#p1\_fib\_recursive

steps = 0

def fibonacci(n):

global steps

steps += 1 # Increment steps count

if n <= 1:

return n

return fibonacci(n - 1) + fibonacci(n - 2)

n = int(input("Enter the number of Fibonacci terms: "))

print("Fibonacci Series:", end=" ")

for i in range(n):

fibonacci(i) # Call Fibonacci function to calculate the value and track steps

print(fibonacci(i), end=" ")

print() # Move to the next line after printing the series

# Reset steps count before getting the Fibonacci value at a specific position

steps = 0

pos = int(input("Enter position (0 to " + str(n - 1) + "): "))

result = fibonacci(pos)

print("Fibonacci(" + str(pos) + ") = " + str(result))

print("Steps taken to calculate Fibonacci(" + str(pos) + "): " + str(steps))

#p1\_fib\_iterative

def fibonacci(n):

if n == 0:

return 0

if n == 1:

return 1

a, b = 0, 1

for i in range(2, n + 1):

a, b = b, a + b

return b

n = int(input("Enter the number of Fibonacci terms: "))

fibonacci\_series = [0] \* n # Initialize the list with zeros

for i in range(n): #Fibonacci series

fibonacci\_series[i] = fibonacci(i)

print("Fibonacci Series:", fibonacci\_series)

pos = int(input(f"Enter position (0 to {n - 1}): "))

print(f"Fibonacci({pos}) = {fibonacci(pos)} (Steps: {n - 1})")

#p2\_job\_sequencing

def schedule\_jobs(jobs, n):

# Sorting jobs based on profit using a simple bubble sort method

for i in range(n):

for j in range(i + 1, n):

if jobs[i][2] < jobs[j][2]:

# Swap jobs[i] and jobs[j] if jobs[i] has less profit

temp = jobs[i]

jobs[i] = jobs[j]

jobs[j] = temp

scheduled\_jobs = [0] \* n # Initialize an empty schedule with 0 (unoccupied slots)

total\_profit = 0

# Try to schedule each job

for i in range(n):

job\_id, deadline, profit = jobs[i]

if deadline > n:

deadline = n # Make sure the deadline doesn't exceed total jobs

# Find the first available slot for the job, from the latest possible slot

for j in range(deadline - 1, -1, -1):

if scheduled\_jobs[j] == 0: # If the slot is empty

scheduled\_jobs[j] = job\_id # Assign the job to this slot

total\_profit += profit # Add the job's profit

break

# Print scheduled jobs

print("Scheduled jobs:", end=" ")

for i in range(n):

if scheduled\_jobs[i] != 0:

print(scheduled\_jobs[i], end=" ")

print()

print("Total Profit:", total\_profit)

# Driver code

n = int(input("Enter number of jobs: "))

jobs = [] # Initialize an empty list of jobs

# Input job details manually (without using split)

for i in range(n):

job\_input = input("Enter job ID, deadline, profit (separated by spaces): ")

# Manually extract job details from the input

job\_id = 0

deadline = 0

profit = 0

space\_count = 0

num = ""

# Parse the input string character by character

for char in job\_input:

if char == " ":

space\_count += 1

if space\_count == 1:

job\_id = int(num)

elif space\_count == 2:

deadline = int(num)

num = "" # Reset num after each space

else:

num += char # Add digit to num

# The last part will be profit after the second space

profit = int(num)

jobs.append([job\_id, deadline, profit]) # Add job to the jobs list

# Call the function to schedule the jobs

schedule\_jobs(jobs, n)

p3\_fractional\_knapsack

# Input number of items

n = int(input("Enter number of items: "))

# Initialize an empty list to store items

items = []

# Input profit and weight for each item

for i in range(n):

# Read profit and weight

data = input(f"Enter profit and weight for item {i + 1}: ")

# Manually split the string to extract profit and weight

space\_index = 0

# Find the space separating profit and weight

for index in range(len(data)):

if data[index] == ' ':

space\_index = index

break

# Extract profit (before the space)

profit = 0

for index in range(space\_index):

profit = profit \* 10 + (int(data[index]) - int('0')) # Convert char to integer

# Extract weight (after the space)

weight = 0

for index in range(space\_index + 1, len(data)):

weight = weight \* 10 + (int(data[index]) - int('0')) # Convert char to integer

# Calculate profit/weight ratio

ratio = profit / weight

# Manually add the item (profit, weight, ratio) to the list

item = [profit, weight, ratio]

items = items + [item] # Concatenate instead of using append

# Input capacity of the knapsack

capacity = int(input("Enter capacity: "))

