Algorithms

Divide and Conquer

Dr. Mudassir Shabbir

LUMS University

February 11, 2026

Announcements

- Midterm Exam/Long Quiz 1 on Sun 02/22, 2026 noon - 1:45p.
- Homework 2 (no-submission practice problems) will be released later this week.
- Talk on *Graphs, Geometry, and Machine Learning* in LCE Auditorium at 6pm.

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 consecutive pairs. — Cost: $O(n)$
- Remember the smallest.

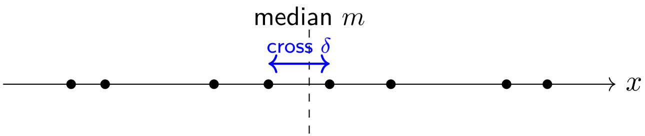
Total is $O(n \log n)$

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 $\delta = \min(\delta_L, \delta_R, |\min R - \max L|)$.

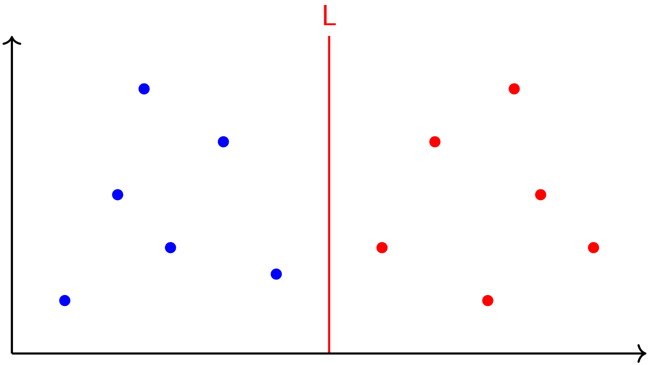
Running Time

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



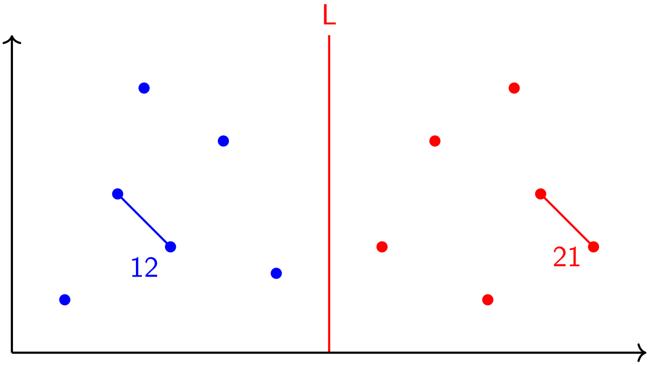
Closest Pair of Points

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



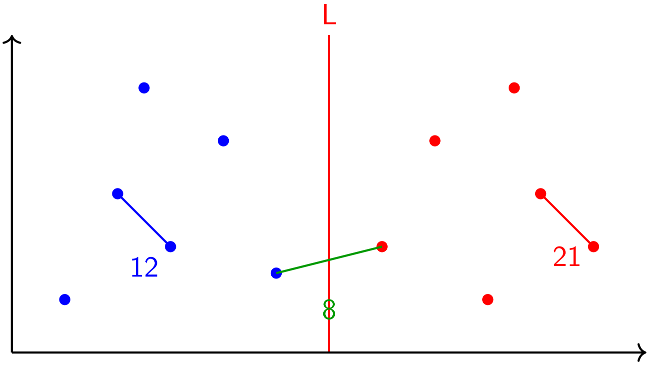
Closest Pair of Points

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

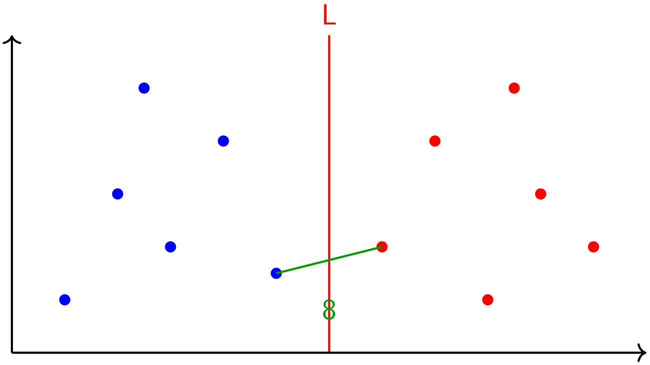
Running Time?

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

Time for combine?

