Algorithms

Divide and Conquer

Dr. Mudassir Shabbir

LUMS University

February 4, 2026

Announcements

- Midterm Exam/Long Quiz 1 on Sun 02/22, 2026 noon - 1:45p.
- Worksheet 1 grades will be posted tomorrow.

Contestation Rule: You have 10 day after the grades are published to contest any grade.

Recap: Algorithm Design Paradigms

- **Prune and Search**
- **Divide and Conquer:**
  - Recursively break problem into smaller subproblems.
  - Combine subproblem solutions to solve original problem.
- Examples:
  - Quick Sort, Merge Sort,
  - Counting Inversions
  - Closest Pair of Points in 2D

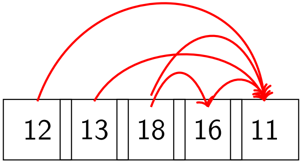
Inversions: Definition & Example

Definition

An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

Example

- Array $A = [12,\ 13,\ 18,\ 16,\ 11]$
- Inversions: $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$



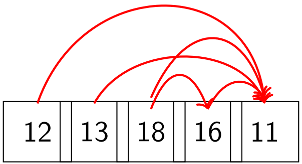
Counting Inversions

Maximum possible inversions

- Max Number of inversions: $\binom{n}{2} = \dfrac{n(n-1)}{2}$.

Brute force idea

- Check all pairs (i, j) with $i < j$; for each pair, test if $A[i] > A[j]$.
- Total pairs: $\binom{n}{2} = \Theta(n^2) \;\Rightarrow\;$ running time $\Theta(n^2)$; extra space $O(1)$.



Brute Force for Inversions

```
countInversions(A):
    n = length(A)
    count = 0
    for i = 1 to n:
        for j = i+1 to n:
            if A[i] > A[j]:
                count = count + 1
    return count
```

Cost: $\Theta(n^2)$ comparisons.

Brute Force for Inversions

```
countInversions(A):
    n = length(A)
    count = 0
    for i = 1 to n:
        for j = i+1 to n:
            if A[i] > A[j]:
                count = count + 1
    return count
```

Cost: $\Theta(n^2)$ comparisons.

Can we do better?

Brute Force for Inversions

```
countInversions(A):
    n = length(A)
    count = 0
    for i = 1 to n:
        for j = i+1 to n:
            if A[i] > A[j]:
                count = count + 1
    return count
```

Cost: $\Theta(n^2)$ comparisons.

Can we do better? — try **Divide & Conquer** (merge-and-count).

Divide and Conquer

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

- Divide into 2 sublists of equal size
- Recursively count the inversions
  - 5 blue-blue inversions
  - 8 red-red inversions

Divide and Conquer

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

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

- Divide into 2 sublists of equal size
- Recursively count the inversions
  - 5 blue-blue inversions
  - 8 red-red inversions

We also need to count 9 blue-red inversions? Total $= 5 + 8 + 9 = 22$.

Divide and Conquer: Counting Inversions

```
countInversions(A):
    n = length(A)
    if n <= 1:
        return 0
    B = A[1..n/2]
    R = A[n/2+1..n]
    BB = countInversions(B)
    RR = countInversions(R)
    BR = countBlueRedInversions(B,R)
    return BB + RR + BR
```

Divide and Conquer: Counting Inversions

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

Cost: $O(1)$

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

Cost: $2 \cdot T(n/2)$

Cost: ???

- 5 blue-blue inversions, 8 red-red inversions
- 9 blue-red inversions
- **Total = 5 + 8 + 9 = 22.**

Counting Blue–Red Inversions

Observation

- Comparing every blue element to every red element *seems* quadratic: $O(|B| \cdot |R|) = O(n^2)$.

Key idea (assume sorted halves)

- for $x \in B$, BR inversions contributed by x is $\mathrm{rank}_R(x)$.
- Using a two-pointer merge-style scan to compute ranks, we can count all BR inversions in **linear time** $O(n)$ once halves are sorted.

Divide and Conquer: Counting Inversions

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

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

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

- Counting blue-red inversions amounts to counting the rank of each blue element in the red half.
- This can be done in **linear time** $O(n)$ via a merge-style two-pointer scan.

Finding Inversions

Combine: count blue-red 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.

Variation of mergesort

Merge and Count

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.

