1a. Best-First Search

In [1]:
```python
from queue import PriorityQueue


def best_first_search(graph,start,goal,heuristic):
    visited = set()
    pq = PriorityQueue()
    pq.put((heuristic[start],start))

    while not pq.empty():
        h,node = pq.get()

        if node == goal:
            print("Goal Reached :",node)
            return

        if node not in visited:
            for neighbor in graph[node]:
                if neighbor not in visited:
                    pq.put((heuristic[neighbor],neighbor))
            print("Visiting Node : ",node)
            visited.add(node)
    print("Goal Not Found!!")
```

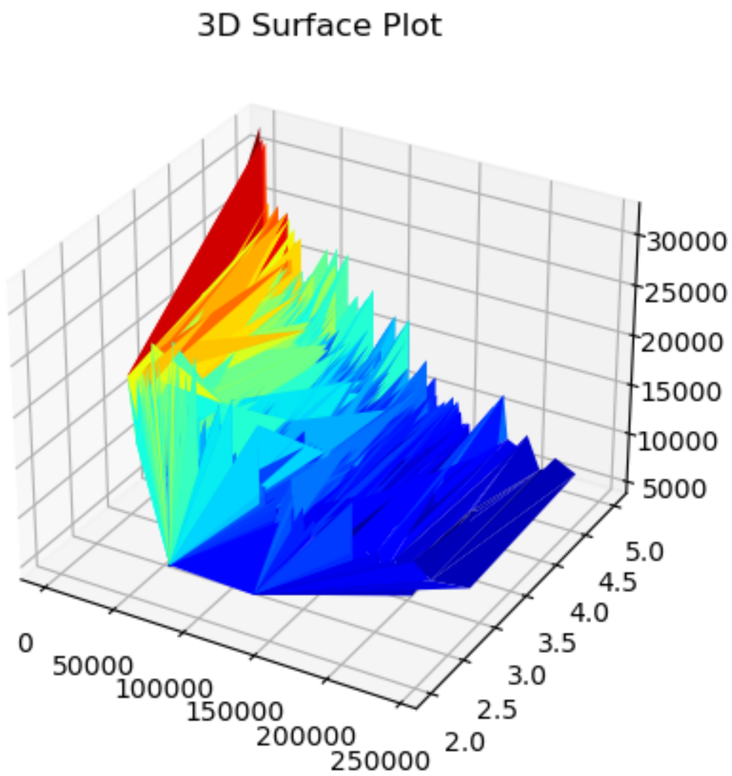
In [2]:
```python
graph = {
    'S':['A','B'],
    'A': ['C', 'D'],
    'B': ['E', 'F'],
    'C': [],
    'D': [],
    'E': ['H'],
    'F': ['I', 'G'],
    'H':[],
    'I':[],
    'G':[],
}

start_node = 'S'
goal_node = 'G'

#Heuristic values from curr node -> goal node
heuristic_values = {
    'S': 13,
    'A': 12,
    'B': 4,
    'C': 7,
    'D': 3,
    'E': 8,
    'F': 2,
    'H': 4,
    'I': 9,
    'G': 0,
}

best_first_search(graph, start_node, goal_node, heuristic_values)
```

```
Visiting Node :  S
Visiting Node :  B
Visiting Node :  F
Goal Reached : G
```

1b. 3D-Plot

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

In [4]:
```
dataset = pd.read_csv('./corolla.csv')
x = dataset['KM']
y = dataset['Doors']
z = dataset['Price']

ax = plt.axes(projection='3d')
ax.plot_trisurf(x,y,z,cmap="jet")
ax.set_title("3D Surface Plot")

plt.show()
```



