演算法與程式解題實務

Mong-Jen Kao (高孟駿)

Monday 18:30 – 21:20

The Divide-and-Conquer Paradigm

- The divide-and-conquer is a *powerful technique* commonly used *for designing efficient algorithms*.

  It consists of three steps.

  - ***Divide*** –
    to divide the problem instance into sub-instances of smaller sizes.
  - ***Conquer*** – to conquer the sub-instances separately.
  - ***Merge*** –
    to merge the answer of the sub-instances for the original instance.

Divide-and-Conquer

– More Examples

More on recursion for problem solving.

Example 1.

Fast Exponentiation

Computing the power of a number (matrix) fast.

Fast Exponentiation

- Given a number a and an integer $N > 0$, compute a^N.
  - Naive approach - $\Theta(N)$ time.
  - Divide and Conquer - $\Theta(\log N)$ time.

<u>At most one recursion</u> should be made here.

$$a^N = \begin{cases} 1, & \text{if } N = 0, \\ a^{N/2} \cdot a^{N/2}, & \text{if } N \text{ is even}, N > 0, \\ a^{N/2} \cdot a^{N/2} \cdot a, & \text{if } N \text{ is odd}, N > 0. \end{cases}$$

$T(n) = T(n/2) + \Theta(1)$ and $T(n) = \Theta(\log n)$.

Application – Fibonacci Numbers

- For any $n \geq 0$, the n-th Fibonacci number F_n is defined as follows.

$$F_n = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ F_{n-1} + F_{n-2}, & \text{if } n \geq 2. \end{cases}$$

  - Naive approach - $\Theta(n)$ time.
  - We can observe that, for any $n \geq 2$,

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}.$$

Hence, via fast exponentiation, this can be computed in $\Theta(\log n)$ time.

Example 2.

Maximum Sum Segment

The Maximum Sum Segment Problem

- Given a sequence of numbers $a_1, a_2, \ldots, a_n$, find a segment $[\ell, r] \subseteq [1, n]$ such that

$$\sum_{\ell \leq i \leq r} a_i$$

is maximized.

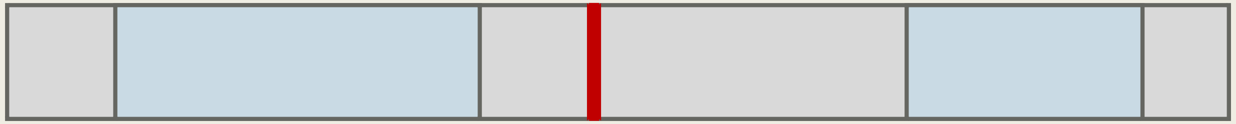
Has a maximum sum of 6.

-2	1	-3	4	-1	2	1	-5	4	-6	2	3	-2	1

Naïve approach takes $\Theta(n^2)$ time.

The Maximum Sum Segment Problem

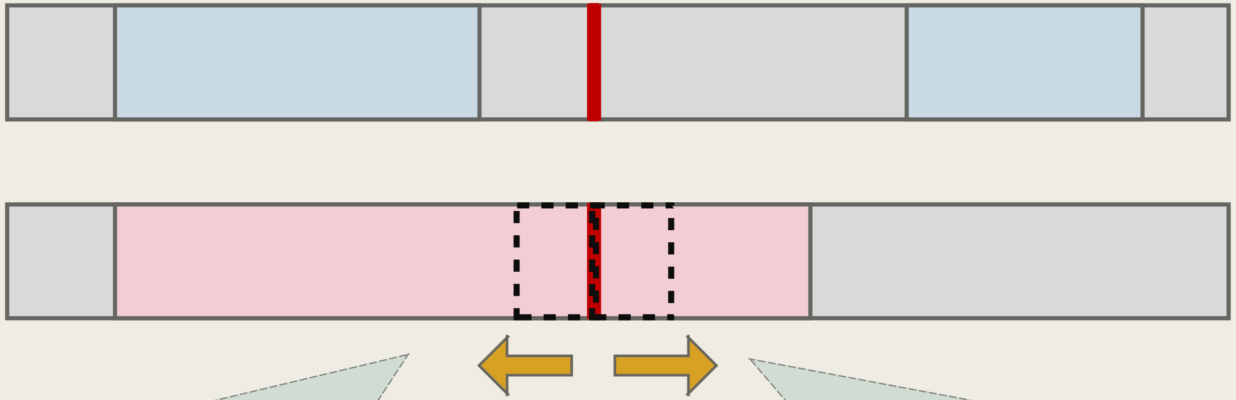
- This problem can be solved via divide-and-conquer in $\Theta(n \log n)$ time.
  - ***Divide*** –
    Divide the current instance into two halves.
  - ***Conquer*** –
    Recursively solve the two sub-instances to obtain the best segment for them.



