1 Non-recursion

```
nterms = int(input("How many terms? "))
n1, n2 = 0, 1
count = 0
if nterms <= 0:
print("Please enter a positive integer")
elif nterms == 1:
print("Fibonacci sequence up to", nterms, ":")
print(n1)
else:
print("Fibonacci sequence:")
while count < nterms:
print(n1)
nth = n1 + n2
n1 = n2
n2 = nth
count += 1
```

5 N-Queens matrix

```
global N
N = 4
def printSolution(board):
for i in range(N):
for j in range(N):
if board[i][j] == 1:
print("Q",end=" ")
else:
print(".",end=" ")
print()
def isSafe(board, row, col):
for i in range(col):
if board[row][i] == 1:
return False
for i, j in zip(range(row, -1, -1),
range(col, -1, -1)):
if board[i][j] == 1:
return False
for i, j in zip(range(row, N, 1),
range(col, -1, -1)):
if board[i][j] == 1:
return False
return True
def solveNQUtil(board, col):
if col >= N:
return True
for i in range(N):
if isSafe(board, i, col):
board[i][col] = 1
if solveNQUtil(board, col + 1)== True:
return True
board[i][col] = 0
return False
def solveNQ():
board = [[0, 0, 0, 0],[0, 0, 0, 0],
[0, 0, 0, 0],[0, 0, 0, 0]]
if solveNQUtil(board, 0) == False:
print("Solution does not exist")
return False
printSolution(board)
return True
if __name__ == '__main__':
solveNQ()
```

1 Recursion

```
def recur_fibo(n):
if n <= 1:
return n
else:
return recur_fibo(n - 1) + recur_fibo(n - 2)
nterms = 7
if nterms <= 0:
print("Please enter a positive integer")
else:
print("Fibonacci sequence:")
for i in range(nterms):
print(recur_fibo(i))
```

4 0-1 Knapsack problem using dynamic

```
def knapSack(W, wt, val, n):
dp = [0 for i in range(W + 1)]
for i in range(1, n + 1):
for w in range(W, 0, -1):
if wt[i - 1] <= w:
dp[w] = max(dp[w], dp[w - wt[i - 1]] + val[i - 1])
return dp[W]
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

3 fractional Knapsack problem