02a. A-Star Algorithm

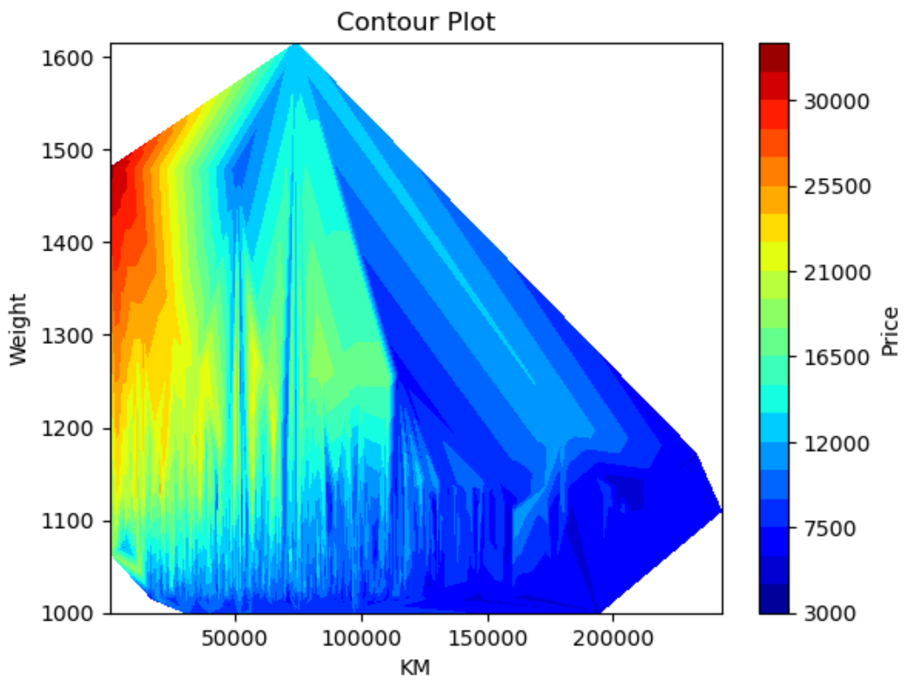
In [1]:
```
import heapq

def a_star_search(graph, start, goal, heuristic, cost):
    # Priority queue for exploring nodes
    priority_queue = []
    heapq.heappush(priority_queue, (0 + heuristic[start], start))
    visited = set()
    g_cost = {start: 0}
    parent = {start: None}

    while priority_queue:
        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node in visited:
            continue

        visited.add(current_node)

        if current_node == goal:
            break

        for neighbor in graph[current_node]:
            new_cost = g_cost[current_node] + cost[(current_node, neighbor)]
            if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_cost
                f_cost = new_cost + heuristic[neighbor]
                heapq.heappush(priority_queue, (f_cost, neighbor))
                parent[neighbor] = current_node

    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = parent[node]
    path.reverse()

    return path
```

In [2]:
```
# Example graph
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}

# Example heuristic values (assumed for demonstration)
heuristic = {
    'A': 6,
    'B': 4,
    'C': 4,
    'D': 0,
    'E': 2,
    'F': 3,
    'G': 1
}

# Example costs between nodes (assumed for demonstration)
cost = {
    ('A', 'B'): 1,
    ('A', 'C'): 1,
    ('B', 'D'): 1,
    ('B', 'E'): 3,
    ('C', 'F'): 5,
    ('C', 'G'): 2
}

start = 'A'
goal = 'D'

path = a_star_search(graph, start, goal, heuristic, cost)
print("A* Search Path:", path)
```

```
A* Search Path: ['A', 'B', 'D']
```

02b. Contour Plot

In [3]:
```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

dataset = pd.read_csv('./corolla.csv')
x = dataset['KM']
y = dataset['Weight']
z = dataset['Price']

plt.tricontourf(x, y, z, levels=20, cmap='jet')
plt.colorbar(label='Price')
plt.xlabel('KM')
plt.ylabel('Weight')
plt.title('Contour Plot')
plt.show()
```



3a. MinMax Algorithm

In [1]:
```
def minmax(depth, nodeIndex, maximizingPlayer, values, path):
    if depth == 3:
        return values[nodeIndex], path + [nodeIndex]

    if maximizingPlayer:
        best = float('-inf')
        best_path = []
        for i in range(2):
            val, new_path = minmax(depth + 1, nodeIndex * 2 + i, False, values, path + [nodeIndex])
            if val > best:
                best = val
                best_path = new_path
        return best, best_path
    else:
        best = float('inf')
        best_path = []
        for i in range(2):
            val, new_path = minmax(depth + 1, nodeIndex * 2 + i, True, values, path + [nodeIndex])
            if val < best:
                best = val
                best_path = new_path
        return best, best_path
```

