403: Algorithms and Data Structures

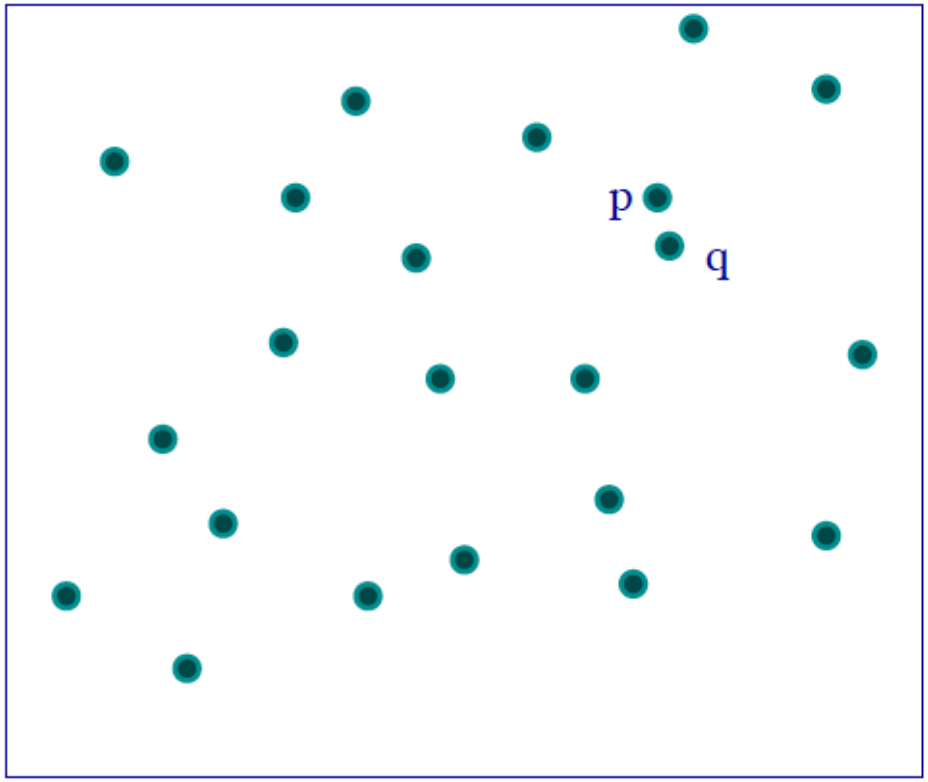
The closest pair in 2D (Programming Assignment)

UAlbany

Computer Science

Closest pair in 2D

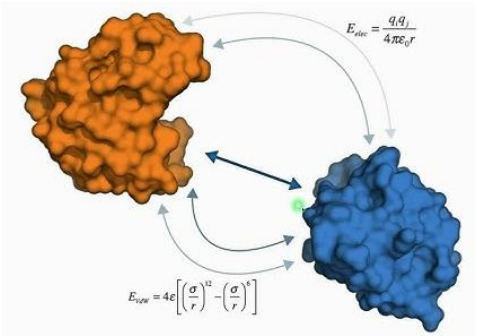
- Given n points in 2-dimensions, find the two whose distance is smallest.
- Euclidean distance