```
class Item:
def __init__(self, value, weight):
self.value = value
self.weight = weight
def fractionalKnapsack(W, arr):
arr.sort(key=lambda x: x.value / x.weight, reverse=True)
final_value = 0.0
for item in arr:
if item.weight <= W:
W -= item.weight
final_value += item.value
else:
final_value += item.value * W / item.weight
break
return final_value
if __name__ == "__main__":
W = 50
arr = [Item(60, 10), Item(100, 20), Item(120, 30)]
max_val = fractionalKnapsack(W, arr)
print("Maximum value we can obtain =", int(max_val))
```

6 Multiply two matrix
#include <iostream>
#include <pthread.h>
#include <cstdlib>
using namespace std;
#define MAX 4
#define MAX_THREAD 4
int matA[MAX][MAX];
int matB[MAX][MAX];
int matC[MAX][MAX];
int step_i = 0;
void multi(void* arg){
int i = step_i++;
for (int j = 0; j < MAX; j++) {
for (int k = 0; k < MAX; k++) {
matC[i][j] += matA[i][k] * matB[k][j];}}}
int main() {
for (int i = 0; i < MAX; i++) {
for (int j = 0; j < MAX; j++) {
matA[i][j] = rand() % 10;
matB[i][j] = rand() % 10; } }
cout << "Matrix A" << endl;
for (int i = 0; i < MAX; i++) {
for (int j = 0; j < MAX; j++) {
cout << matA[i][j] << " ";}
cout << endl;}
cout << "Matrix B" << endl;
for (int i = 0; i < MAX; i++) {
for (int j = 0; j < MAX; j++) {
cout << matB[i][j] << " ";}
cout << endl;}
pthread_t threads[MAX_THREAD];
for (int i = 0; i < MAX_THREAD; i++) {
int* p = nullptr;
pthread_create(&threads[i], nullptr,
(void*(*)(void*))multi, (void*)p);}
for (int i = 0; i < MAX_THREAD; i++) {
pthread_join(threads[i], nullptr);}
cout << "Multiplication of A and B" << endl;
for (int i = 0; i < MAX; i++) {
for (int j = 0; j < MAX; j++) {
cout << matC[i][j] << " ";}
cout << endl;}
return 0;}
	2 Huffman Encoding
	import heapq
	from collections import defaultdict
	class Node:
	def __init__(self, left=None, right=None, value=None, frequency=None):
	self.left = left
	self.right = right
	self.value = value
	self.frequency = frequency
	def children(self):
	return (self.left, self.right)
	class Huffman_Encoding:
	def __init__(self, string):
	self.string = string
	self.encoding = {}
	def build_tree(self):
	freq = defaultdict(int)
	for char in self.string:
	freq[char] += 1
	heap = [(f, Node(value=c)) for c, f in freq.items()]
	heapq.heapify(heap)
	while len(heap) > 1:
	freq1, node1 = heapq.heappop(heap)
	freq2, node2 = heapq.heappop(heap)
	merged_node = Node(left=node1, right=node2)
	heapq.heappush(heap, (freq1 + freq2, merged_node))
	return heap[0][1]
	def huffman_encoding(self, node, binary_str=""):
	if node.value is not None:
	self.encoding[node.value] = binary_str
	if node.left:
	self.huffman_encoding(node.left, binary_str + "0")
	if node.right:
	self.huffman_encoding(node.right, binary_str + "1")
	def encode(self):
	root = self.build_tree()
	self.huffman_encoding(root)
	for char, binary in self.encoding.items():
	print(f"Char: {char} | Huffman code: {binary}")
	# Input string AAAAAAABBCCCCCCDDDEEEEEEEEE
	string = input("Enter string to be encoded: ")
	encode = Huffman_Encoding(string)
	encode.encode()

5 SALES * import pandas as pd import numpy as np import matplotlib.pyplot as plt import seaborn as sns *df = pd.read_csv('sales_data_sample.csv', encoding='unicode_escape') df.head() *df.info() *df_drop = ['ADDRESSLINE1', 'ADDRESSLINE2', 'POSTALCODE', 'CITY', 'TERRITORY', 'PHONE', 'STATE', 'CONTACTFIRSTNAME', 'CONTACTLASTNAME', 'CUSTOMERNAME', 'ORDERNUMBER'] df = df.drop(df_drop, axis=1) *from sklearn.preprocessing import LabelEncoder def convert_categories(col): le = LabelEncoder()	df[col] = le.fit_transform(df[col].values) *from sklearn.cluster import KMeans wcss = [] for k in range(1,15): kmeans = KMeans(n_clusters=k,init='k-means++', random_state=15) kmeans.fit(data) wcss.append(kmeans.inertia_) *k = list(range(1,15)) plt.plot(k,wcss) plt.xlabel('Clusters') plt.ylabel('scores') plt.title('Finding right number of clusters') plt.grid() plt.show()

1 UBER *import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
*df = pd.read_csv("uber.csv")
df.head()
*df.drop(columns=['Unnamed: 0','key'],inplace=True)
* df.dropna(how='any',inplace=True)
* df.isnull().sum()*
* for col in df.select_dtypes(exclude=['object']):plt.figure()
sns.boxplot(data=df,x=col)
* temp = distance(df['pickup_latitude'],df['pickup_
longitude'],df['dropoff_latitude'],df['dropoff_longitude'])
temp.head()
* df_new = df.copy()
df_new['Distance'] = temp
df = df_new
* sns.boxplot(data=df,x='Distance')
* df = df[(df['Distance'] < 200) & (df['Distance'] > 0)]
* df.corr()
* sns.scatterplot(y=df['fare_amount'],x=df['Distance'])
* from sklearn.preprocessing import StandardScaler
x_train = std_x.fit_transform(x_train)
* from sklearn.linear_model import LinearRegression
* fit_predict(LinearRegression())
* from sklearn.ensemble import RandomForestRegressor
fit_predict(RandomForestRegressor())

2 email * import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
* df = pd.read_csv(emails.csv')
df.head()
* df.isnull().sum()
* df.dropna(how='any',inplace=True)
* x = df.iloc[:,1:-1].values
y = df.iloc[:,-1].values
* from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x,
y,test_size=0.25,random_state=10)
* from sklearn.metrics import
ConfusionMatrixDisplay,confusion_matrix,
accuracy_score,precision_score,recall_score,
plot_precision_recall_curve,plot_roc_curve
def report(classifier):
y_pred = classifier.predict(x_test)
cm = confusion_matrix(y_test,y_pred)
display = CMatDisp(cm,display_labels=classifier.classes_)
display.plot()
plot_precision_recall_curve(classifier,x_test,y_test)
plot_roc_curve(classifier,x_test,y_test)
* from sklearn.neighbors import KNeighborsClassifier
* kNN = KNeighborsClassifier(n_neighbors=10)
kNN.fit(x_train,y_train)
* report(kNN)

3 Bank * import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
* df = pd.read_csv('Churn_Modelling.csv')
df.head()
* plt.xlabel('Exited')
plt.ylabel('Count')
df['Exited'].value_counts().plot.bar()
plt.show()
* df['Geography'].value_counts()
* df = pd.concat([df,pd.get_dummies(df['
Geography'],prefix='Geo')],axis=1)
* from sklearn.model_selection import train_test_split
* import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
* model.fit(x_train,y_train,batch_size=64,
validation_split=0.1,epochs=100)
* accuracy_score(y_test,y_pred)
* cm = confusion_matrix(y_test,y_pred)
display = ConfusionMatrixDisplay(cm)
display.plot()

4 KNN diabetes* import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
* df=pd.read_csv("diabetes.csv")
* df.shape
* df.describe()
* df["Glucose"]
* sc_X=StandardScaler()
X_train=sc_X.fit_transform(X_train)
X_test=sc_X.transform(X_test)
* knn=KNeighborsClassifier(n_neighbors=11)
* y_pred=knn.predict(X_test)
* cf_matrix=confusion_matrix(y_test,y_pred)
* ax = sns.heatmap(cf_matrix, annot=True, cmap='Blues')
* accuracy_score(y_test,y_pred)
* precision_score(y_test,y_pred)
* error_rate=1-accuracy_score(y_test,y_pred)

6 titanic *import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
titanic_data=pd.read_csv('titanic_data.csv')
*titanic_data.describe()
*titanic_data.info()
*font = {'weight' : 'bold','size'   : 22}
*plt.figure(figsize=(12,9))
*plt.xlabel('Survived Or Not')

plt.title("Survival Percentage", fontdict=font)
*titanic_data.isnull().any()
*plt.figure(figsize=(12,9))
sns.countplot(x='Survived',data=titanic_data)
label=['Not Survived','Survived']
plt.xticks(titanic_data['Survived'].unique(), label, size=13)
plt.show()
*titanic_data.drop(['PassengerId','Pclass','Name','Ticket',
'Embarked','Sex'],axis=1,inplace=True)
*final_prediction = model.predict(new_data)
*final_prediction