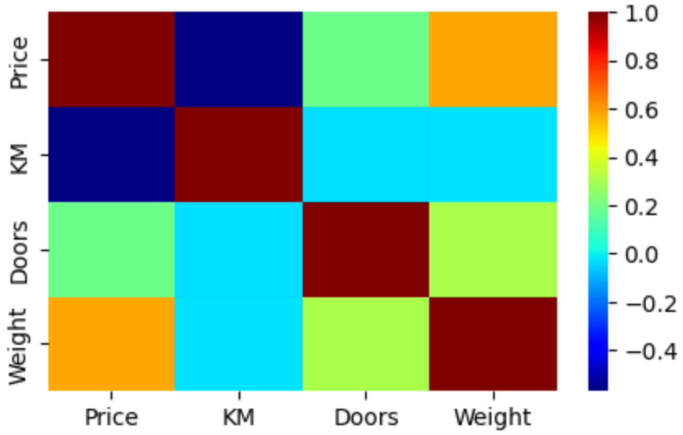
In [2]:
```
# Example tree with depth 3 and 8 terminal nodes
values = [3, 5, 2, 9, 12, 5, 23, 23]

# Start the Min-Max algorithm
optimal_value, optimal_path = minmax(0, 0, True, values, [])
print("The optimal value is:", optimal_value)
print("The path taken is:", optimal_path)
```

```
The optimal value is: 12
The path taken is: [0, 1, 2, 4]
```

3b. Heat Map

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [5]:
```
data=pd.read_csv("./corolla.csv")
plt.figure(figsize = ( 5 , 3 ))
sns.heatmap(data[["Price","KM","Doors", "Weight"]].corr(),cmap='jet')
plt.show()
```



4a. Alpha-Beta Pruning

In [1]:
```
def alphabeta(depth, nodeIndex, maximizingPlayer, values, alpha, beta, path):
    if depth == 3:
        return values[nodeIndex], path + [nodeIndex]

    if maximizingPlayer:
        best = float('-inf')
        best_path = []
        for i in range(2):
            val, new_path = alphabeta(depth + 1, nodeIndex * 2 + i, False, values, alph
            if val > best:
                best = val
                best_path = new_path
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best, best_path
    else:
        best = float('inf')
        best_path = []
        for i in range(2):
            val, new_path = alphabeta(depth + 1, nodeIndex * 2 + i, True, values, alpha
            if val < best:
                best = val
                best_path = new_path
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best, best_path
```

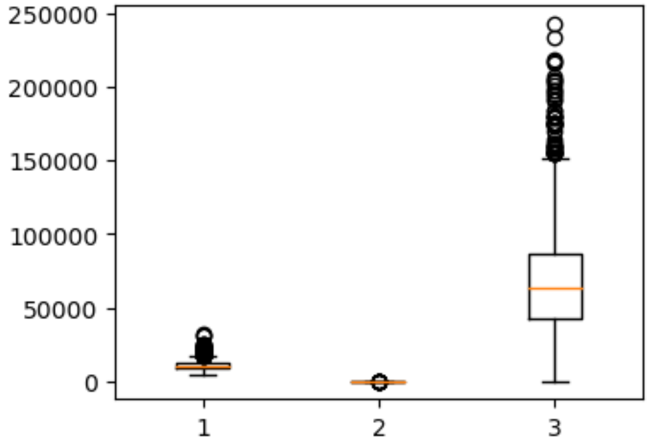
In [2]:
```
# Example tree with depth 3 and 8 terminal nodes
values = [3, 5, 2, 9, 12, 5, 23, 23]

# Start the Alpha-Beta Pruning algorithm
optimal_value, optimal_path = alphabeta(0, 0, True, values, float('-inf'), float('inf')
print("The optimal value is:", optimal_value)
print("The path taken is:", optimal_path)
```

```
The optimal value is: 12
The path taken is: [0, 1, 2, 4]
```

4b. Box Plot

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

In [5]:
```
data=pd.read_csv('./corolla.csv')
plt.figure(figsize = ( 4 , 3 ))
plt.boxplot([data["Price"],data["HP"],data["KM"]])
# plt.xticks([1,2,3],["Price","HP","KM"])
plt.show()
```