$$d(p,q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$$

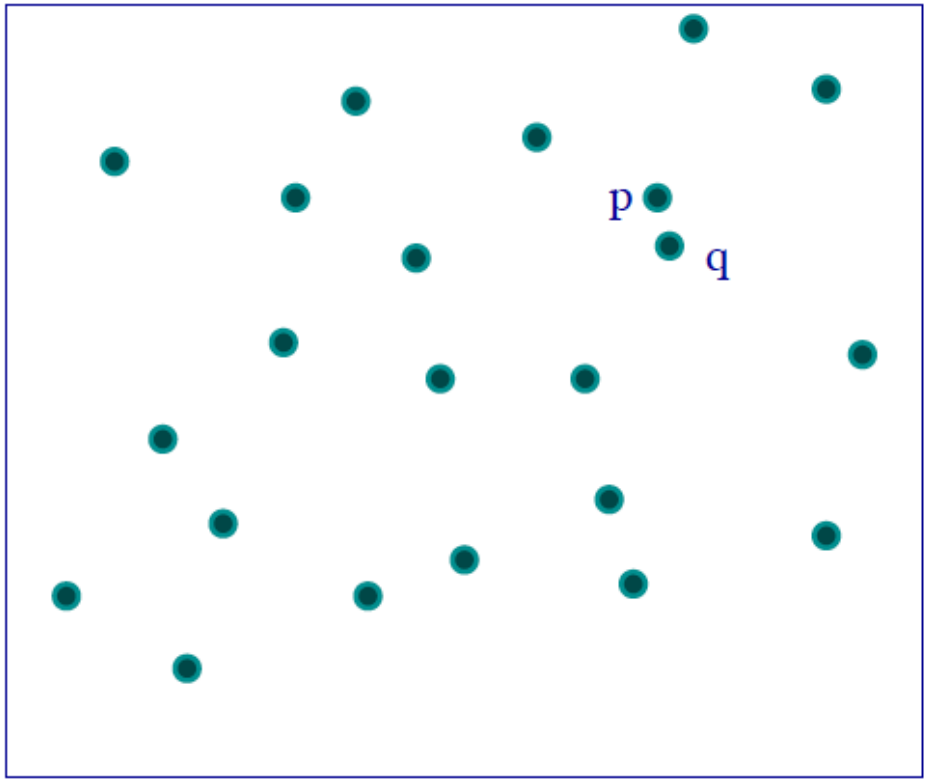
Why is this problem important?

- **Collision detection:** In simulations/games, identifying objects that are closest to each other - potential collisions
- **Air traffic control:** Monitoring aircraft that are getting too close together to prevent mid-air collisions.
- **Molecular docking:** In drug discovery, finding the closest molecules in a protein-ligand interaction to identify potential binding sites.
- **Hierarchical clustering:** Clustering data points by finding the closest pairs and progressively merging them into larger clusters.
- **Image processing:** Identifying nearby pixels with similar color values for image analysis tasks.
- …

