1. Competition

```
# -*- coding: UTF-8 -*-
# description: 在一场比赛中，有n个检查点，比赛的要求是到达n-1个检查点即可，这些检查点排列在x
轴上，位置分别为x1,x2,...,xn,且允许以任意
#               顺序访问检查点，比赛的开始位置为a,现在问完成比赛所需经过的最小距离是多少。

# example: input:3 10  (输入包含两行，第一行为两个参数n,a,其中1<=n<=1e5,-1e6<=a<=1e6)
#                 1 7 12  (第二行为n个整数，表示x1,x2,...,xn, 且-1e7<=xi<=1e6)
#
#           output:7    (输出一个整数表示问题的解)


"""
@param string line 一个测试用例
@return string 处理后的结果
"""


def solution(line):
    points = [int(x) for y in line.split('\n') for x in y.strip().split()]
    n = points.pop(0)
    a = points.pop(0)
    if a <= points[0]:
        return points[-2] - a
    elif a >= points[-1]:
        return a - points[1]
    else:  # 排除上面两种情况，说明起码有两个点，而a在两者间
        tmp_1 = min(a - points[0] + points[-2] - points[0], abs(points[-2] - a)
+ points[-2] - points[0])
        tmp_2 = min(abs(a - points[1]) + points[-1] - points[1], points[-1] - a
+ points[-1] - points[1])
        return min(tmp_1, tmp_2)


test = "3 10\n 1 7 12"
print(solution(test))


# 一共有n个点，设从小到大排序，x1到xn。由于要访问n-1个点，即抛弃一个点不访问。
# 设抛弃的点在中间，即2到n-1,由于要访问1和n,这些中间的点可以顺便访问，而不增加负担，
# 反而可以访问中间的点而不访问最小点或者最大点来减小负担。所以不访问的那个点要么是1,要么是n。而
这依赖于a的位置。

# 假设a小于等于点1，则不访问的点是n。假设a大于等于点n,则不访问的点是1。这些都是平凡的情况。
# 设a在点1和点n之间，不失一般性，设不访问的点是n,则要访问点1到点n-1。
# 则最佳的访问的候选情况只有两种，一种是先向左访问到1，再向右访问到n-1，另外一种则恰好相反。
# 因为如果在中间某点转向，则有一段距离会被重复三次。一定是在点1或者点n-1处转向。
# 这两种访问中，前者重复了一遍a到点1，后者重复了一遍a到点n，比较则可知道最佳的访问。
# 则综上有: a<=x1时: d=X_(n-1) - a
#           a>=xn时: d= a - x2
#           x1<a<xn时: d在以下情况选最小的:
#                     1.不访问点n,则为 min{ a-x1+x(n-1)-x1, |x(n-1)-a|+x(n-1)-x1 }
#                     2.不访问点1,则为 min{ |a-x2|+xn-x2, xn-a+xn-x2}
```

2. pagoda