- This problem can be solved via divide-and-conquer in $\Theta(n \log n)$ time.
  - ***<u>Merge</u>* –**

    The optimal segment is the best of the following segments.
    - The best segment for the two sub-instances.
    - The best segment that spans over the two sub-instances.

Can be computed in $\Theta(n)$ time.



Best segment ***that ends*** at mid.

Best segment that ***starts from*** mid+1.

The Maximum Sum Segment Problem

- Given a sequence of numbers $a_1, a_2, \dots, a_n$,
  find a segment $[\ell, r] \subseteq [1, n]$ such that $\sum_{\ell \le i \le r} a_i$ is maximized.

- This problem can be solved via divide-and-conquer in $\Theta(n \log n)$ time.

  – Can we do better than $\Theta(n \log n)$?

  – **Yes**.

    With a cleaver observation, we can do it in $\Theta(n)$ time.

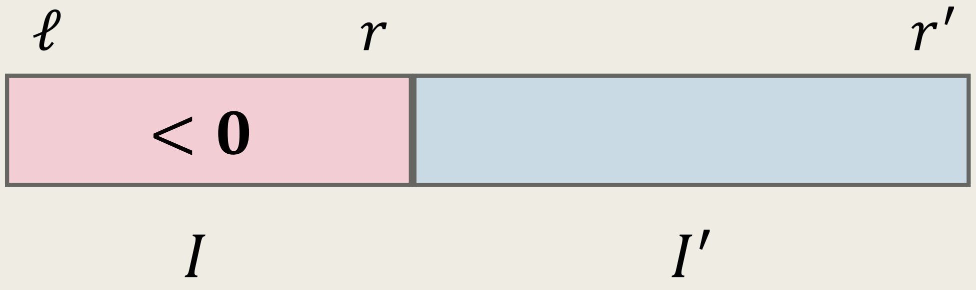
Maximum Sum Segment in $O(n)$ Time

- Let $S(I) \coloneqq \sum_{i \in I} a_i$ denote the sum of segment I.

- <u>*An Observation*</u>.

  If $I = [\ell, r]$ is a segment with $S(I) < 0$, then for any $r' > r$, we always have that

$$S(\ [\ell, r']\ ) < S(\ [r+1, r']\ ).$$



I' is always better than $I \cup I'$.

- Consider the following algorithm.

- MaximumSumSegment($A[\,1, 2, \dots, n\,]$)

A. best_sum ← 0.
   current_sum ← 0.

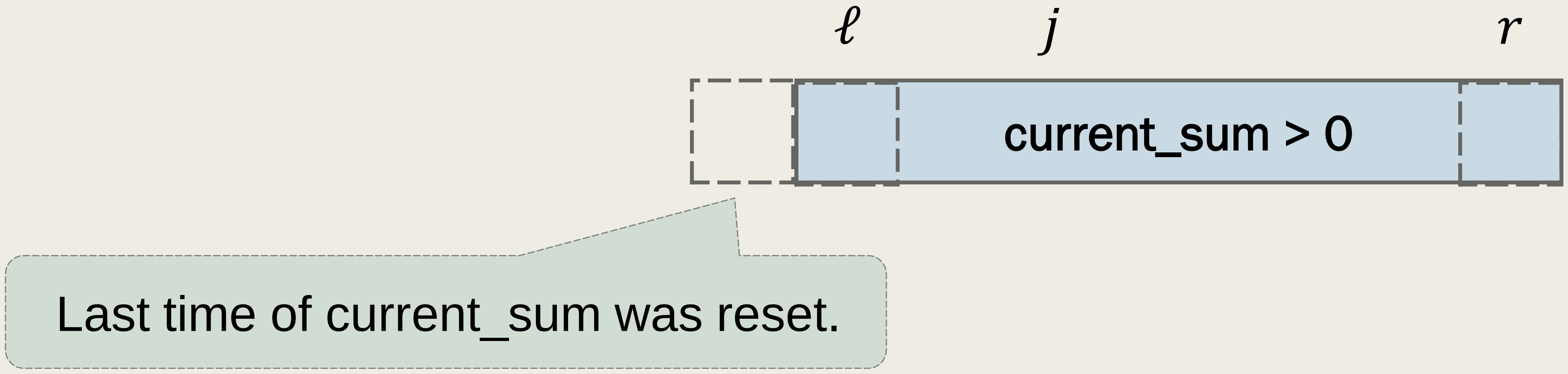
B. For i = 1 to n, do the following.

   a) current_sum ← max(0, current_sum + a_i).