05a. Naive Bayes Classifier - Titanic Dataset

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
```

In [2]:
```
# Load the dataset
df = pd.read_csv("titanic.csv")
df = df[['Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]
```

In [3]:
```
# Handle missing values
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Fare'].fillna(df['Fare'].median(), inplace=True)
df = df.drop(["Embarked"], axis = 1)
df.head()
```

Out[3]:

	Survived	Pclass	Age	SibSp	Parch	Fare
0	0	3	22.0	1	0	7.2500
1	1	1	38.0	1	0	71.2833
2	1	3	26.0	0	0	7.9250
3	1	1	35.0	1	0	53.1000
4	0	3	35.0	0	0	8.0500

In [4]:
```
# Split the data into train and test sets
X = df.drop('Survived', axis=1)
y = df['Survived']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s
```

In [5]:
```
# Initialize and fit the Gaussian Naive Bayes classifier
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

Out[5]:
```
▾ GaussianNB
GaussianNB()
```

In [6]:
```
# Make predictions on the test set
y_pred = classifier.predict(X_test)
```

In [7]:
```
# Evaluate the model
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Confusion Matrix:
 [[88 17]
 [36 38]]
Accuracy: 0.7039106145251397
```

06. KNN with Glass Dataset

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

In [2]:
```
df=pd.read_csv("./glass.csv")
df.head()
```

Out[2]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	1
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1

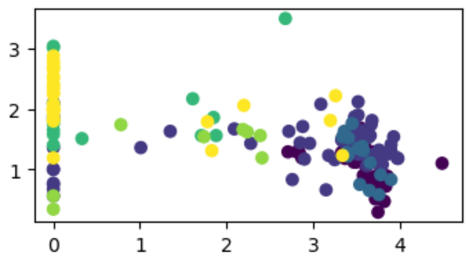
In [3]:
```
#Euclidean Distance

def ec(x1,x2):
    return np.sqrt(np.sum((x1-x2)**2))
```

In [4]:
```
from collections import Counter

class KNN:
    def __init__(self,k=3):
        self.k=k

    def fit(self,X,y):
        self.X_train=X
        self.y_train=y

    def predict(self,X):
        predictions=[self._predict(x) for x in X]
        return predictions

    def _predict(self,x):
        #Compute distance from one given point to all the points in X_train
        distances=[ec(x1=x,x2=x_train) for x_train in self.X_train]

        #Get k closest indices and labels
        k_indices=np.argsort(distances)[:self.k]
        k_labels=[self.y_train[i] for i in k_indices]

        #Get most common class label
        co=Counter(k_labels).most_common()
        return co[0][0]
```

In [5]:
```
#Split Data

X=df.drop("Type",axis=1).values
y=df['Type'].values
X_train,X_test,Y_train,Y_test=train_test_split(X,y,test_size=0.3,random_state=40)
```

In [6]:
```
#Fit Model

clf=KNN(k=3)
clf.fit(X_train,Y_train)
predictions=clf.predict(X_test)
print(predictions)
plt.figure(figsize = (4,2))
plt.scatter(X[:,2],X[:,3],c=y)
plt.show()
```

```
[2, 1, 6, 5, 5, 3, 2, 2, 7, 2, 1, 1, 2, 2, 2, 2, 1, 2, 7, 3, 1, 1, 1, 2, 5, 6, 1, 2, 1, 5, 1, 2, 2, 1, 1, 1,
6, 2, 1, 1, 2, 3, 2, 2, 6, 3, 2, 7, 1, 1, 3, 1, 2, 2, 1, 3, 7, 2, 1, 3, 1, 7, 1, 2, 2]
```



In [7]:
```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_pred=predictions,y_true=Y_test))
```

```
0.6307692307692307
```

08. Unsupervised K-means clustering on Iris dataset

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

In [2]:
```
df=pd.read_csv("./iris.csv")
df.head()
```

Out[2]:

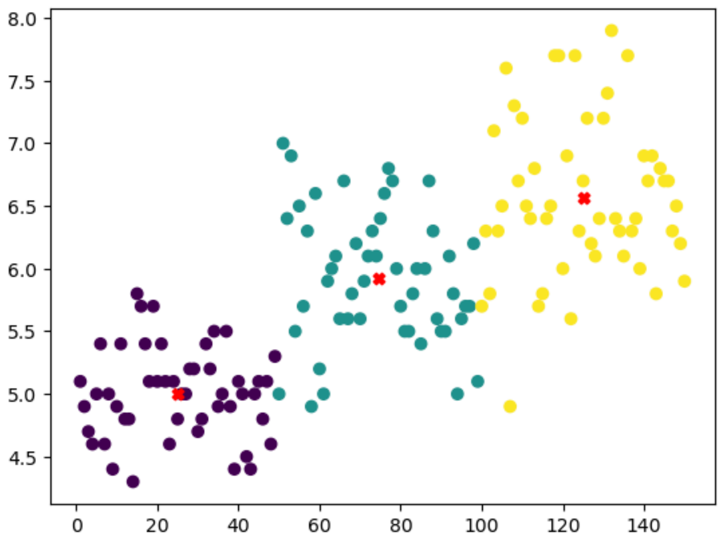
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