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

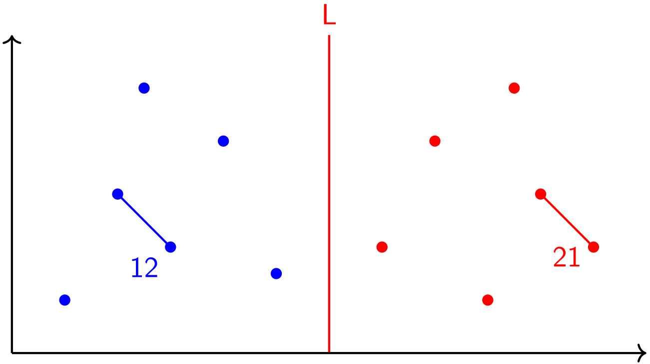
Combine: how to do this without comparing each point on left to each point on right?



Closest Pair of Points

Let δ be the minimum between pair on left and pair on right

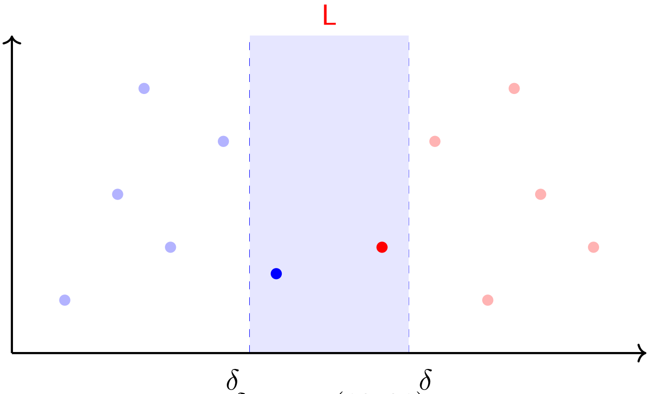
If there exists a pair with one point in each side and whose distance $< \delta$, find that pair.



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

Closest Pair of Points

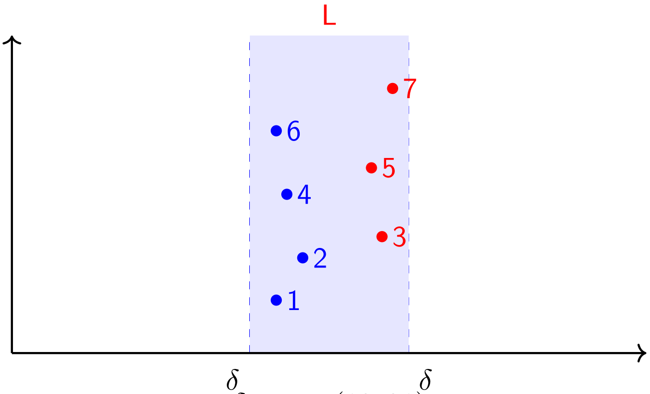
Observation: only need to consider points within δ of line L.



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

Closest Pair of Points

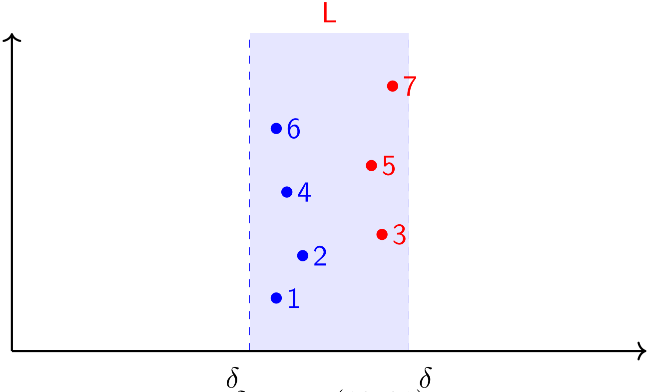
Sort points in 2δ-strip by their y coordinate.



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

Closest Pair of Points

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



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

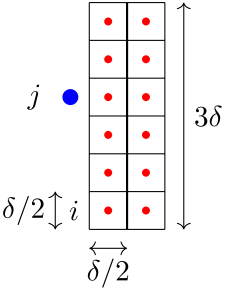
Closest Pair of Points

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

Closest-Pair$(p_1, \ldots, p_n)$
Compute separation line L such that half the points are on one side and half on the other side. $O(n \log n)$
$\delta_1 =$ **Closest-Pair**(left half) $2T(n/2)$
$\delta_2 =$ **Closest-Pair**(right half)
$\delta = \min(\delta_1, \delta_2)$ $O(n)$
Delete all points further than δ from separation line L $O(n \log n)$
Sort remaining points by y-coordinate. $O(n)$
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.
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.

