Name: NAGAVELLI VARUN

HTNO:2403A52036

Batch no:03

Sub-Group:O

O.1 — [S14O1] Point-in-polygon (ray casting)

Your Task:

Implement ray-casting point-in-polygon; treat points on edges as inside.

Code:

```
def point_on_segment(px, py, x1, y1, x2, y2):
    """Check if point (px,py) lies on segment (x1,y1)-(x2,y2)."""
    if cross != 0:
        return False

    if min(x1, x2) <= px <= max(x1, x2) and min(y1, y2) <= py <= max(y1, y2):
        return True
    return False


def point_in_polygon(px, py, poly):
    """Ray-casting point-in-polygon with edge inclusion."""
    n = len(poly)
    inside = False


    for i in range(n):
        x1, y1 = poly[i]
        x2, y2 = poly[(i + 1) % n]
        if point_on_segment(px, py, x1, y1, x2, y2):
            return True
```

```python
    for i in range(n):
        x1, y1 = poly[i]
        x2, y2 = poly[(i + 1) % n]

        if y1 > y2:
            x1, x2 = x2, x1
            y1, y2 = y2, y1
        if y1 <= py < y2:
            x_inters = x1 + (py - y1) * (x2 - x1) / (y2 - y1)
            if px < x_inters:
                inside = not inside
    return inside


def points_in_polygon(poly, pts):
    """Return boolean list for each point wrt polygon."""
    return [point_in_polygon(px, py, poly) for (px, py) in pts]

poly = [(0,0),(4,0),(4,4),(0,4)]
pts = [(2,2),(5,5)]

print(points_in_polygon(poly, pts))
```

OUTPUT:

[True,False]

OBSERAVATION:

*** Correctly includes edge/vertex points using point_on_segment.**

*** Ray-casting logic is implemented properly (avoids double-counting edges).**

*** Works for both convex and concave polygons.**

*** Complexity is optimal (O(n)).**

*** Only improvement: add floating-point tolerance for numerical stability.**

O.2 — [S14O2] Compute rolling median (w=3)

Your Task:

Return the median for each sliding window; prefer an efficient approach.

Code:

```python
import bisect

def rolling_median(nums, w=3):
    """Compute rolling medians with window size w."""
    if w <= 0:
        raise ValueError("Window size must be positive")
    if len(nums) < w:
        return []

    window = sorted(nums[:w])
    medians = []

    for i in range(w, len(nums) + 1):

        if w % 2 == 1:
            medians.append(window[w // 2])
        else:
            medians.append((window[w // 2 - 1] + window[w // 2]) / 2)

        if i == len(nums):
            break
```

```python
        out_num, in_num = nums[i - w], nums[i]
        window.pop(bisect.bisect_left(window, out_num))
        bisect.insort(window, in_num)

    return medians


print(rolling_median([1, 3, 2, 5, 4], 3))
print(rolling_median([1, 2, 3], 3))
print(rolling_median([5, 4], 3))
print(rolling_median([], 3))
print(rolling_median([10, 20, 30, 40], 2))
```

OUTPUT:

[2, 3, 4]

[2]

[]

[]

[15.0, 25.0, 35.0]

OBSERVATION:

*** Correctly computes sliding medians using a sorted window with bisect (O(n log w)).**

*** Handles odd and even window sizes.**

*** Covers edge cases: empty list, list shorter than w, exact-size list.**

*** Produces expected outputs on test cases ([2, 3, 4] for sample).**

*** For very large n, a two-heaps approach may be more scalable, but this solution is clean and efficient enough for typical telecom monitoring workloads.**