# 一次元累積和

def Ruiseki(nums):

"""リストnumsの累積和を計算したリストcum\_sumを返す"""

tmp\_sum = 0

cum\_sum = [0]

for i in nums:

tmp\_sum += i

cum\_sum.append(tmp\_sum)

return cum\_sum

nums = [2, 3 ,41, 6, 0, -2, 7, 0]

cum\_sum = Ruiseki(nums)

# [0, 2, 5, 46, 52, 52, 50, 57, 57]

# numsの3番目から7番目までの区間[41, 6, 0, -2, 7]の和は

# cum\_sum[7] - cum\_sum[2]

# 二次元累積和 # 鉄則本p63の図が分かりやすい。

def YokoRuiseki(nums):

"""一次元リストnumsの横方向累積和を計算したリストcum\_sumを返す"""

tmp\_sum = 0

cum\_sum = [0]

for i in nums:

tmp\_sum += i

cum\_sum.append(tmp\_sum)

return cum\_sum

def TateRuiseki(field, H, W): # H, W は横累積処理をする前の元々の状態のH, W

"""二次元リストfieldの縦方向累積和を計算した二次元リストfieldを返す"""

field = [[0 for i in range(W + 1)]] + field

for col in range(W + 1):

tmp\_cum = 0

for row in range(H + 1):

tmp\_cum += field[row][col]

field[row][col] = tmp\_cum

return field

def calc\_area\_sum(leftup, rightdown, R):

a, b = leftup

c, d = rightdown

return R[c + 1][d + 1] - R[a][d + 1] - R[c + 1][b] + R[a][b]

H = 4

W = 6

field = [

[4,2,3,6,5,8],

[9,1,1,2,4,1],

[5,0,2,1,0,7],

[6,8,3,2,9,1]

]

field\_after\_yokorui = [YokoRuiseki(row) for row in field]

field\_R = TateRuiseki(field\_after\_yokorui, H, W)

leftup = (1,2) #元の長方形の左上座標の(行, 列)の「index」

rightdown = (2, 4) #元の長方形の右下座標の(行, 列)の「index」

ans = calc\_area\_sum(leftup, rightdown, field\_R)

print(ans)

#[1, 2, 4]

#[2, 1, 0] の部分。（元々のfieldの）

# 10

# 約数列挙

def div\_enu(N):

"""自然数Nの約数を列挙した集合を返す"""

div\_set = set()

for div in range(1, int(N \*\* (0.5) + 1)):

if N % div == 0:

div\_set.add(div)

div\_set.add(N // div)

return div\_set

# 素数判定 (試し割り法で, √Nまでの数で割っていき, 1以外の約数が無ければ素数)

def is\_prime(n):

"""自然数nが素数ならTrue"""

if n == 1:

return False

for div in range(2, int(n \*\* (0.5)) + 1):

if n % div == 0:

return False

else:

return True

# 素数列挙【エラトステネスの篩】

def Sieve\_of\_Eratosthenes(N):

""" N以下(N <= 2)の素数を列挙したリストを返す(鉄則本p158)"""

# 2以上N以下の整数を全て書いてみる (先頭の0と1はダミー的存在)

field = [True for i in range(N + 1)]

# baseにマルを付け, 「それ以外の」baseの倍数を消す (base自身は消さないように注意)

for base in range(2, int(N \*\* 0.5) + 1): # baseは√Nまででよい

if not field[base]: # baseが既に消されていたらcontinue

continue

for del\_num in range(base \* 2, N + 1, base): # baseの倍数を消す (base自身は消さないように注意)

field[del\_num] = False

# Trueなら対応する数字を入れる

prime\_nums = [i for i in range(2, N + 1) if field[i]]

return prime\_nums

# 素因数分解

def factorization(N):

""" 自然数N(>=2)を素因数分解した結果の, 素因子が入ったリストを返す O(√N)""”

factor = []

for div in range(2, int(N \*\* (0.5)) + 1):

while N % div == 0:

N //= div

factor.append(div)

if N != 1:

factor.append(N)

return factor

#print(factorization(12))

# [2, 2, 3]

# bit全探索

def bit\_allsearch(N):

""" bit全探索。 文字列が入ったリストを返す"""

bin\_list = []

for i in range(2 \*\* N):

tmp = bin(i)[2:] # bin()で2進数(文字列)に変換後, 0b以降だけ取る。

bin\_list.append("0" \* (N - len(tmp)) + tmp) # 先頭に0を追加して, 桁数をNに合わせる

return bin\_list

# bit\_allsearch(3)

# ['000', '001', '010', '011', '100', '101', '110', '111'] 2 \*\* 3 = 8通り

# 90度回転

def Turn\_90(A, H, W):

"""二次元配列A(H行W列)を時計回りに90度回転させる"""

after\_H = W # 回転後の配列の高さ

after\_W = H # 回転後の配列の横幅

A\_after\_turn = [["" for col in range(after\_W)] for row in range(after\_H)]

for row in range(H):

for col in range(W):

A\_after\_turn[col][H - row - 1] = A[row][col]

return A\_after\_turn

# 優先度付きキュー (ヒープ)

import heapq

a = [4,6,5,3,2]

heapq.heapify(a) #再代入の必要なし。#aは必ずリスト

print(a)

→[2,3,5,4,6]

常に一番左に最小値が来る。ほかの要素はばらばら。

あらかじめマイナスを付しておけば, 最大値も取り出せる。

データ型自体はlist型のままである。

(!!注意!!)