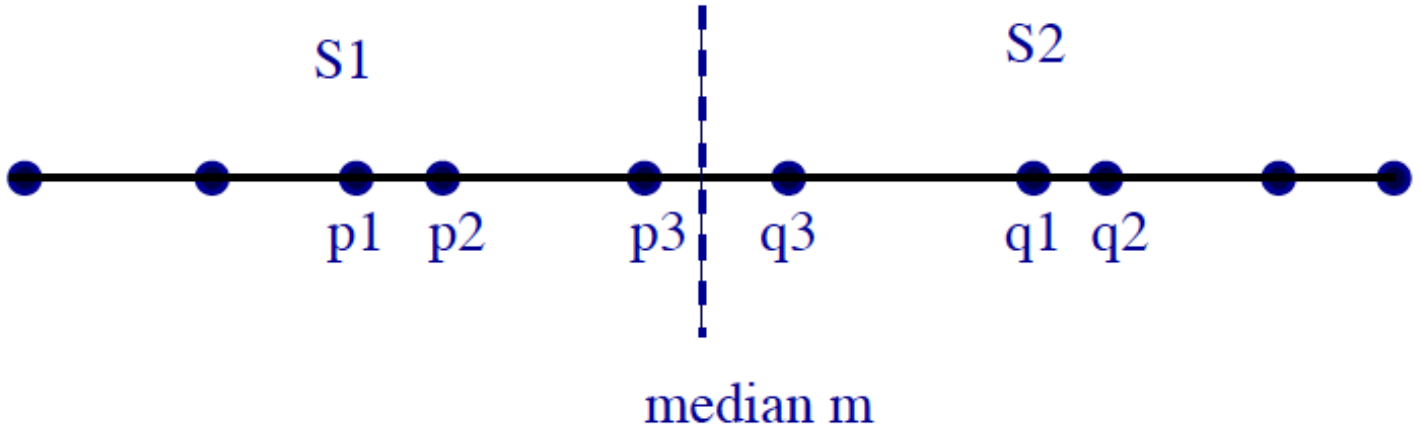


Closest pair in 2D

- Brute force?
  - Consider all pairs
- Complexity?
  - $O(n^2)$

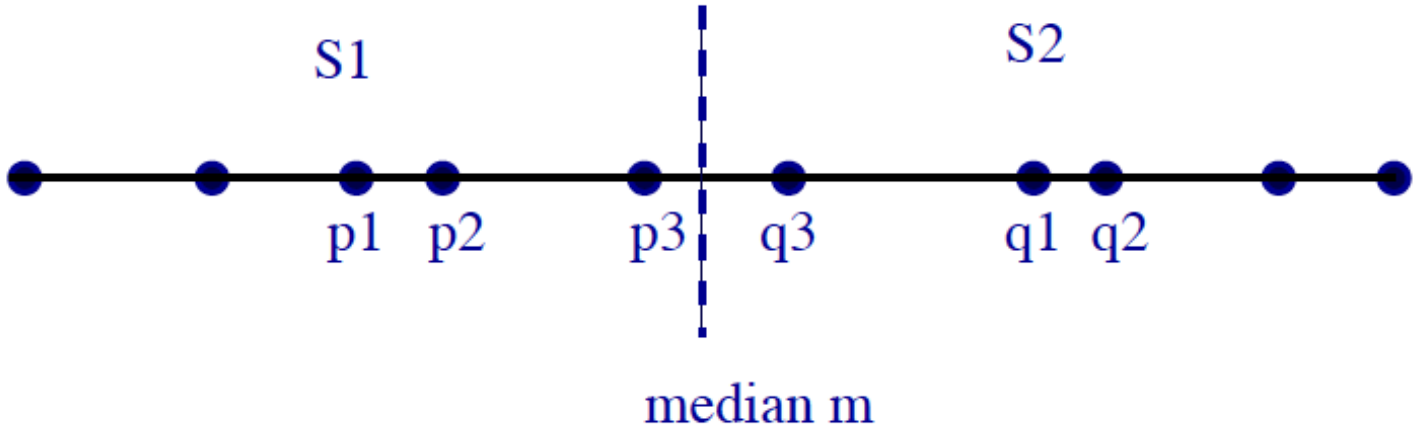


Divide-And-Conquer (1D)



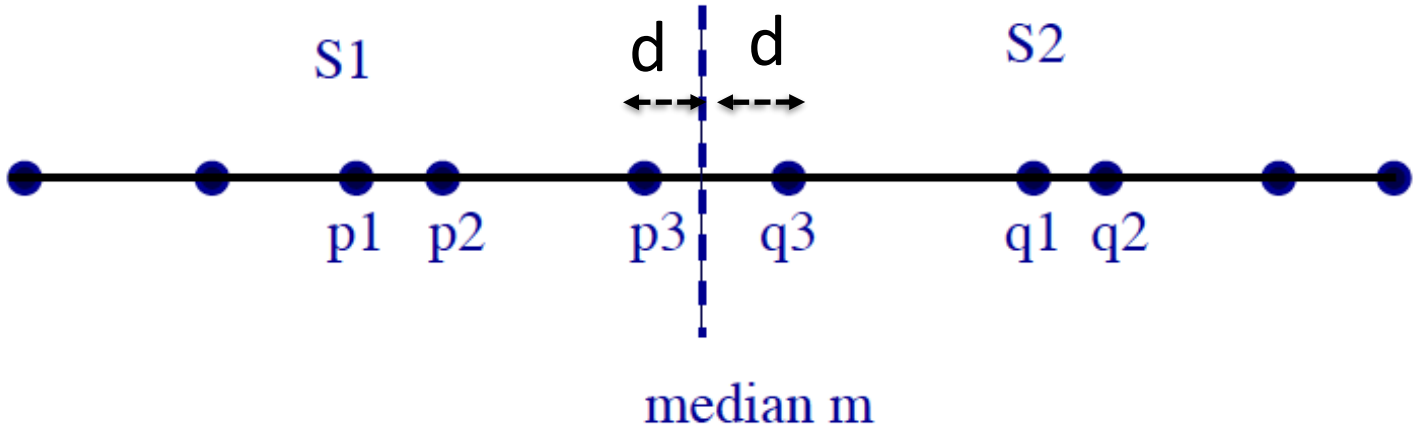
- We can simply sort and consider consecutive pairs O(nlogn)
  - Does not generlize to 2D