In [6]:
```
# K-means Function

def kmeans(X, K, max_iters):
    # Use the first K data points as the initial centroids
    centroids = X[:K]

    for _ in range(max_iters):

        expanded_x = X[:, np.newaxis]
        euc_dist = np.linalg.norm(expanded_x - centroids, axis=2)

        # Assign each data point to the nearest centroid
        labels = np.argmin(euc_dist, axis=1)

        # Update the centroids based on the assigned points
        new_centroids = np.array([X[labels == k].mean(axis=0) for k in range(K)])

        # If the centroids did not change, stop iterating
        if np.all(centroids == new_centroids):
            break

        centroids = new_centroids

    return labels, centroids
```

In [7]:
```
# Fit Model

X=np.array(df.iloc[:,:-1].values)
labels,c=kmeans(X,3,200)
print(labels)
print(c)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
[[ 25.          5.00612245   3.42040816   1.46530612   0.24489796]
 [ 74.5         5.922        2.78         4.206        1.304     ]
 [125.          6.57058824   2.97058824   5.52352941   2.01176471]]
```

In [8]:
```
#Plot Graph

plt.scatter(X[:,0],X[:,1],c=labels)
plt.scatter(c[:,0],c[:,1],marker="X",color="red")
plt.show()
```



Agglomerative Clustering using single linkage and complete Linkage

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import load_iris
```

In [2]:

```
# data = pd.read_csv("iris.csv")
iris = load_iris()
data = iris.data[:6]
data
```

Out[2]:

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4]])
```

In [3]:

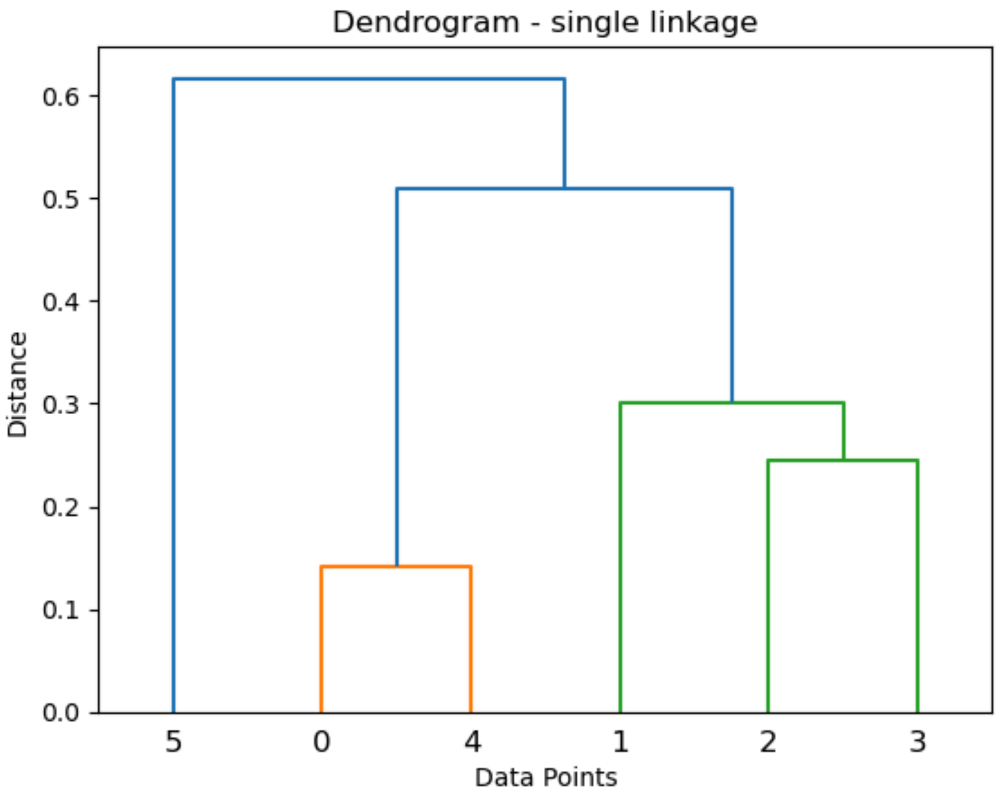
```
# Proximity Matrix

def proximity_matrix(data):
  n = data.shape[0]
  proximity_matrix = np.zeros((n, n))
  for i in range(n):
    for j in range(i+1, n):
        proximity_matrix[i, j] = np.linalg.norm(data[i] - data[j])
        proximity_matrix[j, i] = proximity_matrix[i, j]
  return proximity_matrix
```

