```python
import time
# Non-Recursive
def fibonacci_non_recursive(n):
    a=0
    b=1
    if n<0:
        print("Incorrect input")
    elif n==0:
        return 0
    elif n==1:
        return 1
    else:
        for i in range(2,n+1):
            c=a+b
            a=b
            b=c
        return c

n=int(input("Enter value :"))

start_time=time.time()

print("Non Recursive :",fibonacci_non_recursive(n))
end_time=time.time()
print(f"Time required by non recursive is {end_time-start_time}")

# Recursive
def recursive(n):
    if n<0:
        print("Invalid input")
    elif n==0:
        return 0
    elif n==1:
        return 1
    else:
        return recursive(n-1)+recursive(n-2)

start_time=time.time()
print("Recursive :",recursive(n))
end_time=time.time()
print(f"Time required by recursive is {end_time-start_time}")


# For the non-recursive version:
```

```python
# Time Complexity: O(n) since we need to iterate from 2 to n to calculate the Fibonacci number at position n.
# Space Complexity: O(n) since we are storing all the Fibonacci numbers up to the nth position in the list.

# For the recursive version:

# Time Complexity: O(2^n) since at each level of the recursion tree, the number of function calls doubles.
# Space Complexity: O(n) considering the space taken up by the function call stack.

# In terms of efficiency, the non-recursive version is much more efficient than the recursive one for large values of n. This is because the recursive version recalculates the same values multiple times, leading to exponential time complexity.


# Function to solve the fractional knapsack problem
def fractional_knapsack(items, capacity):
    # n = int(input("Enter the number of items: "))
    # items = []
    # for i in range(n):
    #     print(f"For item {i + 1}:")
    #     weight = float(input("Enter the weight: "))
    #     value = float(input("Enter the value: "))
    #     items.append((weight, value))

    # capacity = float(input("Enter the capacity of the knapsack: "))
    # Sort items based on the value-to-weight ratio in descending order
    items.sort(key=lambda x: x[1] / x[0], reverse=True)

    total_value = 0.0
    knapsack = []

    for item in items:
        weight, value = item
        if capacity >= weight:
            # Take the whole item if it fits in the knapsack
            total_value += value
            knapsack.append((weight, value))
            capacity -= weight
        else:
            # Take a fraction of the item to fill the remaining capacity
            fraction = capacity / weight
            total_value += fraction * value
            knapsack.append((fraction*weight, fraction * value))
            break

    return total_value, knapsack
# Example usage:
        #(weight,value)
items = [(10, 60), (20, 100), (30, 120)]
capacity = 50
max_value, selected_items = fractional_knapsack(items, capacity)
print("Maximum value:", max_value)
print("Selected items:", selected_items)





def knapsack_dp(capacity, weights, values, n):
    dp = [0 for _ in range(capacity + 1)]

    for i in range(1, n + 1):
        for w in range(capacity, 0, -1):
            if weights[i - 1] <= w:
                dp[w] = max(dp[w], dp[w - weights[i - 1]] + values[i - 1])

    return dp[capacity]


# Driver code
if __name__ == '__main__':
    values = [60, 100, 120]
    weights = [10, 20, 30]
    knapsack_capacity = 50
    number_of_items = len(values)
    print(knapsack_dp(knapsack_capacity, weights, values, number_of_items))
```

```python
def is_safe(board, row, col, n):
    # Check the column for conflicts
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check the upper-left diagonal for conflicts
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check the upper-right diagonal for conflicts
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j] == 1:
            return False
    return True

# we are only checking upper diagonal because we will be placing queens from top
to bottom

def solve_n_queens(board, row, n):
    if row == n:
        return True  # All queens are placed successfully

    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1  # Place the queen
            if solve_n_queens(board, row + 1, n):
                return True  # If placing queen in the current position leads to
a solution, return True
            board[row][col] = 0  # If not, backtrack and try the next column

    return False  # If no column is suitable, return False

def print_solution(board):
    for row in board:
        print(" ".join(["Q" if cell == 1 else "." for cell in row]))

def n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    # Place the first queen (you can choose any square)
