#1PRGM

import time

import matplotlib.pyplot as plt

def linearsearch(arr,n,key):

for i in range(n):

if arr[i]==key:

return i+1

return-1

def linear\_search\_n(r):

results=[]

for \_ in range(r):

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

arr=list(map(int,input("\n Enter the elements of an array:").split()))

key=int(input("\nEnter the key element to be searched:"))

repeat=10000

result=-1

start=time.time()

for \_ in range(repeat):

result=linearsearch(arr,n,key)

end=time.time()

if result!=-1:

print(f"key {key} found at position{result}")

else:

print(f"key {key} not found")

time\_taken=(end-start)\*1000

print(f"Time taken to search a key element={time\_taken}milli seconds")

results.append((n,time\_taken))

return results

def plot\_results(results):

n\_vlaues=[result[0] for result in results]

times=[result[1] for result in results]

plt.figure()

plt.plot(n\_values,time,'0\_')

plt.xlabel('Number of elements(n)')

plt.ylabel('Time taken(milli seconds)')

plt.title('linear search Time Complexity')

plt.grid(True)

plt.show()

r=int(input("Enter the number of runs:"))

results=linear\_search\_n(r)

plot\_results(results)

#2PRGM

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 binary\_search(array,key):

while len(array)>0:

mid=(len(array))//2

if array[mid]==key:

return True

elif array[mid]<key:

array=array[:mid]

else:

array=array[mid+1:]

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= binary\_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("binary search time efficiency")

plt.show()

#3PRGM

def toh(n,source,temp,dest):

global count

if n>0:

toh(n-1,source,dest,temp)

print(f"Move disk {n} {source}->{dest}")

count+=1

toh(n-1,temp,source,dest)

source="s"

temp="t"

dest="d"

count=0

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

print("Sequence is:")

toh(n,source,temp,dest)

print("The Number of Moves:",count)

#4 PRGM

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 selectionSort(Array, size):

for ind in range(size):

min\_index = ind

for j in range(ind + 1, size):

if Array[j] < Array[min\_index]:

min\_index = j

(Array[ind], Array[min\_index]) = (Array[min\_index], Array[ind])

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)

start = timeit.default\_timer()

selectionSort(Array, n)

times = timeit.default\_timer() - start

print("Sorted Array:")

print(Array)

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('Selection Sort Time efficiency')

plt.show()

#5PRGM

def power\_bruteforce(a,n):

result=1

for i in range(n):

result\*=a

return result

def power\_divide\_conquer(a,n):

if n==0:

return 1

elif n%2==0:

return power\_divide\_conquer(a\*a,n//2)

else:

return a\*power\_divide\_conquer(a\*a,n//2)

a,n=map(int,input("Enter the value of a and n:").split())

result\_brute=power\_bruteforce(a,n)

result\_divide\_conquer=power\_divide\_conquer(a,n)

print("Result using brute force:",result\_brute)

print("Result using divide and conquer:",result\_divide\_conquer)

#6 PRGM

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=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 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)

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(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('Quick Sort Time Efiiciency')

plt.show()

#7PRGM

def factorial(n):

fact =1

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

fact\*= 1

return fact

def binomialCoeff\_bruteForce(n,k):

return factorial(n)

def binomialCoeff\_DP(n, k):

c= [[0 for j in range (k + 1) ] for i 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 the value of n: "))

k = int(input("Enter the value of k: "))

result\_bruteForce = binomialCoeff\_bruteForce(n, k)

result\_DP = binomialCoeff\_DP(n, k)

print(f"Binomial Coefficient (Brute Force): {result\_bruteForce}")

print(f"Binomial Coefficient (Dynamic Programming): {result\_DP}")

#8 prgm

INF=9999

def printsolution(V,D):

print("The All Pair Shortest Path:")

for i in range(V):

for j in range(V):

if D[i][j]==INF:

print("%7s"%"INF",end="")

else:

print("%7d"%D[i][j],end="")

print()

def floyd(V,C):

D=[[0]\*V for \_ in range(V)]

for i in range(V):

for j in range(V):

D[i][j]=C[i][j]

for k in range(V):

for i in range(V):

for j in range(V):

if D[i][j]>(D[i][k]+D[k][j]):

