Assignment 1

Prepared by Ali Younis & Manuel Martinez Garcia

Problem 1: Deck of Cards Simulation

Link to ChatGPT conversation for building Poker Logic

Building a Deck of Cards

Below is some code I made creating a sort of dealer and card logic.

In [1]:
```
import random
from collections import Counter

rank_caller = {i: str(i) for i in range(2, 11)}  # 2-10
rank_caller.update({
    11: "Jack",
    12: "Queen",
    13: "King",
    14: "Ace"
})

class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __repr__(self):
        return f"{rank_caller[self.rank]} of {self.suit}"

class Deck:
    def __init__(self, n=3):
        self.deck = []
        self.generate_deck(n)

    def generate_deck(self, n):
        ranks = [i for i in range(2,15)]
        suits = ["Hearts", "Clubs", "Diamonds", "Spades"]

        for _ in range(n):
```

```
            for rank in ranks:
                for suit in suits:
                    self.deck.append(Card(rank, suit))

        random.shuffle(self.deck)

    def deal_hand(self, hand_size=7):
        hand = self.deck[:hand_size]
        self.deck = self.deck[hand_size:]
        return hand
```

Poker Logic

Below is the logic for deciding the best hand in 7-choose-5 poker based on this priority. J's are Wild Cards.

1. **Five of a Kind**
    - Example: A-A-A-A-A
2. **Straight Flush**
    - Five consecutive cards, all of the same suit.
    - Example: 9-8-7-6-5 (all Hearts).
3. **Four of a Kind**
    - Four cards of the same rank.
    - Example: K-K-K-K-5.
4. **Full House**
    - Three of one rank + a pair of another.
    - Example: Q-Q-Q-2-2.
5. **Flush**
    - Any five cards of the same suit, *not in sequence*.
    - Example: A-10-7-5-3 (all Clubs).
6. **Straight**
    - Five consecutive cards, *not all the same suit*.
    - Example: 9-8-7-6-5 (mixed suits).
    - *Ace is high (A-K-Q-J-10).*
7. **Three of a Kind**
    - Three cards of the same rank.

- Example: 7-7-7-J-2.

8. **Two Pair**

    - Two different pairs + a fifth card.
    - Example: A-A-8-8-4.

9. **One Pair**

    - A single pair + three unrelated cards.
    - Example: 9-9-K-Q-3.

10. **High Card**

    - None of the above. Ranked by highest card.
    - Example: A-K-5-4-2 (mixed suits).