Sorting Lower Bounds

Comparison-Based Sorting: The Question

- We want to understand a fundamental limitation of sorting.
- Consider any algorithm that sorts using only:

comparisons between elements

- Examples:
  - Bubble Sort, Merge Sort, Heap Sort, QuickSort
- Question:

Can we sort faster than $O(n \log n)$ using comparisons?

The Comparison Model

- Input: n distinct elements
- Allowed operation: compare two elements

$$a_i \; ? \; a_j$$

- Each comparison has two possible outcomes:

$$a_i < a_j \quad \text{or} \quad a_i > a_j$$

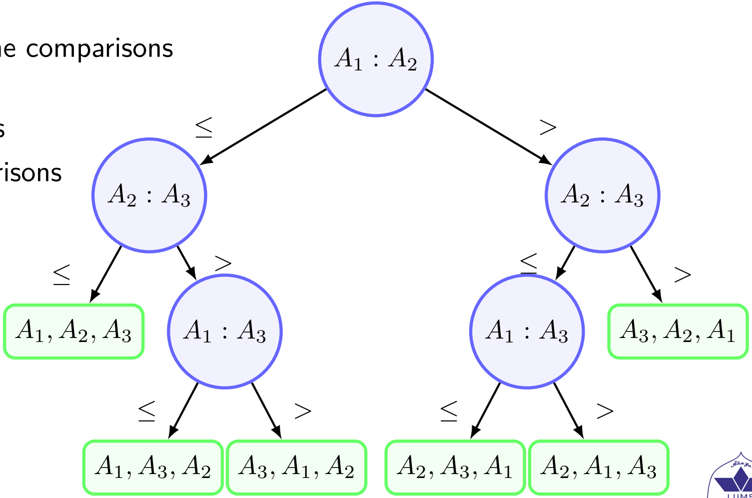
- Algorithm's behavior depends only on outcomes of comparisons

Key Idea

Any comparison-based sorting algorithm can be modeled as a **decision tree**.

Decision Tree for Sorting Three Elements

- The decision tree models the comparisons made by the algorithm
- Internal nodes: comparisons
- Edges: outcomes of comparisons
- Leaves: final sorted order



Key Observation

- Input elements are distinct
- There are $n!$ possible input permutations
- A correct sorting algorithm must:

distinguish all $n!$ permutations

- Therefore:

The decision tree must have **at least** $n!$ **leaves**

Decision Tree Height

- Each comparison gives at most 2 outcomes
- A binary tree of height h has at most:

$$2^h \text{ leaves}$$

- Since the tree must have $\geq n!$ leaves:

$$2^h \geq n!$$

- Taking logarithms:

$$h \geq \log_2(n!)$$

Worst-case number of comparisons $\geq \log_2(n!)$

Lower Bounding $\log(n!)$

Using Stirling's approximation:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Taking logarithms:

$$\log(n!) = \Theta(n \log n)$$

More precisely:

$$\log(n!) \geq n \log n - n$$

Conclusion

Any comparison-based sorting algorithm needs:

$$\Omega(n \log n) \text{ comparisons in the worst case}$$

Sorting Lower Bound Theorem

Theorem

Any comparison-based sorting algorithm on n distinct elements requires $\Omega(n \log n)$ comparisons in the worst case.

- Merge Sort: $\Theta(n \log n)$ optimal
- Heap Sort: $\Theta(n \log n)$ optimal
- QuickSort: $\Theta(n \log n)$ average, $\Theta(n^2)$ worst

Takeaway

To beat $n \log n$, you must **leave the comparison model**.

Escaping the Lower Bound

- Counting Sort
- Radix Sort
- Bucket Sort

Why do these work? Because they use information beyond comparisons.

Sorting Faster Than $O(n \log n)$?

- Comparison-based sorting has a lower bound: $\Omega(n \log n)$.
- But *not all sorting algorithms are created equal* aka non-comparison-based sorting.
- If keys come from a small range, we can do better.

Counting Sort: $O(n + k)$ **time**

Stable Sorting

Definition

A sorting algorithm is **stable** if it preserves the relative order of elements with equal keys.

Example

Input:	(2,a)	(1,b)	(2,c)	(1,d)
Stable sort:	(1,b)	(1,d)	(2,a)	(2,c)

- Elements with equal keys keep their original order.
- Required for Radix Sort to work correctly.

In-Place Sorting

Definition

A sorting algorithm is **in-place** if it uses only $O(1)$ extra memory beyond the input array.

