#!/bin/python3

from collections import deque

INF = 1000000

def Dijkstra(Graph, source):

global INF

Q = []

for v in Graph.keys():

dist[v] = INF

prev[v] = None

Q.append(v)

dist[source] = 0

while len(Q) > 0:

u = vertex in Q with min dist[u]

Q.pop(Q.index(u))

for each neighbor v of u:

alt = dist[u] + length(u, v)

if alt < dist[v]:

dist[v] = alt

prev[v] = u

return dist[], prev[]

q = int(input().strip())

G = dict()

for a1 in range(q):

node\_1, node\_2 = map(int, input().strip().split())

if node\_1 not in G.keys():

G[node\_1] = []

G[node\_1].append(node\_2)

from time import process\_time

from random import randint

import heapq

def partition(arr,low,high):

i = ( low-1 )

pivot = arr[high]

for j in range(low , high):

if arr[j] <= pivot:

i = i+1

arr[i],arr[j] = arr[j],arr[i]

arr[i+1],arr[high] = arr[high],arr[i+1]

return ( i+1 )

def quickSort(arr,low,high):

if low < high:

pi = partition(arr,low,high)

quickSort(arr, low, pi-1)

quickSort(arr, pi+1, high)

ls = [randint(i - 100, i) for i in range(50000, -1, -1)]

unsorted = ls.copy()

st = process\_time()

quickSort(unsorted, 0, len(unsorted) - 1)

print(process\_time() - st)

from collections import deque

import time

def costsOfPaths(source, destination, vertices):

vertex = [source, 0]

costs = []

q = deque()

while True:

for w in range(len(vertices[vertex[0]])):

if vertices[vertex[0]][w] != 0:

if w == destination:

costs.append(vertex[1] + vertices[vertex[0]][w])

else:

q.appendleft([w, vertex[1] + vertices[vertex[0]][w]])

if len(q) == 0: return costs

vertex = q.pop()

return costs

vertices = [

[0, 1, 2, 0, 10, 0],

[0, 0, 0, 0, 0, 500],

[0, 0, 0, 300, 0, 0],

[0, 0, 0, 0, 0, 4],

[0, 100, 3, 60, 0, 0],

[0, 0, 0, 0, 0, 0]

]

print(min(costsOfPaths(0, 5, vertices)))

print(time.process\_time())

'''

6

0 1 2 0 10 0

0 0 0 0 0 500

0 0 0 300 0 0

0 0 0 0 0 4

0 100 2 60 0 0

0 0 0 0 0 0

'''

def get\_value(mat, i, j, m, n):

if j > n:

return 0

if i > m:

return 0

return mat[i][j]

def find\_total\_paths(mat, m, n):

for i in range(n + 1):

mat[m][i] = 1 if mat[m][i] != 0 else 0

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

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

if mat[i][j] != 0:

mat[i][j] = get\_value(mat, i + 1, j, m, n) + \

get\_value(mat, i, j + 1, m, n)

return mat[0][0]

# take input

mat = [

[9, 9, 0, 9],

[9, 9, 9, 9],

[9, 0, 0, 9],

[9, 9, 9, 9]

]

m = n = len(mat)

print(find\_total\_paths(mat, m - 1, n - 1))

#Levenshtein distance

import operator

def Compute(s, t):

n, m = len(s), len(t)

if (n == 0):

return m

if (m == 0):

return n

d = [[i] for i in range(n + 1)]

for j in range(m + 1): d[0].append(j)

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

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

cost = 0 if (t[j - 1] == s[i - 1]) else 1

d[i].append(min([d[i - 1][j] + 1,

d[i][j - 1] + 1,

d[i - 1][j - 1] + cost]))

return d[n][m]

data = ["abcd", "abc", "abcde", "ab"]

pattern = "abcde"

letsee = {}

for d in data:

letsee[d] = Compute(d, pattern)

print(letsee)

for j in sorted(letsee.items(), key=operator.itemgetter(1)):

print(j[0])

def compareStrings(a, b):

if len(a) > len(b): a, b = b, a

for i in range(len(a)):

if a[i] != b[i]:

return b[:i]

return a

data = "raghavendrasagarraghavendrasaggurusagar"

m = ""

suffixes = []

for i in range(len(data)):

suffixes.append(data[i:])

suffixes.sort()

for i in range(len(suffixes)-1):

c = compareStrings(suffixes[i], suffixes[i+1])

if len(c) > len(m): m = c

print(m)

def readBits(binaries, k):

"""Reads the bits and manages to compose it"""

indices = []

for col in range(8):

for row in range(len(binaries)):

if binaries[row][col] == '1':

indices.append(row)

if len(indices) >= k:

for index in range(len(binaries)):

if index not in indices:

binaries.pop(index)

indices.clear()

return(binaries)

n, s = map(int, input().split())

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

binary = ['{0:08b}'.format(i) for i in alist]

bins = list(readBits(binary, s))

i = int(bins[0], 2)

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

i &= int(bins[j], 2)

print(i)

from multiprocessing import Process, Queue

from time import process\_time

def do\_sum(q, l):

q.put(sum(l))

def main():

my\_list = range(500000000)

q = Queue()

st = process\_time()

p1 = Process(target=do\_sum, args=(q, my\_list[:5000000]))

p2 = Process(target=do\_sum, args=(q, my\_list[5000000:]))

p1.start()

p2.start()

p1.join()

p2.join()

r1 = q.get()

r2 = q.get()

'''

r1 = sum(my\_list[:500000])

r2 = sum(my\_list[500000:])

'''

t = process\_time() - st

print("{0:0.9f}".format(t))

print(r1 + r2)

if \_\_name\_\_ == '\_\_main\_\_':

main()

dp = []

mx = []

def printSubsetsRec(arr, i, sum, p):

global dp, mx

if (i == 0 and sum != 0 and dp[0][sum]):

p.append(arr[i])

if len(mx) < len(p): mx = p.copy()

p.clear()

return

if (i == 0 and sum == 0):

if len(mx) < len(p): mx = p.copy()

p.clear()

return

if (dp[i-1][sum]):

b = p

printSubsetsRec(arr, i-1, sum, b)

if (sum >= arr[i] and dp[i-1][sum-arr[i]]):

p.append(arr[i])

printSubsetsRec(arr, i-1, sum-arr[i], p)

def printAllSubsets(arr, n, sum):

global dp

if (n == 0 or sum < 0):

return

dp = [False] \* n

for i in range(n):

dp[i] = [False] \* (sum + 1)

dp[i][0] = True

if (arr[0] <= sum):

dp[0][arr[0]] = True

for i in range(1, n):

for j in range(sum + 1):

dp[i][j] = dp[i-1][j] or dp[i-1][j-arr[i]] if (arr[i] <= j) else dp[i - 1][j]

if (dp[n-1][sum] == False):

return []

printSubsetsRec(arr, n-1, sum, [])

printAllSubsets([2, 3, 4, 5], 4, 10)

print(mx)

#Cashier algo

def cashier(x, C):

C.sort(reverse=True)

S = []

while x > 0:

ck = None

while len(C) > 0:

if C[0] <= x:

ck = C[0]

break

else:

C.pop(0)

if ck == None:

return []

else:

x = x - ck

S += [ck]

return S

C = [5, 2, 10, 1]

print(cashier(16, C))