In [2]:
```python
def get_best_hand(hand):
    best_hand = []
    best_hand_type = "High Card" #default
    JACK_JACK = 11

    ### Helper Functions ###
    def extract_jack(hand):
        jacks, nojacks = [], []
        for card in hand:
            if card.rank == JACK_JACK:
                jacks.append(card)
            else:
                nojacks.append(card)
        return nojacks, jacks
    def highest_rank(ranks): # pick highest candidate
        return max(ranks) if ranks else None

    nojacks, jacks = extract_jack(hand)
    jacks_count = len(jacks)
    rank_counts = Counter(card.rank for card in nojacks)
    suit_counts = Counter(card.suit for card in nojacks)

    ### 1. Five of a Kind ###
    five_of_kind    = [rank for rank, count in rank_counts.items() if count >= 5 - jacks_count]
    if five_of_kind:
        target = highest_rank(five_of_kind)
        best_hand = [card for card in nojacks if card.rank == target] + jacks
        best_hand_type = "Five of a Kind"
        return best_hand[:5], best_hand_type

    ### 2. Straight Flush ###
    flush = [s for s, cnt in suit_counts.items() if cnt >= 5 - jacks_count]
```

```python
    if flush:
        flush_suit = flush[0]
        flush_ranks = [c.rank for c in nojacks if c.suit == flush_suit]
        if flush_ranks:
            flush_ranks = sorted(set(flush_ranks), reverse=True)  # unique, high->low

            possible_hand = [flush_ranks[0]]  # ranks only
            feasible = True
            i = 0
            jacks_at_hand = jacks.copy()
            jacks_used = 0

            # try to extend downward
            while feasible and i + 1 < len(flush_ranks) and len(possible_hand) + jacks_used < 5:
                curr = flush_ranks[i]
                nextv = flush_ranks[i + 1]
                if nextv == curr:
                    i += 1
                    continue
                gap = curr - nextv - 1  # how many ranks missing between curr and nextv
                if gap <= len(jacks_at_hand):
                    # fill the gap with wilds
                    use = min(gap, len(jacks_at_hand))
                    jacks_at_hand = jacks_at_hand[use:]
                    jacks_used += use
                    i += 1
                    possible_hand.append(nextv)
                else:
                    feasible = False

            # If we still need cards to reach 5 and have remaining lower ranks, try to append them
            while feasible and len(possible_hand) + jacks_used < 5 and i + 1 < len(flush_ranks):
                i += 1
                # fill full gap between last and this next rank
                curr = possible_hand[-1]
                nextv = flush_ranks[i]
                if nextv == curr:
                    continue
                gap = curr - nextv - 1
                if gap <= len(jacks_at_hand):
                    use = min(gap, len(jacks_at_hand))
                    jacks_at_hand = jacks_at_hand[use:]
                    jacks_used += use
                    possible_hand.append(nextv)
                else:
                    break
            # If still short, you might allow extending past the smallest rank (e.g., 5,4,3,2,A) — omitted for minimal change.
            if len(possible_hand) + jacks_used >= 5:
                best_hand = [c for c in nojacks if (c.suit == flush_suit and c.rank in possible_hand)]
```

```python
            best_hand += jacks[:jacks_used]
            best_hand_type = "Straight Flush"
            return best_hand[:5], best_hand_type


    ### 3. Four of a Kind ###
    four_of_kind    = [rank for rank, count in rank_counts.items() if count >= 4 - jacks_count]
    if four_of_kind:
        target = highest_rank(four_of_kind)
        best_hand = [card for card in nojacks if card.rank == target] + jacks
        best_hand_type = "Four of a Kind"
        return best_hand[:5], best_hand_type

    ### 4. Full House ###
    three_of_kind = [r for r, cnt in rank_counts.items() if cnt >= 3]
    pairs = [r for r, cnt in rank_counts.items() if cnt >= 2 and r not in three_of_kind]
    if three_of_kind and pairs:
        t, p = max(three_of_kind), max(pairs)
        best_hand = [c for c in hand if (c.rank == t or c.rank == p)]
        best_hand_type = "Full House"
        return best_hand[:5], best_hand_type

    ### 5. Flush ####
    flush = [suit for suit, count in suit_counts.items() if count >= 5 - jacks_count]
    if flush:
        # choose the suit with most cards to maximize quality
        flush_suit = max(flush, key=lambda s: suit_counts[s])
        suited = sorted([c for c in nojacks if c.suit == flush_suit],
                        key=lambda c: c.rank, reverse=True)
        best_hand = suited[:5]

        # top up with jacks if short
        short = 5 - len(best_hand)
        if short > 0:
            best_hand += jacks[:short]
        best_hand_type = "Flush"
        return best_hand[:5], best_hand_type

    #### 6. Sequence ###
    rank_values = [c.rank for c in nojacks]
    if rank_values:
        # unique ranks, sorted high -> low
        rank_values = sorted(set(rank_values), reverse=True)

        possible_hand = [rank_values[0]]  # store ranks only
        feasible = True
        i = 0
        jacks_at_hand = jacks.copy()
        jacks_used = 0
```

```python
    # extend downward while we have next ranks and haven't reached 5 cards total (incl. wilds)
    while feasible and i + 1 < len(rank_values) and len(possible_hand) + jacks_used < 5:
        curr = rank_values[i]
        nextv = rank_values[i + 1]

        if nextv == curr:
            i += 1
            continue

        gap = curr - nextv - 1  # number of missing ranks between curr and nextv
        if gap < 0:
            # should not happen with sorted unique, but guard anyway
            i += 1
            continue

        if gap <= len(jacks_at_hand):
            use = gap
            jacks_at_hand = jacks_at_hand[use:]
            jacks_used += use
            i += 1
            possible_hand.append(nextv)
        else:
            feasible = False

    # If we still need cards (<5), try stepping one more rank down if we can fill the whole gap
    while feasible and len(possible_hand) + jacks_used < 5 and i + 1 < len(rank_values):
        i += 1
        curr = possible_hand[-1]
        nextv = rank_values[i]
        if nextv == curr:
            continue
        gap = curr - nextv - 1
        if gap <= len(jacks_at_hand):
            use = gap
            jacks_at_hand = jacks_at_hand[use:]
            jacks_used += use
            possible_hand.append(nextv)
        else:
            break

    if len(possible_hand) + jacks_used >= 5:
        # build final straight: take cards with ranks in possible_hand + add exactly jacks_used
        best_hand = [c for c in nojacks if c.rank in possible_hand]
        best_hand += jacks[:jacks_used]
        best_hand_type = "Sequence"
        return best_hand[:5], best_hand_type

### 7. Three of a Kind ###
```

```
    three_of_kind = [rank for rank, count in rank_counts.items() if count >= 3 - jacks_count]
    if three_of_kind:
        target = max(three_of_kind)
        best_hand = [c for c in nojacks if c.rank == target] + jacks
        best_hand_type = "Three of a Kind"
        return best_hand[:5], best_hand_type


    # 8. Two Pair
    pairs = [r for r, cnt in rank_counts.items() if cnt >= 2]
    if len(pairs) >= 2:
        p1, p2 = sorted(pairs, reverse=True)[:2]
        best_hand = [c for c in hand if (c.rank == p1 or c.rank == p2)]
        best_hand_type = "Two Pair"
        return best_hand[:5], best_hand_type

    # 8. One Pair
    elif pairs:
        p = max(pairs)
        best_hand = [c for c in hand if c.rank == p]
        best_hand_type = "One Pair"
        return best_hand[:5], best_hand_type

    rest = sorted(nojacks, key=lambda c: c.rank, reverse=True)[:5]
    short = 5 - len(rest)
    if short > 0:
        rest += jacks[:short]  # treat jacks as high
    return rest[:5], best_hand_type
```