Divide-And-Conquer (1D)



- DIVIDE: split array in two equal parts
- CONQUER: recursively find closest pair in parts
- COMBINE:
  - Let **d** be the smallest distance in **S1** and **S2**
  - If **dist(p3,q3)<d** return **dist(p3,q3)** else **d**

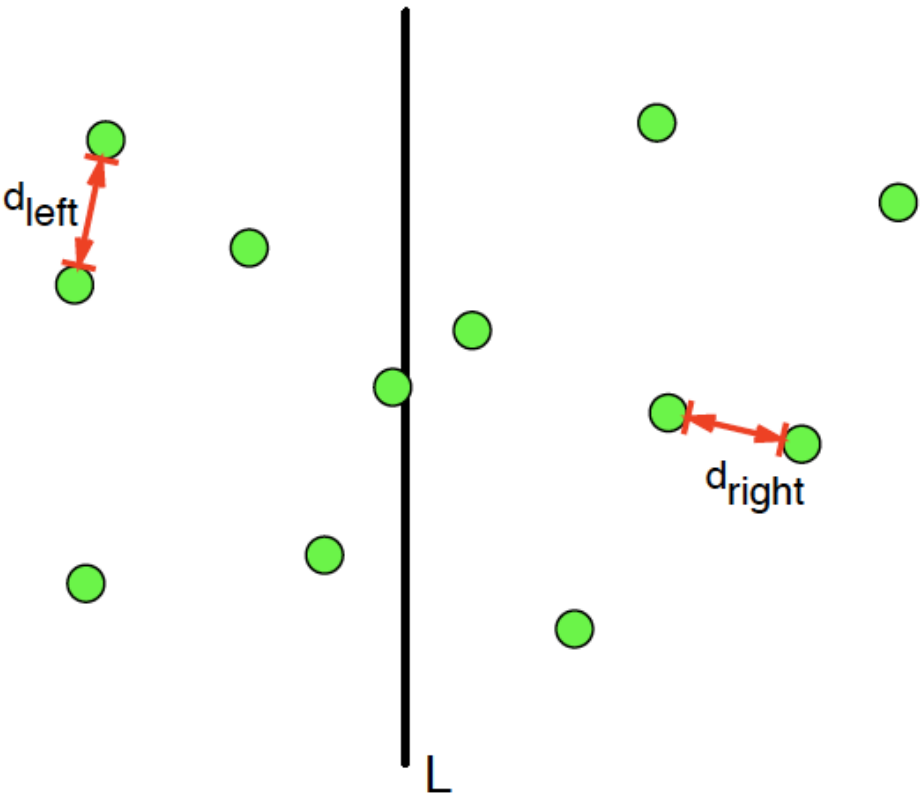
Divide-And-Conquer (1D)



- Key observation: If m is the dividing coordinate, then p3, q3 must be within d of m.
  - **p3** must be the rightmost point in **S1**
  - **q3** must be the leftmost point in **S2**
  - **Hard to generalize to 2D**
- **How many points of S1 can be in (m-d,m]?**