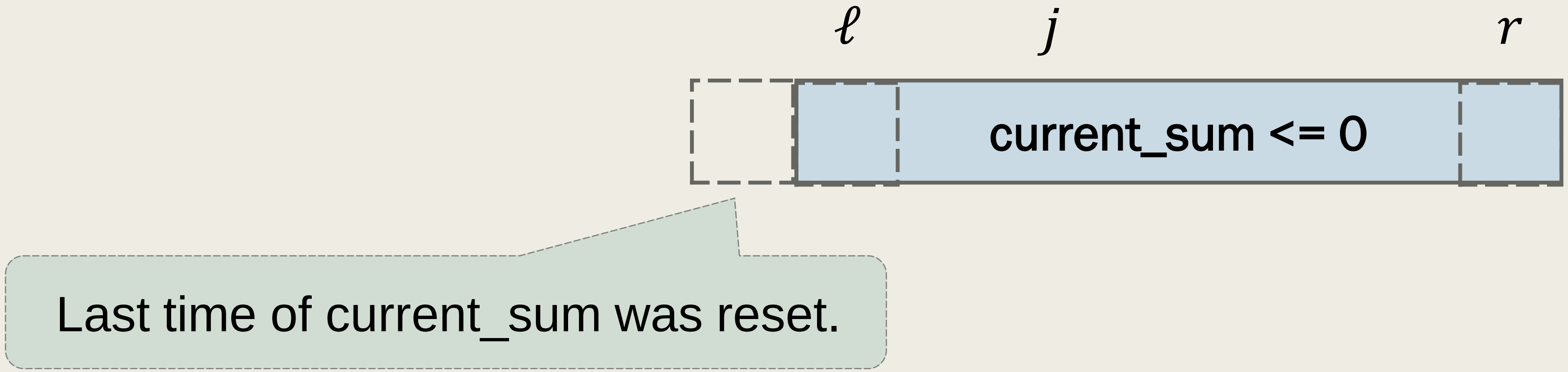
   b) best_sum ← max(current_sum, best_sum).

C. Output best_sum.

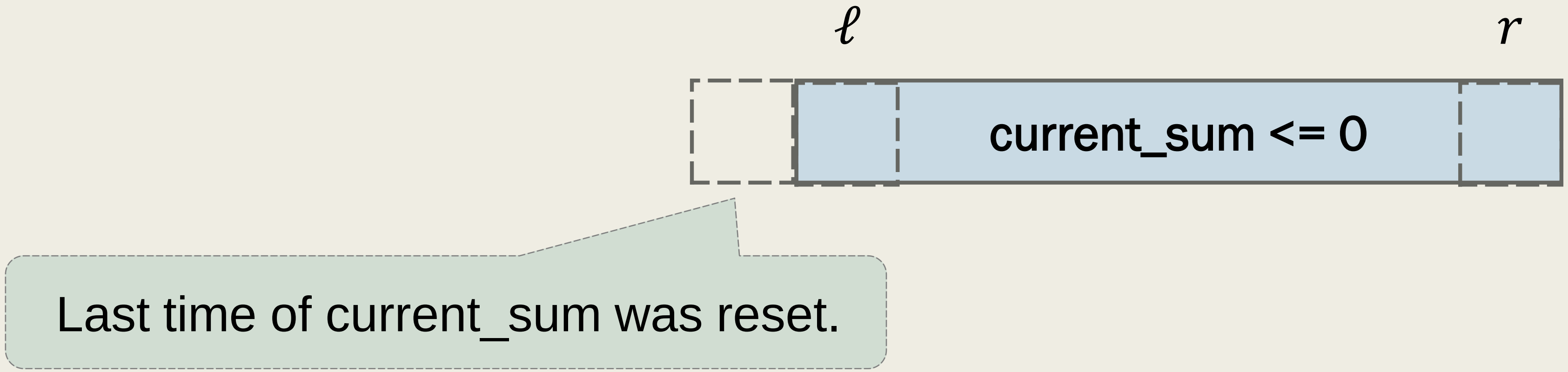
This algorithm can be modified to output the index of the segment.



- For any $\ell \leq j \leq r$, we have $S([\ell, j]) > 0$.
  - This implies that $S([j+1, r]) < S([\ell, r])$.
  - If a segment starts at $j+1$ and contains r, then ***<u>extending the left-end</u>*** to ℓ will strictly increase its sum.



- Suppose that current_sum was reset at $a_{\ell-1}$ and a_r, and not in-between.
  - Then, for any $\ell \le j < r$, we have $S([j+1, r]) < S([\ell, r]) \le 0$.
  - If a segment starts at $j+1$ and contains r, then <u>***changing its left-end***</u> to $r+1$ never decreases its sum.



- Let $t_1 = 0, t_2, \dots, t_k = n + 1$ be the set of indexes for which current_sum was reset and $[\ell, r]$ be a maximum sum segment.
  - Then, we have

$$t_i + 1 = \ell \le r \le t_{i+1}$$

    for some $1 \le i < k$.

