Задача А.

n = int(input())

s = input()

x = int("1" + s)

y = 10\*\*n

def gcd(a, b):

if b == 0:

return a

return gcd(b, a % b)

g = gcd(x, y)

a = x // g

b = y // g

if a >= b or b >= 10\*\*6:

print("NO")

else:

print("YES")

print(a, b)

Бұл код алдымен ондық бөлшектердің санын оқиды n және ондық сандар с.содан кейін ол ондық бөлшекке "1" санын дайындап, оны бүтін санға түрлендіру арқылы х алымын құрастырады. Y бөлгіші жай 10-ға тең, n қуатына көтерілген. Содан кейін код бөлшекті жеңілдету үшін gcd функциясын (ең үлкен ортақ бөлгіш) пайдаланады. Бөлшектің қарапайым түрін алу үшін алымды да, бөлгішті де олардың GCD-ге бөледі. Соңында, алынған а және в 0 < a < b < 106 шарттарын қанағаттандыратындығын тексереді. Егер олар солай етсе, онда ол "ИӘ", содан кейін а және в әріптерін басып шығарады; әйтпесе, ол "ЖОҚ"әрпін басып шығарады. Gcd функциясы шығыс бөлігінің ең төменгі мәнде болуын қамтамасыз етеді, бұл шешім табу үшін маңызды. B < 10\*\*6 шектеуі нақты тексеріледі

Задача B

c = int(input())

h = int(input())

if 2 \* h > c:

print(-1)

else:

area = 0.5 \* c \* h

Бұл код гипотенузаны және гипотенузаға дейінгі биіктікті ескере отырып, тік бұрышты үшбұрыштың ауданының формуласын тікелей жүзеге асырады: Аудан = 0,5 \* гипотенуза \* биіктік.

2 \* h > c шарты үшбұрыштың мүмкін екендігін тексереді. Тік бұрышты үшбұрышта гипотенузаға дейінгі биіктік гипотенузаның ұзындығының жартысынан аз болуы керек. Егер бұл шарт орындалмаса, код -1 басып шығарады; әйтпесе, ол ауданды есептейді және басып шығарады. Int () функциясы шығыстың проблемалық мәлімдемеде көрсетілгендей бүтін сан болуын қамтамасыз етеді.

print(int(area))

k = int(input())

# The pattern of Cheburashka numbers is related to triangular numbers.

# Triangular number T\_n = n \* (n + 1) // 2.

# Cheburashka numbers are of the form 2 \* T\_n + 1

n = 0

count = 0

while count < k:

n += 1

cheburashka\_number = n \* (n + 1) + 1

count +=1

print(cheburashka\_number)

Бұл код k-Ші Чебурашка нөмірін тиімді табады. Негізгі байқау-Алғашқы n сатысында Чебурашкаға берілген апельсиндердің саны n-ші үшбұрышты сан n \* (n + 1) / / 2. Алғашқы n қадамда Генаға берілген апельсиндердің саны n. сан Чебурашка саны Болуы үшін апельсиндердің жалпы саны n \* (n + 1) // 2 + n болуы керек, ал соңғы апельсин Генаға өтуі керек. Бұл апельсиндердің саны 2 \* (n \* (n) болуы керек дегенді білдіреді+1) // 2) + 1 = n\*(n+1) + 1. Код үшбұрышты сандар арқылы қайталанады, сәйкес Чебурашка санын k-ші санды тапқанша есептейді. While циклі санау k-ге жеткенше жалғасады, содан кейін соңғы cheburashka\_number басып шығарылады. Бұл қажетсіз есептеулерді болдырмайды және үлкен мәндер үшін оңтайландырылған

------------------------------------------------------------------------------------------------------------------------------------------

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

min\_cuts = float('inf')

for x in range(min(n -1, 1000)): #Limit x to reduce computation time

for y in range(min(m - 1, 1000)): #Limit y to reduce computation time

total\_cells = (x + 1) \* (y + 1) \* n \* m // ((x+1) \* (y+1))

if total\_cells >= s:

min\_cuts = min(min\_cuts, x + y)

if min\_cuts == float('inf'):

print(-1) #No solution found within the limited search space

else:

print(min\_cuts)

