Objectives

- Divide and conquer algorithms
  - Counting inversions
  - Closest pairs of points

Review

- What is a recurrence relation?
- How can you compute D&C running times?

Counting Inversions: Divide-and-Conquer

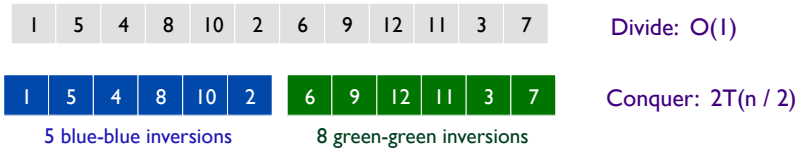
Towards a solution…

Assume number represents where item *should* be in the list, i.e., where it is in someone else's list

What was the approach so far?

Counting Inversions: Divide-and-Conquer

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities



9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: $\Theta(n^2)$

What would make figuring out blue-green inversions easier?

Total = 5 + 8 + 9 = 22

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted
- Count inversions where a_i and a_j are in different halves
- Merge two sorted halves into sorted whole

to maintain sorted invariant

- What does sorting do for us?
- What is our algorithm for counting the inversions and merging?

Note: these lists are only subsets of the original list

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted
- Count inversions where a_i and a_j are in different halves
- Merge two sorted halves into sorted whole

to maintain sorted invariant

Count: O(n)

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Merge: O(n)

We'll run through an example in a bit…

Merge and Count

```
Merge-and-Count(A,B):
    i=0
    j=0
    inversions = 0
    output = []
    while i < A.size and j < B.size:
       output.append( min(A[i], B[j]) )
       if B[j] < A[i]:
          inversions += A.size – i
       update i or j
    Append the remainder of the non-exhausted list to
    the output
    return inversions and output
```

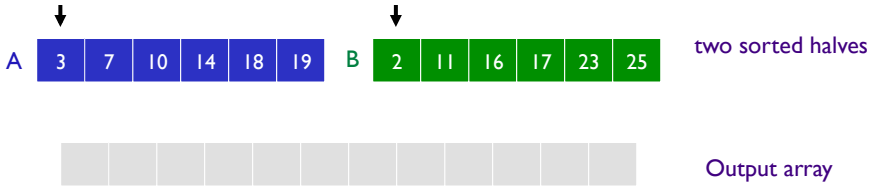

Merge and Count

Precondition: A and B are sorted

```
Merge-and-Count(A,B):
    i=0 (front of list A)
    j=0 (front of list B)
    inversions = 0
    output = []
    while A not empty and B not empty:
       output.append( min(A[i], B[j]) )
       if B[j] < A[i]:
          inversions += A.size – i (remaining elements in A)
       update i or j (whichever had smaller element)
    Append the remainder of the non-exhausted list to
    the output
    return inversions and output
```

Merge and Count Step

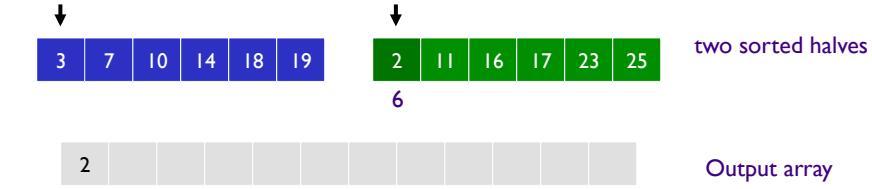
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total:

Merge and Count

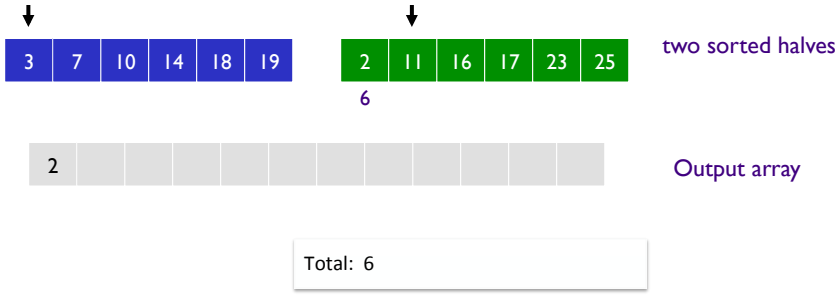
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

