• Captions
• Google sheet
• Assignment
• Loop invariants
• Merge sort
• Record
Closest Pair Algorithm

https://cs.pomona.edu/classes/cs140/
Notes

• Assignment due tomorrow
• Checkpoint 1 next Wednesday
• Slack, checked out pinned messages
Outline

Topics and Learning Objectives

• Learn more about Divide and Conquer paradigm
• Learn about the closest-pair problem and its $O(n \log n)$ algorithm
  • Gain experience analyzing the run time of algorithms
  • Gain experience proving the correctness of algorithms

Exercise

• Closest Pair
Closest Pair Problem

• **Input:** $P$, a set of $n$ points that lie in a (two-dimensional) plane

• **Output:** a pair of points $(p, q)$ that are the “closest”
  • Distance is measured using Euclidean distance:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$
Closest Pair Problem

- What is the brute force method for this search?
- What is the asymptotic running time of the brute force method?

Can we do better than $O(n^2)$?
One-dimensional closest pair

How would you find the closest two points?
• Sort by position: $O(n \lg n)$
• Return the closest two using a linear scan: $O(n)$
• Total time: $O(n \lg n) + O(n) = O(n \lg n)$

Any problems using this approach for the two-dimensional case?
• How do you sort the points?
• Sorting does not generalize to higher dimensions!
1. Which two are closest on the y-axis?
1. Which two are closest on the y-axis?
1. Which two are closest on the y-axis?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?
Closet Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)
   1. Sort by x-coordinate
   2. Sort other by y-coordinate

\[ O(n \lg n) \]
P : [p0(1,10), p1(2,8), p2(7,3), p3(5,7), p4(8,4), p5(3,5), p6(10,9), p7(9,1)]

Sorted by x coordinate

Px : [p0(1,10), p1(2,8), p5(3,5), p3(5,7), p2(7,3), p4(8,4), p7(9,1), p6(10,9)]

Sorted by y coordinate

Py : [p7(9,1), p2(7,3), p4(8,4), p5(3,5), p3(5,7), p1(2,8), p6(10,9), p0(1,10)]
Closet Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)
   1. Sort by x-coordinate
   2. Sort other by y-coordinate

- Can we still end up with a $O(n \log n)$ algorithm for finding the closest pair?
- Does the closeness of two points on one axis matter?
1. FUNCTION FindClosestPair(points)
2.    points_x = copy_and_sort_by_x(points)
3.    points_y = copy_and_sort_by_y(points)
4.    RETURN ClosestPair(points_x, points_y)
Closet Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)
   1. Sort by x-coordinate
   2. Sort other by y-coordinate

   • Can we still end up with a $O(n \lg n)$ algorithm for finding the closest pair?
   • Does the closeness of two points on one axis matter?

2. Apply the Divide-and-Conquer method
Divide-and-Conquer

1. **DIVIDE** into smaller subproblems
2. **CONQUER** the subproblems via recursive calls
3. **COMBINE** solutions from the subproblems

• How would you divide the problems?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?

4. How would you divide the search space?
1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?

4. How would you divide the search space?

This is not the average x-value.
1. **FUNCTION** ClosestPair(px, py)
2. \[ n = px.\text{length} \]
3. **IF** \( n == 2 \)
4. \[ \text{RETURN} \ px[0], px[1], \text{dist}(px[0], px[1]) \]
5. 
6. 
7. 
8. \[ \text{pl, ql, dl = ClosestPair(left\_px, left\_py)} \]
9. 
10. 
11. 
12. \[ \text{pr, qr, dr = ClosestPair(right\_px, right\_py)} \]