一度ヒープにしたつもりでも,

a.append(1)　とかやってしまうと, 普通に末尾に1が追加されて[2,3,5,4,6,1]となってしまう。

(a.remove(値)など, リストの関数全般にも同じことがいえる。)

heappopの動作もその直後の一回分おかしくなるので,

ヒープの恩恵を得たいときは

【しっかりヒープ用の構造(場合によってはリストと別物の構造)を作って

『heappopとheappushのみ』使うこと！！！】

(どうしてもremoveとかするなら,した後にheapifyし直す)

空リストにheappushしていくなら, heapifyは必要ない。

なるだけヒープは一次元で扱う方が良いと思うが, 二次元にしたい場合,

二次元リストを一気にheapifyすることはできない(エラー)ので,空リストにheappushしていく。

heapq.heappop(list) O(logN) (空リストからheappopするとエラーなので注意)

heapq.heappush(list, elem) O(logN)

list[0] O(1) (削除せずに取得するだけでいいなら)

heapq.heapify(list) O(N) ← 計算量注意

# Pythonで標準装備されていない多重集合(重複する値を保持でき, 順序も意識できる)

# \_\_init\_\_のmax\_queryに注意

# 鉄則本A55

class BinaryTrie: # (https://kanpurin.hatenablog.com/entry/2021/12/22/001854)

def \_\_init\_\_(self, max\_query=2\*10\*\*5, bitlen=30):

n = max\_query \* bitlen

self.nodes = [-1] \* (2 \* n)

self.cnt = [0] \* n

self.id = 0

self.bitlen = bitlen

#全体のサイズ

def size(self):

return self.cnt[0]

# 値xの個数

def count(self,x):

pt = 0

for i in range(self.bitlen-1,-1,-1):

y = x>>i&1

if self.nodes[2\*pt+y] == -1:

return 0

pt = self.nodes[2\*pt+y]

return self.cnt[pt]

# 値xの挿入

def insert(self,x):

pt = 0

for i in range(self.bitlen-1,-1,-1):

y = x>>i&1

if self.nodes[2\*pt+y] == -1:

self.id += 1

self.nodes[2\*pt+y] = self.id

self.cnt[pt] += 1

pt = self.nodes[2\*pt+y]

self.cnt[pt] += 1

# 値xの削除

# 値xが存在しないときは何もしない

def erase(self,x):

if self.count(x) == 0:

return

pt = 0

for i in range(self.bitlen-1,-1,-1):

y = x>>i&1

self.cnt[pt] -= 1

pt = self.nodes[2\*pt+y]

self.cnt[pt] -= 1

# 昇順x番目の値(1-indexed)

def kth\_elm(self,x):

assert 1 <= x <= self.size()

pt, ans = 0, 0

for i in range(self.bitlen-1,-1,-1):

ans <<= 1

if self.nodes[2\*pt] != -1 and self.cnt[self.nodes[2\*pt]] > 0:

if self.cnt[self.nodes[2\*pt]] >= x:

pt = self.nodes[2\*pt]

bt = BinaryTrie()

bt.insert(4)

bt.insert(3)

bt.insert(7)

bt.insert(7)

bt.insert(5)

print(bt.size()) # 5

print(bt.count(7)) # 2

print(bt.kth\_elm(3)) # 5

昇順で並べたときに

3番目(1番目, 2番目...の数え方)になる値は5

print(bt.lower\_bound(1)) # 1

値1以上の最小要素は,値3になり,

値3は昇順で1番目(1番目, 2番目...の数え方)。

else:

x -= self.cnt[self.nodes[2\*pt]]

pt = self.nodes[2\*pt+1]

ans += 1

else:

pt = self.nodes[2\*pt+1]

ans += 1

return ans

# 値x以上の最小要素が昇順何番目か(1-indexed)

# 値x以上の要素がない時はsize+1を返す

def lower\_bound(self,x):

pt, ans = 0, 1

for i in range(self.bitlen-1,-1,-1):

if pt == -1: break

if x>>i&1 and self.nodes[2\*pt] != -1:

ans += self.cnt[self.nodes[2\*pt]]

pt = self.nodes[2\*pt+(x>>i&1)]

return ans

# ソートで第一キーを昇順に, 第二キーを降順にしたい場合

a = [

[1, 80],

[1, 90],

[1, 20],

[2, 100],

]

Sorted(a, key = lambda x:(x[0], -x[1]))

# [[1, 90], [1, 80], [1, 20], [2, 100]]