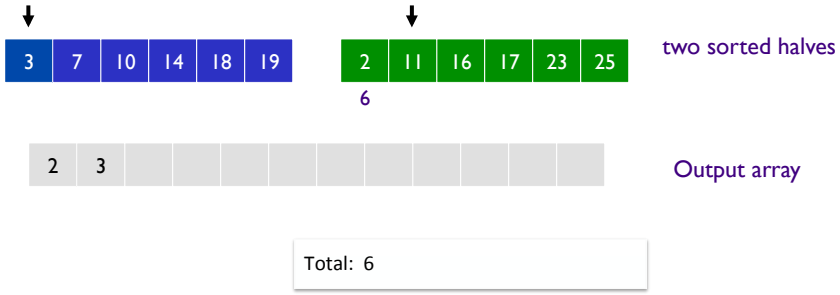
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

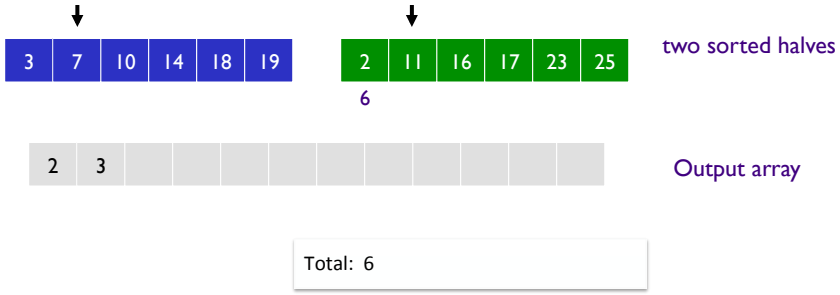
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

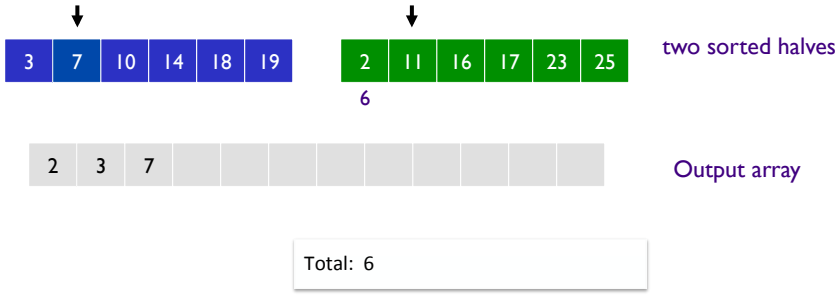
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



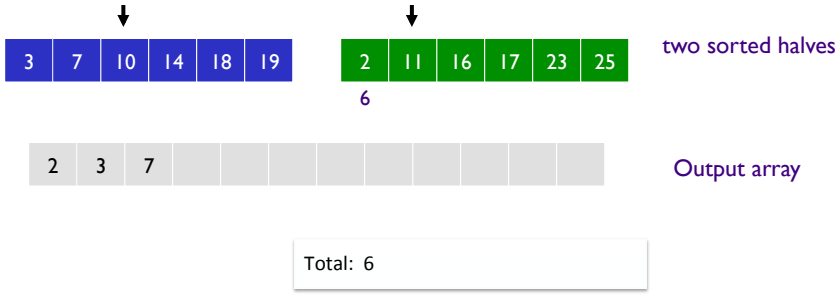
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



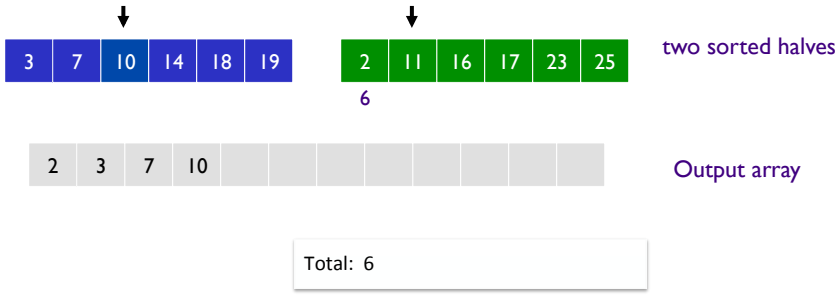
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



