DAA Programs

PROGRAM 1.  
  
import timeit  
import matplotlib.pyplot as plt  
  
def Input(Array, n):  
 for i in range(n):  
 ele = int(input("Arr : "))  
 Array.append(ele)  
  
def linear\_search(Array, key):  
 for x in Array:  
 if x == key:  
 return True  
 return False  
  
N = []  
CPU = []  
  
trail = int(input("Enter number of trials: "))  
  
for t in range(trail):  
 Array = []  
 print("\n-----> TRIAL NO :", t + 1)  
 n = int(input("Enter number of elements: "))  
 Input(Array, n)  
 print("Array:", Array)  
 key = int(input("Enter key to search: "))  
  
 start = timeit.default\_timer()  
 found = linear\_search(Array, key)  
 end = timeit.default\_timer()  
  
 print("Element Found:", found)  
 elapsed\_time = (end - start) \* 1\_000\_000   
 N.append(n)  
 CPU.append(round(elapsed\_time, 2))  
  
  
print("\nN\tCPU Time (µs)")  
for i in range(trail):  
 print(N[i], "\t", CPU[i])  
  
  
plt.plot(N, CPU, label='Time vs Size')  
plt.scatter(N, CPU, color="red", marker="\*", s=50)  
plt.xlabel('Array Size - N')  
plt.ylabel('CPU Time (µs)')  
plt.title('Linear Search Time Efficiency')  
plt.grid(True)  
plt.legend()  
plt.show()  
  
  
program 2  
  
import timeit  
import matplotlib.pyplot as plt  
  
def Input(Array, n):  
 for i in range(n):  
 ele = int(input("Arr : "))  
 Array.append(ele)  
def binary\_search(Array, key):  
 low = 0  
 high = len(Array) - 1  
  
 while low <= high:  
 mid = (low + high) // 2  
 if Array[mid] == key:  
 return True  
 elif Array[mid] < key:  
 low = mid + 1  
 else:  
 high = mid - 1  
 return False  
  
N = []  
CPU = []  
  
trail = int(input("Enter no. of trials: "))  
  
for t in range(trail):  
 Array = []  
 print("\n-----> TRIAL NO :", t + 1)  
 n = int(input("Enter number of elements: "))  
 Input(Array, n)  
 Array.sort()   
 print("Sorted Array:", Array)  
 key = int(input("Enter key to search: "))  
 start = timeit.default\_timer()  
 s = binary\_search(Array, key)  
 times = timeit.default\_timer() - start  
  
 print("Element Found =", s)  
 N.append(n)  
 CPU.append(round(times \* 1000000, 2))  
  
print("\nN\tCPU Time (µs)")  
for t in range(trail):  
 print(N[t], "\t", CPU[t])  
  
plt.plot(N, CPU, label="Time vs Size")  
plt.scatter(N, CPU, color="red", marker="\*", s=50)  
plt.xlabel('Array Size - N')  
plt.ylabel('CPU Processing Time (µs)')  
plt.title('Binary Search Time Efficiency')  
plt.grid(True)  
plt.legend()  
plt.show()  
  
program 3  
def TowerOfHanoi(n, source, destination, auxiliary):  
 if n == 1:  
 print("Move disk 1 from source", source, "to destination", destination)  
 return  
 TowerOfHanoi(n - 1, source, auxiliary, destination)  
 print("Move disk", n, "from source", source, "to destination", destination)  
 TowerOfHanoi(n - 1, auxiliary, destination, source)  
n = int(input("Enter number of disks: "))  
TowerOfHanoi(n, 'A', 'B', 'C')  
  
program 4  
import timeit  
import matplotlib.pyplot as plt  
def Input(Array, n):  
 for i in range(0, n):  
 ele = int(input("Arr : "))  
 Array.append(ele)  
  
   
def linear\_search(Array, key):  
 for x in Array:  
 if x == key:  
 return True  
 return False  
  
