Q1: Algorithm Analysis

Background on Algorithm Analysis (50 points total)

- Constant Time (O(1)): Regardless of the input size, the algorithm takes a constant amount of time to complete.
  - Accessing an element in an array by index.
  - Inserting or deleting an element at the beginning of a linked list.
- Linear Time (O(n)): The runtime grows linearly with the size of the input.
  - Iterating through a list or array once.
  - Searching for an element in an unsorted list by traversing it.
- Logarithmic Time (O(log n)): As the input size grows, the runtime increases logarithmically.
  - Binary search in a sorted array.
  - Operations in a balanced binary search tree (BST).
- Quadratic Time (O(n^2)): The runtime grows quadratically with the input size.
  - Nested loops where each loop iterates through the input.
  - Some sorting algorithms like Bubble Sort or Selection Sort.
- Exponential Time (O(2^n)): The runtime doubles with each addition to the input size.
  - Algorithms involving recursive solutions that make repeated calls.
  - Solving the Traveling Salesman Problem using brute force.
- Factorial Time (O(n!)): The runtime grows factorially with the input size.
  - Permutation problems that explore all possible combinations, like the brute force solution for the Traveling Salesman Problem with no optimizations.

Bubble Sort (Example)

Use the below analysis and explanation as a template for answering the following questions on algorithm analysis. You are to analyze each algorithm line by line, calculate the runtime complexity, and provide an explanation in markdown.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):  # O(n)
        for j in range(0, n-i-1):  # O(n)
            if arr[j] > arr[j+1]:  # O(1)
                arr[j], arr[j+1] = arr[j+1], arr[j]  # O(1)

# Total Runtime Complexity:
# Big O notation: O(n^2)
# Big Omega notation: Ω(n)
# Big Theta notation: Θ(n^2)
```

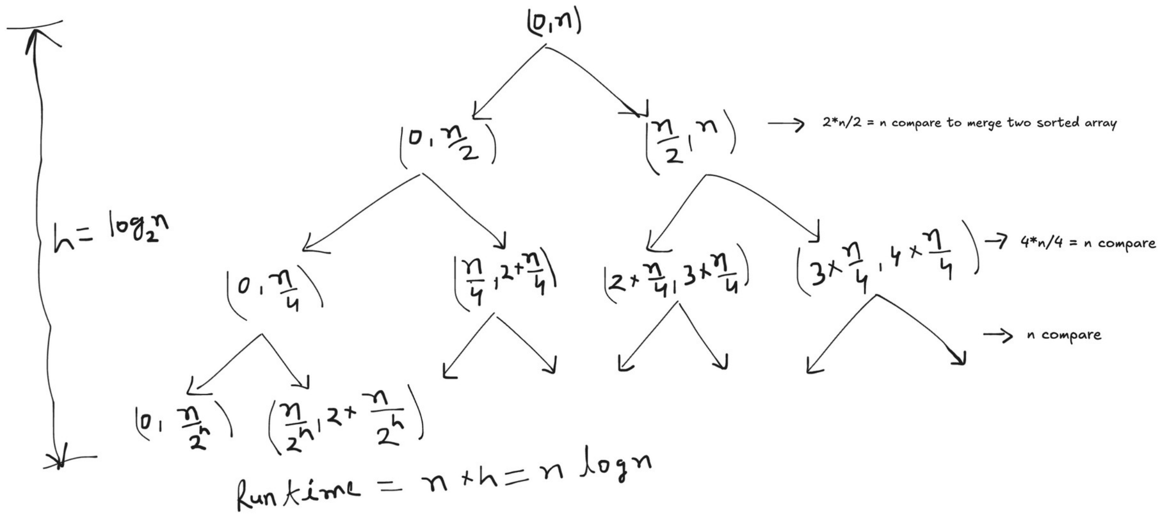
Explanation:

- The outer loop iterates 'n' times, where 'n' is the length of the array.
- The inner loop also iterates 'n' times in the worst case.
- The comparisons and swapping operations inside the inner loop are constant time ('O(1)').

This results in a total runtime complexity of O(n^2) in the worst case, Ω(n) in the best case (already sorted), and Θ(n^2) in the average and worst cases, as both loops iterate over the input array.

1A Merge Sort (10 points)

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Divide phase runs as per height of tree i.e. logn
        merge_sort(left_half)  # O(1)
        merge_sort(right_half) # O(1)

        i = j = k = 0

        # Merge phase takes O(n) time
        while i < len(left_half) and j < len(right_half):  # O(n/2)
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Either one of next two while loop runs
        while i < len(left_half):  # O(n/2)
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half): # O(n/2)
            arr[k] = right_half[j]
            j += 1
            k += 1

# Total Runtime Complexity:
# Big O notation: O(nlogn)
# Big Omega notation:  Ω(nlogn)
# Big Theta notation: Θ(nlogn)
```



