Objectives

- Divide and conquer
  - Counting inversions
  - Closest pair of points

Review: Comparing Rankings

- To determine similarity of rankings, need a metric
- Similarity metric: number of inversions between two rankings
  - My rank: 1, 2, …, n
  - Your rank: $a_1, a_2, \ldots, a_n$
  - Movies i and j *inverted* if $i < j$ but $a_i > a_j$

Naïve/Brute force solution?

Movies

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions:
3-2, 4-2

Review: Counting Inversions

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

1	5	4	8	10	2	6	9	12	11	3	7

Should be at position 5

Towards a solution…

Review: Counting Inversions

- 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

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1)

1	5	4	8	10	2	6	9	12	11	3	7

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

Combine: seems like $\Theta(n^2)$

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Total = 5 + 8 + 9 = 22

What would make figuring out blue-green inversions easier?

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

A | 3 | 7 | 10 | 14 | 18 | 19 | B | 2 | 11 | 16 | 17 | 23 | 25 |

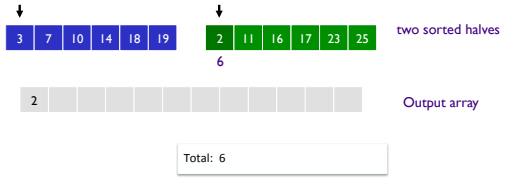
two sorted halves

Output array

Total:

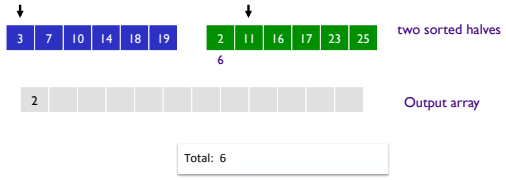
Merge and Count

- 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



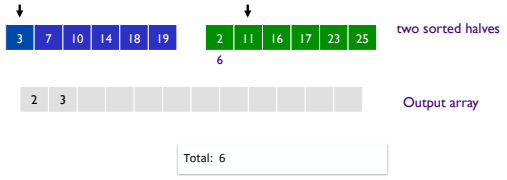
Merge and Count

- 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



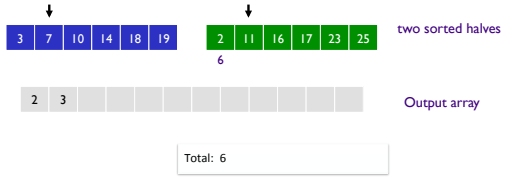
Merge and Count

- 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



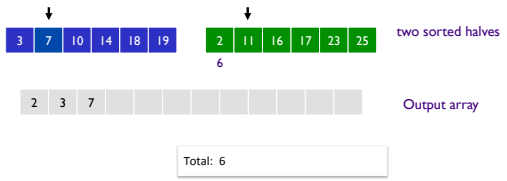
Merge and Count

- 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



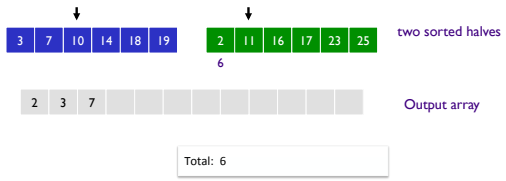
Merge and Count

- 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



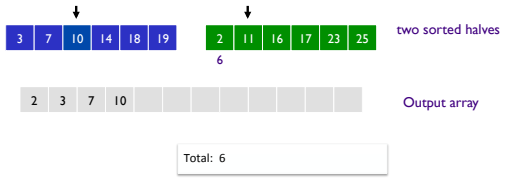
Merge and Count

- 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

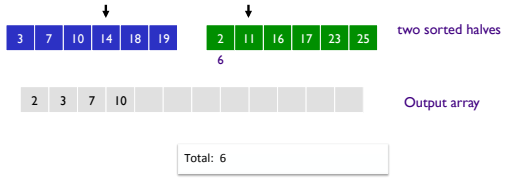
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

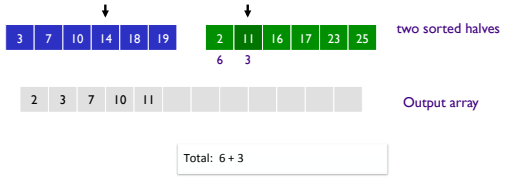
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

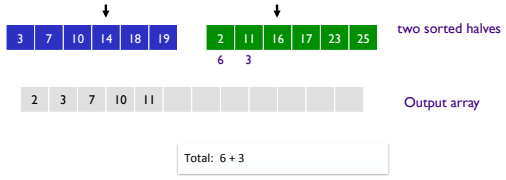
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

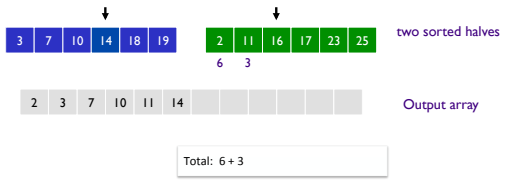
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