  
N = []  
CPU = []  
trail = int(input("Enter no. of trails : "))  
  
  
for t in range(0, trail):  
 Array = []  
 print("-----> TRAIL NO : ", t + 1)  
 n = int(input("Enter number of elements : "))  
 Input(Array, n)  
 print(Array)  
 key = int(input("Enter key :"))  
 start = timeit.default\_timer()  
 s = linear\_search(Array, key)  
 print("Element Found = ", s)  
 times = timeit.default\_timer() - start  
 N.append(n)  
 CPU.append(round(float(times) \* 1000000, 2))  
 print("N CPU")  
  
   
for t in range(0, trail):  
 print(N[t], CPU[t])  
plt.plot(N, CPU)  
plt.scatter(N, CPU, color= "red", marker= "\*", s=50)  
plt.xlabel('Array Size - N')  
plt.ylabel('CPU Processing Time')  
plt.title('Linear Search Time efficiency')  
plt.show()  
  
program 5  
  
def bpower(a, n):  
 pow = 1  
 for i in range(n):  
 pow \*= a  
 return pow  
  
  
def dpower(x, y):  
 if y == 0:  
 return 1  
 elif y % 2 == 0:  
 half = dpower(x, y // 2)  
 return half \* half  
 else:  
 half = dpower(x, y // 2)  
 return x \* half \* half  
  
  
a = int(input("Enter a: "))  
n = int(input("Enter n: "))  
  
print("Brute Force method a^n:", bpower(a, n))  
print("Divide and Conquer a^n:", dpower(a, n))  
  
program 6  
import timeit  
import random  
import matplotlib.pyplot as plt  
  
  
def Input(Array, n):  
   
 for i in range(0, n):  
 ele = random.randrange(1, 50)  
   
 Array.append(ele)  
  
  
def partition(Array, low, high):  
 i = low - 1  
 pivot = Array[high]   
 for j in range(low, high):  
   
 if Array[j] <= pivot:  
   
 i += 1  
 Array[i], Array[j] = Array[j], Array[i]  
 Array[i + 1], Array[high] = Array[high], Array[i + 1]  
 return i + 1  
  
  
def quickSort(Array, low, high):  
 if low < high:  
   
 pi = partition(Array, low, high)  
   
 quickSort(Array, low, pi - 1)  
 quickSort(Array, pi + 1, high)  
  
  
N = []  
CPU = []  
  
trail = int(input("Enter number of trials: "))  
  
for t in range(trail):  
 Array = []  
 print("-----> TRIAL NO:", t + 1)  
 n = int(input("Enter number of elements: "))  
 Input(Array, n)  
 start = timeit.default\_timer()  
 quickSort(Array, 0, n - 1)  
 times = timeit.default\_timer() - start  
 print("Sorted Array:")  
 print(Array)  
 N.append(n)  
 CPU.append(round(times \* 1000000, 2))   
  
  
print("N CPU")  
for t in range(trail):  
 print(N[t], CPU[t])  
  
  
plt.plot(N, CPU)  
plt.scatter(N, CPU, color="red", marker="\*", s=50)  
plt.xlabel('Array Size - N')  
plt.ylabel('CPU Processing Time (μs)')  
plt.title('Quick Sort Time Efficiency')  
plt.show()  
  
program 7  
def binomialCoeff\_BF(n, k):  
 if k > n:  
 return 0  
 if k == 0 or k == n:  
 return 1  
   
 return binomialCoeff\_BF(n - 1, k - 1) + binomialCoeff\_BF(n - 1, k)  
  
  
def binomialCoef\_DC(n, k):  
 C = [[0 for \_ in range(k + 1)] for \_ in range(n + 1)]  
  
   
 for i in range(n + 1):  
 for j in range(min(i, k) + 1):  
   
 if j == 0 or j == i:  
 C[i][j] = 1  
 else:  
   
 C[i][j] = C[i - 1][j - 1] + C[i - 1][j]  
 return C[n][k]  
  
  
n = int(input("Enter n: "))  
k = int(input("Enter k: "))  
  
print("Brute Force method C(n, k):", binomialCoeff\_BF(n, k))  
print("Divide and Conquer C(n, k):", binomialCoef\_DC(n, k))  
  
  
program 8  
nV = 4  
INF = 999  
  
  
def floyd(G):  
 dist = list(map(lambda p: list(map(lambda q: q, p)), G))  
 # Adding vertices individually  
 for r in range(nV):  
 for p in range(nV):  
 for q in range(nV):  
 dist[p][q] = min(dist[p][q], dist[p][r] + dist[r][q])  
 sol(dist)  
  
  
def sol(dist):  
 for p in range(nV):  
 for q in range(nV):  
 if dist[p][q] == INF:  
 print("INF", end=" ")  
 else:  
 print(dist[p][q], end=" ")  
 print(" ")  
  