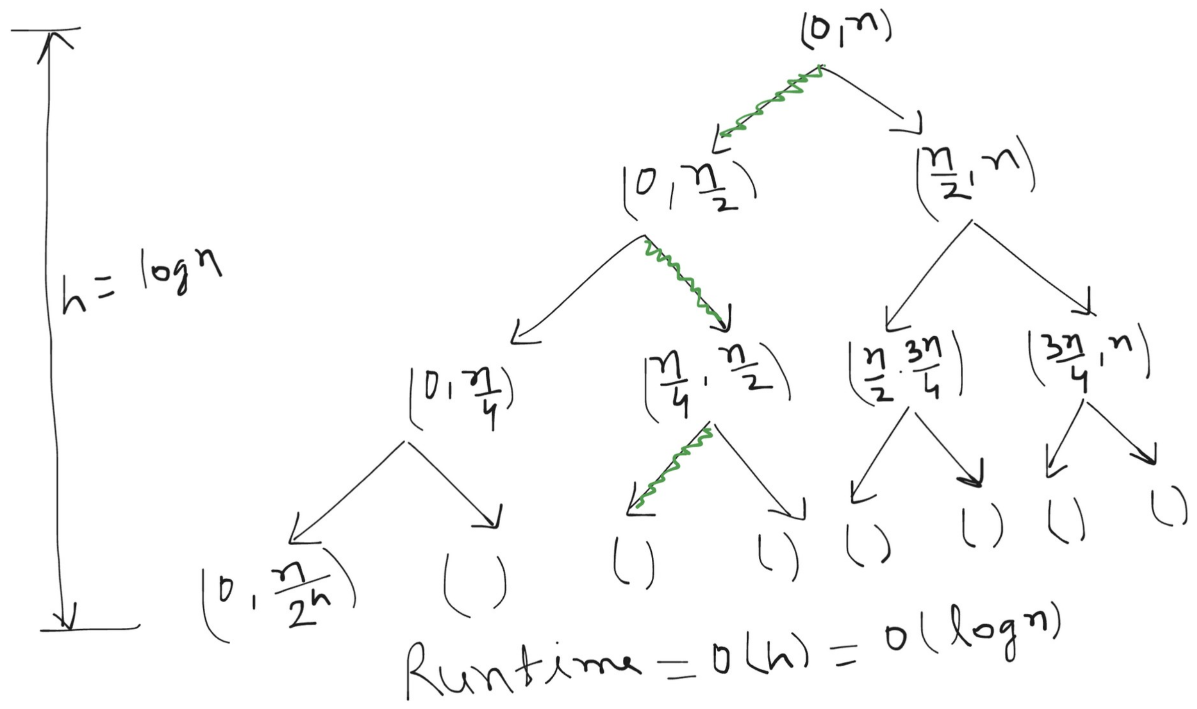
Explanation:

- Merge sort dividing phase, divices problem into two parts starts to begin with size (0,n)
- On first level, it splits into two halves of size (0,n/2) and (n/2,n)
- On second level, it further splits into two halves which gives four nodes i.e. (0,n/4), (n/4,2*n/4), (2*n/4, 3*n/4) and (3*n/4, n)
- If we generalize it then on level `h`, it divides into (0,n/2^h) blocks
- On the last level with height `h`, since size of array is one, therefore `n/2^h = 1` i.e. `n = 2^h` i.e. `h=logn`
- On every level, we run merge phase. For example, at level 1, we merge (0,n/2) and (n/2,n). First while loop will do `n/2` comparision. Then one of the next two while loop runs with `n/2` comparision which makes total `n` compare.
- It means at every level, we do run `n` comparision.
- Total run time complexity became `n*h` i.e. `n*logn`
- Since run time execution is same irrespective of input therefore best case, worst case and avergae case will give the same runtime complexity.

1B Binary Search (10 points)

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # This is and following are constant 
unit of work.
        mid_val = arr[mid]

        if mid_val == target:
            return mid
        elif mid_val < target:  # This and following else case force 
execution to choose one path.
```

```
            low = mid + 1
        else:
            high = mid - 1

    return -1

# Total Runtime Complexity:
# Big O notation: O(logn)
# Big Omega notation: Ω(1)
# Big Theta notation: θ(logn)
```



Explanation:

- At every level during the division phase, Binary search choose one path.
- It means, it doesn't explore the entire binary tree instead it only traverse one path
- At every level, it does constant time of work.
- Therefore the runtime execution is same as height of the tree which is `logn`
- But if middle element is same as the target then it will complete in O(1). Therefore best case is `O(1)`
- If target does not exist then it will exhaust all the nodes in the path therefore in the worst case it will be `O(logn)`
- In the average case it might find the target somewhere in the middle of path therefore on average we can estimate it as `O(logn)`

1C Linear Search (10 points)