How do we create these arrays?
1. How do we create left_px?
2. How do we create right_px?
3. How do we create left_py?
4. How do we create right_py?
1. **FUNCTION** ClosestPair(px, py)
2. \[n = \text{px}.\text{length}\]
3. **IF** \[n == 2\]
4. **RETURN** px[0], px[1], dist(px[0], px[1])
5. \[\text{left}_\text{px} = \text{px}[0 ..< \text{n}/2]\]
6. \[\text{left}_\text{py} = [p \text{ FOR } p \text{ IN } \text{py} \text{ IF } p.x < \text{px}[\text{n}/2].x]\]
7. \[\text{pl}, \text{ql}, \text{dl} = \text{ClosestPair(left}_\text{px}, \text{left}_\text{py})\]
8. \[\text{right}_\text{px} = \text{px}[\text{n}/2 ..< \text{n}]\]
9. \[\text{right}_\text{py} = [p \text{ FOR } p \text{ IN } \text{py} \text{ IF } p.x \geq \text{px}[\text{n}/2].x]\]
10. \[\text{pr}, \text{qr}, \text{dr} = \text{ClosestPair(right}_\text{px}, \text{right}_\text{py})\]

**Median x value**
Any problems with our current approach?
FUNCTION ClosestPair(px, py)

n = px.length

IF n == 2
    RETURN px[0], px[1], dist(px[0], px[1])

left_px = px[0 ..< n//2]
left_py = [p FOR p IN py IF p.x < px[n//2].x]
pl, ql, dl = ClosestPair(left_px, left_py)

right_px = px[n//2 ..< n]
right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
pr, qr, dr = ClosestPair(right_px, right_py)

d = min(dl, dr)
ps, qs, ds = ClosestSplitPair(px, py, d)

RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)

What time complexity does this process need such that the overall algorithm runs in O(n lg n)?

Hint: think about Merge Sort.
Exercise Question 1

Running time needed for ClosestSplitPair?
Merge Sort and It’s Recurrence Equation
FUNCTION RecursiveFunction(some_input)
    IF base_case:
        # Usually O(1)
        RETURN base_case_work(some_input)

    # Two recursive calls, each with half the data
    one = RecursiveFunction(some_input.first_half)
    two = RecursiveFunction(some_input.second_half)

    # Combine results from recursive calls (usually O(n))
    one_and_two = Combine(one, two)

    RETURN one_and_two
1. **FUNCTION** ClosestPair(px, py)
2. \[ n = \text{px}.\text{length} \]
3. \[ \text{IF } n == 2 \]
4. \[ \text{RETURN } \text{px}[0], \text{px}[1], \text{dist}(\text{px}[0], \text{px}[1]) \]
5. \[ \]
6. \[ \text{left\_px} = \text{px}[0..<n//2] \]
7. \[ \text{left\_py} = [p \text{ FOR } p \text{ IN } \text{py} \text{ IF } p.x < \text{px}[n//2].x] \]
8. \[ \text{pl}, \text{ql}, \text{dl} = \text{ClosestPair(} \text{left\_px}, \text{left\_py}) \]
9. \[ \]
10. \[ \text{right\_px} = \text{px}[n//2..<n] \]
11. \[ \text{right\_py} = [p \text{ FOR } p \text{ IN } \text{py} \text{ IF } p.x \geq \text{px}[n//2].x] \]
12. \[ \text{pr}, \text{qr}, \text{dr} = \text{ClosestPair(} \text{right\_px}, \text{right\_py}) \]
13. \[ \]
14. \[ d = \text{min}(\text{dl}, \text{dr}) \]
15. \[ \text{ps}, \text{qs}, \text{ds} = \text{ClosestSplitPair(} \text{px}, \text{py}, d) \]
16. \[ \]
17. \[ \text{RETURN } \text{Closest(} \text{pl}, \text{ql}, \text{dl}, \text{pr}, \text{qr}, \text{dr}, \text{ps}, \text{qs}, \text{ds}) \]
Key Idea

• In ClosestSplitPair we only need to check for pairs that are closer than those found in the recursive calls to ClosestPair.