```
# -*- coding: UTF-8 -*-
# description: 小Q正在攀登一座宝塔，塔总共有n层，但是每两层之间的净高却不相同,所以造成了小Q爬过每层的时间也不同，如果某一层的高度为x,
#               那么爬过这一层所需的时间也是x。此外，小Q还会使用一种魔法，每用一次就可以让他向上跳一层或两层。但是每次跳完后，小Q都会将魔法力
#               用完，必须爬过至少一层才能再次跳跃，可以认为小Q需要跳两次一层才休息,最后也可以跳到塔外即超过塔高，跳是不消耗时间的。
#               现在，小Q想用最短的时间爬到塔顶，这个最短时间是多少?

# example: input:5          (输入的第一行是一个数n,n表示塔的层数，n<=1e4)
#                 3 5 1 8 4  (第二行为n个整数，h1,h2,...,hn,为从下往上每层的高度，1<=hi<=100)
#
#           output:1         (输出一个数表示最短时间)


"""
@param string line 一个测试用例
@return string 处理后的结果
"""


def solution(line):
    heights = [int(x) for y in line.split('\n') for x in y.strip().split()]
    num_layers = heights[0]
    if num_layers <= 2:
        return 0
    dp = [[0 for i in range(3)] for j in range(num_layers + 1)]
    dp[2] = [0, heights[1], heights[2]]
    dp[3] = [heights[1], heights[2], heights[3]]
    for i in range(4, num_layers + 1):  # 注意dp和heights从1开始计数,0处填充的是其他数据
        dp[i][0] = min(dp[i - 1][1], dp[i - 2][2])
        dp[i][1] = min(dp[i - 1][2], dp[i - 2][1] + heights[i - 1], dp[i - 2][0] + heights[i - 1])
        dp[i][2] = min(dp[i - 1][0] + heights[i], dp[i - 2][1] + heights[i])
    return min(dp[num_layers])


test = "5\n 3 5 18 8 4"
print(solution(test))

# 只要出现了爬的状态，则魔力变为2。最开始的状态是dp[0][2]=0
# 在n层的状态由以下构成{dp[n-1][2],   dp[n-1][1],   dp[n-1][0]+height[n],
#                     dp[n-2][2],   dp[n-2][1]+height[n],   dp[n-2][1]+height[n-1],   dp[n-2][0]+height[n-1]}
# 可以分成以下三组:
# dp[n][0]= min{dp[n-1][1],  dp[n-2][2]}
# dp[n][1]= min{dp[n-1][2],  dp[n-2][1]+height[n-1], dp[n-2][0]+height[n-1]}
# dp[n][2]= min{dp[n-1][0]+height[n],    dp[n-2][1]+height[n]}
#
# 那最短的时间就是 min{dp[n][0],dp[n][1],dp[n][2]}
```

3. card

```
# -*- coding: UTF-8 -*-
# description: 小Q有一叠纸牌,一共有n张,从上往下依次编号为1到n。现在小Q要对这叠纸牌反复做以下操作:
#                   把当前位于顶端的牌扔掉,然后把新的顶端的牌放到新叠牌的底部。
#              小Q会一直操作到只剩下一张牌为止。小Q想知道每次丢掉的牌的编号。

# example: input:7          (输入为一行,只有一个数字n, 1<=n<=1e6)
#
#          output:1 3 5 7 4 2 6         (输出n个用空格间隔的整数,表示每次丢掉的牌的编号)



"""
@param string line 一个测试用例
@return string 处理后的结果
"""



def solution(line):
    cards = [str(x + 1) for x in range(int(line))]
    res = []
    while len(cards) > 2:
        res += [cards.pop(0)]
        cards.append(cards.pop(0))
    return " ".join(res + cards)


print(solution(8))
```

4. checkerboard

```
# -*- coding: UTF-8 -*-
# description: 妞妞公主有一块黑白棋盘,该棋盘共有n行m列,任意相邻的两个格子都是不同的颜色(黑或白),坐标为(1,1)的格子是白色的。
#              这一天牛牛来看妞妞公主,和妞妞公主说:只要你告诉我n和m,我就能马上算出黑色方块和白色方块的数量
#              妞妞公主说:这太简单了。这样吧,我在这n行m列中选择一个左下角坐标为(x0,y0),右上角坐标为(x1,y1)的矩形,把这个矩形里的共
#                       (x1-x0+1)*(y1-y0+1)个方块全部涂白,你还能算出黑色方块和白色方块的数量吗?
#              牛牛说:so easy,你可以在执行涂白操作后再选一个左下角坐标为(x2,y2),右上角坐标为(x3,y3)的矩形,把这个矩形里面的方块
#                       全部涂黑,我仍然能够马上算出黑色方块和白色方块的数量。
#              妞妞公主表示不相信,开始提问了T次。请问你能帮牛牛算出每次提问时棋盘的黑白方块数量吗?

# example: input:3  (输入一共包含3*T+1行,第一行输入表示T,表示提问的次数,接下来的3*T行表示T次提问,这里是3次提问)
#                1 3      (第(1+3*i)行是两个整数n,m, 表示第i次提问时棋盘的大小)
```

```
#               1 1 1 3  (第(2+3*i)行是四个整数x0,y0,x1,y1, 表示第i次提问时涂白操作时选取的两个坐标)
#               1 1 1 3  (第(3+3*i)行是四个整数x2,y2,x3,y3, 表示第i次提问时涂黑操作时选取的两个坐标)
#               3 3
#               1 1 2 3
#               2 1 3 3
#               3 4
#               2 1 2 4
#               1 2 3 3
#              (数据限制: 1<=T<=1e4, 1<=x<=n<=1e9, 1<=y<=m<=1e9, x0<=x1, y0<=y1, x2<=x3, y2<=y3)
#
#          output:0 3    (输出T行，每行两个整数表示白色方块的数量和黑色方块的数量)
#                 3 6
#                 4 8

"""
