

June 7, 2016

```
In [5]: import math
```

1 Algorithm Design Technique: Divide and Conquer

- divide into subproblems of same type
- recurse
- combine

1.1 Problem (Maxima)

Given a set P of n points in 2D, we say point p dominates q iff $p.x > q.x$ and $p.y > q.y$.

We say point q is *maximal* if $q \in P$ and no point in P dominates q .

Find all maximal points.

```
In [6]: # O(N^2)
# brute force
def find_maximal_1(points):
    big_points = []
    for point1 in points:
        for point2 in points:
            if point1 != point2 and (point1[0] < point2[0] or point1[1] < point2[1]):
                break
        else:
            big_points.append(point1)
    return big_points

# O(N log N)
# couple of ways e.g. convex hull, 2D range tree (?), sort-then-line-sweep
def find_maximal_2(points):
    def f(points):
        n = len(points)
        if n <= 1:
            return points
        ls = f(points[:n/2])
        rs = f(points[n/2:])
        values = [x for x in ls if x[1] > rs[0][1]] + rs
    return values
```

```

        return f(sorted(points))

In [7]: print find_maximal_1([(1, 1), (0, 0), (0, 1), (1, 0)])
        print find_maximal_2([(1, 1), (0, 0), (0, 1), (1, 0)])

[(1, 1)]
[(1, 1)]

```

1.2 Closest Pair

Given a set of n points in 2D, find the pair of points s.t. the Euclidean distance, or $(x - y)^2$, is minimized.

```

In [14]: def dist(px, py):
        return (px[0] - py[0]) ** 2 + (px[1] - py[1]) ** 2

# O(N^2)
# brute force
def find_closest_1(points):
    assert len(points) >= 2
    best = float('inf')
    for i in range(len(points)):
        for j in range(i+1, len(points)):
            if dist(points[i], points[j]) < best:
                best = dist(points[i], points[j])
    return math.sqrt(best)

# O(N log N)
# Shamos' Algorithm
# This is because  $T(n) = 2 T(n/2) + O(n \log n) = O(n^2 \log n)$ 
# But optimally, we don't need to sort within this algorithm, so  $T(n) = 2$ 
def find_closest_2(points):
    sorted_by_ys = sorted(points, key=lambda p:p[1])

    def f(points):
        if len(points) <= 1:
            return float('inf')
        if len(points) == 2:
            return math.sqrt(dist(points[0], points[1]))
        if len(points) == 3:
            return math.sqrt(min(dist(points[0], points[1]), dist(points[0], points[2]), dist(points[1], points[2])))
        x_m = points[len(points)/2][0]
        p_l = [p for p in points if p[0] <= x_m]
        p_r = [p for p in points if p[0] > x_m]
        closest_l = f(p_l)
        closest_r = f(p_r)
        delta = min(closest_l, closest_r)
    return f(sorted_by_ys)

```

```

# first idea: look at pairs within some  $[x - \text{delta}, x + \text{delta}]$  range
within_delta_l = [p for p in p_l if x_m - delta <= p[0]]
within_delta_r = [p for p in p_r if p[0] <= x_m + delta]

# second idea: if we have sliding window upwards of length delta x
# each window contains <= 8 points, because of Pidgeonhole Principle
combined = within_delta_l + within_delta_r
windows = filter(lambda p: p in combined, sorted_by_ys) # "sorting by y"
best = delta
for i in range(len(windows)): #  $O(n)$ 
    for j in range(i, min(len(windows), i+8)): #  $O(1)$ 
        for k in range(j+1, min(len(windows), i+8)): #  $O(1)$ 
            best = min(best, dist(windows[i], windows[k]))
return best

return f(sorted(points))

```

In [15]: print find_closest_1([(0, 0), (1, 1), (2, 2), (3, 3)])

```

print find_closest_2([(0, 0), (1, 1), (2, 2), (3, 3)])

```

1.41421356237

1.41421356237