Debugging / Quick Test

In [3]:
```
best_hands = []
for _ in range(500):
    hand = Deck().deal_hand()
    best_hand, best_hand_type = get_best_hand(hand)
    best_hands.append(best_hand_type)

Counter(best_hands)
```

Out[3]:
```
Counter({'One Pair': 111,
         'Three of a Kind': 107,
         'Two Pair': 91,
         'High Card': 60,
         'Flush': 48,
         'Four of a Kind': 41,
         'Full House': 22,
         'Sequence': 16,
         'Five of a Kind': 2,
         'Straight Flush': 2})
```

Question:

Please write code that uses a Monte Carlo simulation with at least 100,000 trials to solve the following two questions.

> a.) Please compute the probability, $P(H)$, where H is the event that the best possible 5-card hand constructible from a randomly dealt 7-card hand is either a Five of a Kind or Four of a Kind.

In [4]:
```
best_hands = []
n = 10**5
for _ in range(n):
    hand = Deck().deal_hand()
    best_hand, best_hand_type = get_best_hand(hand)
    best_hands.append(best_hand_type)

prob = Counter(best_hands)['Five of a Kind'] + Counter(best_hands)['Four of a Kind']
prob /= n

print(f"The probability that the best possible 5-card hand is either a Five or Four of a Kind is {prob:.6f}")
```

```
The probability that the best possible 5-card hand is either a Five or Four of a Kind is 0.094210
```

> b.) Please compute the probability, $P(H)$, where H is the event that the best possible 5-card hand constructible from a randomly dealt 7-card hand is a High Card.

In [5]:
```
best_hands = []
for _ in range(n):
    hand = Deck().deal_hand()
    best_hand, best_hand_type = get_best_hand(hand)
    best_hands.append(best_hand_type)

prob = Counter(best_hands)['High Card'] / n

print(f"The probability that the best possible 5-card hand is a High Card is {prob:.6f}")
```

```
The probability that the best possible 5-card hand is a High Card is 0.131450
```

Problem 2: Battery Packaging Simulation

Skeleton / Notes

- setup:
  - 100 batteries per box
  - each battery ~ N(1, 0.05^2)
  - so total weight ~ Normal(mean=100, std=0.5)
  - choose N trials for Monte Carlo (like 200000)
- Monte Carlo:
  1. simulate N samples (N×100 normals)
  2. (a) prob(total > 101) = count/N
  3. (b) fee = max(X-100,0); conditional expectation = mean of fee when X>100
- Analytical:
  - use pdf/cdf for standard normal
  - (a) prob = 1 - Phi((101-100)/0.5)
  - (c) E[(X-100)+] = σφ(z) + (μ-100)(1-Φ(z))
  - (b) conditional = that / P(X>100)
- Finally: print MC vs analytic results This concludes the human portion/student skeleton. The following conversation with a language model (https://chatgpt.com/share/68db3ffd-12f8-8009-b80e-2f9f5eef14d6) was used to generate the body of the code.