Example

Array:

4	1	3	2

- Elements are rearranged within the same array.
- No auxiliary array proportional to input size.
- Examples: Insertion Sort, Selection Sort, Heap Sort.

Counting Sort: High-Level Idea

- Input: array $A[1..n]$ of integers.
- Assume each $A[i] \in \{0, 1, \ldots, k\}$.
- Count how many times each value appears.
- Use counts to place elements in sorted order.

Key difference: No comparisons between elements.

Counting Sort: Example

Input array

2	5	3	0	2	3	0	3

- $n = 8$
- Values lie in range $\{0, 1, 2, 3, 4, 5\}$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	0	0	0	0	0	0
index	0	1	2	3	4	5

Initialize $C[0..k] = 0$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	0	0	1	0	0	0
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	0	0	1	0	0	1
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	0	0	1	1	0	1
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3
C:	1	0	1	1	0	1		
index	0	1	2	3	4	5		

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:

2	5	3	0	2	3	0	3

C:

C:	1	0	2	1	0	1
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	1	0	2	2	0	1
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	2	0	2	2	0	1
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 1 — Counting Frequencies

Input:	2	5	3	0	2	3	0	3

C:	2	0	2	3	0	1
index	0	1	2	3	4	5

$$C[A[j]] \leftarrow C[A[j]] + 1$$

Counting Sort: Step 2 — Prefix Sums

Transform counts into positions

C:	2	0	2	3	0	1
Prefix:	2	2	4	7	7	8

- Prefix sum tells us:
- how many elements $\leq x$ exist.
- where value x should end in the output.

Counting Sort: Step 3 — Build Output

- Scan input from right to left.
- Place each element in its correct position.
- Decrement its count.

This preserves stability.

Counting Sort: Final Output

0	0	2	2	3	3	3	5

Sorted array

Counting Sort: Algorithm

```
CountingSort(A, k):
    C[0..k] = 0
    for j = 1 to n:
        C[A[j]]++

    for i = 1 to k:
        C[i] = C[i] + C[i-1]

    for j = n downto 1:
        B[C[A[j]]] = A[j]
        C[A[j]]--

    return B
```

Counting Sort: Running Time

- Counting frequencies: $O(n)$
- Prefix sums: $O(k)$
- Output construction: $O(n)$

Total Cost

$T(n, k) = O(n + k)$

- Linear time if $k = O(n)$.

When Is Counting Sort a Good Idea?

- Keys are integers from a small range.
- Stability matters (e.g., radix sort).
- Comparisons are expensive or meaningless.

Examples:

- Grades (e.g., 0-100)
- Characters (ASCII / Unicode blocks)

Counting Sort: Limitations

- Requires extra memory: $O(n + k)$
- Not comparison-based
- Inefficient if $k \gg n$

But: foundational building block for **Radix Sort**.

Radix Sort: Idea

- Sort numbers digit by digit.
- Use a **stable** sorting algorithm for each digit.
- Process digits from **least significant** to **most significant**.

Key Insight

Stability ensures earlier digit order is preserved.

Radix Sort: Example Input

Input:

170	45	75	90	802	24	2	66

Digits processed right-to-left (units, tens, hundreds)

Radix Sort: Pass 1 (Units Digit)

Input:	170	45	75	90	802	24	2	66
After units:	170	90	802	2	24	45	75	66

- Sorted by last digit using Counting Sort.
- Relative order of equal digits preserved.

Radix Sort: Pass 2 (Tens Digit)

After tens:	802	2	24	45	66	170	75	90

- Sorted by tens digit.
- Units-digit order remains intact.

Radix Sort: Pass 3 (Hundreds Digit)

Final output:

2	24	45	66	75	90	170	802

Fully sorted array

Why Does Radix Sort Work?

- Each pass sorts by one digit.
- Stability preserves ordering from previous passes.
- After processing all digits, numbers are fully sorted.

Invariant

After pass i, the array is sorted by the last i digits.

Radix Sort: Running Time

- Let:
  - n = number of elements
  - d = number of digits
  - k = range of each digit
- Each digit pass uses Counting Sort: $O(n + k)$

Total Time

$$T(n) = O(d(n + k))$$

- Linear if d and k are constants.

Counting Sort vs Radix Sort

	Counting Sort	Radix Sort
Comparison-based	No	No
Stable	Yes	Yes
Handles large keys	No	Yes
Uses Counting Sort	—	Yes

Radix Sort $=$ Counting Sort $\times$ Digits