```
# 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 までによい
        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]
            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
```

```
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)。
```

```
# 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]]
```