Left:	3	7	10	14	18	19
Right:	2	11	16	17	23	25

```
numLeft = 6
Total:
```

Merge and Count

Left:	3	7	10	14	18	19
Right:	2	11	16	17	23	25
Merged:	2					

```
numLeft = 6
Total: 6
```

Merge and Count

Left:	3	7	10	14	18	19
Right:		11	16	17	23	25
Merged:	2	3				

```
numLeft = 5

Total: 6
```

Merge and Count

Left:

7	10	14	18	19

Right:

11	16	17	23	25

Merged:

2	3	7

```
numLeft = 4
Total: 6
```

Merge and Count

Left: | 10 | | 14 | 18 | 19 |

Right: | 11 | 16 | 17 | 23 | 25 |

Merged: | 2 | 3 | 7 | 10 |

`numLeft = 3`

`Total: 6`

Merge and Count

Left:			14	18	19
Right:		11	16	17	23	25
Merged:	2	3	7	10	11

`numLeft = 3`

`Total: 6 + 3`

Merge and Count

Left:			14		18	19
Right:			16	17	23	25
Merged:	2	3	7	10	11	14

```
numLeft = 2
Total: 6 + 3
```

Merge and Count

Left:

18	19

Right:

16	17	23	25

Merged:

2	3	7	10	11	14	16

`numLeft = 2`

`Total: 6 + 3 + 2`

Merge and Count

Left: 18 | 19

Right: 17 | 23 | 25

Merged: 2 | 3 | 7 | 10 | 11 | 14 | 16 | 17

```
numLeft = 2

Total: 6 + 3 + 2 + 2
```

Merge and Count

Left:

18	19

Right:

23	25

Merged:

2	3	7	10	11	14	16	17	18

`numLeft = 1`

`Total: 6 + 3 + 2 + 2`

Merge and Count

Left:				19						
Right:					23	25				
Merged:	2	3	7	10	11	14	16	17	18	19

`numLeft = 0`

`Total: 6 + 3 + 2 + 2`

Merge and Count

Right:			23		25						
Merged:	2	3	7	10	11	14	16	17	18	19	23

`numLeft = 0`

`Total: 6 + 3 + 2 + 2`

Merge and Count

Right:

25

Merged:

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

`numLeft = 0`

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

Counting Inversions: Implementation

```
Sort-and-Count(L)
if list L has one element then
    return (0, L)
end if

Divide the list into two halves A and B
(r_A, A) ← Sort-and-Count(A)
(r_B, B) ← Sort-and-Count(B)
(r_C, L) ← Merge-and-Count(A, B)
r = r_A + r_B + r_C
return (r, L)
```

Cost of Sort-and-Count?

Sort-and-Count(L)
if list L has one element **then**
    return (0, L)
end if

Divide the list into two halves A and B
$(r_A, A) \leftarrow$ **Sort-and-Count**(A)
$(r_B, B) \leftarrow$ **Sort-and-Count**(B)
$(r_C, L) \leftarrow$ **Merge-and-Count**(A, B)
$r = r_A + r_B + r_C$
return (r, L)

Recurrence: $T(n) = 2T(n/2) + O(n)$
Solution: $T(n) = O(n \log n)$

Closest Pair of Points

Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

Closest Pair of Points

1-dimensional version

Closest Pair of Points

1-D version.

- Sort points Cost: $O(n \log n)$
- For each point, find the distance between a point and the point that follows it. Cost: $O(n)$
- Remember the smallest.

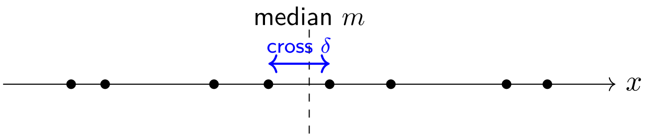
Total is $O(n \log n)$

1D via Divide & Conquer

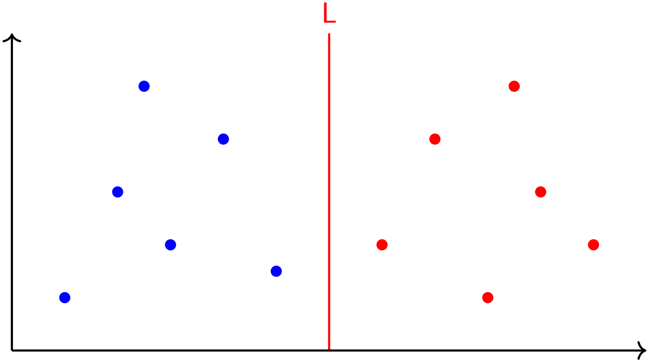
- Divide by the median m; recursively compute δ_L and δ_R.
- Combine: the only cross-pair to check is $(\max L, \min R)$.
- Return $\delta = \min(\delta_L, \delta_R, |\min R - \max L|)$.