- Hence, the algorithm produces the maximum sum segment.

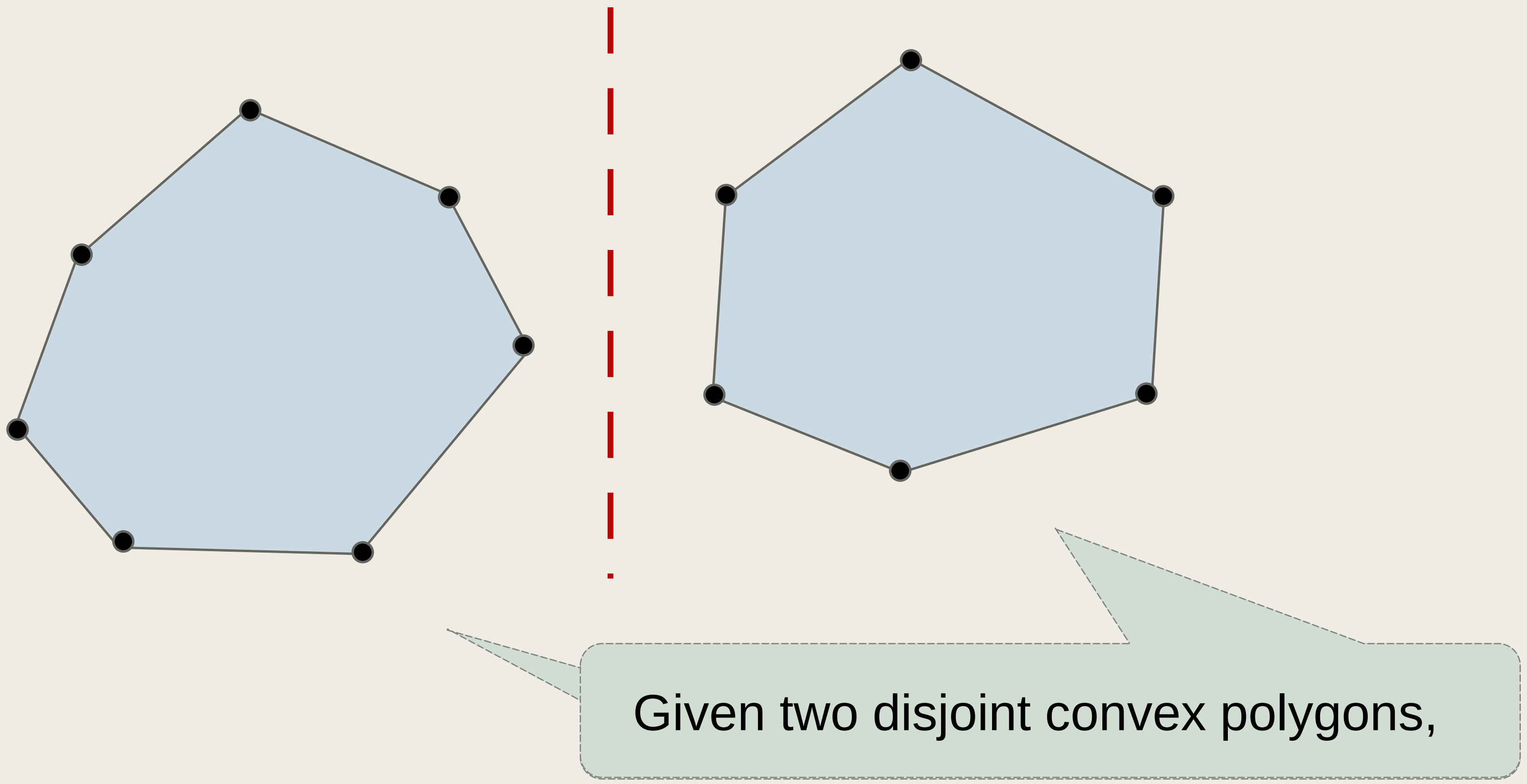
Example 3.

Convex Hull (revisited)