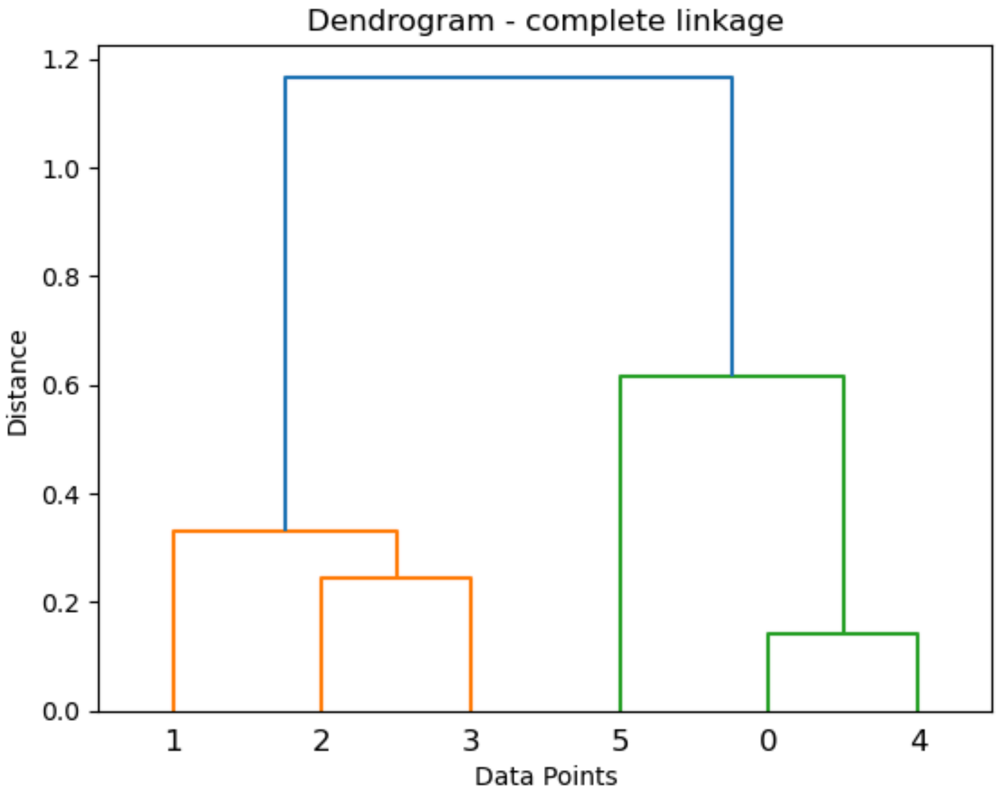
In [4]:

```
# Plot Dendogram

def plot_dendrogram(data, method):
  linkage_matrix = linkage(data, method=method)
  dendrogram(linkage_matrix)
  plt.title(f'Dendrogram - {method} linkage')
  plt.xlabel('Data Points')
  plt.ylabel('Distance')
  plt.show()
```

In [5]:
```
# Calculate the proximity matrix
print("Proximity matrix:")
print(proximity_matrix(data))

# Plot the dendrogram using single-linkage
plot_dendrogram(data, 'single')

# Plot the dendrogram using complete-linkage
plot_dendrogram(data, 'complete')
```

```
Proximity matrix:
[[0.         0.53851648 0.50990195 0.64807407 0.14142136 0.6164414 ]
 [0.53851648 0.         0.3        0.33166248 0.60827625 1.09087121]
 [0.50990195 0.3        0.         0.24494897 0.50990195 1.08627805]
 [0.64807407 0.33166248 0.24494897 0.         0.64807407 1.16619038]
 [0.14142136 0.60827625 0.50990195 0.64807407 0.         0.6164414 ]
 [0.6164414  1.09087121 1.08627805 1.16619038 0.6164414  0.        ]]
```



