```python
import numpy as np

def array_compare(X_n, Y_n, mean_num):
    """
    Compare arrays based on a mean value and calculate statistics for
values above and below the mean.

    Parameters:
    -----------
    X_n : array-like
        Input array of numerical values.
    Y_n : array-like
        Binary array (0s and 1s) corresponding to X_n.
    mean_num : float
        Mean value to split the array into two parts.

    Returns:
    --------
    list
        A nested list containing:
        - [[mean_num]]: The mean value used for comparison
        - [lower_percentage]: Percentage of 0s and 1s in the lower
partition (≤ mean)
        - [higher_percentage]: Percentage of 0s and 1s in the higher
partition in the form of (> mean)

    Notes:
    ------
    - Values equal to mean_num are included in the lower partition
    - Percentages are returned as [percentage_of_0s, percentage_of_1s]
    """
    X_n = np.array(X_n)
    Y_n = np.array(Y_n)

    # Create masks for partitioning
    lower_mean_mask = X_n <= mean_num
    higher_mean_mask = X_n > mean_num

    # Split X array into lower and higher partitions
    lower_array = X_n[lower_mean_mask]
    higher_array = X_n[higher_mean_mask]

    # Calculate sizes of partitions
    sum_low = len(lower_array)
    sum_high = len(higher_array)

    # Split Y array according to masks
    higher_y = Y_n[higher_mean_mask]
    lower_y = Y_n[lower_mean_mask]
```

```
    # Count occurrences of 1s and 0s in each partition
    lower_1 = np.sum(lower_y == 1)
    higher_1 = np.sum(higher_y == 1)
    lower_0 = np.sum(lower_y == 0)
    higher_0 = np.sum(higher_y == 0)

    # Calculate percentages for each partition
    lower_percentage = [lower_0/sum_low, lower_1/sum_low]
    higher_percentage = [higher_0/sum_high, higher_1/sum_high]

    return [[mean_num], [lower_percentage], [higher_percentage]]

def generate_arrays(random_seed=9999, use_random=False):
    """
    Generate or return predefined arrays for testing.

    Parameters:
    -----------
    random_seed : int, optional (default=89)
        Seed for random number generation.
    use_random : bool, optional (default=False)
        If True, generates random arrays. If False, returns predefined
arrays.

    Returns:
    --------
    tuple
        - X : numpy.ndarray
            2D array of features
        - y : numpy.ndarray
            1D array of binary labels (0s and 1s)
    """
    if use_random:
        np.random.seed(random_seed)
        random_sample_num = np.random.randint(4, 20)
        random_features_num = np.random.randint(2, 20)
        X = np.random.uniform(0, 10, (random_sample_num,
random_features_num))
        y = np.random.randint(0, 2, random_sample_num)
    else:
        X = np.array([[3.3,2.1],[1.0,1.1],[1.5,1.5],[3.9,3.0],
[4.4,1.6],
                      [5.1,0.6],[4.0,2.9],[2.0,1.0],[3.1,0.5]])
        y = np.array([0,1,1,0,1,0,0,1,0])

    return X, y

def sorted_corresponding_mean_calculation(X, y):
```

```
    """
    Calculate means at transition points between 0s and 1s in sorted
arrays. And returns mean value and statistic of corresponding y array

    Parameters:
    -----------
    X : numpy.ndarray
        2D input array of features
    y : numpy.ndarray
        1D array of binary labels (0s and 1s)

    Returns:
    --------
    list
        List of results from array_compare for each identified mean
value
        at transition points.

    Notes:
    ------
    The function:
    1. Splits X into columns and sorts them with corresponding y
values
    2. Identifies pairs of adjacent points where y values change (0->1
or 1->0)
    3. Calculates means of these pairs
    4. Applies array_compare using these means
    """
    # Split into columns and sort
    column_split = [X[:, x] for x in range(X.shape[1])]
    sorted_columns = []
    sorted_y = []

    # Sort columns and corresponding y values
    for column in column_split:
        sorted_indices = np.argsort(column)
        sorted_columns.append(column[sorted_indices])
        sorted_y.append(y[sorted_indices])

    # Find transition points and calculate means
    all_groups = []
    all_means = []
    for i, (col, y_col) in enumerate(zip(sorted_columns, sorted_y)):
        column_groups = []
        column_means = []

        # Find transitions between 0s and 1s
        for j in range(len(y_col)):
            if y_col[j] == 1:
```

```python
                if j > 0 and y_col[j - 1] == 0:
                    pair = col[j - 1:j + 1]
                    column_groups.append(pair)
                if j < len(y_col) - 1 and y_col[j + 1] == 0:
                    pair = col[j:j + 2]
                    column_groups.append(pair)

        # Calculate means for transition points
        for g in column_groups:
            column_means.append(np.mean(g))

        all_groups.append(column_groups)
        all_means.append(column_means)

    # Calculate results for each mean value
    results = []
    for col_idx, means_in_column in enumerate(all_means):
        for mean_idx, mean_value in enumerate(means_in_column):
            results.append(array_compare(sorted_columns[col_idx],
                                         sorted_y[col_idx],
                                         mean_value))
    return results
# test
X, y = generate_arrays(use_random=False,random_seed=1222222)
results = sorted_corresponding_mean_calculation(X, y)
print(results)
```

```
[[[np.float64(2.55)], [[np.float64(0.0), np.float64(1.0)]],
[[np.float64(0.8333333333333334), np.float64(0.16666666666666666)]]],
[[np.float64(4.2)], [[np.float64(0.5714285714285714),
np.float64(0.42857142857142855)]], [[np.float64(0.5),
np.float64(0.5)]]], [[np.float64(4.75)], [[np.float64(0.5),
np.float64(0.5)]], [[np.float64(1.0), np.float64(0.0)]]],
[[np.float64(0.8)], [[np.float64(1.0), np.float64(0.0)]],
[[np.float64(0.42857142857142855), np.float64(0.5714285714285714)]]],
[[np.float64(1.85)], [[np.float64(0.3333333333333333),
np.float64(0.6666666666666666)]], [[np.float64(1.0),
np.float64(0.0)]]]]
```