```
def linear_search(arr, target):
    for i in range(len(arr)): # O(n)
        if arr[i] == target:  # O(1)
            return i
    return -1

# Total Runtime Complexity:
# Big O notation: O(n)
# Big Omega notation: Ω(1)
# Big Theta notation: θ(n)
```

Explanation:

- first for loop runs n times
- If condition runs in constant time
- Therefore overall time complexity is `O(n)`
- If target exist as first element then time complexity will be `O(1)`
- If target exist as last element then time complexity will be `O(n)`
- If target exist somewhere in middle then time complexity will be O(n/k) but on average we can say that `O(n)`

1D Selection Sort (10 points)

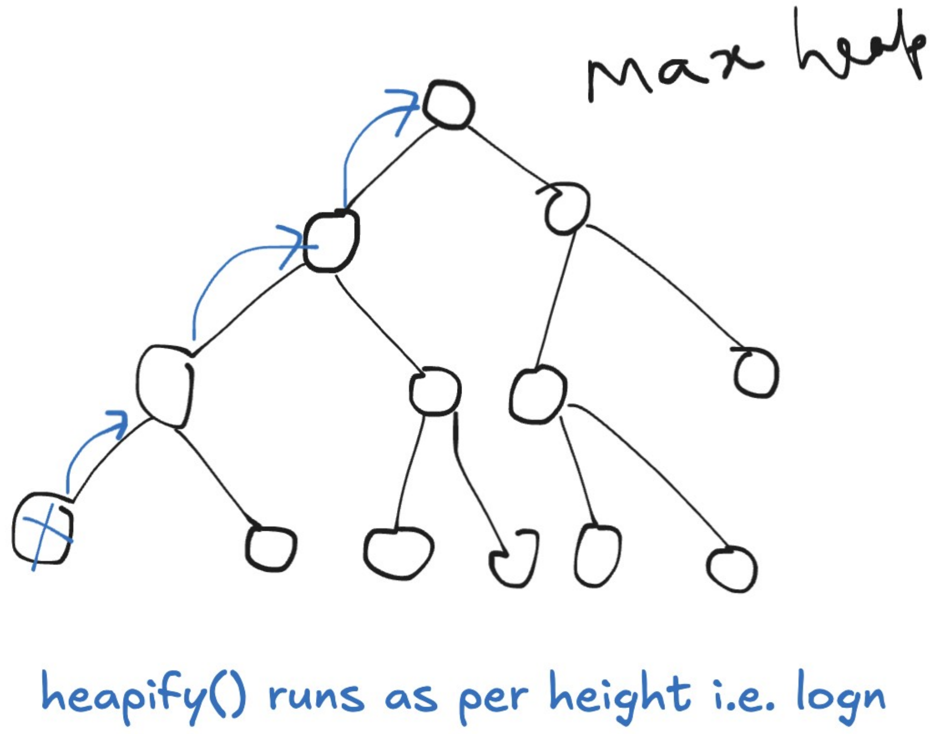
```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1): # O(n)
        min_index = i
        for j in range(i + 1, n): # n + (n-1) + (n-2) +...+2+1 = 
n*(n+1)/2
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Total Runtime Complexity:
# Big O notation: O(n*n)
# Big Omega notation: Ω(n*n)
# Big Theta notation: θ(n*n)
```

Explanation:

- First for loop runs `n` times.
- Second for loop first time runs `n`, second time `n-1` and at the end `1` time.
- Total comparsion = `n` + `(n-1)` + `(n-2)`+.....+`1` = `n+(n+1)/2` ~ `O(n*n)`
- It means run time complexity is `O(n*n)`
- Since second loop always runs with same size irrespective of input therefore total time complexity will be same in best. worst and avg case.

1E Heap Sort (10 points)

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest) # Recursive call runs as per height
of complete binary tree i.e. h = logn


def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1): # Build heap phase: O(n/2)
        heapify(arr, n, i) # O(logn)

    for i in range(n - 1, 0, -1): # Sort phase: O(n)
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0) # After ith largest is choosen, run rest on
[i,0) in O(logn)

# Total Runtime Complexity:
# Big O notation: O(n*logn)
# Big Omega notation: Ω(n*logn)
# Big Theta notation: θ(n*logn)
```



Explanation:

- During the build phase, heapify(n,i) runs starting from heapify(n,n), heapify(n,n-1), heapify(n,n-2)....heapify(n,n/2-1)
- Build phase loop runs n/2 times with heapify runs logn therefore overall time complexity would be O(nlogn)
- Sort phase run n times to pick the next max element and place at the end. After that heapify() scope is reduced to the (n-1,0)
- It means in sort phase heapify heapify(i,0) starting from heapify(n-1,0), heapify(n-2,0), heapify(n-3,0).....heapify(0,0)
- Sort phase loop runs n times with heapify logn therefore overall time complexity would be O(nlogn)
- Therefore overall time complexity is O(2*nlogn) i.e. O(nlogn)
- Please note; heap sort runs inplace. Therefore once max heap is built, the input array is converted into max heap.
- Pleasse also note; max heap converted array is not equal to sorted array.
- Therefore irrespective of input array, overall time complexity of heapify() stays O(logn).