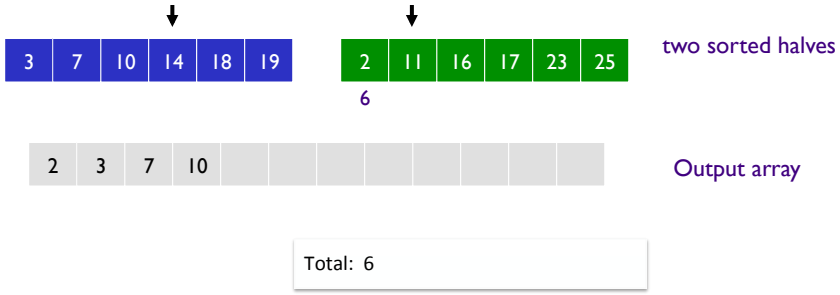
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



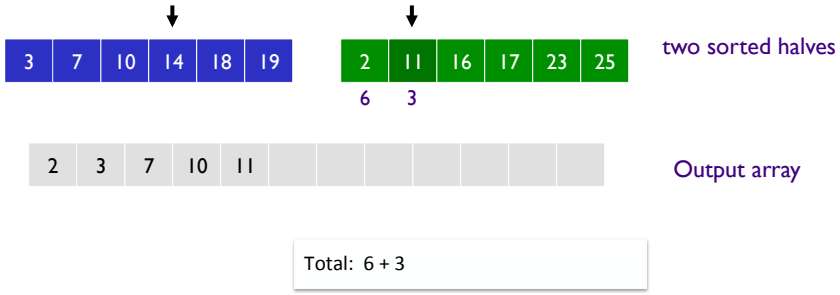
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



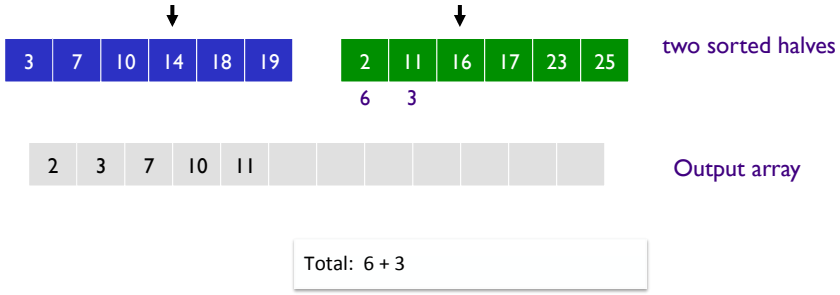
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



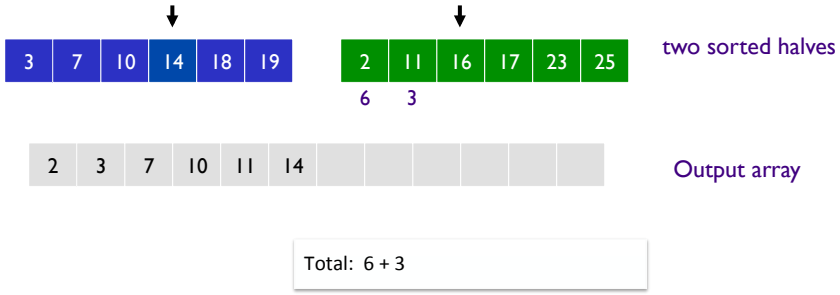
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



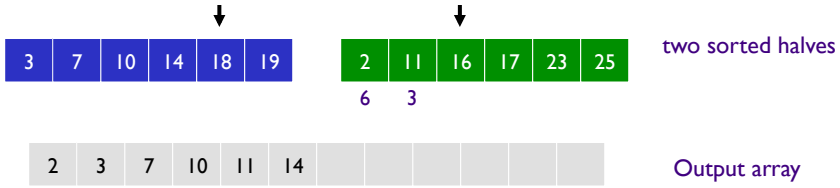
Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Merge and Count

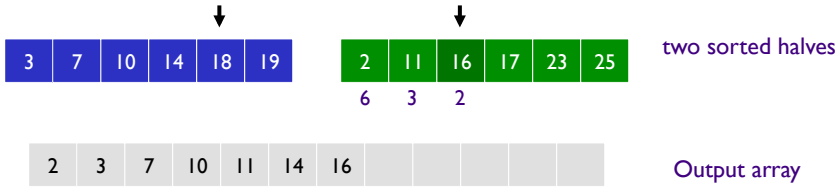
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