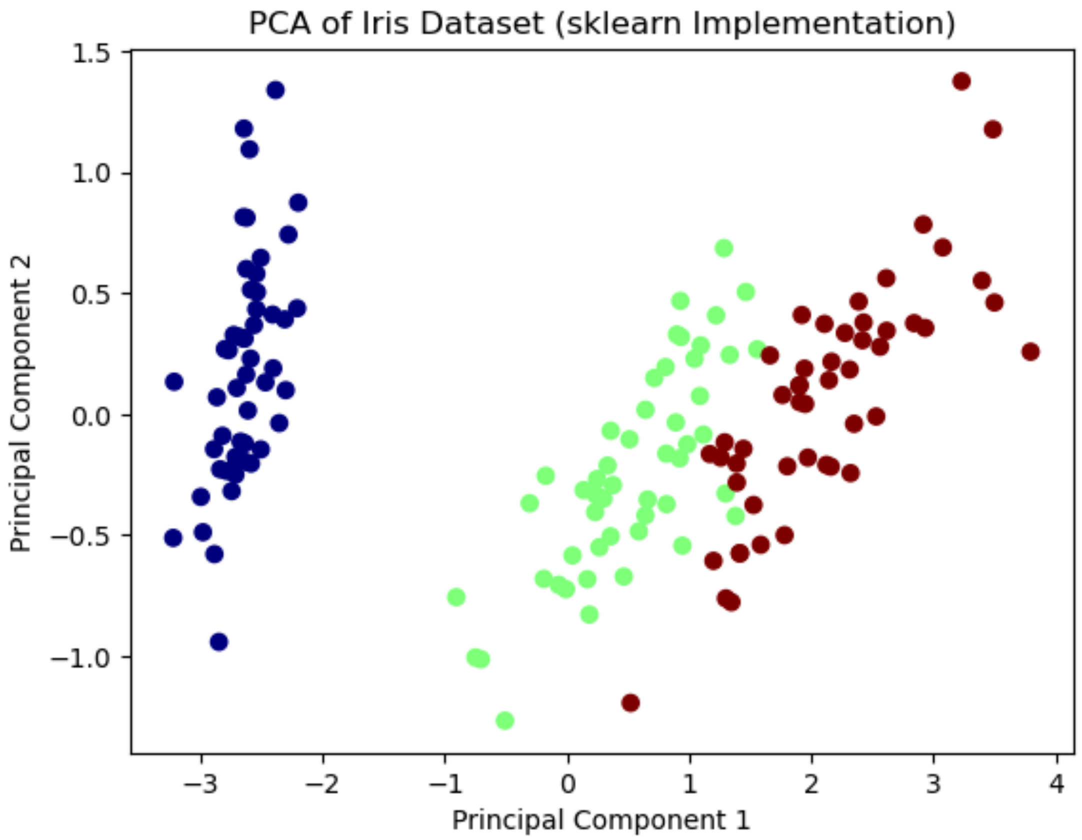


11a. PCA - Principal Component Analysis

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA as SklearnPCA

# Load the Iris dataset
X = load_iris().data
y = load_iris().target

# Perform PCA using sklearn
pca = SklearnPCA(n_components=2)
X_projected = pca.fit_transform(X)

print("Shape of Data:", X.shape)
print("Shape of transformed Data:", X_projected.shape)