• This is easier (faster) than trying to find the closest split pair without any extra information!

\[ \delta = \min[d(p_l, q_l), d(p_r, q_r)] \]
FUNCTION ClosestSplitPair(px, py, d)
    n = px.length
    x_median = px[n//2].x
    middle_py = [p FOR p IN py IF x_median - d < p.x < x_median + d]

    closest_d = INFINITY, closest_p = closest_q = NONE
    FOR i IN [0 ..< middle_py.length - 1]
        FOR j IN [1 ..= min(7, middle_py.length - i)]
            p = middle_py[i], q = middle_py[i + j]
            IF dist(p, q) < closest_d
                closest_d = dist(p, q)
                closest_p = p, closest_q = q

    RETURN closest_p, closest_q, closest_d
FUNCTION ClosestSplitPair(px, py, d)
  n = px.length
  x_median = px[n//2].x
  middle_py = [p FOR p IN py
                IF x_median - d < p.x < x_median + d]

  closest_d = INFINITY, closest_p = closest_q = NONE
  FOR i IN [0 .. middle_py.length - 1]
      FOR j IN [1 ..= min(7, middle_py.length - i)]
          p = middle_py[i], q = middle_py[i + j]
          IF dist(p, q) < closest_d
              closest_d = dist(p, q)
              closest_p = p, closest_q = q

  RETURN closest_p, closest_q, closest_d
Exercise Question 2
Running Time of Nested For-Loops
FUNCTION ClosestSplitPair(px, py, d)
  
  n = px.length
  x_median = px[n//2].x
  
  middle_py = [p FOR p IN py
                IF x_median - d < p.x < x_median + d]

  closest_d = INFINITY, closest_p = closest_q = NONE
  
  FOR i IN [0..middle_py.length - 1]
    FOR j IN [1..min(7, middle_py.length - i)]
      p = middle_py[i], q = middle_py[i + j]
      IF dist(p, q) < closest_d
        closest_d = dist(p, q)
        closest_p = p, closest_q = q

  RETURN closest_p, closest_q, closest_d
Claim

Let \( p \in \text{left}, q \in \text{right} \) be a split pair with \( d(p, q) < d \)
Then

A. \( p \) and \( q \) \( \in \text{middle}_py \), and
B. \( p \) and \( q \) are at most 7 positions apart in \text{middle}_py

If the claim is true:

Corollary 1: If the closest pair of \( P \) is in a split pair, then our \text{ClosestSplitPair} procedure finds it.

Corollary 2: \text{ClosestPair} is correct and runs in \( O(n \log n) \) same recursion tree as merge sort
Proof—Part A

Let $p \in \text{left}$, $q \in \text{right}$ be a split pair with $d(p, q) < d$. Then

A. $p$ and $q \in \text{middle}_py$, and

If $p = (x_1, y_1) \in \text{left AND } q = (x_2, y_2) \in \text{right AND } d(p, q) < d$ Then

$x_{\text{median}} - d < x_1 \leq x_{\text{median}} \text{ and } x_{\text{median}} \leq x_2 < x_{\text{median}} + d$

Otherwise, $p$ and $q$ would not be the closest pair with $d(p, q) < d$
Proof—Part A

Let \( p \in \text{left}, \ q \in \text{right} \) be a split pair with \( d(p, q) < d \).

Then

A. \( p \) and \( q \) ∈ \( \text{middle}_\text{py} \), and

If \( p = (x_1, y_1) \in \text{left AND} \ q = (x_2, y_2) \in \text{right AND} \ d(p, q) < d \)

Then

\[
\begin{align*}
    x_{\text{median}} - d &< x_1 \leq x_{\text{median}} \quad \text{and} \\
    x_{\text{median}} &\leq x_2 < x_{\text{median}} + d
\end{align*}
\]

Otherwise, \( p \) and \( q \) would not be the closest pair with \( d(p, q) < d \).
Claim

Let \( p \in \text{left}, \ q \in \text{right} \) be a split pair with \( d(p, q) < d \) Then

A. \( p \) and \( q \) \in \text{middle}_p y, \) and

B. \( p \) and \( q \) are at most 7 positions apart in \text{middle}_p y

If the claim is true:

Corollary 1: If the closest pair of \( P \) is in a split pair, then our \text{ClosestSplitPair} procedure finds it.