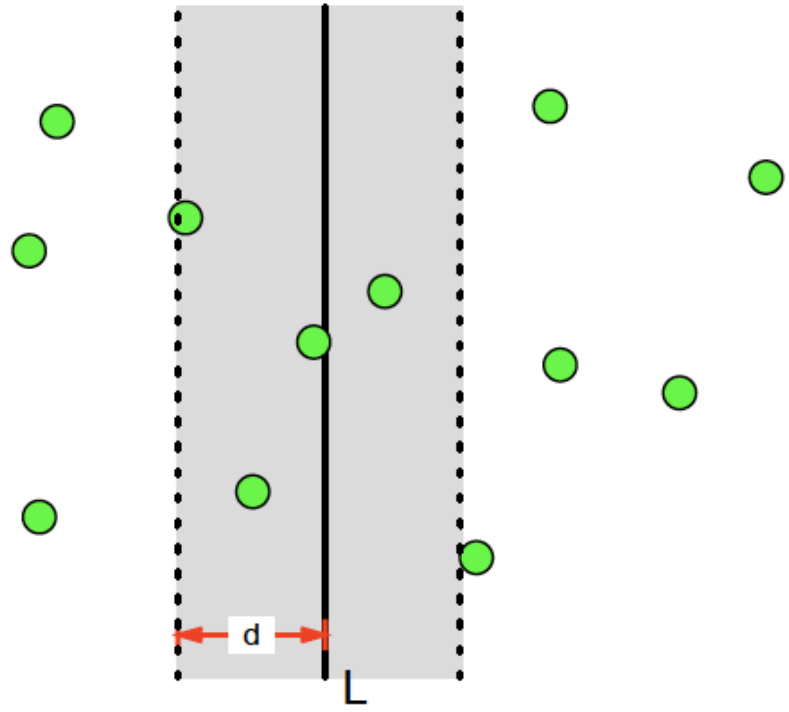
Divide-And-Conquer (2D)

- DIVIDE: split points in two equal parts with line L
- CONQUER: recursively find closest pair in parts
- COMBINE:
  - $d=\min(d_{left}, d_{right})$
  - d is the answer **unless L splits points that are close**



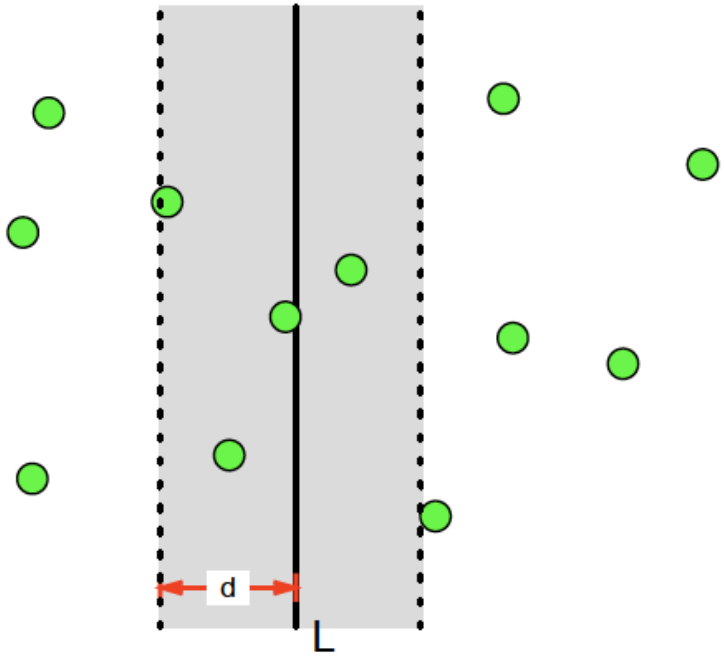
Region near L

- If there is a pair (p,q) within distance d split by L, then both p and q must be within d from L
- Let Sy be an array of points in the region sorted by y coordinate
- size of Sy might be O(|S|)
  - Cannot check all pairs