    board[0][0] = 1

    if solve_n_queens(board, 1, n):
        print_solution(board)
    else:
        print("No solution exists")

n = int(input("Enter number of queens :"))  # Change this value to the desired
board size
n_queens(n)

import random
import time

# Deterministic Quicksort
def deterministic_quicksort(arr):
    if len(arr) <= 1:         # Important Base condition
        return arr
    pivot = arr[len(arr) // 2] #square bracket
    #list comprehension for partitioning
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return deterministic_quicksort(left) + middle +
deterministic_quicksort(right)

# Randomized Quicksort
def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = random.choice(arr)
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return randomized_quicksort(left) + middle + randomized_quicksort(right)

# Analysis
def analyze_sorting_algorithm(sort_function, array):
    start_time = time.time()
    sorted_array = sort_function(array)
    end_time = time.time()
    return sorted_array, end_time - start_time

# Test the algorithms
```

```python
test_array = [3, 2, 1, 5, 4, 8, 7, 6]
sorted_array_det, time_taken_det =
analyze_sorting_algorithm(deterministic_quicksort, test_array)
sorted_array_rand, time_taken_rand =
analyze_sorting_algorithm(randomized_quicksort, test_array)

# Print results
print("Deterministic Quicksort:")
print("Sorted Array:", sorted_array_det)
print(f"Time taken: {time_taken_det:.6f} seconds")
print("\nRandomized Quicksort:")
print("Sorted Array:", sorted_array_rand)
print(f"Time taken: {time_taken_rand:.6f} seconds")

def generate_random_array(start, end, size):
  array = []
  for i in range(size):
    array.append(random.randint(start, end))
  return array

# In the analyze_sorting_algorithm function, the sorting function is passed as an
argument. This function can be any sorting algorithm, including the
deterministic_quicksort function. When you call sort_function(array), it is
replaced by the actual sorting function you pass to it. Therefore, if you pass
deterministic_quicksort as the sort_function argument, it will be executed during
the sort_function(array) call.

# Here's how the execution flows:

# start_time = time.time() captures the current time before the sorting operation
begins.

# sorted_array = sort_function(array) calls the sorting function that was passed
as an argument. If deterministic_quicksort was passed as sort_function, it will
execute the code inside the deterministic_quicksort function, which sorts the
array using the deterministic Quicksort algorithm.

# end_time = time.time() records the time after the sorting is completed.

# end_time - start_time calculates the time taken for the sorting process.

# Finally, the function returns both the sorted array and the time taken for the
sorting operation as a tuple.
```

```
# So, the deterministic_quicksort function gets executed inside the
analyze_sorting_algorithm function when you pass it as an argument to
sort_function. The timing is done around the execution of the sorting function to
measure the time taken by that specific sorting algorithm.

# # Deterministic (Classic) Quick Sort:
# def partition(arr, low, high):
#     pivot = arr[low]
#     left = low + 1
#     right = high
#     done = False

#     while not done:
#         while left <= right and arr[left] <= pivot:
#             left = left + 1
#         while arr[right] >= pivot and right >= left:
#             right = right -1
#         if right < left:
#             done= True
#         else:
#             arr[left], arr[right] = arr[right], arr[left]

#     arr[low], arr[right] = arr[right], arr[low]
#     return right

# def deterministic_quick_sort(arr, low, high):
#     if low < high:
#         pivot_index = partition(arr, low, high)
#         deterministic_quick_sort(arr, low, pivot_index - 1)
#         deterministic_quick_sort(arr, pivot_index + 1, high)

# # Example usage:
# arr = [3, 6, 8, 10, 1, 2, 1]
# deterministic_quick_sort(arr, 0, len(arr) - 1)
# print("Deterministic Sorted array:", arr)


# # Randomized Quick Sort:

# import random

# def randomized_partition(arr, low, high):
#     random_index = random.randint(low, high)
#     arr[random_index], arr[low] = arr[low], arr[random_index]
#     return partition(arr, low, high)
# def randomized_quick_sort(arr, low, high):
#     if low < high:
#         pivot_index = randomized_partition(arr, low, high)
#         randomized_quick_sort(arr, low, pivot_index - 1)
#         randomized_quick_sort(arr, pivot_index + 1, high)

# # Example usage:
# arr = [3, 6, 8, 10, 1, 2, 1]
# randomized_quick_sort(arr, 0, len(arr) - 1)
# print("Randomized Sorted array:", arr)
```