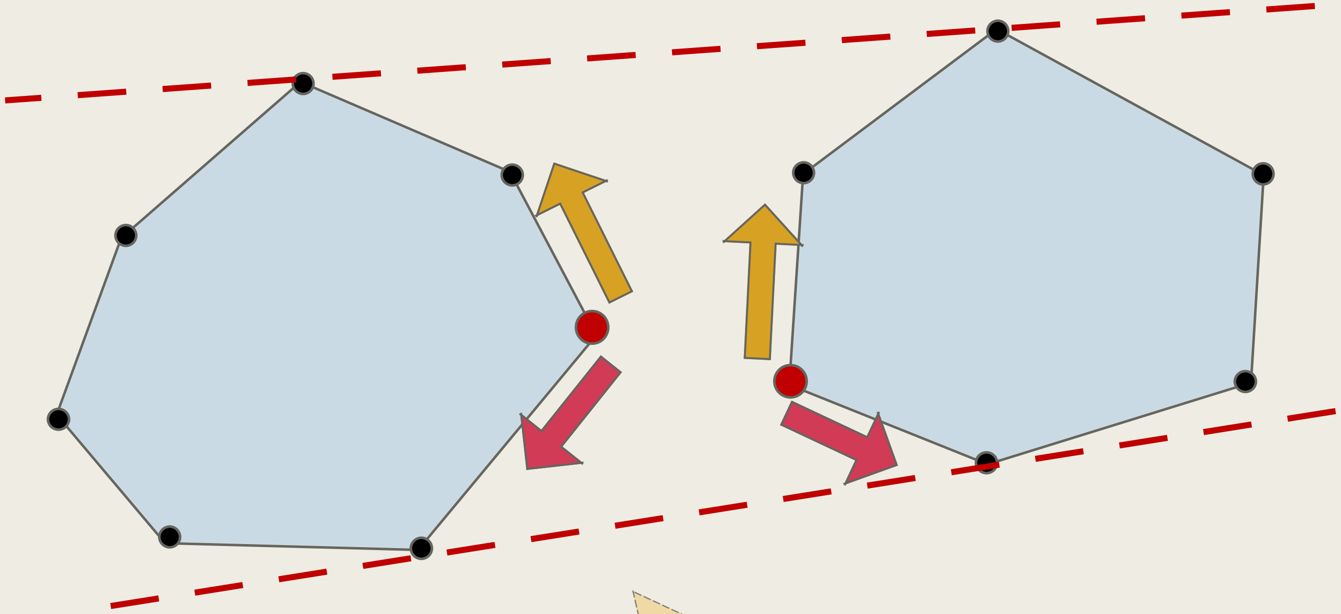
Computing the Convex Hull via divide and conquer.

Convex Hull

- By the following property, the convex hull problem can be solved by divide-and-conquer technique.

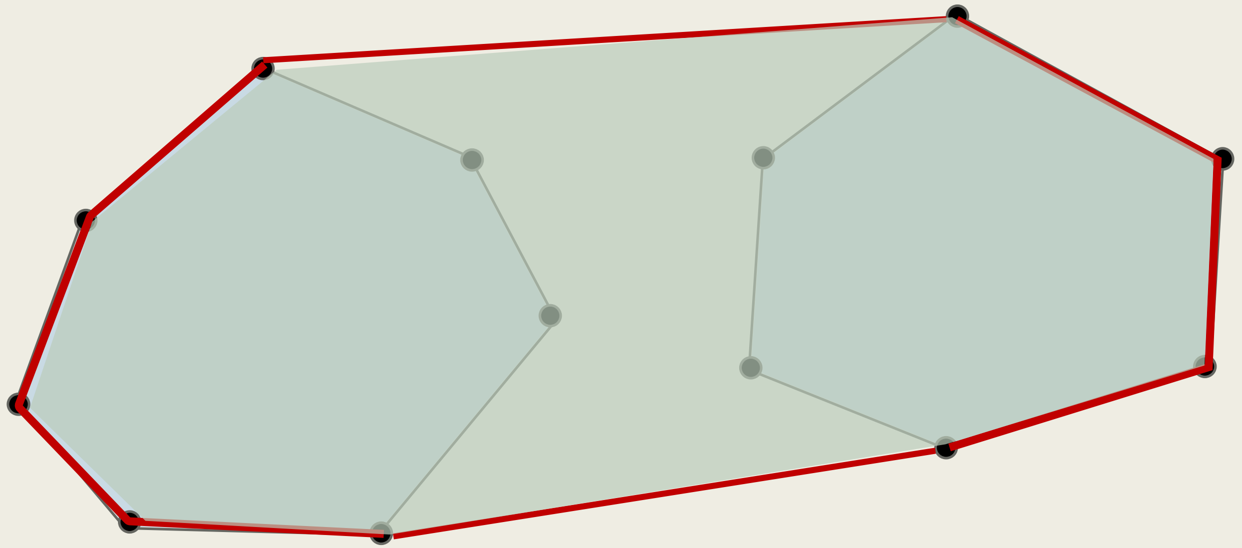


- By the following property, the convex hull problem can be solved by divide-and-conquer technique.



The common tangent lines can be computed by <u>*two-pointer method*</u> in $O(n)$ time.

- By the following property, the convex hull problem can be solved by divide-and-conquer technique.



Then the two convex hulls can be merged.

Convex Hull

- By the above property, the convex hull problem can be solved by divide-and-conquer technique in $\Theta(n \log n)$ time.