Running Time

$T(n) = 2T(n/2) + O(1)$ if the median is known; with sorting once, we still get $O(n \log n)$ overall.

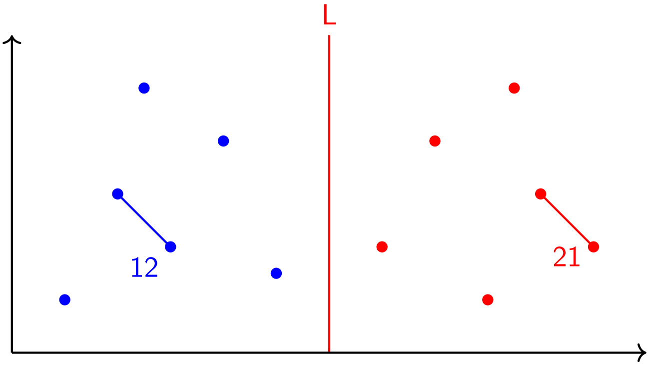


Divide: draw vertical line L so that $n/2$ points on each side.



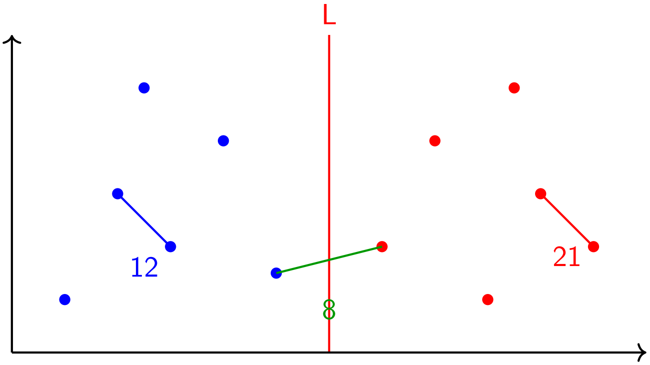
Closest Pair of Points

Solve: recursively find closest pair in each side.



Closest Pair of Points

Combine: find closest pair with one point from each side. Return closest of three pairs.



Closest Pair of Points

Running Time?

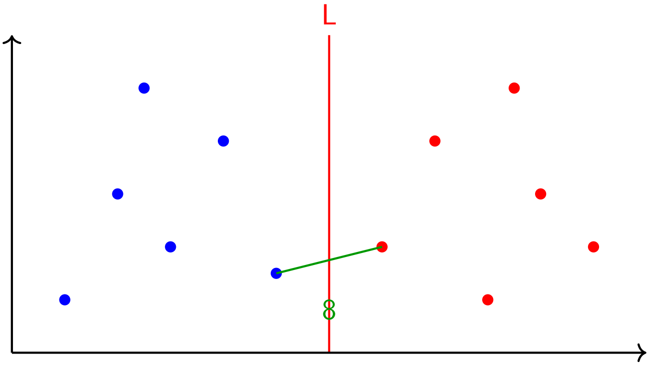
$$T(n) \leq 2T(n/2) + ???$$

Time for combine?

Goal: implement combine in linear time, to get $O(n \log n)$ overall

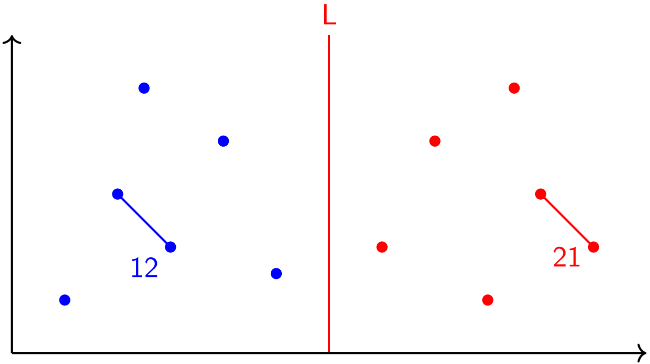
Closest Pair of Points

Combine: how to do this without comparing each point on left to each point on right?