In [2]:
```python
import numpy as np
from math import erf, sqrt, exp, pi

# --- setup ---
np.random.seed(42)                # reproducibility
n_per_box = 100
mu_each = 1.0
sd_each = 0.05

mu_total = n_per_box * mu_each
sd_total = np.sqrt(n_per_box) * sd_each   # = 0.5

N = 200_000  # number of Monte Carlo trials; bump this if you want tighter MC error

# --- helpers for standard normal ---
```

```python
def phi(z):
    # standard normal pdf
    return (1.0 / sqrt(2*pi)) * exp(-0.5 * z*z)

def Phi(z):
    # standard normal cdf
    return 0.5 * (1.0 + erf(z / sqrt(2.0)))

# --- Monte Carlo simulation ---
# simulate N boxes: draw N x 100 normals with mean=1, sd=0.05 and sum to get totals
samples = np.random.normal(loc=mu_each, scale=sd_each, size=(N, n_per_box)).sum(axis=1)

# (a) estimate P(total > 101)
p_mc_gt_101 = np.mean(samples > 101.0)

# (b) conditional expectation of extra fee given X>100, where fee = max(X-100, 0)
fees = np.clip(samples - 100.0, 0.0, None)
mask_gt_100 = samples > 100.0
cond_fee_mc = fees[mask_gt_100].mean()

# --- Analytical values ---
# (a) analytic P(X>101)
z_101 = (101.0 - mu_total) / sd_total
p_analytic_gt_101 = 1.0 - Phi(z_101)

# (c) analytic E[(X-100)+] using the known formula
# For X ~ N(mu, sigma^2), k=100:
# E[(X-k)+] = sigma * phi((k - mu)/sigma) + (mu - k) * (1 - Phi((k - mu)/sigma))
k = 100.0
z_k = (k - mu_total) / sd_total
expected_positive_part = sd_total * phi(z_k) + (mu_total - k) * (1.0 - Phi(z_k))

# (b) analytic conditional expectation E[X-100 | X>100]
p_gt_100 = 1.0 - Phi(z_k)
cond_fee_analytic = expected_positive_part / p_gt_100

# --- print results side by side ---
print("Monte Carlo (N = {:,} trials)".format(N))
print(f"(a) P(X > 101) ≈ {p_mc_gt_101:.6f}")
print(f"(b) E[X - 100 | X > 100] ≈ ${cond_fee_mc:.6f}")

print("\nAnalytic")
print(f"(a) P(X > 101) = {p_analytic_gt_101:.6f}")
print(f"(b) E[X - 100 | X > 100] = ${cond_fee_analytic:.6f}")
print(f"(c) E[(X - 100)+] = ${expected_positive_part:.6f}")

# (optional) quick sanity checks:
# - MC estimate of E[(X-100)+] should be close to analytic when N is large
mc_pos_part = fees.mean()
```

```
print("\nSanity check: MC E[(X-100)+] ≈ ${:.6f}".format(mc_pos_part))
```

```
Monte Carlo (N = 200,000 trials)
(a) P(X > 101) ≈ 0.022915
(b) E[X - 100 | X > 100] ≈ $0.397774

Analytic
(a) P(X > 101) = 0.022750
(b) E[X - 100 | X > 100] = $0.398942
(c) E[(X - 100)+] = $0.199471

Sanity check: MC E[(X-100)+] ≈ $0.198947
```

Problem 3: Food Truck Simulation

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
lamb = 2            # rate
total_days = 100    # of Wednesdays
t_open = 0          # 11:00 AM
t_close = 120       # 1:00 PM
window_start = 55   # 11:55 AM
window_end = 65     # 12:05 Pm

rng = np.random.default_rng(174)
```

In [3]:
```
def generate_arrivals(rate, t0, t1, rng):
    arrivals = []
    t = t0
    while True:
        t += rng.exponential(1 / rate)
        if t > t1:
            break
        arrivals.append(t)
    return np.array(arrivals)
```

a.) Denote E_i as the number of customers that arrive at the food truck between 11:55 AM and 12:05 PM on the i-th day. Do the following:

> i.) Compute $\frac{1}{100}\sum_{i=1}^{100} E_i$ and compute the sample variance for $\mathbb{E}_i$'s.

In [4]:
```
E_i = []
all_inters = []

for _ in range(total_days):
    arrivals = generate_arrivals(lamb, t_open, t_close, rng)
```

```python
    E_i.append(np.sum((arrivals >= window_start) & (arrivals <= window_end)))
    if len(arrivals) >=2:
        all_inters.extend(np.diff(arrivals))
E_i = np.array(E_i)