# Plot the results
pc1 = X_projected[:, 0]
pc2 = X_projected[:, 1]

plt.scatter(pc1, pc2, c=y, cmap="jet")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA of Iris Dataset (sklearn Implementation)")
plt.show()
```

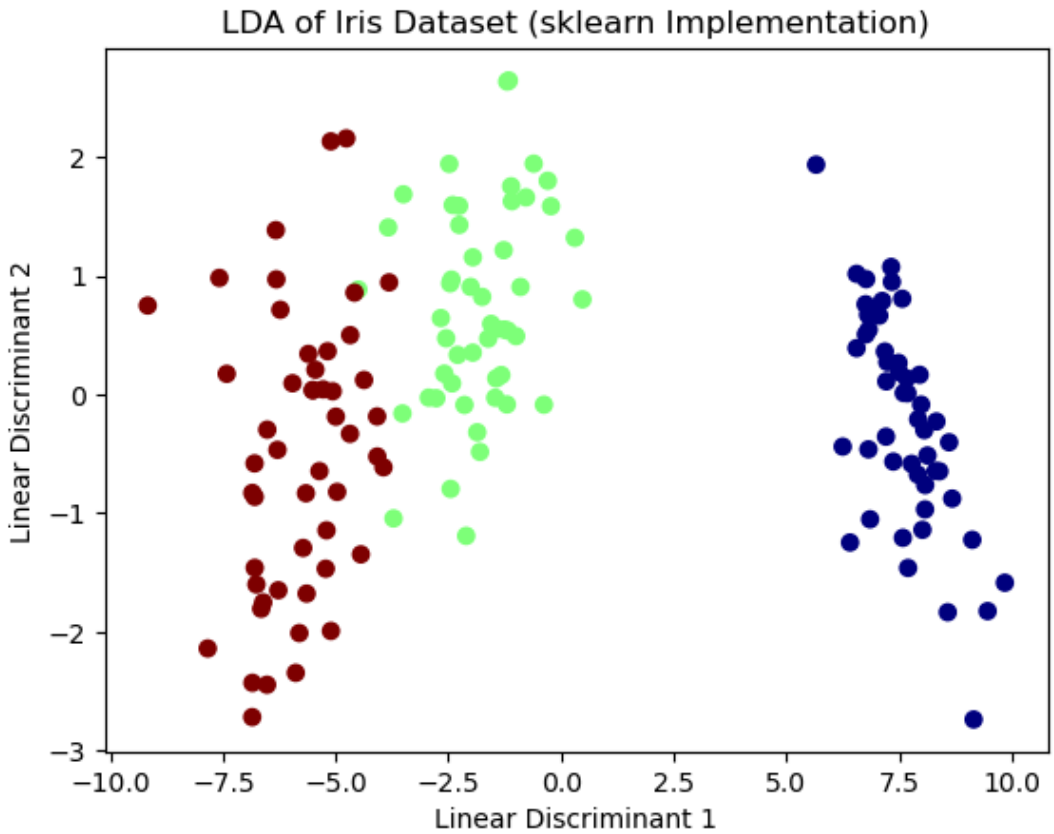
```
Shape of Data: (150, 4)
Shape of transformed Data: (150, 2)
```



11b. LDA - Linear Discriminant Analysis

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Load the Iris dataset
X = load_iris().data
y = load_iris().target

# Perform LDA using sklearn
lda = LinearDiscriminantAnalysis(n_components=2)
X_projected = lda.fit_transform(X, y)

print("Shape of Data:", X.shape)
print("Shape of transformed Data:", X_projected.shape)

# Plot the results
ld1 = X_projected[:, 0]
ld2 = X_projected[:, 1]

plt.scatter(ld1, ld2, c=y, cmap="jet")
plt.xlabel("Linear Discriminant 1")
plt.ylabel("Linear Discriminant 2")
plt.title("LDA of Iris Dataset (sklearn Implementation)")
plt.show()
```

```
Shape of Data: (150, 4)
Shape of transformed Data: (150, 2)
```



In [5]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

class PCA:
    def fit_transform(self, X, n_components=2):
        # Mean center the data
        mean = np.mean(X, axis=0)
        X_centered = X - mean

        # Calculate covariance matrix
        cov = np.cov(X_centered.T)

        # Calculate eigenvalues and eigenvectors
        eigenvalues, eigenvectors = np.linalg.eig(cov)

        # Sort the eigenvectors in decreasing order of eigenvalues
        idxs = np.argsort(eigenvalues)[::-1]
        eigenvectors = eigenvectors[:, idxs]

        # Select the top n_components eigenvectors
        components = eigenvectors[:, :n_components]

        # Transform the data
        X_projected = np.dot(X_centered, components)
        return X_projected

# Load dataset
X = load_iris().data
y = load_iris().target

# Perform PCA
pca = PCA()
X_projected = pca.fit_transform(X)

print("Shape of Data:", X.shape)
print("Shape of transformed Data:", X_projected.shape)

# Plot the results
plt.figure(figsize = (5,3))
plt.scatter(X_projected[:, 0], X_projected[:, 1], c=y, cmap="jet")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label='Class')
plt.show()
```

```
Shape of Data: (150, 4)
Shape of transformed Data: (150, 2)
```