Q2: Algorithm Design & Analysis (20 points)

For this question, we will take the above process one step further. We are going to provide you with a common algorithm, and you are to complete the following:

- Construct pseudocode for the algorithm.
- Analyze your pseudocode line by line.
- Provide the runtime complexity for the algorithm.
- Provide an explanation justifying your analysis of the algorithm.

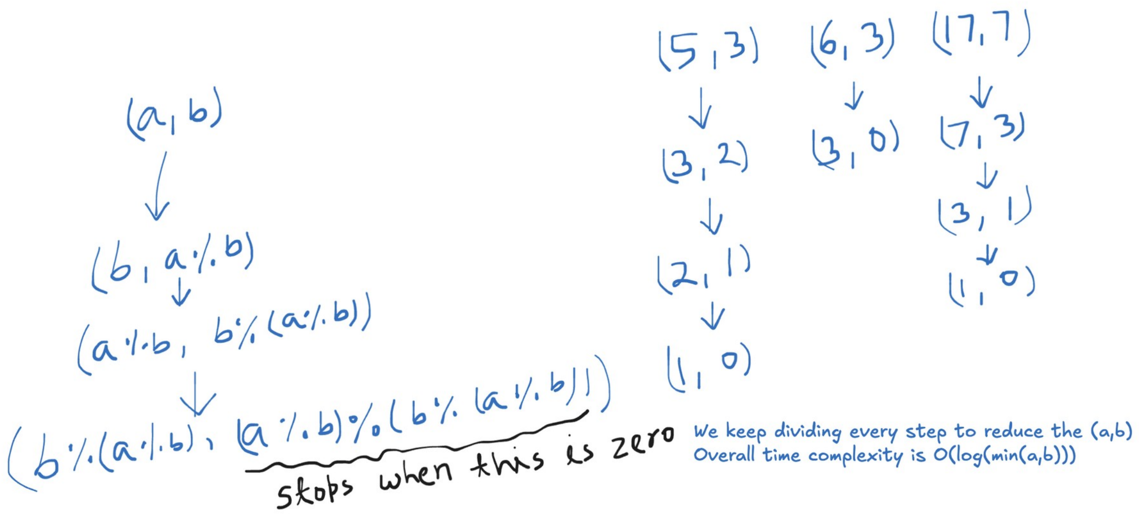
You may select from the following:

1. Euclidean Algorithm

- **Category**: Number theory

- **Purpose**: Finds the greatest common divisor (GCD) of two numbers using repeated division. Following is pseudo code

```
def gcd(a,b): # a = b*q + r, assumption is that a > b
    if (a < b): # To handle bad input if a < b. It runs only once.
        return gcd(b,a)
    if (b == 0):
        return a
    return gcd(b, a % b)
```



We keep dividing every step to reduce the (a,b)
Overall time complexity is O(log(min(a,b)))

Total Runtime Complexity: Big O notation: O(log(min(a,b))) Big Omega notation: Ω(1) Big Theta notation: Θ(O(log(min(a,b))))

Explanation:

- If b is 1 then gcd will return `1` in just two steps. Therefore in the best case, time compleixty will be O(1)
- Both worst case and average case will run O(log(min(a,b)))

2. Fibonacci Sequence (Iterative or Recursive)

- **Category**: Sequence generation

- **Purpose**: Generates the Fibonacci sequence, where each number is the sum of the two preceding ones.

Following is pseudo code for iterative version

```
def fib(n):
    # Base case
    if(n < 2):
        return n
    # initialization phase
    dp = [-1]*n
    dp[0] = 0
    dp[1] = 1
    for (i in range(2, n)): # O(n)
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n-1]
```

Total Runtime Complexity: Big O notation: O(n) Big Omega notation: Ω(n) Big Theta notation: Θ(n)

Explanation:

- for loop runs n-2 times
- Therefore overall time complexity is O(n)

3. Prime Number Checking (Trial Division)

- **Category**: Number theory

- **Purpose**: Determines whether a number is prime by checking divisibility up to its square root.

Following is pseudo algorithm

```
def is_prime(n):
    sqrt = math.sqrt(n)
    for i in range(2, sqrt+1): # Run sqrt times
        if(n % i == 0)
            return False
    return True
```

Total Runtime Complexity: Big O notation: O(sqrt(n)) Big Omega notation: Ω(1) Big Theta notation: Θ(sqrt(n))

Explanation:

- for loop runs sqrt(n) times
- Therefore overall time complexity is O(sqrt(n))
- In best case if number is even number then O(1) it will return False
- In worst case it will try all divisor from 2 to sqrt(n)

- In average case, it will also try all divisor from 2 to sqrt(n)

Q3: Probabilities and Distributions (30 points total)

Objective: Demonstrate understanding of probability distributions, moments, and their data science applications.