Blockchain :

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.7;

contract Student{
    struct details{
        uint id;
        string name;
    }
    details[] stud;
    uint nextId=0;

    function create(string memory name)public {
        stud.push(details(nextId++,name));
    }
    function read(uint id)public view returns(uint,string memory ){
        for(uint i=0;i<stud.length;i++){
            if(stud[i].id==id){
                return(stud[i].id,stud[i].name);
            }
        }
        return (0, "");
    }
    function update(uint id,string memory name) public {
        for(uint i=0;i<stud.length;i++){
            if(stud[i].id==id){
                stud[i].name=name;
            }
        }
    }
    function del(uint id) public {
        delete stud[id];
    }
}
```

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.7.0;
contract Bank{
    struct client_details{
        uint client_id;
        address client_address;
        uint client_balance;
    }
    client_details[] clients;
    address payable manager;
    uint counter =0;
    modifier onlyManager{
        require(msg.sender==manager,"Only for manager");
        _;
    }
    modifier onlyClient{
        bool isClient=false;
        for(uint i=0;i<clients.length;i++){
            if(clients[i].client_address==msg.sender){
                isClient=true;
            }
        }
        require(isClient,"Only for clients");
        _;
    }
    receive() external payable { }
    function setManager(address payable manaddr)public {
        manager=manaddr;
    }
    function joinClient()public payable{
        clients.push(client_details(counter,msg.sender,address(msg.sender).balanc
e));
    }
    function deposit() public payable {
        payable(address(this)).transfer(msg.value);
    }
    function withdraw(uint amount)public payable {
        payable(address(msg.sender)).transfer(amount * 1 ether);
    }
    function getBalanceContract() public view returns(uint){
        return address(this).balance;
    }
}
```

ML :

Practical 1:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns


df= pd.read_csv("uber.csv")

df.head()


df=df.drop(['Unnamed: 0', 'key'],axis=1)


df.info()


df.pickup_datetime=pd.to_datetime(df.pickup_datetime)


df =df.assign(hour = df.pickup_datetime.dt.hour,

             day = df.pickup_datetime.dt.day)


df.head()

df=df.drop(['pickup_datetime'],axis=1)


df.head()


from math import *

def distance_transform(longitude1,latitude1,longitude2,latitude2):

    travel_dist=[]

    for pos in range(len(longitude1)):

        long1,lati1,long2,lati2=map(radians,[longitude1[pos],latitude1[pos],longitude2[pos],latitude2[pos]])

        dist_long=long2-long1

        dist_lati=lati2-lati1
        a = sin(dist_lati/2)**2 + cos(lati1)*cos(lati2)*sin(dist_long/2)**2

        c = 2* asin(sqrt(a)) * 6371

        travel_dist.append(c)

    return travel_dist


df['dist_travel'] = distance_transform(df['pickup_longitude'].to_numpy(),

                                df['pickup_latitude'].to_numpy(),

                                df['dropoff_longitude'].to_numpy(),

                                df['dropoff_latitude'].to_numpy())


df.head()


df.plot(kind='box',subplots=True,layout=(7,2),figsize=(15,20))


def remove(df1,col):

    Q1=df1[col].quantile(0.25)

    Q3=df1[col].quantile(0.75)

    IQR = Q3 -Q1

    lower = Q1 - 1.5*IQR

    upper = Q3 + 1.5*IQR

    df[col]=np.clip(df1[col],lower,upper)

    return df

def treat_outlier(df,col_list):

    for c in col_list:

        df=remove(df,c)

    return df


df = treat_outlier(df , df.iloc[:,0::])


df.columns
df.dropna(inplace=True)

x=df.drop(['fare_amount'],axis=1)

y=df['fare_amount']


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=40)


from sklearn.linear_model import LinearRegression

regression =LinearRegression()

regression.fit(X_train,Y_train)


regression.coef_


pred=regression.predict(X_test)



pred


from sklearn.ensemble import RandomForestRegressor

random=RandomForestRegressor(n_estimators=100)


random.fit(X_train,Y_train)


random.predict(X_test)
```

Practical 2:

```
import pandas as pd

import numpy as np
```

```
df =pd.read_csv('emails.csv')

df.head()

df.dropna(inplace=True)

df=df.drop(['Email No.'],axis=1)

df.isnull().sum()

x=df.drop(['Prediction'],axis=1)
y=df['Prediction']

from sklearn.preprocessing import scale
x= scale(x)

from sklearn.model_selection import train_test_split
X_train,X_test,Y_train,Y_test=train_test_split(x,y,test_size=0.3)

from sklearn.neighbors import KNeighborsClassifier
knn=KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train,Y_train)

pred=knn.predict(X_test)



from sklearn.metrics import accuracy_score
print("Accuracy of KNN is:",accuracy_score(Y_test,pred))

from sklearn.svm import SVC
```

```
model=SVC(C=1)
model.fit(X_train,Y_train)

pred=model.predict(X_test)

print("Accuracy of SVM is:",accuracy_score(Y_test,pred))
```

Practical 3 :