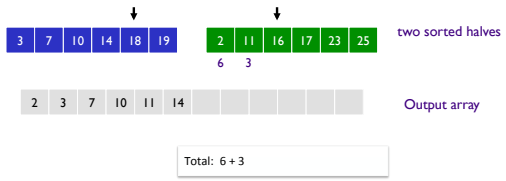
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

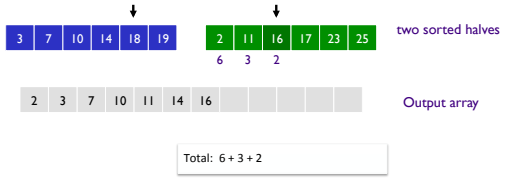
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

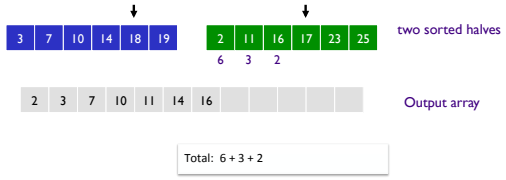
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2

Merge and Count

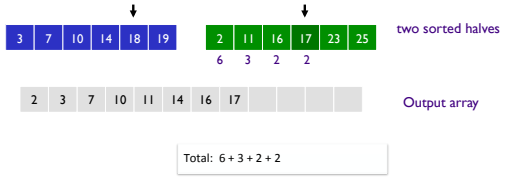
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2

Merge and Count

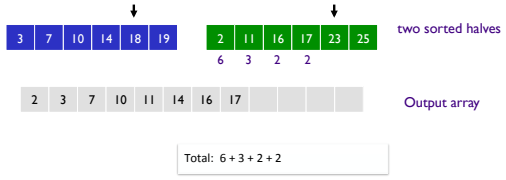
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

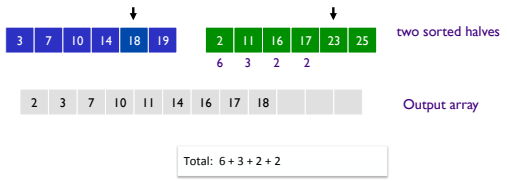
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

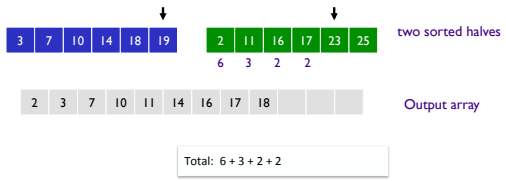
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

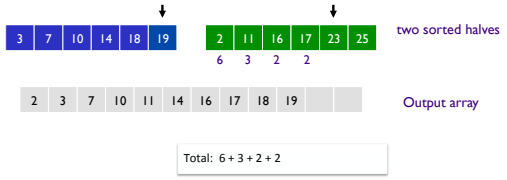
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

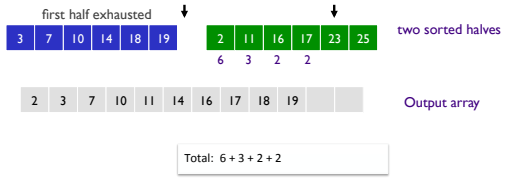
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

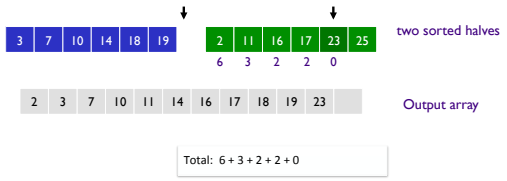
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

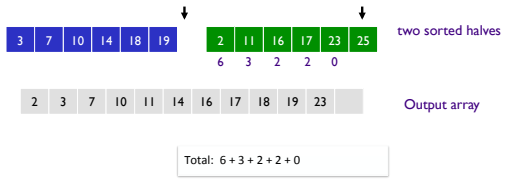
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0

Merge and Count

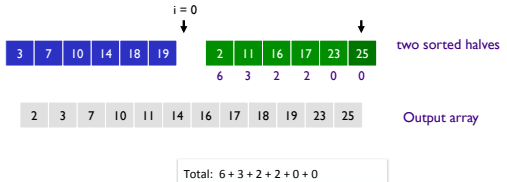
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0

Merge and Count

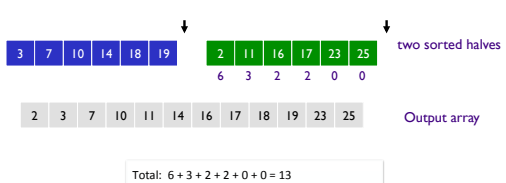
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



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



Total: 6 + 3 + 2 + 2 + 0 + 0 = 13

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
   (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
```

- 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
   (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
```

Recurrence relation?
Runtime of algorithm?