3A Dataset Identification (10 points)

Find a publicly available dataset that exhibits characteristics of a particular probability distribution (e.g., Poisson, Normal, Exponential).

Over the course of this question you will help prove that this dataset aligns with that chosen distribution.

```python
# Import Python Modules
import pandas as pd


# Import dataset
penguins = pd.read_csv('penguins.csv')

# Get the unique species
unique_species = penguins['species'].unique()
print("unique_species ", unique_species)

penguins.head()
```

```
unique_species  ['Adelie' 'Chinstrap' 'Gentoo']

  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  \
0  Adelie  Torgersen            39.1           18.7              181.0

1  Adelie  Torgersen            39.5           17.4              186.0

2  Adelie  Torgersen            40.3           18.0              195.0

3  Adelie  Torgersen             NaN            NaN                NaN

4  Adelie  Torgersen            36.7           19.3              193.0


   body_mass_g     sex
0       3750.0    MALE
1       3800.0  FEMALE
2       3250.0  FEMALE
```

```
3          NaN     NaN
4       3450.0  FEMALE
```

Describe your dataset:

Palmer Penguins dataset

This dataset contains data for 344 penguins observed in the Palmer Archipelago, Antarctica. It includes measurements like culmen (bill) length, culmen depth, flipper length, and body mass, along with species and sex.

Many of these continuous measurements, especially when considered for a single species, tend to be **normally distributed**.

We can download csv dataset from https://raw.githubusercontent.com/mwaskom/seaborn-data/master/penguins.csv

3B Compute the first four moments for each feature (column). (10 points)

Select a minimum of two features for your response.

Following are first four moments

1. Mean
2. Variance
3. Skewness
4. Kurtosis

We can use Pandas api to calculate the moment based on observed data we loaded into Pandas dataframe. Please note; following are list of Penguins species

1. Adelie
2. Chinstrap
3. Gentoo

The distinct biological differences between species mean that lumping them all together can obscure important patterns, create misleading averages, and hide the true nature of the distributions for each group. Therefore we will calculate these statistics for each species.

```
# Perform computation here.
import numpy as np

# We only compute moments for numerical columns
numerical_features = penguins.select_dtypes(include=np.number).columns

for feature in numerical_features:
    print(f"\nFeature: '{feature}'")

    # Drop NA values for the calculation of moments for that specific
```

```
column
    clean_feature_data = penguins[[feature,'species']].dropna()

    # Group by species, then select the feature, then aggregate to get
moments
    moments_by_species = clean_feature_data.groupby('species')
[feature].agg(
        Mean='mean',
        Variance='var',
        Skewness='skew',
        Kurtosis=pd.Series.kurtosis # or data.kurt()
    )
    print(moments_by_species)
```

```
Feature: 'bill_length_mm'
                Mean   Variance  Skewness  Kurtosis
species
Adelie     38.791391   7.093725  0.161674 -0.157204
Chinstrap  48.833824  11.150630 -0.090575  0.035860
Gentoo     47.504878   9.497845  0.651131  1.297106

Feature: 'bill_depth_mm'
                Mean  Variance  Skewness  Kurtosis
species
Adelie     18.346358  1.480237  0.321239 -0.060308
Chinstrap  18.420588  1.289122  0.006879 -0.874796
Gentoo     14.982114  0.962792  0.324231 -0.583455

Feature: 'flipper_length_mm'
                 Mean   Variance  Skewness  Kurtosis
species
Adelie     189.953642  42.764503  0.087337  0.331959
Chinstrap  195.823529  50.863916 -0.009472  0.046358
Gentoo     217.186992  42.054911  0.394850 -0.575221

Feature: 'body_mass_g'
                  Mean       Variance  Skewness  Kurtosis
species
Adelie     3700.662252  210282.891832  0.285336 -0.573738
Chinstrap  3733.088235  147713.454785  0.247433  0.593379
Gentoo     5076.016260  254133.180061  0.069635 -0.722791
```

3C Interpret these moments in the context of the dataset and compare them to the theoretical values of the selected distribution. (10 points)

At a minimum you should perform the following:

- Compute the theoretical values of the first four moments.
- Plot the data and fit the chosen distribution visually (using histograms, bar charts, etc.)
- Provide a written analysis of the differences between the theoretical and calculated values.

Calculated moments

These are the moments (mean, variance, skewness, kurtosis) computed directly from given sampled data i.e. it doesn't use entire population instead uses observed sampled data. $\acute{x}$ is used to represent sample mean and s^2 to represent sample variance.

