draw {inconclusive_list}'

    return stats[0][0] # Else, return the dominant strategy
```

In [7]:
```
def construct_CI(stat1, stat2, conf):
    """ Uses Welch's approximation to construct a joint CI of two means, unknown populatio

    xbar1 = stat1[1][0]
    xbar2 = stat2[1][0]
    s1 = stat1[1][1]
    s2 = stat2[1][1]
    n = stat1[1][2]
    m = stat2[1][2]
```

```
    r = (s1**2 /n + s2**2 /m)**2 /(s1**4 /(n**2 * (n-1)) + s2**4 /(m**2 * (m-1)))

    q = scipy.stats.t.ppf(conf, df = r)

    poolsd = np.sqrt(s1**2 /n + s2**2 /m)

    CI = (xbar1 - xbar2 - q * poolsd, xbar1 - xbar2 + q * poolsd)

#     print(f'{stat1[0]} {stat2[0]}: {CI}')
#     print('\n')

    return CI
```

In [8]:

```
def in_range(CI):
    """ Helper function to check whether 0 is within the Confidence Interval """

    if CI[0] <= 0 and CI[1] >= 0:
        return True
    return False
```

In [9]:

```
for p in [0.51, 0.505, 0.501, 0.5005, 0.5]:

    counter = 0

    for j in range(TRIALRUNS):


        data = defaultdict(list) # Using defaultdict makes the code more adaptable to difi

        sample = list()
        victory = list()

        for i in range(RUNS):
            obs1 = random.sample(CHOICE, 1)[0]
            obs2 = random.uniform(0, 1)

            data[obs1].append(winloss(obs1, obs2, p))

        # This time we record the data for both players because we are interested in each
        # i.e. we are not as interested in, or equally interested in each's dominant strat

        stats = dict()
        for choice in CHOICE:
            tmp = list()
            tmp.append(s.mean(data[choice]))
            tmp.append(s.stdev(data[choice]))
            tmp.append(len(data[choice]))

            stats[choice] = tmp

        result = final_choice(list(stats.items()), CONF)
        if result == 'A':
            counter += 1

    statdict[p] = counter/TRIALRUNS
```

In [10]:

```
statdict
```

Out[10]:

```
defaultdict(int,
            {0.51: 0.92, 0.505: 0.47, 0.501: 0.12, 0.5005: 0.06, 0.5: 0.06})
```

However, our test was conducted on random variables with variance = 1. Different scenarios (random variables with different variances) need to undergo re-testing.