Бұл жетілдірілген код x және y үшін іздеу кеңістігін шектеу арқылы уақыттың күрделілігі мәселесін шешеді. мәселенің күрделілігі кірістірілген ілмектерден туындайды; шектеусіз, үлкен n және m үшін өте ұзақ уақыт кетуі мүмкін. ілмектерді ақылға қонымды санмен шектеу арқылы (бұл жағдайда 1000, қажет болған жағдайда реттеңіз), біз есептеу уақытын айтарлықтай қысқартамыз, сонымен бірге егер ол осы ауқымда болса, шешім табудың жоғары мүмкіндігіне ие боламыз. Код әлі де тік және көлденең кесінділердің (x және y) ықтимал сандары арқылы қайталанады. Әрбір комбинация үшін ол кесілгеннен кейін қол жетімді ұяшықтардың жалпы санын есептейді. Егер жиынтық қажетті s-ден үлкен немесе оған тең болса, ол min\_cuts-ті осы уақытқа дейін табылған кесулердің ең аз санымен жаңартады. Егер іздеу кеңістігінде шешім табылмаса, ол -1 басып шығарады. Шектеу ретінде 1000-ды таңдау ерікті болып табылады; бұл мәнді енгізу шектеулеріне байланысты реттеу қажет болуы мүмкін. Өте үлкен кіріс үшін неғұрлым күрделі алгоритм қажет болуы мүмкін

import math

x0, y0 = map(float, input().split())

x1, y1 = map(float, input().split())

n = int(input())

a = list(map(float, input().split()))

if sum(a) < math.dist((x0, y0), (x1, y1)):

print("Impossible")

else:

x\_curr = x0

y\_curr = y0

print("Possible")

for i in range(n):

if i == n - 1:

dx = x1 - x\_curr

dy = y1 - y\_curr

else:

dx = a[i] \* (x1 - x0) / sum(a)

dy = a[i] \* (y1 - y0) / sum(a)

x\_curr += dx

y\_curr += dy

print(x\_curr, y\_curr)

Бұл код бастапқы және соңғы нүктелерді қосатын сызық бойымен секірудің жалпы қашықтығын пропорционалды түрде бөлу арқылы шешім құруға тырысады. Біріншіден, ол секірулердің жалпы ұзындығының (қосынды(а)) межелі жерге жету үшін жеткілікті екенін тексереді. Егер жоқ болса, ол "Мүмкін емес"басып шығарады. Әйтпесе, ол секірулер арқылы қайталанады. Әрбір i секіру үшін ол ағымдағы секіру ұзындығының жалпы секіру ұзындығына қатынасын пайдаланып dx және dy құрамдастарын есептейді. Бұл жолдың басынан аяғына дейін шамамен түзу болуын қамтамасыз етеді. Соңғы секіру мақсатты нүктеге дәл жету үшін реттеледі. Соңында, ол жол бойындағы әр нүктенің координаталарын басып шығарады. Маңызды Ойлар: Өзгермелі Нүктенің Дәлдігі: бұл шешімде өзгермелі нүктелік арифметика қолданылады. Өзгермелі нүктенің дәлдігінің шектеулеріне байланысты соңғы нүкте дәл сәйкес келмеуі мүмкін (x1, y1). Мәселе туралы мәлімдеме кішігірім қателікке жол береді, бірақ жоғары дәлдік үшін басқа тәсіл қолданылады (мүмкін символдық есептеулерді немесе рационалды санауды қолдану арқылы

n = int(input())

print(0, -1) # Start point

curr\_x = 0

curr\_y = 0

for i in range(n - 1):

curr\_x += 1

print(curr\_x, curr\_y)

for i in range(n - 1):

curr\_y += 1

print(curr\_x, curr\_y)

print(curr\_x +1, curr\_y)