@param string line 一个测试用例
@return string 处理后的结果
"""


def solution(line):
    def square_num(i1, j1, i2, j2):
        total = (i2 - i1 + 1) * (j2 - j1 + 1)
        if total & 1:  # 总的格子数是奇数
            if (i1 + j1) & 1:  # 奇数说明左下角为黑色
                return total // 2 + 1, total // 2  # 返回黑色格和白色格子数量
            else:
                return total // 2, total // 2 + 1
        else:
            return total // 2, total // 2

    input_data = [int(x) for y in line.split('\n') for x in y.strip().split()]
    T = input_data.pop(0)
    res = ""
    for times in range(T):
        tmp_matrix = input_data[10 * times:10 * (times + 1)]  # n,m,加上八个坐标值，一共10个元素
        n, m = tmp_matrix[:2]
        x0, y0, x1, y1 = tmp_matrix[2:6]
        x2, y2, x3, y3 = tmp_matrix[6:]
        x4, y4, x5, y5 = max(x0, x2), max(y0, y2), min(x1, x3), min(y1, y3)
        t1 = square_num(1, 1, m, n)[1]
        t2 = square_num(x0, y0, x1, y1)[0]
        t3 = square_num(x2, y2, x3, y3)[1]
        t4 = square_num(x4, y4, x5, y5)[0] if x5 >= x4 and y5 >= y4 else 0
        white_num = t1 - t3 + (t2 - t4)
        black_num = n * m - white_num
        res += str(white_num) + " " + str(black_num) + "\n"
    return res


test1 = "3\n1 3\n1 1 1 3\n1 1 1 3\n3 3\n1 1 2 3\n2 1 3 3\n3 4\n2 1 2 4\n1 2 3 3"
print(solution(test1))
# output:
# 0 3
```

```
# 3 6
# 4 8
test2 = "5\n2 2\n1 1 2 2\n1 1 2 2\n3 4\n2 2 3 2\n3 1 4 3\n1 5\n1 1 5 1\n3 1 5
1\n4 4\n1 1 4 2\n1 3 4 4\n3 4\n1 2 4 2\n1 3 4 4\n3 4\n1 2 4 2\n2 1 3 3"
print(solution(test2))
# output:
# 0 4
# 3 9
# 2 3
# 8 8
# 4 8


# 总的格子数量是不变的，就是n*m，除了白色就是黑色，求得黑色方格就能知道白色方格，反之亦然。而黑白的地位是等价的，不妨求白格数量。
# 设所有矩阵从1开始计数，分析最初的矩阵，可以发现两个涂色规律:
# 1. 设某个格子的坐标为(i,j)，则i+j是偶数的话，该位置是白色，i+j是奇数的话，该位置是黑色。可以画斜率为1的直线，很容易看出。
# 2. 不妨设一个矩阵的左下角格子是白色，若矩阵的格子总数为偶数，那么该矩阵内黑白格子数相同。
#                                 若矩阵的格子总数为奇数，那么该矩阵内白色格子比黑色格子多一个。
#    由于黑白地位等价，以上命题黑白可以互换，也是成立的。
# 所以根据以上两个观察，可以得到每次操作后黑白格子的数量。

# 设最初的黑白矩阵为A,被涂白的矩阵为B,被涂黑的矩阵为C。重点是B和C的交叉区域，即两者重叠的地方，不妨设为D。
# A,B,C区域已给出，而B与C重叠的区域D则为: 左下角为(max(x0,x2), max(y0,y2))， 右上角为(min(x1,x3), min(y1,y3))。
# 要求白色格子的数量，就先求出A的白色格子数量t1，再求出B区域原来的黑色格子数量t2，再求出C区域原来的白色格子数量t3，再求出D区域原来的黑色格子t4。
# 那么最终白色方块的数量就是t1- t3 + (t2-t4)，即总的白格数量减去涂黑操作影响的白格子，再加上涂白操作里面受到影响的黑格子数量(重叠区域的黑格要减去)。
```

5. sequence

```
# -*- coding: UTF-8 -*-
# description: 小Q得到了一个长度为n的序列A， A中的数各不相同，对于A中的每一个Ai， 求min |Ai-Aj|,其中1<=j<i
#               以及令上式取到最小值的j(记为Pi)。若最小值点不唯一，则选择使Aj较小的那个。

# example: input:3    (输入为两行，第一行是整数n,第二行n个数表示A1到An,用空格隔开，其中n<=1e5,|Ai|<=1e9)
#                1 5 3
#
#          output:4 1 (输出为n-1行，每行2个用空格隔开的整数，分别表示当i取2到n时，对应的min|Ai-Aj|和Pi的值，其中1<=j<i)
#                 2 1

"""
@param string line 一个测试用例
@return string 处理后的结果
"""


def solution(line):
```

```
    A=  [int(x) for y in line.split('\n') for x in y.strip().split()]
    n = A.pop(0)
    res =""
    for i in range(1,n):
        tmp = float('inf')
        tmp_aj,tmp_pi=0,0
        for j in range(i):
            if abs(A[i]-A[j])<tmp:
                tmp = abs(A[i]-A[j])
                tmp_aj = A[j]
                tmp_pi = j
            elif abs(A[i] - A[j])==tmp and tmp_aj>A[j]:
                tmp_aj =A[j]
                tmp_pi = j
        res += str(tmp)+" "+str(tmp_pi+1)+"\n"
    return res

test="3\n1 5 3"
print(solution(test))
```