  
G = [  
 [0, 5, INF, INF],  
 [50, 0, 15, 5],  
 [30, INF, 0, 15],  
 [15, INF, 5, 0]  
]  
  
floyd(G)  
  
program 9  
import timeit  
  
def polynomial\_BF(poly, x, n):  
   
 result = 0  
   
 for i in range(n):  
   
 Sum = poly[i]  
   
 for j in range(n - i - 1):  
 Sum = Sum \* x  
   
 result = result + Sum  
   
 print("Value of polynomial 2x^3 - 6x^2 + 2x - 1 for x = 3 using [BRUTE FORCE method]:", result)  
  
def horner(poly, x, n):  
   
 res = poly[0]  
   
 for i in range(1, n):  
 res = res \* x + poly[i]  
 print("Value of polynomial 2x^3 - 6x^2 + 2x - 1 for x = 3 using [HORNER method]:", res)  
  
  
  
poly = [2, -6, 2, -1]  
x = 3  
n = len(poly)  
  
start1 = timeit.default\_timer()  
polynomial\_BF(poly, x, n)  
t1 = timeit.default\_timer() - start1  
  
start2 = timeit.default\_timer()  
horner(poly, x, n)  
t2 = timeit.default\_timer() - start2  
  
print("Time complexity of Brute Force method O(n^2):", t1)  
print("Time complexity of Horner method O(n):", t2)  
  
  
program 10  
def BoyerMooreHorspool(pattern, text):  
 m = len(pattern)  
 n = len(text)  
 if m > n:  
 return -1  
  
 skip = []  
 for k in range(256):  
 skip.append(m)  
 for k in range(m - 1):  
 skip[ord(pattern[k])] = m - k - 1  
 skip = tuple(skip)  
  
 k = m - 1  
 while k < n:  
 j = m - 1  
 i = k  
 while j >= 0 and text[i] == pattern[j]:  
 j -= 1  
 i -= 1  
 if j == -1:  
 return i + 1  
 k += skip[ord(text[k])]  
  
 return -1  
  
# Main block  
if \_\_name\_\_ == '\_\_main\_\_':  
 text = input("Enter the text:")  
 pattern = input("Enter the key text:")  
 s = BoyerMooreHorspool(pattern, text)  
 print('Text:', text)  
 print('Pattern:', pattern)  
 if s > -1:  
 print('Pattern "' + pattern + '" found at position', s)  
 else:  
 print('Pattern not found.')  
  
  
program 11  
def get\_prefix\_array(pattern):  
 b = len(pattern)  
 prefix = [0] \* b  
 j = 0 # length of previous longest prefix  
 for i in range(1, b):  
 while j > 0 and pattern[i] != pattern[j]:  
 j = prefix[j - 1]  
 if pattern[i] == pattern[j]:  
 j += 1  
 prefix[i] = j  
 return prefix  
  
def kmp\_search(text, pattern):  
 a, b = len(text), len(pattern)  
 prefix = get\_prefix\_array(pattern)  
 result = []  
 i = j = 0  
  
 while i < a:  
 if text[i] == pattern[j]:  
 i += 1  
 j += 1  
 if j == b:  
 result.append(i - j)  
 j = prefix[j - 1]  
 elif i < a and text[i] != pattern[j]:  
 if j != 0:  
 j = prefix[j - 1]  
 else:  
 i += 1  
 return result  
  
text = "ABABDABACDABABCABABCABAB"  
pattern = "ABABCABAB"  
positions = kmp\_search(text, pattern)  
  
print("Text:", text)  
print("Pattern:", pattern)  
for pos in positions:  
 print(f"Pattern found at index {pos}")  
  
program 12  
  
graph = {  
 '5': ['3', '7'],  
 '3': ['2', '4'],  
 '7': ['8'],  
 '2': [],  
 '4': ['8'],  
 '8': []  
}  
  
visited = []   
queue = []   
  
def bfs(visited, graph, node):  
 visited.append(node)  
 queue.append(node)  
 while queue:   
 m = queue.pop(0)  
 print(m, end=" ")  
  
 for neighbour in graph[m]:  
 if neighbour not in visited:  
 visited.append(neighbour)  
 queue.append(neighbour)  
  
  
print("Following is the Breadth-First Search:")  
bfs(visited, graph, '5')  
  