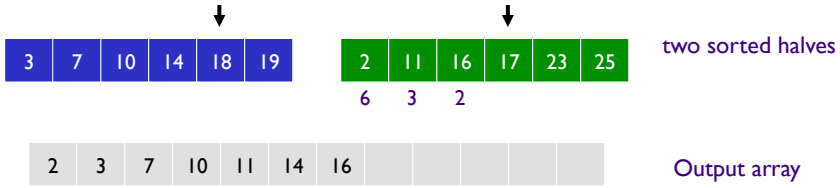
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2

Merge and Count

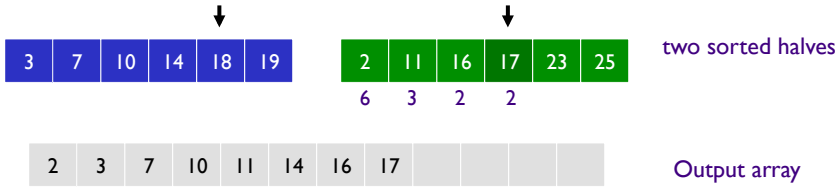
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2

Merge and Count

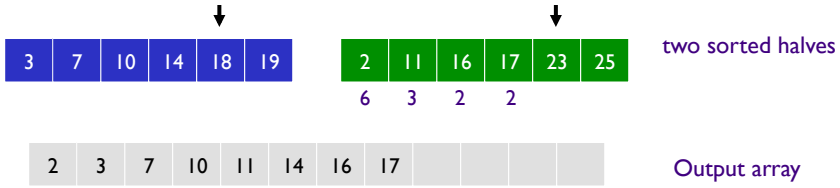
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

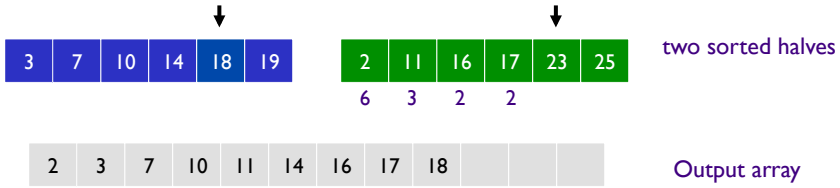
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

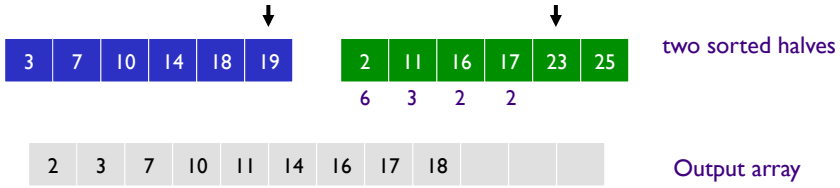
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

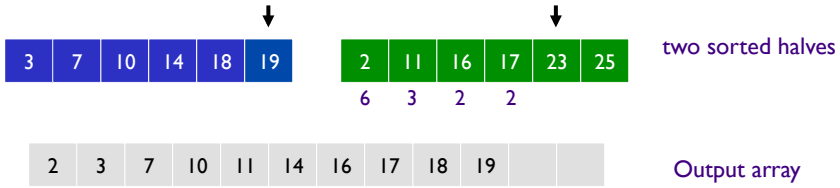
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

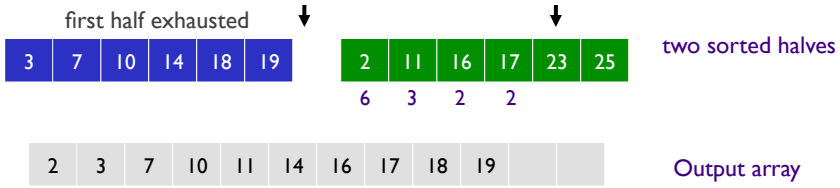
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

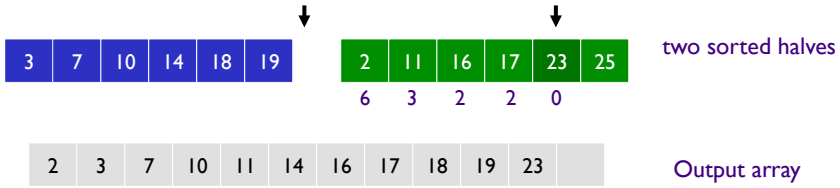
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

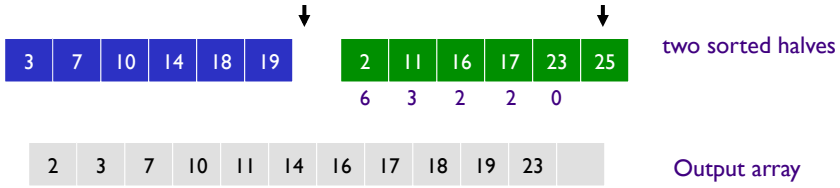
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0

Merge and Count

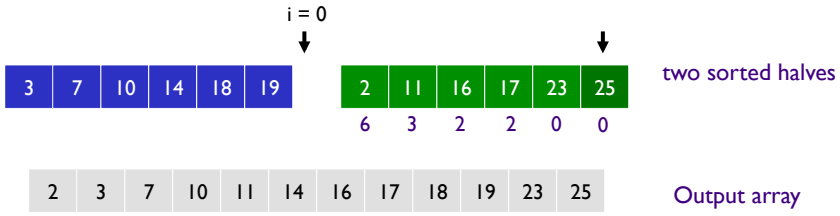
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0

Merge and Count

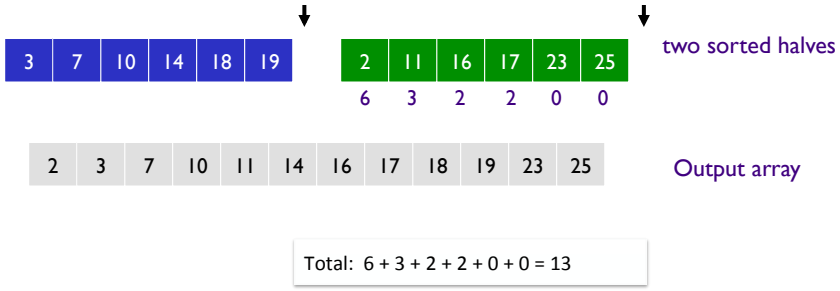
- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0 + 0

Merge and Count

- Given two sorted halves,
  count number of inversions
  where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Counting Inversions: Implementation

```
Sort-and-Count(L)
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (iA, A) = Sort-and-Count(A)
   (iB, B) = Sort-and-Count(B)
   (i, L) = Merge-and-Count(A, B)

   total_inversions = iA + iB + i
   return total_inversions and the sorted list L
```

Counting Inversions: Implementation

```
Sort-and-Count(L)
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (i_A, A) = Sort-and-Count(A)
   (i_B, B) = Sort-and-Count(B)
   (i, L) = Merge-and-Count(A, B)

   total_inversions = i_A + i_B + i
   return total_inversions and the sorted list L
```

- Merge-and-Count
  - Pre-condition. A and B are sorted.
  - Post-condition. L is sorted.

Counting Inversions: Implementation

```
Sort-and-Count(L)
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (i_A, A) = Sort-and-Count(A)
   (i_B, B) = Sort-and-Count(B)
   (i, L) = Merge-and-Count(A, B)

   total_inversions = i_A + i_B + i
   return total_inversions and the sorted list L
```

Recurrence relation?
Runtime of algorithm?

- Merge-and-Count
  - Pre-condition. A and B are sorted.
  - Post-condition. L is sorted.

Analysis

Recurrence Relation:

$T(n) \leq T(n/2) + T(n/2) + O(n)$

➔$T(n) \in O(n \log n)$

```
Sort-and-Count(L)
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (i_A, A) = Sort-and-Count(A)   T(n/2)
   (i_B, B) = Sort-and-Count(B)   T(n/2)
   (i, L) = Merge-and-Count(A, B) O(n)

   total_inversions = i_A + i_B + i
   return total_inversions and the sorted list L
```


CLOSEST PAIR OF POINTS

Computational Geometry

- Algorithms and data structures for geometrical objects
  - Points, line segments, polygons, etc.
  - Common motivator: large data sets → efficiency
- Some Applications
  - Graphics
  - Robotics
    - motion planning and visibility problems
  - Geographic information systems (GIS)
    - geometrical location and search, route planning

Closest Pair of Points

- Closest pair. Given *n* points in the plane, find a pair with smallest Euclidean distance between them.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.

