Daily Coding Problem #150

Problem

This problem was asked by LinkedIn.

Given a list of points, a central point, and an integer k, find the nearest k points from the central point.

For example, given the list of points `[(0, 0), (5, 4), (3, 1)]`, the central point `(1, 2)`, and k = 2, return `[(0, 0), (3, 1)]`.

Solution

One strategy to solve this would use a max-heap or a priority queue. We store the first k elements in the queue with their distances from the center point as their priority. After that, we go through the rest of the points and compare their distances with the max priority of the queue/heap. If the current point is closer, then pop that max element from the queue/heap and add in the current one.

```
import math

def nearest_k_points(points, center, k):
    if len(points) <= k:
        return points

    pq = PriorityQueue()

    for point in points:
        if len(pq) < k:
            pq.push(point, distance(point, center))
        else:
            dist = distance(point, center)
            if dist < pq.peek()[1]:
                pq.pop()
                pq.push(point, dist)

    return [pq.pop()[0] for i in range(k)]

def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
```

Assuming the priority queue uses a heap, this will take O(n log k) time and O(k) space.

An implementation of a priority queue using a heap follows:

```
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, ele, priority):
        self.heap.append((priority, ele))
        self._bubble_up(len(self.heap) - 1)

    def pop(self):
        if self.is_empty():
            raise IndexError("get from empty heap")

        self._swap(0, len(self.heap) - 1)
        priority, ele = self._pop()
        self._bubble_down(0)

        return (ele, priority)

    def peek(self):
        priority, ele = self.heap[0]
        return (ele, priority)

    def is_empty(self):
        return not self.heap

    def _bubble_down(self, ind):
        length = len(self.heap)
        heap = self.heap

        while True:
            lc, rc = self._left_child(ind), self._right_child(ind)
            if lc >= length and rc >= length:
                break
```

```python
            elif lc >= length:
                replace = rc
            elif rc >= length:
                replace = lc
            else:
                replace = min(lc, rc, key=lambda i: self.heap[i])

            if heap[replace] > heap[ind]:
                self._swap(ind, replace)
                ind = replace
            else:
                break

    def _bubble_up(self, ind):
        par = self._parent(ind)
        heap = self.heap

        while par >= 0:
            if heap[ind] > heap[par]:
                self._swap(ind, par)
                ind = par
                par = self._parent(ind)
            else:
                break

    def _parent(self, ind):
        return (ind - 1) // 2

    def _left_child(self, ind):
        return (ind * 2) + 1

    def _right_child(self, ind):
        return (ind * 2) + 2

    def __len__(self):
        return len(self.heap)

    def _swap(self, i, j):
        _, item0 = self.heap[i]
        _, item1 = self.heap[j]

        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def _pop(self):
        priority, ele = self.heap.pop()
        return (priority, ele)
```