  1. Divide the points into two halves according to their x-coordinates.
  2. Recursively compute the convex hulls for the two sub-instances.
  3. Compute the common tangent points and merge the two convex hulls.

- ***Q:*** ***Can we do it faster***, say, in $o(n \log n)$?
  - The answer, however, is no.

Sorting $\propto$ (reducible to) Convex Hull

- ***Q:*** Can we do it faster, say, in $o(n \log n)$?
  - The answer, however, is no.

- We will show that, *sorting is reducible to convex hull*.
  - That is, an algorithm for computing convex hull can be used for sorting as well.
  - Hence, if convex hull can be done in $o(n \log n)$ time, then so is sorting.

Sorting $\propto$ (reducible to) Convex Hull

- We will show that, *<u>sorting</u> is reducible to <u>convex hull</u>*.
  - Given n numbers $a_1, a_2, \ldots, a_n$ to be sorted,
    we construct in $O(n)$ time n points

    $$p_1 = (a_1, a_1^2), p_2 = (a_2, a_2^2), \ldots, p_n = (a_n, a_n^2).$$

  - Since the curve $y = x^2$ is convex,
    all of $p_1, \ldots, p_n$ will be vertices of their convex hull.
  - Hence, traversing the convex hull of $p_1, \ldots, p_n$ will give us
    the sorted order of $a_1, \ldots, a_n$ in $O(n)$ time.

Example 4.

Finding Closest Pair

Computing the closest pair for a set of 2-D points.

Closet Pair for 2-D Points

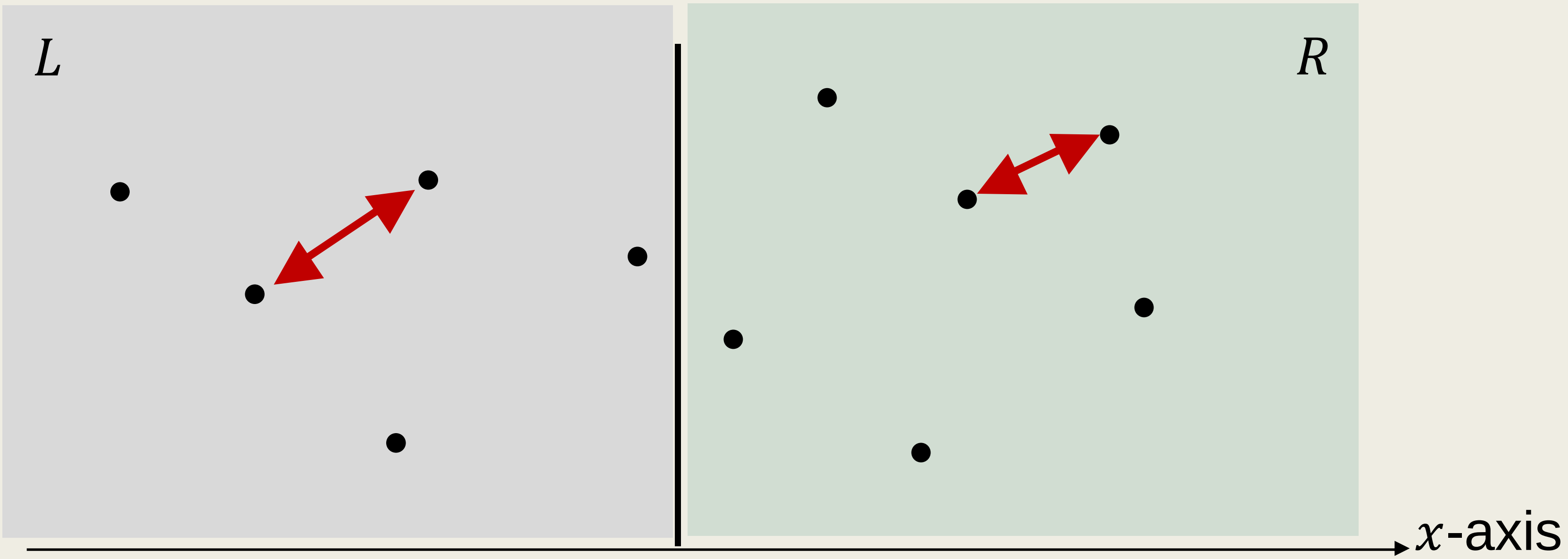
- Given a set of points $p_1, p_2, \ldots, p_n \in \mathbb{R}^2$, find the pair (i, j) with $1 \leq i < j \leq n$ such that

$$d(p_i, p_j) = \min_{1 \leq k < \ell \leq n} d(p_k, p_\ell) \ .$$

  - With a naïve approach,
    the closest pair can be computed in $O(n^2)$ time.
  - In the following,
    we show that this can be computed in $O(n \log n)$ time.