- When we calculate `penguins[feature].mean()`, `penguins[feature].var()`, `penguins[feature].skew()`, and `penguins[feature].kurt()`, we are computing the calculated (sample) moments for that specific column by using all the sampled data.

Note: In the 3B section, we did computed the calculated moments based on `penguins.csv` as sampled data.

Theoretical moments

These are the true, underlying statistical properties of an entire population or a probability distribution. They are often represented by Greek letters (e.g., μ for mean, σ^2 for variance).

Since it is nearly impossible to observe the data for entire population therefore we always estimate the moments of entire population based on given sampled data. That's why we call it Theoretical moments.

If we knew the true underlying probability distribution from which the data was drawn (e.g., if we knew `bill_length_mm` was perfectly Normally distributed with a specific mean and variance), then those parameters would be the theoretical moments.

Since we don't know the true distribution, a common approach is to:

- Assume a distribution type: let's say, the Normal (Gaussian) distribution as reference.
- Fit the assumed distribution to the data: This means finding the parameters of the chosen distribution (e.g., mean and standard deviation for a Normal distribution) that best match the observed data.
- Use the parameters of the fitted distribution as "theoretical" moments.

Examples:

- If you want to know the average height of all adult humans on Earth, then the population is every adult human on Earth.
- If you want to know the average bill length of all Adelie penguins in Antarctica, then the population is every single Adelie penguin in Antarctica.

Use of stats.norm.fit()

When we use stats.norm.fit(data), we're giving it our sample data. The function then estimates the parameters (mean and standard deviation) of a Normal distribution that would be most likely to produce your observed sample. These fitted_mean and fitted_std_dev are therefore estimates of the true population mean (μ) and true population standard deviation (σ), assuming the underlying population is indeed Normally distributed.

How to asses the correctness of chosen distribution?

One of the approach is the visual inspection. Following are few

1. Histograms and Density Plots

- Plot a histogram of the data.
- Overlay the probability density function (PDF) of the chosen distribution (e.g., Normal distribution) using the parameters estimated from our data (like the fitted_mean and fitted_std_dev we calculated earlier).
- **What to look for**: Does the shape of the PDF curve reasonably match the shape of the histogram? Are there major discrepancies in peaks, symmetry, or tails?

1. Q-Q Plots (Quantile-Quantile Plots)

- This is one of the best visual tools for assessing normality (or any other distributional fit).
- It plots the quantiles of your data against the theoretical quantiles of the chosen distribution.

A Perfect Normal Distribution

A perfect Normal Distribution has:

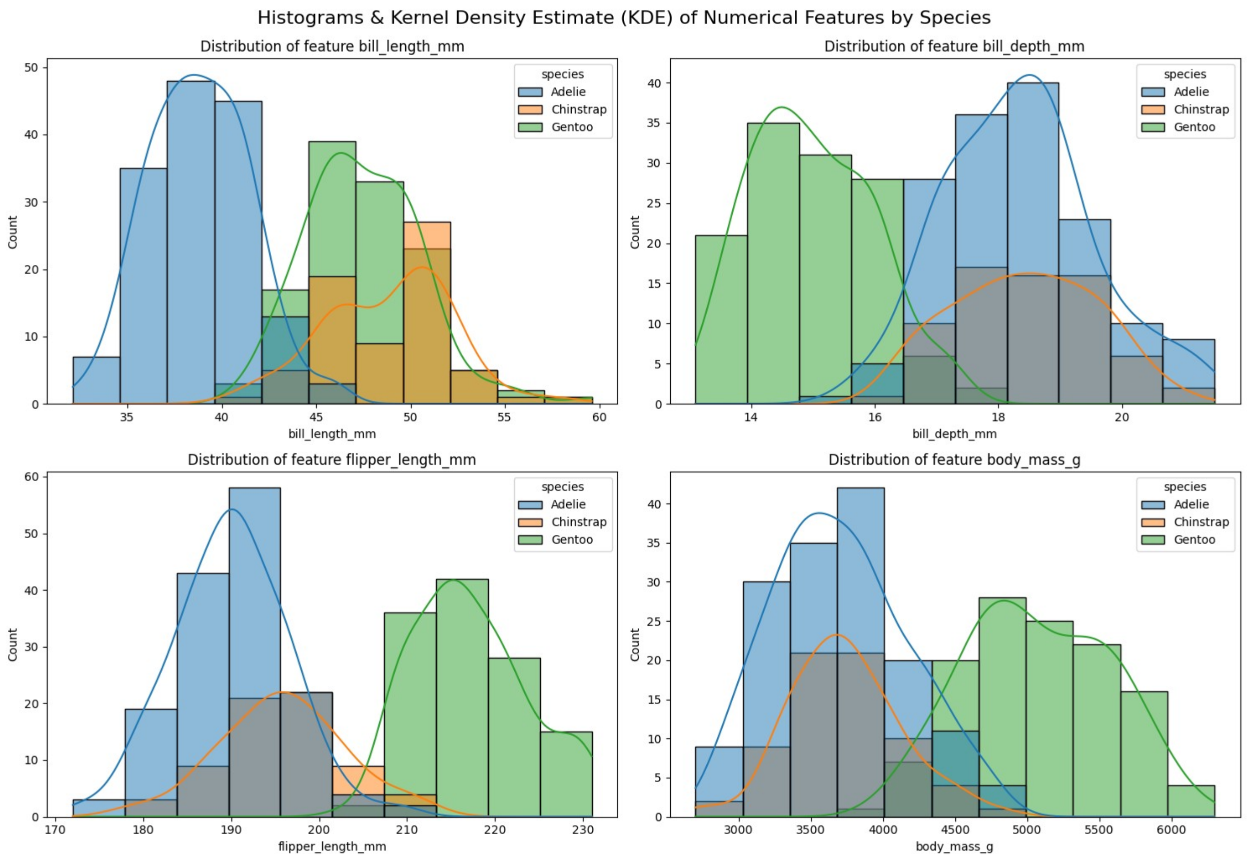
- Skewness = 0 (perfectly symmetrical)
- Excess Kurtosis = 0 (mesokurtic, meaning its tails and peak are like a normal distribution's)

```
from scipy import stats # For fitting normal distribution and
calculating moments
results_list = [] # To store results for a final DataFrame
# Iterate each numerial features
for feature in numerical_features:

    # Drop NA values for the calculation of moments for that specific
column
    clean_feature_data = penguins[[feature,'species']].dropna()

    # Iterate over each species
    for species_name in clean_feature_data['species'].unique():
        # Filter data for the current species and feature
        species_feature_data =
clean_feature_data[clean_feature_data['species'] == species_name]
[feature]

        # Calculated (Empirical) Moments for this species
        calc_mean = species_feature_data.mean()
        calc_variance = species_feature_data.var()
        calc_std = species_feature_data.std()
        calc_skewness = species_feature_data.skew()
        calc_kurtosis = species_feature_data.kurtosis()

        # Fit a normal distribution to the species data to calculate
theoretical moments
        fitted_mean, fitted_std_dev =
stats.norm.fit(species_feature_data)
        theo_mean_species = fitted_mean
        theo_std_dev_species = fitted_std_dev
        theo_variance_species = fitted_std_dev**2
        theo_skewness_species = 0.0  # Theoretical skewness for a
perfect Normal distribution
        theo_kurtosis_species = 0.0  # Theoretical excess kurtosis for
a perfect Normal distribution

        # Storing results
        results_list.append({
            'Feature': feature,
            'Species': species_name,
            'Mean': round(calc_mean, 2),
            'Theo Mean': round(theo_mean_species, 2),
            'Variance': round(calc_variance, 2),
            'Theo Variance': round(theo_variance_species, 2),
            'Std Dev': round(calc_std, 2),
            'Theo Std Dev': round(theo_std_dev_species, 2),
            'Skewness': round(calc_skewness, 2),
            'Theo Skewness': round(theo_skewness_species, 2),
            'Kurtosis': round(calc_kurtosis, 2),
            'Theo Kurtosis': round(theo_kurtosis_species, 2),
        })
# Convert the list of dictionaries to a DataFrame for a nice summary
table
results_df = pd.DataFrame(results_list)
print("\n\n--- Summary Table: Calculated vs. Theoretical Moments by
Feature and Species using Normal Distribution ---\n")
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 175)
print(results_df)
```

```
--- Summary Table: Calculated vs. Theoretical Moments by Feature and
Species using Normal Distribution ---

              Feature    Species     Mean  Theo Mean   Variance  Theo
Variance  Std Dev  Theo Std Dev  Skewness  Theo Skewness  Kurtosis
Theo Kurtosis
0      bill_length_mm     Adelie    38.79      38.79       7.09
7.05     2.66          2.65      0.16            0.0     -0.16
0.0
1      bill_length_mm  Chinstrap    48.83      48.83      11.15
10.99     3.34          3.31     -0.09            0.0      0.04
0.0
2      bill_length_mm     Gentoo    47.50      47.50       9.50
9.42     3.08          3.07      0.65            0.0      1.30
0.0
3       bill_depth_mm     Adelie    18.35      18.35       1.48
1.47     1.22          1.21      0.32            0.0     -0.06
0.0
4       bill_depth_mm  Chinstrap    18.42      18.42       1.29
1.27     1.14          1.13      0.01            0.0     -0.87
0.0
5       bill_depth_mm     Gentoo    14.98      14.98       0.96
0.95     0.98          0.98      0.32            0.0     -0.58
0.0
6   flipper_length_mm     Adelie   189.95     189.95      42.76
42.48     6.54          6.52      0.09            0.0      0.33
0.0
7   flipper_length_mm  Chinstrap   195.82     195.82      50.86
50.12     7.13          7.08     -0.01            0.0      0.05
0.0
8   flipper_length_mm     Gentoo   217.19     217.19      42.05
41.71     6.48          6.46      0.39            0.0     -0.58
0.0
9         body_mass_g     Adelie  3700.66    3700.66  210282.89
208890.29   458.57        457.05      0.29            0.0     -0.57
0.0
10        body_mass_g  Chinstrap  3733.09    3733.09  147713.45
145541.20   384.34        381.50      0.25            0.0      0.59
0.0
11        body_mass_g     Gentoo  5076.02    5076.02  254133.18
252067.06   504.12        502.06      0.07            0.0     -0.72
0.0

%pip install seaborn
import matplotlib.pyplot as plt
import seaborn as sns

# Visualize Distributions for Normality Check
```

```python
plt.figure(figsize=(15, 10))
for i, feature in enumerate(numerical_features):
    df_for_plot = penguins[[feature, 'species']].copy()
    df_for_plot.dropna(subset=[feature, 'species'], inplace=True)
    plt.subplot(2, 2, i + 1)
    sns.histplot(data=df_for_plot, x=feature, kde=True, hue='species')
    plt.title(f'Distribution of feature {feature}')
plt.tight_layout()
plt.suptitle('Histograms & Kernel Density Estimate (KDE) of Numerical
Features by Species', y=1.02, fontsize=16)
plt.show()
```

```
Matplotlib is building the font cache; this may take a moment.
```



Interpretation of Moments for Each Feature

1. Mean and Standard Deviation (and Variance):

Observation: For all features and species, the Calculated Mean is practically identical to the Theoretical Mean, and the Calculated Standard Deviation is very close to the Theoretical Standard Deviation (and thus variance). **Analysis**: This is expected. When you fit a normal distribution to data using stats.norm.fit(), it estimates the mean and standard deviation of that normal distribution directly from the sample's mean and standard deviation using Maximum

Likelihood Estimation (MLE). Therefore, the fitted mean and standard deviation will closely match the calculated mean and standard deviation of the actual data. Small differences might arise due to rounding or slight variations in how the fitting algorithm handles floating-point numbers, but they are generally negligible. This indicates that a normal distribution's central tendency and spread can effectively approximate that of the empirical data, even if the shape isn't perfectly normal.

Skewness (Calculated vs. Theoretical):

The Theo Skewness is hardcoded to 0.0. **Reason**: A perfect, theoretical normal distribution is perfectly symmetrical and thus has a skewness of 0. **Analysis Point**: The important comparison here is how close the Calculated Skewness is to 0.0. Values close to 0 indicate good symmetry, aligning well with the assumption of normality.

Looking result closely shows that following are following Normal distribution.

Feature	Species	Skewness	Theo Skewness
bill_length_mm	Chinstrap	-0.09	0.0
bill_depth_mm	Chinstrap	0.01	0.0
flipper_length_mm	Adelie	0.09	0.0
flipper_length_mm	Chinstrap	-0.01	0.0
body_mass_g	Gentoo	0.07	0.0

Following are not following Normal distribution

Feature	Species	Skewness	Theo Skewness
bill_length_mm	Gentoo	0.65	0.0
bill_depth_mm	Adelie	0.32	0.0
bill_depth_mm	Gentoo	0.32	0.0
flipper_length_mm	Gentoo	0.39	0.0
body_mass_g	Adelie	0.29	0.0
body_mass_g	Chinstrap	0.25	0.0

Kurtosis (Calculated vs. Theoretical):

The Theo Kurtosis (excess kurtosis) is hardcoded to 0.0. **Reason**: A perfect, theoretical normal distribution has an excess kurtosis of 0 (its Pearson's kurtosis is 3). **Analysis Point**: The important comparison here is how close the Calculated Kurtosis (which in pandas is excess kurtosis by default) is to 0.0. Values close to 0 indicate that the "tailedness" and "peakedness" of the data are similar to a normal distribution.

Following features has kurtosis closed to zero.

Feature	Species	Kurtosis	Theo Kurtosis
bill_length_mm	Chinstrap	0.04	0.0
bill_depth_mm	Adelie	-0.06	0.0
flipper_length_mm	Chinstrap	0.05	0.0

Following features has kurtosis non zero.

Feature	Species	Kurtosis	Theo Kurtosis
bill_length_mm	Adelie	-0.16	0.0
bill_length_mm	Gentoo	1.30	0.0
bill_depth_mm	Chinstrap	-0.87	0.0
bill_depth_mm	Gentoo	-0.58	0.0
flipper_length_mm	Adelie	0.33	0.0
flipper_length_mm	Gentoo	-0.58	0.0
body_mass_g	Adelie	-0.57	0.0
body_mass_g	Chinstrap	0.59	0.0
body_mass_g	Gentoo	-0.72	0.0