# Simple Bubble Sort for sorting items by profit/weight ratio in descending order

for i in range(n):

for j in range(i + 1, n):

if items[i][2] < items[j][2]:

# Swap items[i] and items[j] if profit/weight ratio of items[i] < items[j]

temp = items[i]

items[i] = items[j]

items[j] = temp

# Initialize total profit

total\_profit = 0

# Iterate over the sorted items

for i in range(n):

if capacity == 0: # If the knapsack is full

break

profit = items[i][0]

weight = items[i][1]

# Take the minimum of the remaining weight of the knapsack and the item weight

weight\_taken = weight if weight <= capacity else capacity

fraction = weight\_taken / weight

profit\_gained = profit \* fraction

# Print item details without string formatting

print("Item (profit: " + str(profit) + ", weight: " + str(weight) + ") - Taken: " + str(fraction \* 100) + "%, Profit gained: " + str(profit\_gained))

# Add the profit gained and reduce the capacity

total\_profit += profit\_gained

capacity -= weight\_taken

# Print total profit without string formatting

print("Maximum value in Knapsack = " + str(total\_profit))

#p4\_0/1\_knapsack

def knapsack(profits, weights, capacity):

n = len(profits)

# Initialize the DP table

dp = [[0] \* (capacity + 1) for \_ in range(n + 1)]

# Fill the DP table

for i in range(1, n + 1):

for w in range(1, capacity + 1):

if weights[i - 1] <= w:

dp[i][w] = dp[i - 1][w]

dp[i][w] = max(dp[i][w], profits[i - 1] + dp[i - 1][w - weights[i - 1]])

else:

dp[i][w] = dp[i - 1][w]

print("Max Profit:", dp[n][capacity])

print("Items:", end=" ")

# Find the items to include in the knapsack

w = capacity

for i in range(n, 0, -1):

if dp[i][w] != dp[i - 1][w]:

print(i, end=" ") # Print item index (1-based)

w = w - weights[i - 1] # Update remaining weight

return dp[n][capacity]

# Driver code

n = int(input("Enter number of items: "))

profits = [0] \* n

weights = [0] \* n

# Input profit and weight for each item together

for i in range(n):

print(f"Enter profit and weight for item {i + 1} (separated by space):")

# Input profit and weight together

data = input()

# Find where the space is in the string to separate profit and weight

space\_index = data.find(' ')

profit = int(data[:space\_index]) # Extract profit (before space)

weight = int(data[space\_index + 1:]) # Extract weight (after space)

profits[i] = profit

weights[i] = weight

# Input capacity of the knapsack

capacity = int(input("Enter the capacity of the knapsack: "))

# Call the knapsack function

knapsack(profits, weights, capacity)

//Bct\_p3\_bank

//SPDX-License-Identifier: Unlicensed

pragma solidity ^0.8.0;

contract Bank{

mapping(address=>uint) public

balances; function deposit(uint

\_amount) public payable{

balances[msg.sender] += \_amount;

}

function withdraw(uint \_amount) public{

require(balances[msg.sender]>= \_amount, "Not enough ether"); balances[msg.sender] -= \_amount;

}

function getBal() public view

returns(uint){ return

balances[msg.sender];

}

}

//Bct4\_student

// SPDX-License-Identifier: Unlicensed

pragma solidity ^0.8.0;

contract Database {

    struct Student { int ID; string firstName; string lastName; }

    int public studentCount;

    mapping(int => Student) public studentRecords;

    event Action(int id, string action);

    event Received(address user, uint amount);

    function addStudent(int \_id, string calldata \_first, string calldata \_last) public {

        studentRecords[\_id] = Student(\_id, \_first, \_last);

        studentCount++; emit Action(\_id, "added");

    }

    function deleteStudent(int \_id) public {

        require(bytes(studentRecords[\_id].firstName).length != 0, "Not found.");

        delete studentRecords[\_id]; studentCount--; emit Action(\_id, "deleted");

    }

    function updateStudent(int \_id, string calldata \_first, string calldata \_last) public {

        require(bytes(studentRecords[\_id].firstName).length != 0, "Not found.");

        studentRecords[\_id] = Student(\_id, \_first, \_last); emit Action(\_id, "updated");

    }

    receive() external payable { emit Received(msg.sender, msg.value); }

}