```
import pandas as pd
import numpy as np
import seaborn as sns

df=pd.read_csv('Churn_Modelling.csv')

df.head()

df.columns

df=df.drop(['RowNumber', 'CustomerId', 'Surname'],axis=1)

df.info()

state=pd.get_dummies(df['Geography'],drop_first=True)
gender=pd.get_dummies(df['Gender'],drop_first=True)

df=pd.concat([df,state,gender],axis=1)

df=df.drop(['Geography','Gender'],axis=1)

x=df.drop(['Exited'],axis=1)
y=df['Exited']
```

```
from sklearn.model_selection import train_test_split

X_train,X_test,Y_train,Y_test=train_test_split(x,y,test_size=0.2)


from sklearn.preprocessing import StandardScaler

sc=StandardScaler()

X_train = sc.fit_transform(X_train)


X_test=sc.transform(X_test)


from keras.models import Sequential

from keras.layers import Dense


classifier=Sequential()


classifier.add(Dense(activation='relu',input_dim=11,units=6,kernel_initializer='uniform'))


classifier.add(Dense(activation='relu',units=6,kernel_initializer='uniform'))


classifier.add(Dense(activation='sigmoid',units=1,kernel_initializer='uniform'))


classifier.summary()


classifier.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])


classifier.fit(X_train,Y_train,batch_size=10,epochs=50)


pred=classifier.predict(X_test)


pred=(pred>0.5)
```

```
from sklearn.metrics import confusion_matrix

cm =confusion_matrix(Y_test,pred)


sns.heatmap(cm,annot=True)
```

Practical 4 :

```
def f(x):
    return (x+3)**2
def df(x):
    return 2*x+6
def gradient_descent(initial_x,learning_rate,n):
    x=initial_x
    x_history=[x]
    for i in range(n):
        gradient=df(x)
        x = x- learning_rate*gradient
        x_history.append(x)
    return x,x_history


x,x_history=gradient_descent(2,0.1,50)
print("Local min is {:.2f}".format(x))
import numpy as np
x_vals=np.linspace(-1,5,100)


import matplotlib.pyplot as plt
plt.plot(x_vals,f(x_vals))


plt.plot(x_history,f(np.array(x_history)),'rx')
```

Practical 5 :

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt





df =pd.read_csv("sales_data_sample.csv")


df.columns


df=df.drop(['ORDERNUMBER',  'ORDERDATE', 'STATUS',
       'MSRP', 'CUSTOMERNAME', 'PHONE',
       'ADDRESSLINE1', 'ADDRESSLINE2', 'CITY', 'STATE', 'POSTALCODE',
       'COUNTRY', 'TERRITORY', 'CONTACTLASTNAME', 'CONTACTFIRSTNAME'
       ],axis=1)

line=pd.get_dummies(df['PRODUCTLINE'],drop_first=True)
deal=pd.get_dummies(df['DEALSIZE'],drop_first=True)
df=pd.concat([df,line,deal],axis=1)

df.info()

df=df.drop(['PRODUCTLINE','DEALSIZE'],axis=1)
df['PRODUCTCODE']=pd.Categorical(df['PRODUCTCODE']).codes

df.info()
```

```
from sklearn.cluster import KMeans
wcss=[]
K=range(1,10)
for k in K:
    kmeans=KMeans(n_clusters=k)
    kmeans.fit(df)
    wcss.append(kmeans.inertia_)



plt.plot(K,wcss,'-bx')


X_train=df.values
model=KMeans(n_clusters=3)
model=model.fit(X_train)
predictions=model.predict(X_train)
unique,counts=np.unique(predictions,return_counts=True)
counts


from sklearn.decomposition import PCA
pca=PCA(n_components=2)
reduced_X=pd.DataFrame(pca.fit_transform(X_train),columns=['PCA1','PCA2'])


plt.figure(figsize=(14,10))
plt.scatter(reduced_X['PCA1'],reduced_X['PCA2'])


reduced_centers=pca.transform(model.cluster_centers_)
reduced_centers


reduced_X['Cluster']=predictions
```

```
plt.scatter(reduced_X[reduced_X['Cluster']==0].loc[:,'PCA1'],reduced_X[reduced_X['Cluster']==0].loc[:,'P
CA2'],color='slateblue',s=5)

plt.scatter(reduced_X[reduced_X['Cluster']==1].loc[:,'PCA1'],reduced_X[reduced_X['Cluster']==1].loc[:,'P
CA2'],color='red',s=5)

plt.scatter(reduced_X[reduced_X['Cluster']==2].loc[:,'PCA1'],reduced_X[reduced_X['Cluster']==2].loc[:,'P
CA2'],color='yellow',s=5)

plt.scatter(reduced_centers[:,0],reduced_centers[:,1],marker='x',color='black',s=400)
```