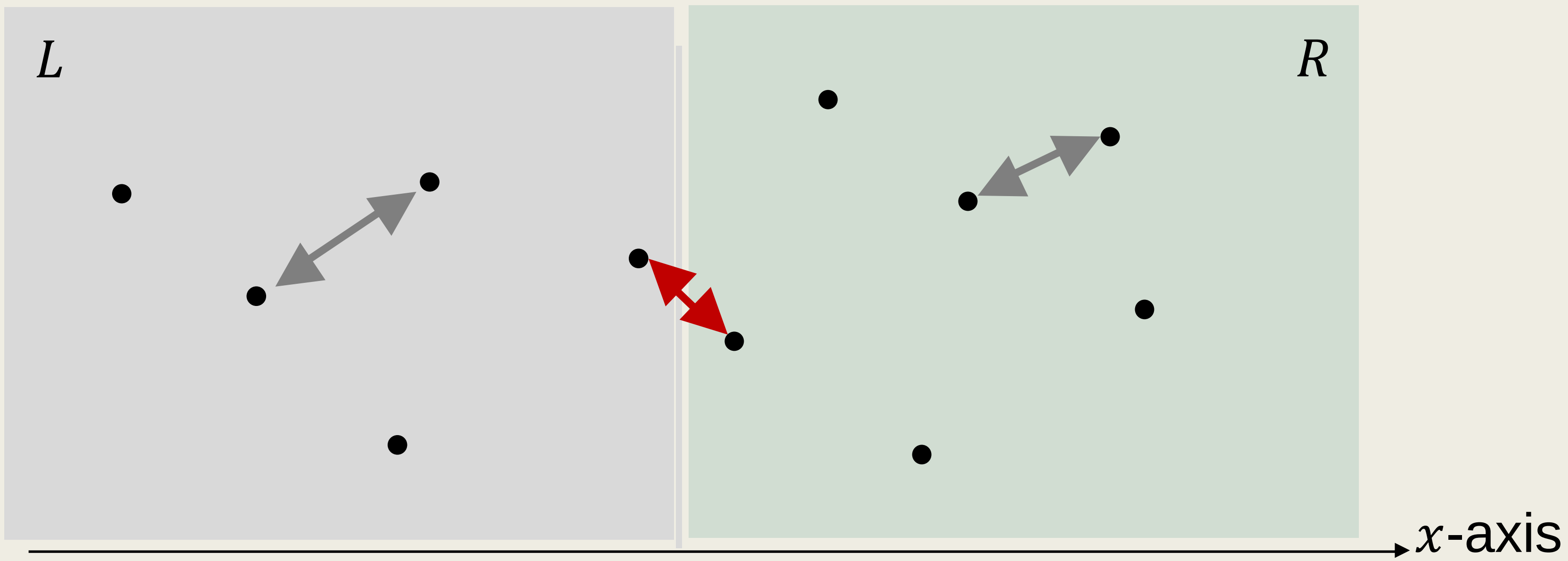
Closet Pair for 2-D Points

- Partition the given points into two equal-sized subsets L and R according to their x-coordinates.
  - There are three cases for a closest pair to reside.