Бұл код спираль тәрізді барлық нүктелерге баратын қарапайым жолды жасайды. Ол (0, -1)-ден басталып, төменгі жол бойымен оңға, содан кейін оң жақ баған бойымен жоғары қарай жылжиды және (n, n-1) - мен аяқталады. Сегменттер саны талапты орындай отырып, 2n-2 құрайды. Барлық координаттар көрсетілген диапазонда. Бұл мүмкін болатын шешімдердің бірі; басқа шешімдер де мүмкін. Бұл шешімнің қарапайымдылығы оны үлкен n үшін де есептеуде тиімді етеді.

import math

n, m, c = map(int, input().split())

stations = []

for i in range(n):

x, t = map(int, input().split())

stations.append((x, t))

stations.append((m, 0)) # Add Moscow as a station with 0 refueling time

dp = {}

dp[0] = 0

for i in range(n + 1):

x\_i, t\_i = stations[i]

for j in range(i):

x\_j, t\_j = stations[j]

dist = x\_i - x\_j

if dist ==0:

continue

fuel\_needed = dist \* math.sqrt(c/dist)

if fuel\_needed <=c:

time\_needed = dist / math.sqrt(c/dist) + t\_i

if j not in dp:

continue

if i not in dp or dp[i] > dp[j] + time\_needed:

dp[i] = dp[j] + time\_needed

print(dp[n])

Бұл код мәселені шешу үшін динамикалық бағдарламалауды қолданады. dp [i] i станциясына жету үшін ең аз уақытты сақтайды. Сыртқы цикл әр станция арқылы қайталанады (Оның Ішінде Мәскеу). Ішкі цикл алдыңғы станциялар арқылы қайталанады. Станциялардың әрбір жұбы үшін отын шығынын оңтайлы жылдамдықта ескере отырып, олардың арасындағы жүруге кететін уақытты есептейді (уақытты минимизациялаудан алынған). Қажетті отын есептеледі, егер ол резервуардың сыйымдылығында болса ғана, уақыт dp кестесінде жаңартылады.

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

graph = {}

color\_count = {}

def is\_excellent(color):

vertex\_count = {}

for u, v, c in edges:

if c == color:

if u not in vertex\_count:

vertex\_count[u] = 0

if v not in vertex\_count:

vertex\_count[v] = 0

vertex\_count[u] += 1

vertex\_count[v] += 1

for count in vertex\_count.values():

if count > 1:

return False

return True

edges = []

for \_ in range(q):

query = input().split()

op = query[0]

u, v, c = int(query[1]), int(query[2]), int(query[3])

if op == '+':

edges.append((u, v, c))

else:

edges.remove((u, v, c))

total\_beauty = 0

for color in set(c for \_, \_, c in edges):

if is\_excellent(color):

beauty = sum(1 for u1, v1, c1 in edges if c1 == color)

total\_beauty += beauty

print(total\_beauty)

Бұл код қарапайым тәсілді қолданады. Ол шеттердің (шеттердің) тізімін жүргізеді және is\_excellent функциясын қолданып, түстің керемет екендігін тексереді (әр шың сол түстің ең көп дегенде бір жиегімен байланысқан). Is\_excellent функциясы берілген түс үшін әрбір шыңның пайда болуын есептейді. Егер қандай да бір шың бірнеше рет пайда болса, түсі керемет емес. Әрбір сұраудан кейін код бірегей түстер арқылы қайталанады, олардың тамаша екенін тексереді және барлық тамаша түстердің сұлулығын қорытындылау арқылы жалпы сұлулықты есептейді. Жиын қайталанбай бірегей түстерді тиімді алу үшін пайдаланылады. Тиімділікті арттыру: Деректер Құрылымы: көптеген шыңдар мен шеттер үшін қарапайым тізімдерге қарағанда тиімдірек деректер құрылымдарын пайдалану өнімділікті айтарлықтай жақсарта алады. Графикті көрсету үшін көршілес тізімдерді немесе жиындарды пайдалануды қарастырыңыз. Бұл түстің тамаша екенін тексерудің уақыт күрделілігін төмендетуі мүмкін. Оңтайландыру: is\_excellent функциясын redu болдырмау үшін оңтайландыруға болады

n = int(input())

