**Maximum number of activities in a room**

s=[int(i) for i in input().split()]

f=[int(i) for i in input().split()]

a=[]

for i in range(len(s)):

l=[]

l.append(s[i])

l.append(f[i])

a.append(l)

a.sort(key=lambda a:a[1])

i=0

l1=[]

l1.append(i+1)

for j in range(1,len(s)):

if(a[j][0]>=a[i][1]):

l1.append(j+1)

i=j

print("The selected activites are ")

print(l1)

**Mobile keypad problem**

row = [0, 0, -1, 0, 1]

col = [0, -1, 0, 1, 0]

def getCountUtil(key,i,j,n):

if (keypad == None or n <= 0):

return 0

if (n == 1):

return 1

k=0

move = 0

ro = 0

co = 0

totalCount = 0

for move in range(5):

ro = i + row[move]

co = j + col[move]

if (ro >= 0 and ro <= 3 and co >= 0 and co <= 2 and keypad[ro][co] != '\*' and keypad[ro][co] != '#'):

totalCount += getCountUtil(keypad, ro, co, n-1)

return totalCount

def count(key,n):

if(key==None or n<0):

return 0

if(n==1):

return 10

i=0

j=0

total=0

for i in range(4):

for j in range(3):

if(key[i][j]!='\*' and key[i][j]!='#'):

total+=getCountUtil(key,i,j,n)

return total

keypad=[['1','2','3'],['4','5','6'],['7','8','9'],['\*','0','#']]

n=int(input())

p=count(keypad,n)

print(p)

**Painters partition**

def numberPainters(l,n,k):

total=0

numPainters=1

for i in l:

total+=i

if(total>k):

total=i

numPainters+=1

return numPainters

def partition(l,n,k):

low=max(l)

high=sum(l)

while(low<high):

mid=low+(high-low)/2

reqPainters=numberPainters(l,n,mid)

if(reqPainters<=k):

high=mid

else:

low=mid+1

return low

print("Enter the value of K")

k=int(input())

print("Enter the lengths of the boards")

l=[int(i) for i in input().split()]

n=int(input())

print("The Minimum Time is ",end=" ")

print(int(partition(l,n,k)))

**painting fence**

n=int(input())

k=int(input())

#Adjacent with same colour

groups=n//2

poss=groups\*k

if(n%2==1):

poss+=k

#Adjacent with different colours

diff=k

for i in range(1,n):

diff\*=(k-1)

poss+=diff

print(poss)

**probability of 2-3 steps**

def findPro(n,p):

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

d[0]=1

d[1]=0

d[2]=p

d[3]=1-p

for i in range(4,n+1):

d[i]=p\*d[i-2]+(1-p)\*d[i-3]

return d[n]

print("Enter the number of steps")

n=int(input())

print("Enter the probability")

p=float(input())

k=findPro(n,p)

print("The probability is",end=" ")

print(round(k,2))

**knapsack**

def knap(w,wt,val,n):

k=[[0 for i in range(w+1)] for j in range(n+1)]

for i in range(n+1):

for j in range(w+1):

if(i==0 or j==0):

k[i][j]=0

elif(wt[i-1]<=j):

k[i][j]=max(val[i-1]+k[i-1][j-wt[i-1]],k[i-1][j])

else:

k[i][j]=k[i-1][j]

#print(k)

return k[n][w]

print("Enter the number of items")

n=int(input())

val=[]

wt=[]

for i in range(n):

l1=[int(i) for i in input().split()]

val.append(l1[1])

wt.append(l1[2])

print("Enter the size of the knapsack")

w=int(input())

print(knap(w,wt,val,n))

**Raju-optimal BST**

def optCost(freq, i, j):

if j < i:

return 0

if j == i:

return freq[i]

fsum = sum(freq[i:j+1])

Min = 999999999999

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

cost = (optCost(freq, i, r - 1) + optCost(freq, r + 1, j))

if cost < Min:

Min = cost

return Min + fsum

n = int(input("test cases: "))

for kaushik in range(n):

k = [int(i) for i in input("keys: ").split()]

f = [int(i) for i in input("freq: ").split()]

print("Cost of Optimal BST is: ",optCost(f,0,len(k)-1))

**All possible subsets**

def subsets(l):

lists=[[]]

for i in range(len(l)+1):

for j in range(i):

lists.append(l[j:i])

return lists

print("Enter the elements")

l=[int(i) for i in input().split()]

print(subsets(l))