D[i][j]=D[i][k]+D[k][j]

printsolution(V,D)

V=int(input("Enter the number of vertices:"))

C=[[0]\*V for \_ in range(V)]

print("Enter the cost matrix:")

print("Enter 9999 for Infinity")

print("Enter 0 for cost(i,i)")

for i in range(V):

C[i]=list(map(int,input().split()))

floyd(V,C)

#9PRGM

import time

import math

def bruteforce(coef,n,x):

sum=0.0

for i in range(n+1):

sum+=coef[i]\*math.pow(x,i)

return sum

​

def hornersrule(coef,n,x):

result=coef[n]

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

result=result\*x+coef[i]

return result

​

n=int(input("Enter the degree of the polynomial:"))

coef=[0]\*(n+1)

print("Enter the coefficients from highest degree to lowest:")

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

coef[i]=int(input())

x=float(input("Enter the value of x:"))

start=time.time()

brute\_force\_result=bruteforce(coef,n,x)

end=time.time()

time\_used=end-start

print(f"Brute force result:{brute\_force\_result:.2f},time used:{time\_used:.6f}seconds")

start=time.time()

horners\_rule\_result=hornersrule(coef,n,x)

end=time.time()

time\_used=end-start

print(f"Hornersrule result:{horners\_rule\_result:.2f},time used:{time\_used:.6f}seconds")

#10 prgm

MAX\_CHARS =256

def max(a,b):

return a if a>b else b

def badCharHeuristic(pat,size,badchar):

for i in range(MAX\_CHARS):

badchar[i]=-1

for i in range(size):

badchar[ord(pat[i])]=i

def patternsearch(text,pat):

m=len(pat)

n=len(text)

badchar=[-1]\*MAX\_CHARS

badCharHeuristic(pat,m,badchar)

s=0 #s is shift of the pattern with respect to text

while s<=(n-m):

j=m-1

while j>=0 and pat[j]==text[s+j]:

j-=1

if j<0:

print("\nPattern occurs at position=",s)

s+=m-badchar[ord(text[s+m])] if (s+m)<n else 1

else:

s+=max(1,j-badchar[ord(text[s+j])])

#main code

text=input("Enter the text:").rstrip('\n')

pat=input("Enter the pattern:").rstrip('\n')

patternsearch(text,pat)

#12prgm

MAX=100

c=[[0]\*MAX for \_ in range(MAX)]

visited=[0]\*MAX

queue=[0]\*MAX

def BFS(v):

front=0

rear=-1

visited[v]=1

queue[rear+1]=v

rear+=1

while front<=rear:

v=queue[front]

front+=1

print(f"{v}",end=" ")

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

if c[v][i]==1 and visited[i]==0:

queue[rear+1]=i

rear+=1

visited[i]=1

if \_\_name\_\_=="\_\_main\_\_":

print("Enter the number of vertices in the graph: ")

n=int(input())

print("Enter the cost matrix of the graph:")

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

c[i]=[0]+list(map(int,input().split()))

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

visited[i]=0

print("Enter the starting vertex:")

v=int(input())

print("BFS traversal of the graph is: ", end="")

BFS(v)

#13 prgm

import sys

def minkey(key,mstSet,n):

min\_value sys.maxsize

for v in range(n):

if mstSet[v]== False and key [v] < min\_value:

min\_value key[v]

min index V

return min\_index

def printMST(parent, c, n):

totalweighte

print("Edge Weight")

for i in range(1, n):

print(str(parent[1]+1)+ str(1+1)+" "+str(c[i][parent[1]]))

totalweight + c[1] [parent[1]]

return totalweight

def primMST(c, n):

parent [None]\*n

key [sys.maxsize]\*n

mstSet [False]\*n

key[0] 0

parent[0] = -1

for count in range(n):

uminkey(key, mstSet, n)

mstSet [u] True

for v in range(n):

if c[u] [v] > 0 and mstSet[v] == False and c[u] [v] <key[v]:

parent[v] = u

key[v] = c[u][v]

totalweight printMST (parent, c, n)

print("Total cost of the minimum spanning tree: " + str(totalweight))