fast closest pair inspired
fast algorithms for these problems

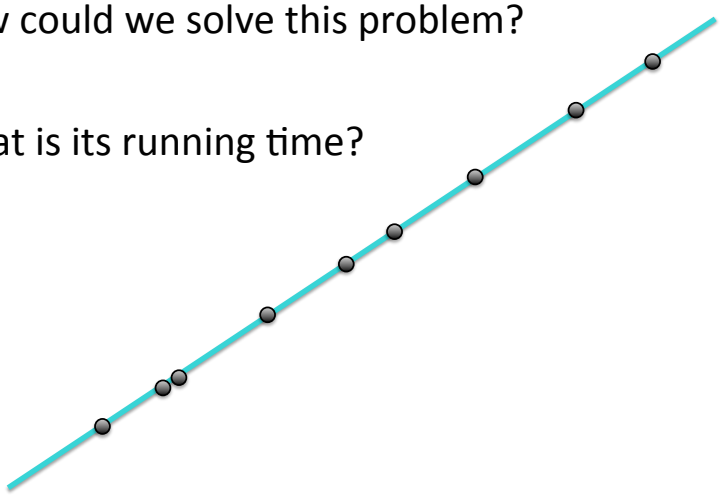
- Brute force?

Closest Pair of Points

- Closest pair. Given *n* points in the plane, find a pair with smallest Euclidean distance between them.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.
- Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons

Simplify: All Points on a Line

- How could we solve this problem?
- What is its running time?



Simplify: All Points on a Line

- How could we solve this problem?
  - Sort the points
    - Monotonically increasing x/y coordinates
    - No closer points than neighbors in sorted list
  - Step through, looking at the distances between each pair
- What is its running time?
  - O(n logn)

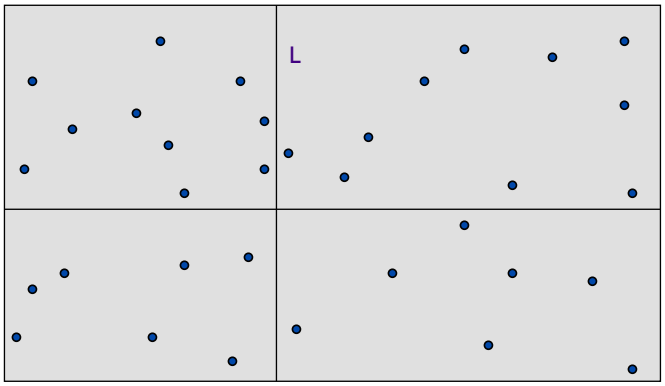
Why won't this work for 2D?

Closest Pair of Points

- Closest pair. Given *n* points in the plane, find a pair with smallest Euclidean distance between them.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.
- Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons
- 1-D version. O(n log n)
  - Easy if points are on a line
- Assumption. No two points have same x coordinate to make presentation cleaner

Closest Pair of Points: First Attempt

- Divide. Sub-divide region into 4 quadrants

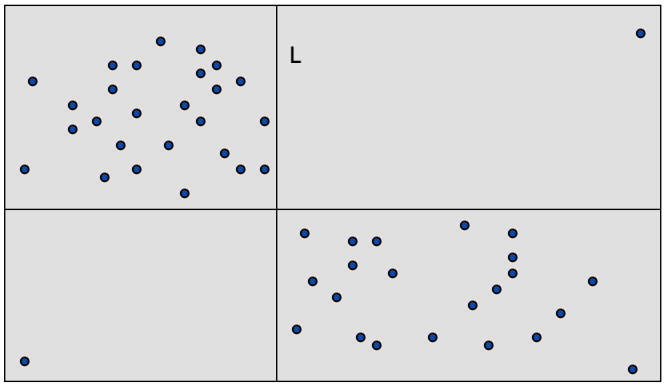


Why does this seem to be a natural first step?
Any problems with implementing this approach?