print(f"(1/100) * sum E_i = {E_i.mean():.4f}")
print(f"Sample Variance(E_i) = {E_i.var(ddof=1):.4f}")
```

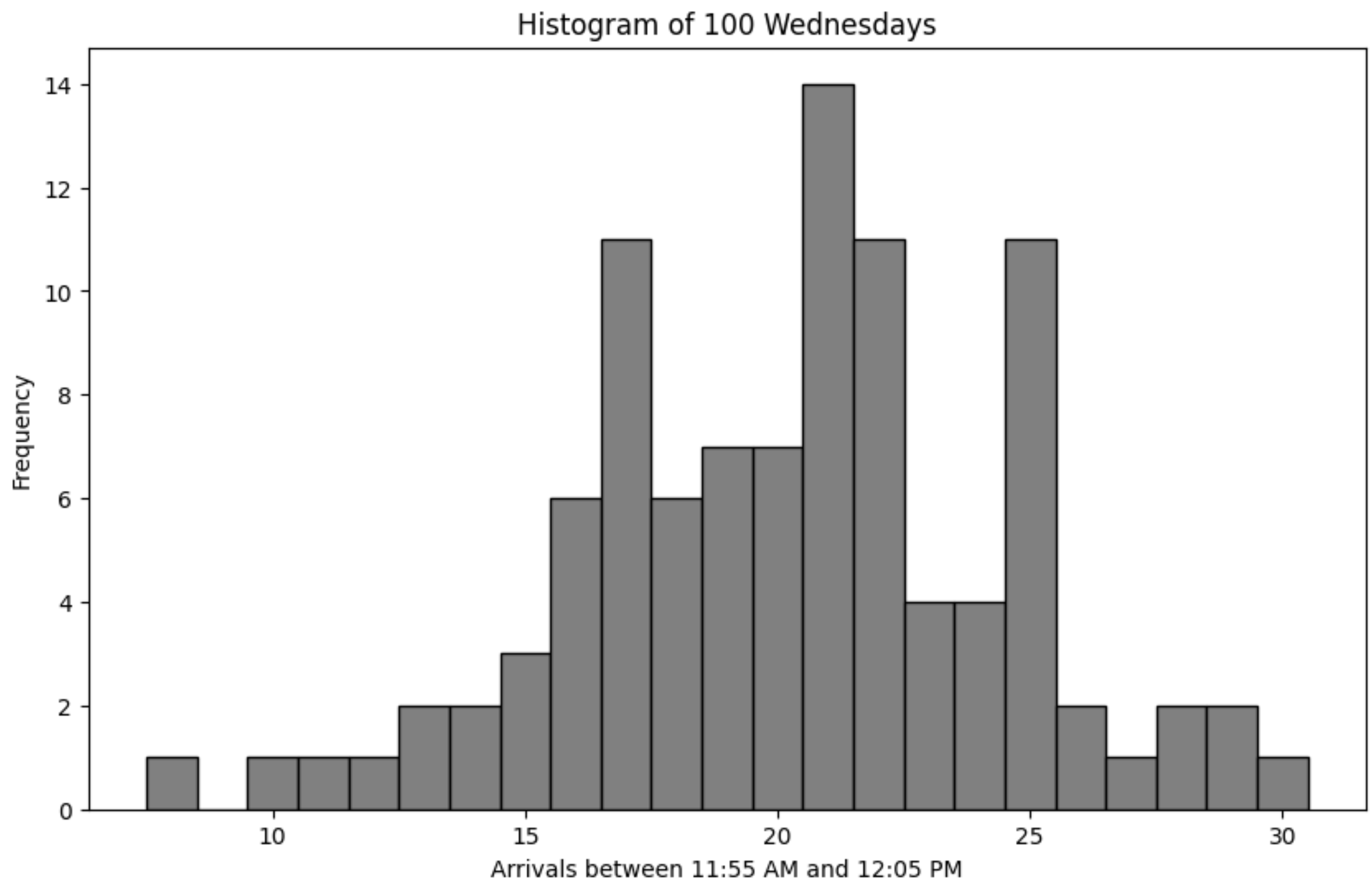
```
(1/100) * sum E_i = 20.2600
Sample Variance(E_i) = 18.1539
```

> ii.) Histogram

In [5]:

```python
bin_range = np.arange(min(E_i), max(E_i) + 2) - 0.5

plt.figure(figsize=(10, 6))
plt.hist(E_i, bins=bin_range, edgecolor='black', color='gray')
plt.xlabel('Arrivals between 11:55 AM and 12:05 PM')
plt.ylabel('Frequency')
plt.title('Histogram of 100 Wednesdays')
plt.show()
```



> iii.) Inter-arrival times over 100 days, proportion > 1 minute, & comparison to e^{-2}

Proportion Longer than 1 minute

In [6]:
```
prop = (np.array(all_inters, dtype=float) > 1).mean()

print(f"Percentage of inter-arrival times longer than a minute = {prop:.4f}")
print(f"e^(-2) = {np.exp(-lamb):.4f}")
```

```
Percentage of inter-arrival times longer than a minute = 0.1327
e^(-2) = 0.1353
```