Corollary 2: \text{ClosestPair} is correct and runs in \( O(n \ lg \ n) \) same recursion tree as merge sort
The diagram illustrates the concept of middle.py, where points p1, q1, d1, and pr, qr, dr are distributed around the center. The distance d between these points and the x_median is highlighted, emphasizing the importance of this measure in the context of the middle.py function.
Proof—Part B

$p$ and $q$ are at most $7$ positions apart in $\text{middle}_\text{py}$

How many other points can possibly be in this area?
Proof—Part B

p and q are at most 7 positions apart in middle_py

Lemma 1: All points of middle_py with a y-coordinate between those of p and q lie within those 8 boxes.

Proof:
1. First, recall that the y-coordinate of p, q differs by less than d.
2. Second, by definition of middle_py, all have an x-coordinate between x_median += δ.
Proof—Part B

p and q are at most 7 positions apart in middle_py

Lemma 1: All points of middle_py with a y-coordinate between those of p and q lie within those 8 boxes.

Lemma 2: At most one point of P can be in each box.

Proof: By contradiction. Suppose points a and b lie in the same box. Then

1. a and b are either both in L or both in R
2. \( d(a, b) \leq d/2 \sqrt{2} < d \)

This is a contradiction! How did we define d?
Max distance within box is $d / \sqrt{2}$
Claim

Let \( p \in \textit{left}, \ q \in \textit{right} \) be a split pair with \( d(p, \ q) < d \) Then

A. \( p \) and \( q \) \( \in \textit{middle}_\textit{py} \), and
B. \( p \) and \( q \) are at most 7 positions apart in \textit{middle}_\textit{py}

If the claim is true:

Corollary 1: If the closest pair of \( P \) is in a split pair, then \texttt{ClosestSplitPair} procedure finds it.

Corollary 2: \texttt{ClosestPair} is correct and runs in \( O(n \log n) \) same recursion tree as merge sort
Closest Pair

1. Copy P and sort one copy by x and the other copy by y in $O(n \lg n)$
2. Divide P into a left and right in $O(n)$
3. Conquer by recursively searching left and right
4. Look for the closest pair in middle_py in $O(n)$
   - Must filter by x
   - And scan through middle_py by looking at adjacent points
Closest Split Pair
FUNCTION ClosestPair(px, py)

n = px.length

IF n == 2

RETURN px[0], px[1], dist(px[0], px[1])

left_px = px[0..<n//2]

left_py = [p FOR p IN py IF p.x < px[n//2].x]

pl, ql, dl = ClosestPair(left_px, left_py)

right_px = px[n//2..<n]

right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]

pr, qr, dr = ClosestPair(right_px, right_py)

d = min(dl, dr)

ps, qs, ds = ClosestSplitPair(px, py, d)

RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)
FUNCTION MergeSort(array)

n = array.length

IF n == 1
    RETURN array

left_sorted = MergeSort(array[0..<n//2])
right_sorted = MergeSort(array[n//2..<n])

array_sorted = Merge(left_sorted, right_sorted)

RETURN array_sorted

T(n) = 2 T(n/2) + O(n)
    = O(n lg n)
FUNCTION RecursiveFunction(some_input):

IF base_case:
    # Usually O(1)
    RETURN base_case_work(some_input)

    # Two recursive calls, each with half the data
    one = RecursiveFunction(some_input.first_half)
    two = RecursiveFunction(some_input.second_half)

    # Combine results from recursive calls (usually O(n))
    one_and_two = Combine(one, two)

RETURN one_and_two

T(n) = 2 T(n/2) + O(n) = O(n lg n)