Title of the Assignment: Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Code: (non-recursion)

```
# Program to display the Fibonacci sequence up to n-th term
nterms = int(input("How many terms? "))
# first two terms
n1, n2 = 0, 1
count = 0
# check if the number of terms is valid
if nterms <= 0:
        print("Please enter a positive integer")
# if there is only one term, return n1
elif nterms == 1:
                print("Fibonacci sequence upto",nterms,":")
print(n1)
# generate fibonacci sequenceelse:
        print("Fibonacci sequence:")
        while count < nterms:
                print(n1)
                nth = n1 + n2
# update values
n1 = n2
n2 = nth
count += 1
```

Output :

How many terms? 7
Fibonacci sequence:
0
1
1
2
3
5
8

Code : (recursion)

```
# Python program to display the Fibonacci sequence
def recur_fibo(n):
if n <= 1:
        return n
else:
        return(recur_fibo(n-1) + recur_fibo(n-2))
nterms = 7
# check if the number of terms is valid
if nterms <= 0:
        print("Plese enter a positive integer")
else:
        print("Fibonacci sequence:")
        for i in range(nterms):
                print(recur_fibo(i))
```

Output :

```
Fibonacci sequence:
0
1
1
2
3
5
8
```

Title of the Assignment: Write a program to solve a fractional Knapsack problem using a greedy method.

Code :

```
class Item:
	def init (self, value, weight):
		self.value = value
		self.weight = weight
def fractionalKnapsack(W, arr):
	# Sorting Item on basis of ratio
	arr.sort(key=lambda x: (x.value/x.weight), reverse=True)
	# Result(value in Knapsack)
	finalvalue = 0.0
	#Looping through all Items
	for item in arr:
		# If adding Item won't overflow,
		# add it completely
		if item.weight <= W:
			W -= item.weight
			finalvalue += item.value
		# If we can't add current Item,
		# add fractional part of it
		else:
			finalvalue += item.value * W / item.weight
			break
	# Returning final value
	return finalvalue
# Driver Code
if name == " main ":
	W = 50
	arr = [Item(60, 10), Item(100, 20), Item(120, 30)]
	# Function call
	max_val = fractionalKnapsack(W, arr)
	print(max_val)
```

Output :

Maximum value we can obtain = 24

Title of the Assignment: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.
Code:

```
# A Dynamic Programming based Python
# Program for 0-1 Knapsack problem
# Returns the maximum value that can
# be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
        dp = [0 for i in range(W+1)] # Making the dp array
        for i in range(1, n+1): # taking first i elements
        for w in range(W, 0, -1):
# previous computation when taking i-1 items
        if wt[i-1] <= w:
        # finding the maximum value
        dp[w] = max(dp[w], dp[w-wt[i-1]]+val[i-1])
return dp[W] # returning the maximum value of knapsack

# Driver code
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

Output :
220

Title of the Assignment: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‟s matrix.

Code:

```
# Python3 program to solve N Queen
# Problem using backtracking
global N
N = 4
	def printSolution(board):
	for i in range(N):
		for j in range(N):
			print(board[i][j], end = " ")
		print()
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):
	# Check this row on left side
	for i in range(col):
		if board[row][i] == 1:
			return False
	# Check upper diagonal on left side
	for i, j in zip(range(row, -1, -1),
		range(col, -1, -1)):
	if board[i][j] == 1:
		return False
	# Check lower diagonal on left side
	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):
	# base case: If all queens are placed
	# then return true
	if col >= N:
		return True
```

```
    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1
            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
            return True
            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0
    # if the queen can not be placed in any row in
    # this column col then return false
    return False
# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
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
# Driver Code
solveNQ()
```

Output :

0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

Title of the Assignment: Write a program for analysis of quick sort by using deterministic and randomized variant.

Code:
```
import random

def deterministic_quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[0]
    lesser = []
    equal = []
    greater = []

    for element in arr:
        if element < pivot:
            lesser.append(element)
        elif element == pivot:
            equal.append(element)
        else:
            greater.append(element)

    return deterministic_quick_sort(lesser) + equal + deterministic_quick_sort(greater)

def randomized_quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = random.choice(arr)
    lesser = []
    equal = []
    greater = []

    for element in arr:
        if element < pivot:
            lesser.append(element)
        elif element == pivot:
            equal.append(element)
```

```python
        else:
            greater.append(element)

    return randomized_quick_sort(lesser) + equal + randomized_quick_sort(greater)

if __name__ == "__main__":
    arr = [3, 6, 8, 10, 1, 2, 1]

    # Deterministic Quick Sort
    sorted_arr_deterministic = deterministic_quick_sort(arr.copy())
    print("Deterministic Quick Sort:")
    print(sorted_arr_deterministic)

    # Randomized Quick Sort
    sorted_arr_randomized = randomized_quick_sort(arr.copy())
    print("\nRandomized Quick Sort:")
    print(sorted_arr_randomized)
```

Output:

```
Deterministic Quick Sort:
[1, 1, 2, 3, 6, 8, 10]

Randomized Quick Sort:
[1, 1, 2, 3, 6, 8, 10]
```

6.Write a program to implement matrix multiplication. Also implement multithreaded matrix multiplication with either one thread per row or one thread per cell. Analyze and compare their performance.

Code:

```
// CPP Program to multiply two matrix using pthreads

#include <bits/stdc++.h>

using namespace std;

// maximum size of matrix

#define MAX 4

// maximum number of threads

#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++; //i denotes row number of resultant matC

for (int j = 0; j < MAX; j++)

for (int k = 0; k < MAX; k++)

matC[i][j] += matA[i][k] * matB[k][j];

}

// Driver Code

int main()

{

// Generating random values in matA and matB

for (int i = 0; i < MAX; i++) {

for (int j = 0; j < MAX; j++) {

matA[i][j] = rand() % 10;
```

```
matB[i][j] = rand() % 10;
}


}
// Displaying matA
cout << endl
<< "Matrix A" << endl;
for (int i = 0; i < MAX; i++) {
for (int j = 0; j < MAX; j++)
cout << matA[i][j] << " ";
cout << endl;
}
// Displaying matB
cout << endl
<< "Matrix B" << endl;
for (int i = 0; i < MAX; i++) {
for (int j = 0; j < MAX; j++)
cout << matB[i][j] << " ";
cout << endl;
}
// declaring four threads
pthread_t threads[MAX_THREAD];
// Creating four threads, each evaluating its own part
for (int i = 0; i < MAX_THREAD; i++) {
int* p;
pthread_create(&threads[i], NULL, multi, (void*)(p));
}
// joining and waiting for all threads to complete
```

```
for (int i = 0; i < MAX_THREAD; i++)
pthread_join(threads[i], NULL);
// Displaying the result matrix
cout << endl
<< "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;
}
```

Output: