Exercice 2 (Régression):

Approximation de la colonne tip dans la base d'exemples Tips en utilisant le modèle d'arbre de décision CART.

In [18]:
```
import numpy as np
import pandas as pd
```

Fonction *compute_avg_leaf_value(Y):*

Cette fonction calcule la valeur moyenne de la variable cible Y, ce qui représente la valeur du nœud feuille dans l'arbre de décision. Elle est utilisée pour déterminer la valeur prédite lorsque l'arbre atteint une feuille.

In [28]:
```
def compute_avg_leaf_value(Y):
    return np.mean(Y)
```

Fonction find_optimal_partition(dataset, num_samples, num_features, min_samples, depth_max):

Cette fonction trouve le meilleur point de séparation dans le jeu de données en fonction du gain d'information. Elle parcourt toutes les caractéristiques et leurs seuils possibles pour trouver la séparation qui maximise la réduction de variance.

In [29]:
```
def find_optimal_partition(dataset, num_samples, num_features, min_samples, depth_max):
    optimal_partition = {}
    max_var_reduction = -float("inf") # Initialise la réduction de variance maximale à moins l'infini

    # Parcours de toutes les caractéristiques
    for feature_idx in range(num_features):
        feature_values = dataset[:, feature_idx]
        possible_splits = np.unique(feature_values)

        ## Parcours de toutes les valeurs uniques de la caractéristique
        for split_threshold in possible_splits:
             # Divise l'ensemble de données en fonction de la valeur actuelle de la caractéristique
            left_data, right_data = perform_split(dataset, feature_idx, split_threshold)

            if len(left_data) > 0 and len(right_data) > 0:  # Assure que la division a donné des ensembles de données gauche et droit non vides
                # Calcul de la réduction de variance après la division
                y, left_y, right_y = dataset[:, -1], left_data[:, -1], right_data[:, -1]
                var_reduction = calculate_variance_reduction(y, left_y, right_y)
                # Met à jour les détails de la partition optimale si la réduction de variance est plus élevée
                if var_reduction > max_var_reduction:
                    optimal_partition["feature_idx"] = feature_idx
                    optimal_partition["threshold"] = split_threshold
                    optimal_partition["left_data"] = left_data
                    optimal_partition["right_data"] = right_data
                    optimal_partition["var_reduction"] = var_reduction
                    max_var_reduction = var_reduction # Met à jour la réduction de variance maximale

    return optimal_partition # Renvoie la meilleure partition trouvée
```

Fonction *perform_split(dataset, feature_idx, threshold)*

Cette fonction divise le jeu de données en deux sous-ensembles en fonction d'une caractéristique et d'un seuil donnés. Elle crée un sous-ensemble dataset_left contenant les lignes où la valeur de la caractéristique est inférieure ou égale au seuil, et un sous-ensemble dataset_right contenant les lignes où la valeur de la caractéristique est supérieure au seuil.

In [30]:
```
def perform_split(dataset, feature_idx, threshold):
    # Sépare l'ensemble de données en deux sous-ensembles en fonction de la caractéristique et du seuil donnés
    left_subset = np.array([row for row in dataset if row[feature_idx] <= threshold])  # Crée le sous-ensemble gauche
    right_subset = np.array([row for row in dataset if row[feature_idx] > threshold]) # Crée le sous-ensemble droit
    return left_subset, right_subset  # Renvoie les sous-ensembles gauche et droit après la division
```

Fonction *calculate_variance_reduction(parent, left_child, right_child):*

Cette fonction calcule la réduction de variance, qui est une mesure de la réduction de la variance de la variable cible après la séparation. Elle prend en entrée le jeu de données parent et ses sous-ensembles gauche et droit.

In [31]:
```
def calculate_variance_reduction(parent, left_child, right_child):
    # Calcule la réduction de la variance résultant de la division des données
    weight_l = len(left_child) / len(parent)
    weight_r = len(right_child) / len(parent)
    # Calcule la réduction de la variance en soustrayant la somme pondérée des variances des enfants de la variance du parent
    reduction = np.var(parent) - (weight_l * np.var(left_child) + weight_r * np.var(right_child))
    return reduction  # Renvoie la réduction de la variance
```

Fonction *construct_decision_tree(dataset, min_samples, depth_max, curr_depth=0):*

Cette fonction récursive construit l'arbre de décision. Elle vérifie les conditions d'arrêt (nombre minimum d'échantillons et profondeur maximale) et si elles ne sont pas satisfaites, elle trouve la meilleure séparation, crée des sous-arbres gauche et droit, et renvoie un nœud de décision. Si les conditions d'arrêt sont satisfaites, elle calcule la valeur du nœud feuille.