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

Analysis

Recurrence Relation:

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

$\rightarrow T(n) \in O(n \log n)$

```
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
   (iA, A) = Sort-and-Count(A)   T(n/2)
   (iB, B) = Sort-and-Count(B)   T(n/2)
   (i, L) = Merge-and-Count(A, B) O(n)

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


CLOSEST PAIRS 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

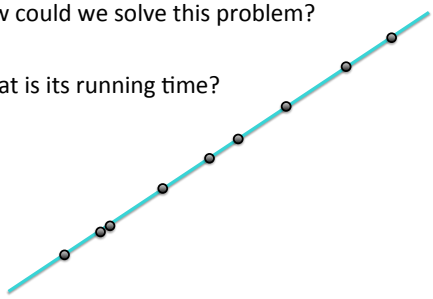
- Brute force?

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)

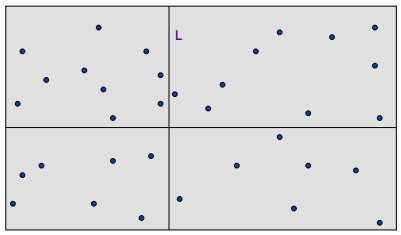
Why won't this work for 2D?

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 clearer.

Closest Pair of Points: First Attempt

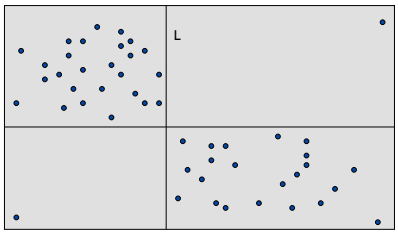
- Divide. Sub-divide region into 4 quadrants



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

Closest Pair of Points: First Attempt

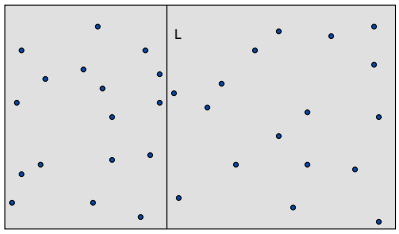
- Divide. Sub-divide region into 4 quadrants
- Obstacle. Impossible to ensure n/4 points in each piece



Closest Pair of Points

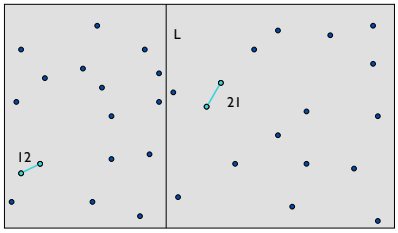
- Divide: draw vertical line L so that roughly ½n points on each side

How do we implement this?



Closest Pair of Points

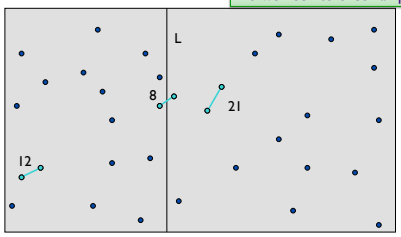
- Divide: draw vertical line L so that roughly ½n points on each side
- Conquer: find closest pair in each side recursively



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

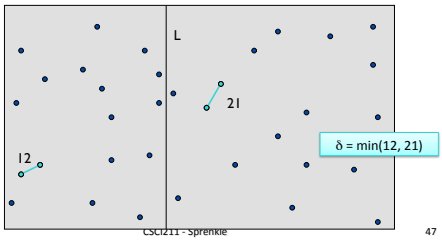
Do we need to check all pairs?



Closest Pair of Points

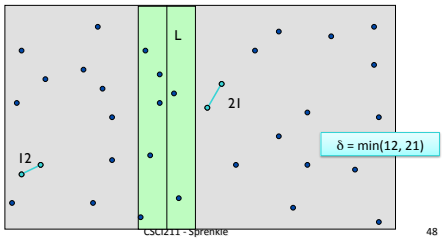
- Find closest pair with one point in each side, assuming that distance < δ

where δ = min(left_min_dist, right_min_dist)



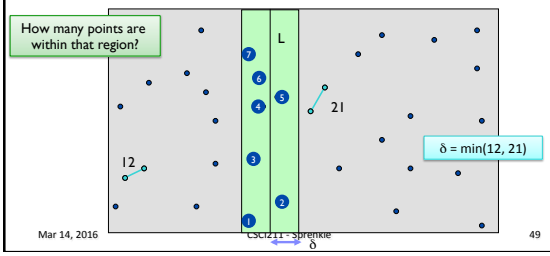
Closest Pair of Points

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



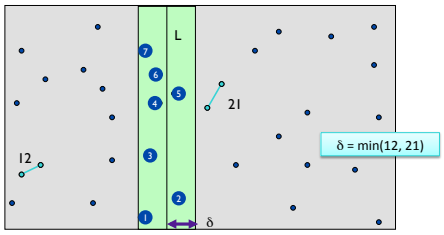
Closest Pair of Points

- 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!



Looking Ahead

- Wiki due tonight
  - Chapter 4.8, 5.1, 5.2, 5.3
- PS7 due Friday