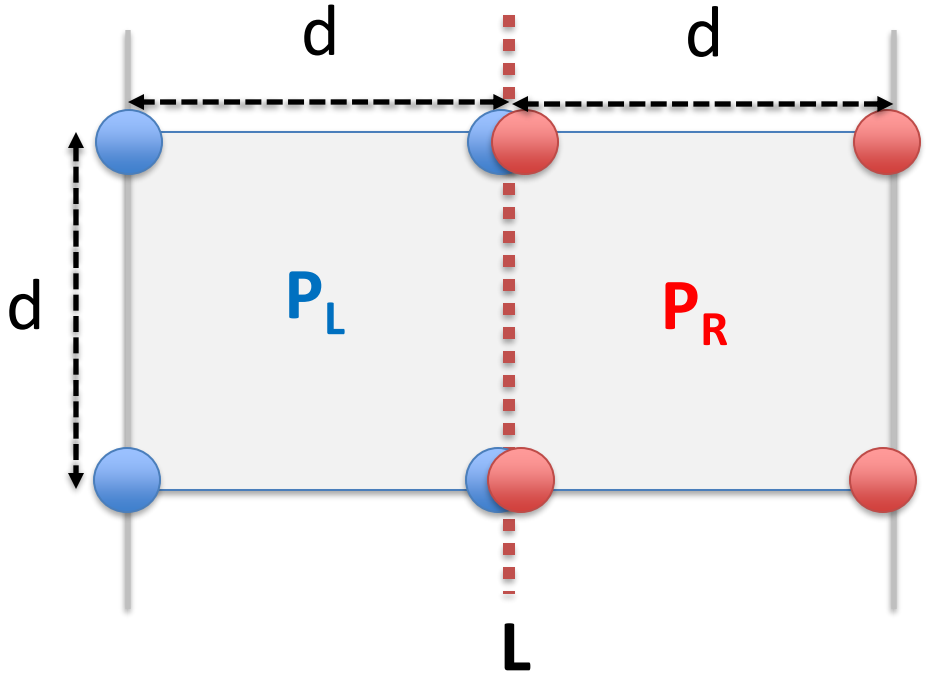
Special structure in Sy

- Sy: points sorted by y near L (in the 2d-wide slab)
- Let: **Sy=p1,p2…pm**, then if **dist(p_i,p_j)<d** then **j-i<=8**
- **"closeby points are closeby in the array"**



Special structure in Sy

- Let d be the smallest distance between pairs on the left and right
- In a box of **2d** x **d** we can pack at most 8 points
  - Points within P_L and P_R have to be at least **d** distance away
  - So, for any point x we only need to check the next 7 points (as ordered by y coordinate in Sy)



Divide and Conquer(2D)

ClosestPair(ptsX, ptsY)

```
1.   if (size(ptsX)<2) return null
2.   if (size(ptsX)==2) return ptsX
3.   m=median of x coordinates
4.   Prepare subsets to the left of m: ptsX-left, ptsY-left and to the right of m: ptsX-right,
     ptsY-right  // They should be sorted but you should not use sorting (see book chapter)
```

DIVIDE

```
5.   pair-left = ClosestPair(ptsX-left, ptsY-left)
6.   pair-right= ClosestPair(ptsX-right, ptsY-right)
```

CONQUER

```
7.   d = min of distances between pair-left and pair-right
8.   res = pair among pair-left and pair-right of the smaller distance
9.   ptsWithinD: an array of points within distance d from m, sorted by y coordinates
10.  for i=1…ptsWithinD.length
11.      for j=i+1…min(ptsWithinD.length,i+7)
12.          if dist(ptsWithinD[i], ptsWithinD[j])<d
13.              res = (ptsWithinD[i], ptsWithinD[j])
14.              d= dist(ptsWithinD[i], ptsWithinD[j])
15.  return res
```

COMBINE: O(n)

Analysis

- Divide set of points in half each time:
  - depth of recursion is O(log n)
- Merge takes O(n) time.
- Recurrence: $T(n) = 2T(n/2) + cn$
- Same as MergeSort: O(n log n) time.