45

Closest Pair of Points: First Attempt

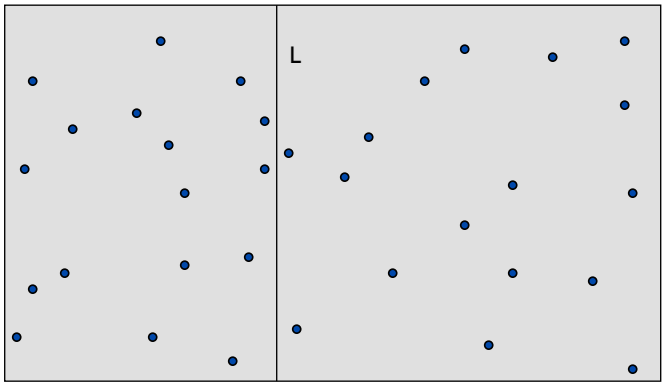
- Divide. Sub-divide region into 4 quadrants
- Obstacle. Impossible to ensure n/4 points in each piece



Closest Pair of Points

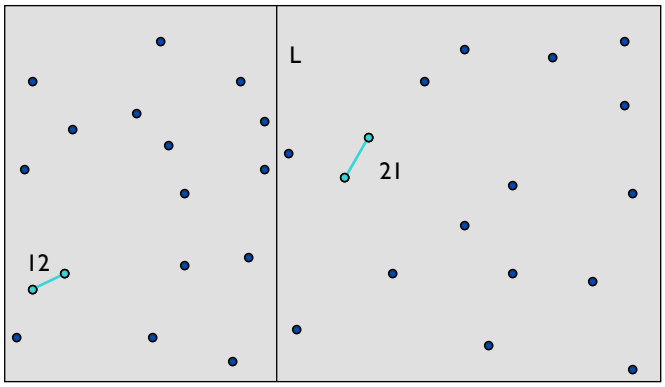
- Divide: draw vertical line L so that roughly ½n points on each side

How do we implement this?



Closest Pair of Points

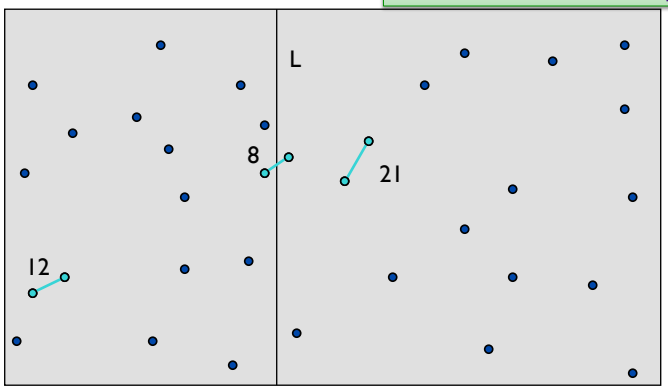
- Divide: draw vertical line L so that roughly ½n points on each side
- Conquer: find closest pair in each side recursively



Closest Pair of Points

- Divide: draw vertical line L so that roughly ½n points on each side
- Conquer: find closest pair in each side recursively
- Combine: find closest pair with one point in each side *seems like $\Theta(n^2)$*
- Return best of 3 solutions

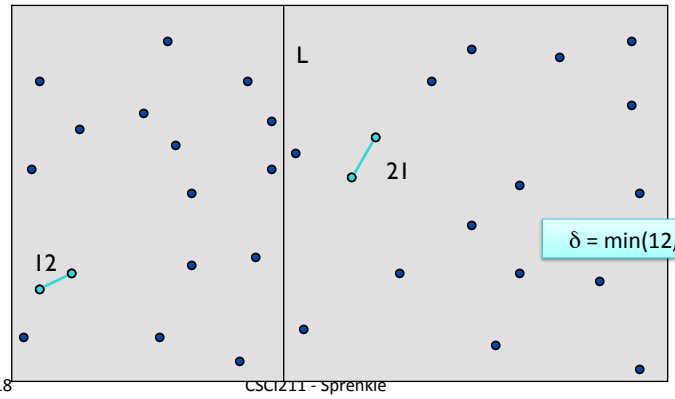
Do we need to check all pairs?



Closest Pair of Points

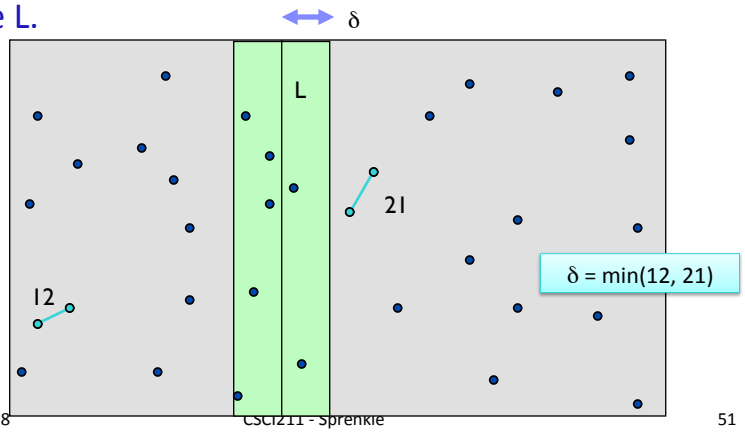
- Find closest pair with one point in each side, assuming that distance < δ

  where δ = min(left_min_dist, right_min_dist)