Program 13  
INF = 9999999  
V = 5  
  
G = [  
 [0, 2, 0, 6, 0],  
 [2, 0, 3, 8, 5],  
 [0, 3, 0, 0, 7],  
 [6, 8, 0, 0, 9],  
 [0, 5, 7, 9, 0]  
]  
  
selected = [0, 0, 0, 0, 0]  
no\_edge = 0  
selected[0] = True  
  
print("Edge : Weight\n")  
  
while no\_edge < V - 1:  
 minimum = INF  
 x = 0  
 y = 0  
  
 for i in range(V):  
 if selected[i]:  
 for j in range(V):  
 if (not selected[j]) and G[i][j]:  
 # Not in selected and there is an edge  
 if minimum > G[i][j]:  
 minimum = G[i][j]  
 x = i  
 y = j  
  
 print(str(x) + " - " + str(y) + " : " + str(G[x][y]))  
 selected[y] = True  
 no\_edge += 1  
  
program 14 a  
from collections import defaultdict  
  
class Graph:  
 def \_\_init\_\_(self, directed=False):  
 self.graph = defaultdict(list)  
 self.directed = directed  
  
 def addEdge(self, frm, to):  
 self.graph[frm].append(to)  
 if not self.directed:  
 self.graph[to].append(frm)  
 else:  
 # Ensure all nodes are in the graph  
 if to not in self.graph:  
 self.graph[to] = self.graph[to]  
  
 def topoSortvisit(self, s, visited, sortlist):  
 visited[s] = True  
 for i in self.graph[s]:  
 if not visited[i]:  
 self.topoSortvisit(i, visited, sortlist)  
 sortlist.insert(0, s)  
  
 def topoSort(self):  
 visited = {i: False for i in self.graph}  
 sortlist = []  
 for v in self.graph:  
 if not visited[v]:  
 self.topoSortvisit(v, visited, sortlist)  
 print(sortlist)  
  
# Main block  
if \_\_name\_\_ == '\_\_main\_\_':  
 g = Graph(directed=True)  
 g.addEdge(1, 2)  
 g.addEdge(1, 3)  
 g.addEdge(2, 4)  
 g.addEdge(2, 5)  
 g.addEdge(3, 4)  
 g.addEdge(3, 6)  
 g.addEdge(4, 6)  
  
 print("Topological Sort:")  
 g.topoSort()  
  
program 14 b  
  
class Graph:  
 def \_\_init\_\_(self, vertices):  
 self.V = vertices  
  
 # A utility function to print the solution  
 def printSolution(self, reach):  
 print("Transitive Closure of the given graph:")  
 for i in range(self.V):  
 for j in range(self.V):  
 print("%3d" % (1 if i == j else reach[i][j]), end=" ")  
 print()  
  
 # Prints transitive closure of graph[][] using Floyd Warshall algorithm  
 def transitiveClosure(self, graph):  
 reach = [row[:] for row in graph]  
 for k in range(self.V):  
 for i in range(self.V):  
 for j in range(self.V):  
 reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])  
 self.printSolution(reach)  
  
# Main module  
g = Graph(4)  
graph = [  
 [1, 1, 0, 1],  
 [0, 1, 1, 0],  
 [0, 0, 1, 1],  
 [0, 0, 0, 1]  
]  
g.transitiveClosure(graph)  
  
  
PROGRAM 15  
  
  
  
from itertools import combinations  
  
def Input(S, n):  
   
 for i in range(n):  
 ele = int(input("Arr : "))  
   
 S.append(ele)  
  
def sub\_set\_sum(size, S, d):  
 count = 0  
 for i in range(size + 1):  
 for my\_sub\_set in combinations(S, i):  
 if sum(my\_sub\_set) == d:  
 print(list(my\_sub\_set))  
 count += 1  
 if count == 0:  
 print("Subset Not found for the given d =", d)  
  
  
S = []  
n = int(input("Enter size: "))  
Input(S, n)  
print("Input array:", S)  
d = int(input("Enter sum d: "))  
print("The result is:")  
sub\_set\_sum(n, S, d)