In [32]:
```
def construct_decision_tree(dataset, min_samples, depth_max, curr_depth=0):
    X, Y = dataset[:, :-1], dataset[:, -1]
    num_samples, num_features = np.shape(X)
    optimal_partition = {}

    if num_samples >= min_samples and curr_depth <= depth_max:
         # Trouve la partition optimale pour diviser les données
        optimal_partition = find_optimal_partition(dataset, num_samples, num_features, min_samples, depth_max)

        if optimal_partition and optimal_partition["var_reduction"] > 0:  # Vérifie si une partition optimale a été trouvée et si la réduction de variance est positive
            # Construction récursive des branches gauche et droite de l'arbre de décision
            left_branch = construct_decision_tree(optimal_partition["left_data"], min_samples, depth_max, curr_depth + 1)
            right_branch = construct_decision_tree(optimal_partition["right_data"], min_samples, depth_max, curr_depth + 1)

            return {
                "feature_idx": optimal_partition["feature_idx"],
                "threshold": optimal_partition["threshold"],
                "left": left_branch,
                "right": right_branch,
                "var_reduction": optimal_partition["var_reduction"]
            }

    return {"value": compute_avg_leaf_value(Y)} # Retourne une feuille avec la valeur moyenne des feuilles
```

Fonction *predict_single_data(x, tree)*

Cette fonction prédit la valeur de la variable cible pour un nouveau point de données x en fonction de l'arbre de décision. Elle traverse récursivement l'arbre jusqu'à atteindre un nœud feu

In [33]:
```
def predict_single_data(x, tree):
    # Vérifie si le nœud est une feuille et renvoie sa valeur si c'est le cas
    if tree["value"] is not None:
        return tree["value"]
        # Obtient la valeur de la fonctionnalité pour l'échantillon
    feature_val = x[tree["feature_idx"]]
     # Détermine la branche à suivre en fonction de la valeur de la fonctionnalité
    if feature_val <= tree["threshold"]:
        return predict_single_data(x, tree["left"])
    else:
        return predict_single_data(x, tree["right"])
```

In [35]:
```
def display_decision_tree(tree=None, indent=" "):
    if tree is None:
        return
    if isinstance(tree, dict):
        if "value" in tree:
            print(tree["value"])
        else:
            print("X_" + str(tree["feature_idx"]), "<=", tree["threshold"], "?", tree["var_reduction"])
            print("%sleft:" % (indent), end="")
            display_decision_tree(tree["left"], indent + indent)
            print("%sright:" % (indent), end="")
            display_decision_tree(tree["right"], indent + indent)
    else:
        print(tree)

# Function to train the decision tree
def train_tree(X, Y, min_samples=2, depth_max=2):
    dataset = np.column_stack((X, Y))
    return construct_decision_tree(dataset, min_samples, depth_max)

# Function to predict using the decision tree
def predict_tree(X, tree):
    return [predict_single_data(x, tree) for x in X]
```

In [37]:
```
# Load the dataset
data = pd.read_csv("tips.csv")

X = data.iloc[:, :-1].values #Extraction des caractéristiques du jeu de données dans la variable "X" en sélectionnant toutes les lignes et toutes les colonnes sauf la dernière
Y = data.iloc[:, -1].values.reshape(-1, 1) #Extraction de la variable cible du jeu de données dans la variable "Y" en sélectionnant toutes les lignes de la dernière colonne et en la remodela

from sklearn.model_selection import train_test_split

# Split the dataset
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2)

# Initialize and fit the model
decision_tree = train_tree(X_train, Y_train, min_samples=3, depth_max=3)

# Affichage de la structure de l'arbre de décision entraîné en utilisant la fonction print_tree.
display_decision_tree(decision_tree)
```

```
X_0 <= 23.95 ? 0.2953852130775213
 left:X_0 <= 16.0 ? 0.08998460746712483
  left:X_0 <= 10.07 ? 0.014016035633128254
    left:X_1 <= 1.0 ? 0.040000000000001
         left:1.3333333333333333
         right:1.8333333333333333
    right:X_0 <= 10.33 ? 0.0024681892034978253
         left:2.2
         right:2.017543859649123
  right:X_1 <= 2.45 ? 0.057251811103968935
    left:X_4 <= Sun ? 0.18750000000000006
         left:3.3333333333333335
         right:2.3333333333333335
    right:X_1 <= 4.3 ? 0.049554183813442676
         left:2.4
         right:3.25
 right:X_3 <= No ? 0.2962799295618467
  left:X_0 <= 26.41 ? 0.1950342562708791
    left:X_0 <= 25.29 ? 0.2232142857142856
         left:3.4285714285714284
         right:2.0
    right:X_4 <= Sun ? 0.3448979591836734
         left:3.8666666666666667
         right:5.166666666666667
  right:X_0 <= 34.63 ? 0.1367134755603563
    left:X_0 <= 30.46 ? 0.17755681818181834
         left:2.909090909090909
         right:2.0
    right:X_0 <= 38.73 ? 0.24489795918367352
         left:4.0
         right:3.0
```