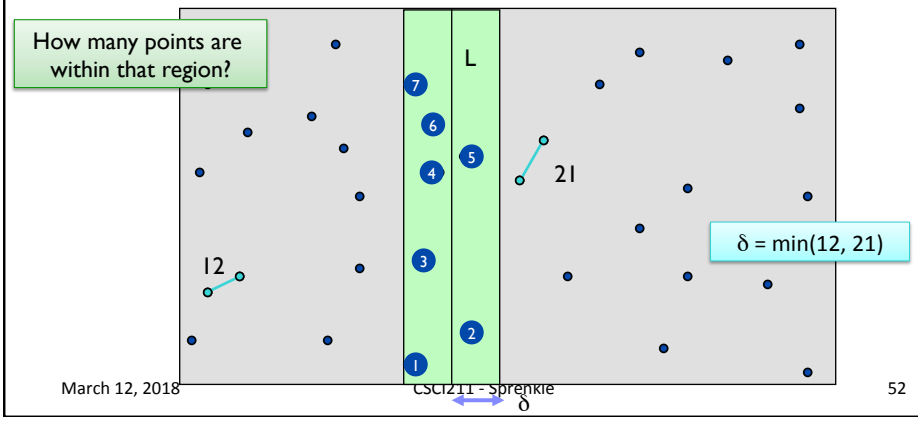
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.
  - Observation: only need to consider points within δ of line L.



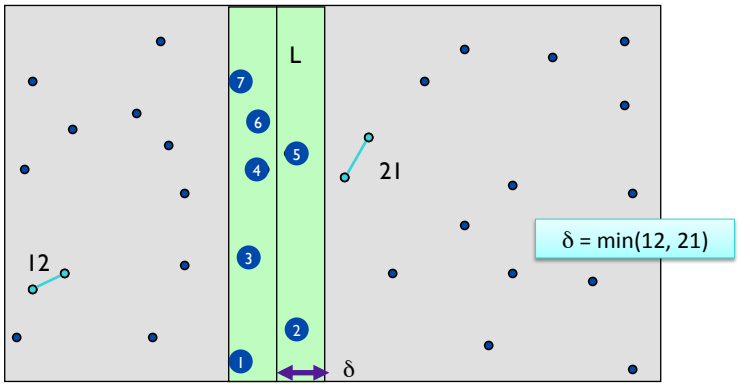
Closest Pair of Points

- Find closest pair w/ 1 point in each side, assuming that distance < δ.
  - Observation: only consider points within δ of line L
  - Sort points in 2δ-strip by their y coordinate



Closest Pair of Points

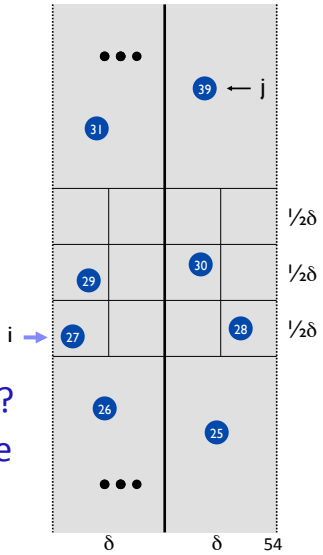
- Find closest pair w/ 1 point in each side, assuming that distance < δ
  - Observation: only consider points within δ of line L
  - Sort points in 2δ-strip by their y coordinate
    - Only checks distances of those within 11 positions in sorted list!



Analyzing Cost of Combining

Prepare minds to be blown…

- Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate
- Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ
  - What is the distance of the box?
  - How many points can be in a box?
  - When do we know that points are > δ apart?



Looking Ahead

- Wiki: 4.8, 5.1-5.3
- PS 7 due Friday
- Exam 2 handed out on Friday
  - Greedy and D&C
  - Due following Friday