**Search Word in matrix**

class wordmatrix:

def \_\_init\_\_(self,n):

self.solution = [[0 for i in range(n)] for j in range(n)]

self.path = 1

def searchword(self,mat,word):

for i in range(len(mat)):

for j in range(len(mat)):

if self.search(mat,word,i,j,0,len(mat)):

return True

return False

def search(self,matrix,word,row,col,index,N):

if (self.solution[row][col]!=0 or word[index]!=matrix[row][col]):

return False

if (index == len(word)-1 ):

self.solution[row][col] = self.path

self.path+=1

return True

self.solution[row][col] = self.path

self.path+=1

if (row+1<N and self.search(matrix, word, row + 1, col, index + 1, N)):

return True

if (row-1>=0 and self.search(matrix, word, row - 1, col, index + 1, N)):

return True

if (col+1< N and self.search(matrix, word, row, col + 1, index + 1, N)):

return True

if (col-1>=0 and self.search(matrix, word, row, col - 1, index + 1, N)):

return True

if (row-1>=0 and col+1<N and self.search(matrix, word, row-1, col+1, index+1, N)):

return True

if (row-1>=0 and col-1>=0 and self.search(matrix, word, row-1, col-1, index+1, N)) :

return True

if (row+1<N and col-1>=0 and self.search(matrix, word, row+1, col-1, index+1, N)) :

return True

if (row+1<N and col+1<N and self.search(matrix, word, row+1, col+1, index+1, N)):

return True

self.solution[row][col] = 0

self.path-=1

return False

def display(self):

for i in range(len(self.solution)):

for j in range(len(self.solution)):

print(self.solution[i][j],end=" ")

print()

a = []

print("elements: ")

while(True):

s = list(input())

key = input("search word: ")

if w.searchword(a,key):

w.display()

else:

print("no match found")

if s!=[]:

a.append(s)

else:

break

w = wordmatrix(len(a))

**Counting Bits**

def binary(n):

if(n==1):

return 1

elif(n==0):

return 0

return binary(n//2)\*10+(n%2)

n=int(input())

l=[]

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

l.append(str(binary(i)).count('1'))

print(l)

**8 queens**

def solve(matrix):

rows = set()

cols = set()

diags = set()

rev\_diags = set()

for i in range(len(matrix)):

for j in range(len(matrix)):

if matrix[i][j]:

rows.add(i)

cols.add(j)

diags.add(i - j)

rev\_diags.add(i + j)

return len(rows) == len(cols) == len(diags) == len(rev\_diags) == len(matrix)

n = int(input("test cases: "))

for k in range(n):

print("data: ")

a = []

for i in range(8):

a.append(int(list(input())[1]))

m = [[0 for i in range(8)] for j in range(8)]

for i in range(8):

m[i][a[i]-1] = 1

for i in range(8):

for j in range(8):

print(m[i][j],end=" ")

print()

if(solve(m)):

print("valid")

else:

print("not valid")

**Hack the money**

**Multiple of 20 or 10**

def solve(n,curr):

if curr==n:

return True

if curr>n:

return False

return solve(n,curr\*10) or solve(n,curr\*20)

n = int(input("test cases: "))

for kaushik in range(n):

a = int(input("number: "))

if a==1:

print("yes")

else:

if(solve(a,1)):

print("yes")

else:

print("no")

**m colouring**

def isSafe(graph, color):

for i in range(len(graph)):

for j in range(i + 1, len(graph)):

if (graph[i][j] and color[j] == color[i]):

return False

return True

def graphColoring(graph, m, i, color):

if (i == len(graph)):

if (isSafe(graph, color)):

printSolution(color)

return True

return False

for j in range(1, m + 1):

color[i] = j

if (graphColoring(graph, m, i + 1, color)):

return True

color[i] = 0

return False

def printSolution(color):

print("Solution Exists:" " Following are the assigned colors ")

for i in range(len(color)):

print(color[i],end=" ")

a = []

print("elements: ")

while(True):

s = [int(i) for i in input().split()]

if s!=[]:

a.append(s)

else:

break

m = int(input("m: "))

color = [0 for i in range(len(a))]

if (not graphColoring(a, m, 0, color)):

print ("Solution does not exist")

**Reverse Pairs**

cnt = 0

def msort(A):

L = len(A)

if L <= 1: # base case

return A

else: # recursive case

return merger(msort(A[:int(L/2)]), msort(A[int(L / 2):]))

def merger(left, right):

global cnt

l, r = 0, 0 # increase l and r iteratively

while l < len(left) and r < len(right):

if left[l] <= 2 \* right[r]:

l += 1

else:

cnt += len(left) - l # COUNT here

r += 1

res = [] # merger

i, j = 0, 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

res += left[i],

i += 1

else:

res += right[j],

j += 1

while i != len(left):

res += left[i],

i += 1

while j != len(right):

res += right[j],

j += 1

return res

nums = list(map(int,input().split()))

msort(nums)

print(cnt)