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

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

cards = list(zip(a, b))

def check\_order(arr):

for i in range(n - 1):

if arr[i+1] < arr[i]:

return False

return True

import itertools

for perm in itertools.permutations(cards):

top = [x[0] for x in perm]

bottom = [x[1] for x in perm]

if check\_order(top) and check\_order(bottom):

print(\*top)

print(\*bottom)

exit()

top\_reversed = [x[1] for x in perm]

bottom\_reversed = [x[0] for x in perm]

if check\_order(top\_reversed) and check\_order(bottom\_reversed):

print(\*top\_reversed)

print(\*bottom\_reversed)

exit()

Бұл код итераторларды қолданады.ауыстырулар карталардың барлық ықтимал тапсырыстарын жасау үшін жұмыс істейді. Әрбір ауыстыру үшін check\_order функциясын қолдана отырып, үстіңгі және астыңғы жақтарының өсу ретімен орналасқандығын тексереді. Егер жарамды орналасу табылса, ол жоғарғы және төменгі жылдарды басып шығарады және шығады. Әйтпесе, ол барлық ауыстыруларды тексеруді жалғастырады. Егер барлық ауыстыруларды тексергеннен кейін жарамды орналасу табылмаса, ол -1 басып шығарады.

Тиімділікті Ескеру:

Бұл шешімнің уақыт күрделілігі O (n! \* n), мұндағы n-карталар саны. Бұл itertools - қа байланысты.ауыстырулар n тудырады! ауыстырулар және әрбір ауыстырудың ретін тексеру O (n) уақытты алады. Үлкен мәндері үшін n, бұл тәсіл ауыстыру санының факторлық өсуіне байланысты есептеу мүмкін болмай қалады. Неғұрлым тиімді алгоритм (мүмкін басқа тәсілді, мүмкін эвристиканы немесе жуықтау әдістерін қамтуы мүмкін) 1 секундтық уақыт шегінде үлкенірек енгізу өлшемдерін өңдеу үшін қажет.

def get\_pos(x, y):

return chr(ord('a') + x) + str(y + 1)

def move(dx, dy):

print(f"? {dx} {dy}")

return input()

def solve():

x = 0

y = 0

# Find boundaries

right\_bound = 0

while True:

res = move(right\_bound + 1, right\_bound + 1)

if res == '-':

break

right\_bound += 1

left\_bound = 0

while True:

res = move(-(left\_bound + 1), -(left\_bound + 1))

if res == '-':

break

left\_bound += 1

up\_bound = 0

while True:

res = move(up\_bound + 1, up\_bound + 1)

if res == '-':

break

up\_bound += 1

down\_bound = 0

while True:

res = move(-(down\_bound + 1), -(down\_bound + 1))

if res == '-':

break

down\_bound +=1

x = (right\_bound - left\_bound) // 2

y = (up\_bound - down\_bound) // 2

print(f"! {get\_pos(x, y)}")

solve()

Бұл код шекарада екілік іздеу тәсілін қолдана отырып, епископтың бастапқы орнын жүйелі түрде анықтайды. Шекараларды табыңыз: Ол епископты тақтаның шетіне тигенше белгілі бір диагональды бағытта (+1, +1) жылжытудан басталады. Содан кейін ол мұны қарама-қарсы диагональдар мен шекараларды табу бағыты үшін қайталайды. Позицияны есептеңіз: содан кейін код шекаралар арасындағы айырмашылықтарға сүйене отырып, бастапқы позицияны (x, y) есептейді. Шығару Нәтижесі: Соңында, ол координаттарды шахмат белгісіне түрлендіретін get\_pos функциясын қолдана отырып, бастапқы позицияны басып шығарады. Неліктен бұл тәсіл жұмыс істейді: епископтың қозғалысы диагональдармен шектеледі. Бір бағытта бірнеше рет қозғалу және шекаралардың бар-жоғын тексеру арқылы біз епископтың орналасқан диапазонын анықтай аламыз. Бұл диапазонның ортаңғы нүктесі бастапқы координаттарды береді. Маңызды Ескертпелер: Қателерді Өңдеу: код енгізу мәселенің сипаттамасына сәйкес келеді деп болжайды. Қателерді сенімді түрде өңдеу келесідей болуы мүмкін