Closest Pair of Points

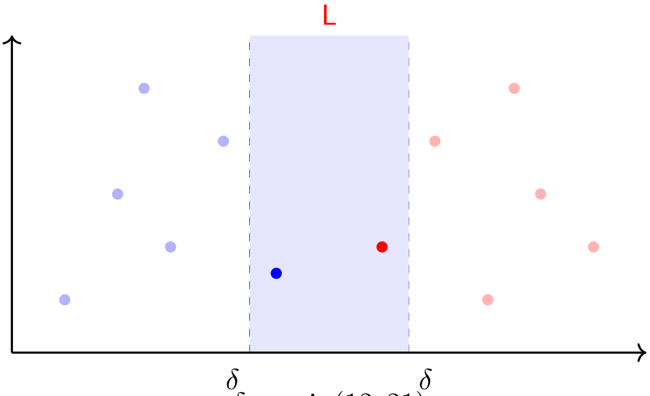
Let δ be the minimum between pair on left and pair on right
If there exists a pair with one point in each side and whose distance $< \delta$, find that pair.



$\delta = \min(12, 21)$

Closest Pair of Points

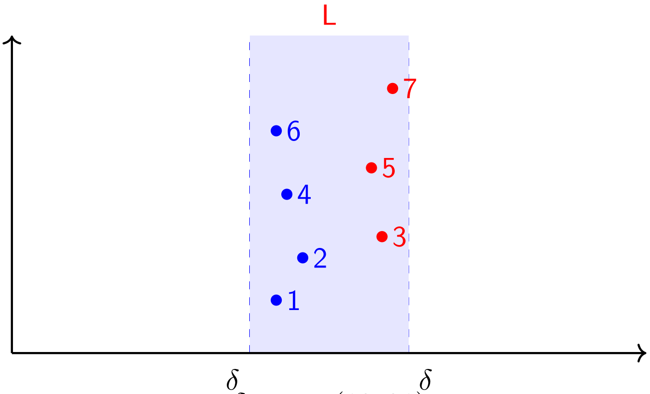
Observation: only need to consider points within δ of line L.



$$\delta = \min(12, 21)$$

Closest Pair of Points

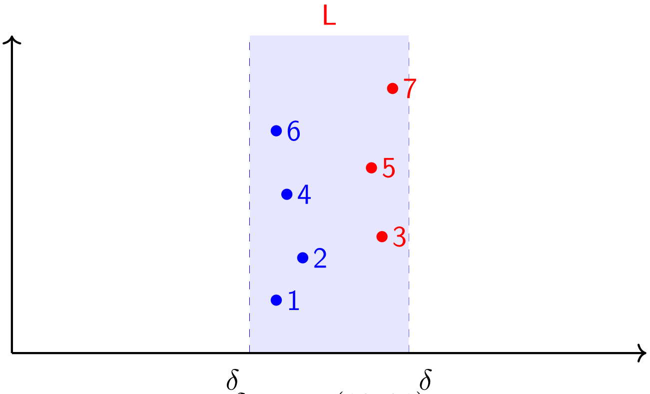
Sort points in 2δ-strip by their y coordinate.



$$\delta = \min(12, 21)$$

Closest Pair of Points

Unbelievable lemma: only need to check distances of those within 11 positions in sorted list!



$\delta = \min(12, 21)$

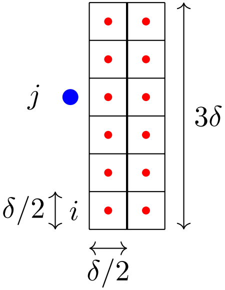
Closest Pair of Points

Let $s_1, s_2, \ldots, s_k$ be the points in the 2δ strip sorted by y-coordinate.

Claim. If $|i - j| > 11$, then the distance between s_i and s_j is at least δ.

Proof:

- No two points lie in same $\delta/2$-by-$\delta/2$ box.
- Two points separated by at least 3 rows have distance $\geq 3\delta/2$.



Closest Pair Algorithm

1: **Closest-Pair**$(p_1, \ldots, p_n)$
2: Compute separation line L such that half the points are on one side and half on the other side. $O(n \log n)$
3: $\delta_1 =$ **Closest-Pair**(left half) $2T(n/2)$
4: $\delta_2 =$ **Closest-Pair**(right half)
5: $\delta = \min(\delta_1, \delta_2)$ $O(n)$
6: Delete all points further than δ from separation line L $O(n \log n)$
7: Sort remaining points by y-coordinate. $O(n)$
8: Scan points in y-order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ, update δ.
9: **return** δ

Recurrence: $T(n) \leq 2T(n/2) + O(n \log n)$
Solution: $T(n) = O(n \log^2 n)$

Closest Pair of Points: Improvement

Can we achieve $O(n \log n)$?

Yes: pre-sort all points by x- and y-coordinates, and filter sorted lists to find the points within δ of L.

See Subhash Suri *(UC Santa Barbara)* Notes for details.