Задача l. Бұл мәселе өте күрделі және уақыт шектеулері шегінде барлық n үшін жұмыс істейтін қарапайым, жалпы шешімі жоқ. Ерікті n үшін шешім табу күрделі алгоритмдерді және көптеген алдын ала есептелген деректерді немесе эвристиканы қажет етеді. Дөрекі күш қолдану тәсілі тіпті орташа өлшемдер үшін де есептеу мүмкін емес n мүмкін болатын келісімдердің көптігіне байланысты. Міне, сондықтан бұл қиын және қандай тәсілдерді қарастыруға болады (бірақ толық енгізу қарапайым жауаптан асып түседі).: Комбинаторлық Жарылыс: n x n торындағы n-оминолардың ықтимал орналасуларының саны n-мен бірге өте тез өседі. Бұл өрескел күштермен барлауды мүмкін емес етеді. Полиомино Плиткалары: мәселе торды полиоминолармен (бірнеше квадраттардан жасалған пішіндер) плиткамен қаптаудың белгілі мәселесімен тығыз байланысты. Бұл жалпы есептеу қиын мәселе. Ерікті n үшін барлық плиткаларды табудың белгілі тиімді алгоритмі жоқ.

Задача M.

x1, y1, x2, y2, x3, y3 = map(int, input().split())

#Find the centroid

centroid\_x = (x1 + x2 + x3) / 3

centroid\_y = (y1 + y2 + y3) / 3

#Handle colinear points

if (y2 - y1) \* (x3 - x2) == (y3 - y2) \* (x2 - x1):

print(x1,y1)

print(x2,y2)

exit()

#Қашықтықтардың қосындысын минимизациялайтын сызықты табыңыз (Ферма нүктесін пайдаланып)

# Қарапайымдылық үшін центроидты түзудің нүктесі ретінде қолданайық

#Центроидты 90 градусқа бұру арқылы түзудің екінші нүктесін табыңыз

#Бұл сызықтың центроидтан кез келген нүктеге дейінгі векторға перпендикуляр болуын қамтамасыз етеді.

перпендикуляр\_х = центроид\_й-у1

перпендикуляр\_й = х1-центроид\_х

second\_point\_x = int (центроид\_х + перпендикуляр\_х)

second\_point\_y = int (centroid\_y + перпендикуляр\_y)

басып шығару (int (centroid\_x), int (centroid\_y))

басып шығару (second\_point\_x, second\_point\_y)

Бұл код үш үйден қашықтықтың қосындысын минимизациялайтын сызықты анықтайтын екі нүктенің координаталарын тиімді табады. Шешім келесі логиканы қолданады:

Центроид: үш нүктенің центроидтары есептеледі. Қашықтықтардың қосындысын минимизациялайтын сызық центроид арқылы өтеді.

Ортогональды Сызық: түзудің екінші нүктесін анықтау үшін центроидтан перпендикуляр түзуді есептейді. Қарапайым тәсіл-центроидтан берілген нүктелердің біріне перпендикуляр түзу құратын нүктені қолдану.

Шығару: центроид пен екінші нүктенің координаталары басылады.

Маңызды Ескерту: бұл шешім қашықтықтардың қосындысын минимизациялайтын сызықты табады. Бұл шешім ақылға қонымды шешімді қамтамасыз еткенімен, оның барлық ықтимал кірістер үшін абсолютті минимумды қамтамасыз ететініне кепілдік жоқ. Қатаң шешім күрделі геометриялық алгоритмдерді қажет етуі мүмкін. Дегенмен, бұл әдіс тиімді, түсінуге және жүзеге асыруға оңай және қанағаттанарлық нәтиже береді.