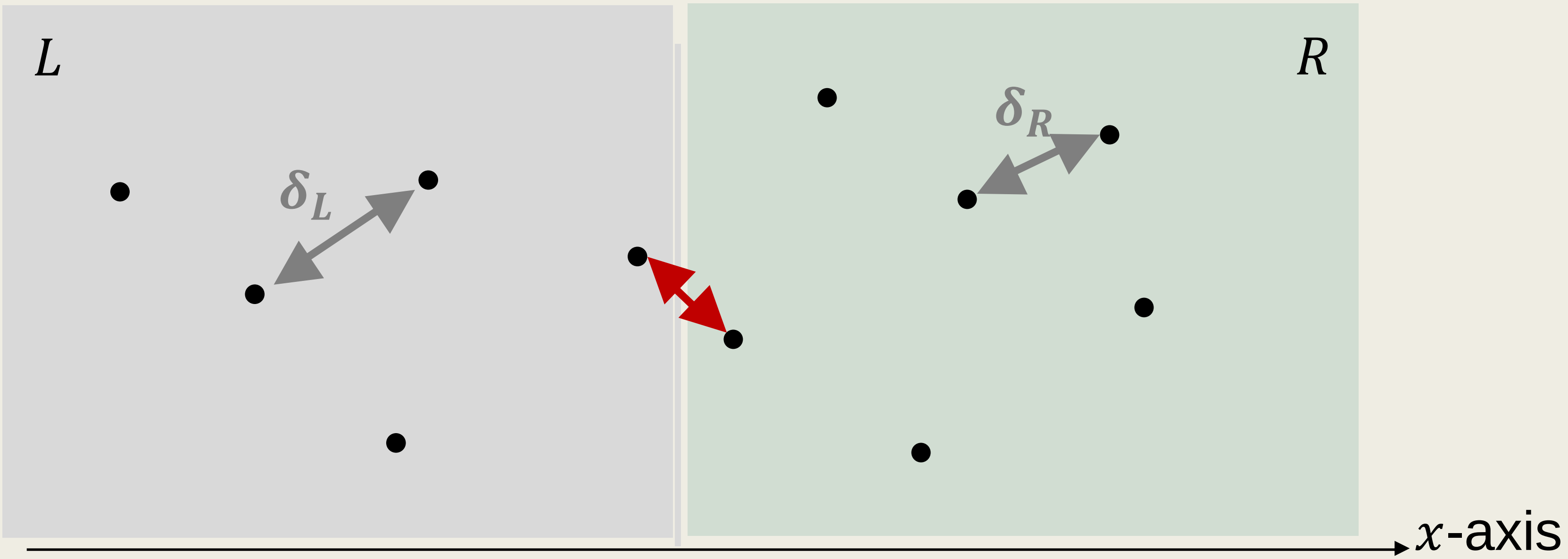


Closet Pair for 2-D Points

- Partition the given points into two equal-sized subsets L and R according to their x-coordinates.
  - There are three cases for a closest pair to reside.



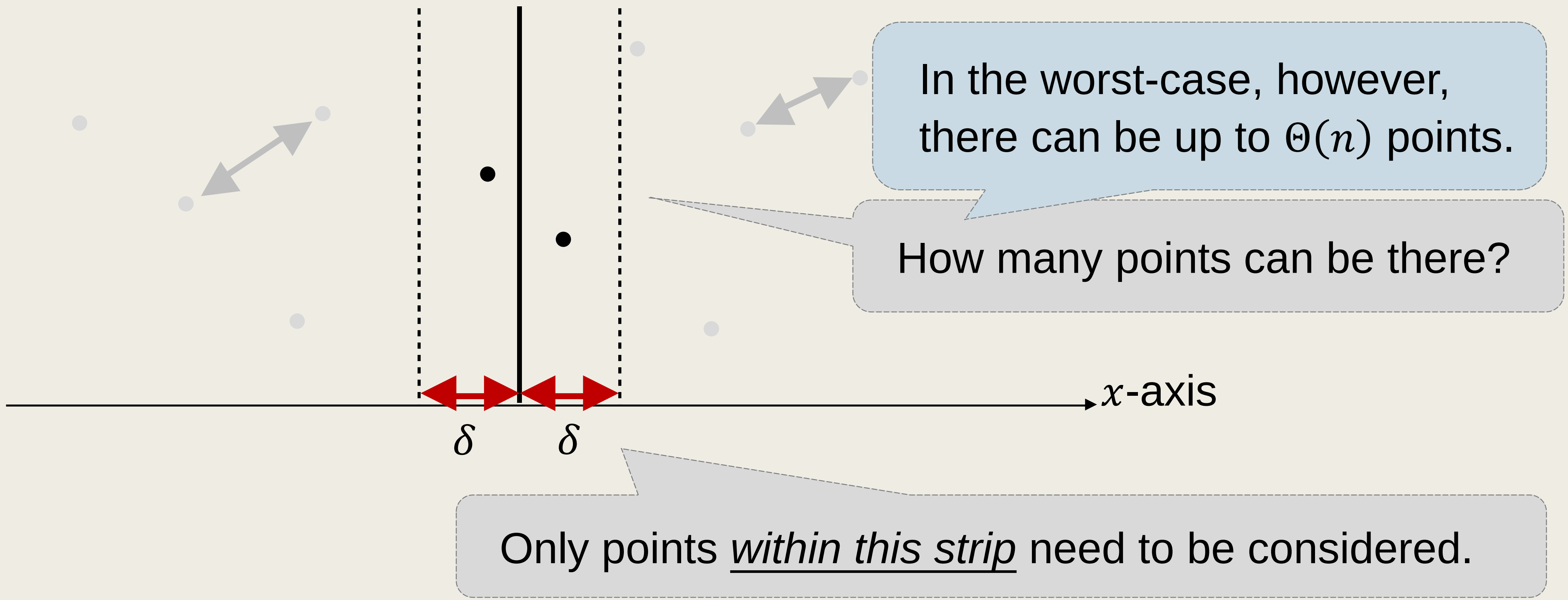
- There are three cases for a closest pair to reside.
  - The closest pairs for L and R can be computed recursively.
    - Let $\delta \coloneqq \min(\delta_L, \delta_R)$.
  - How can we compute the closest pair between L and R fast?



- Let $\delta := \min(\delta_L, \delta_R)$.

Observation 1

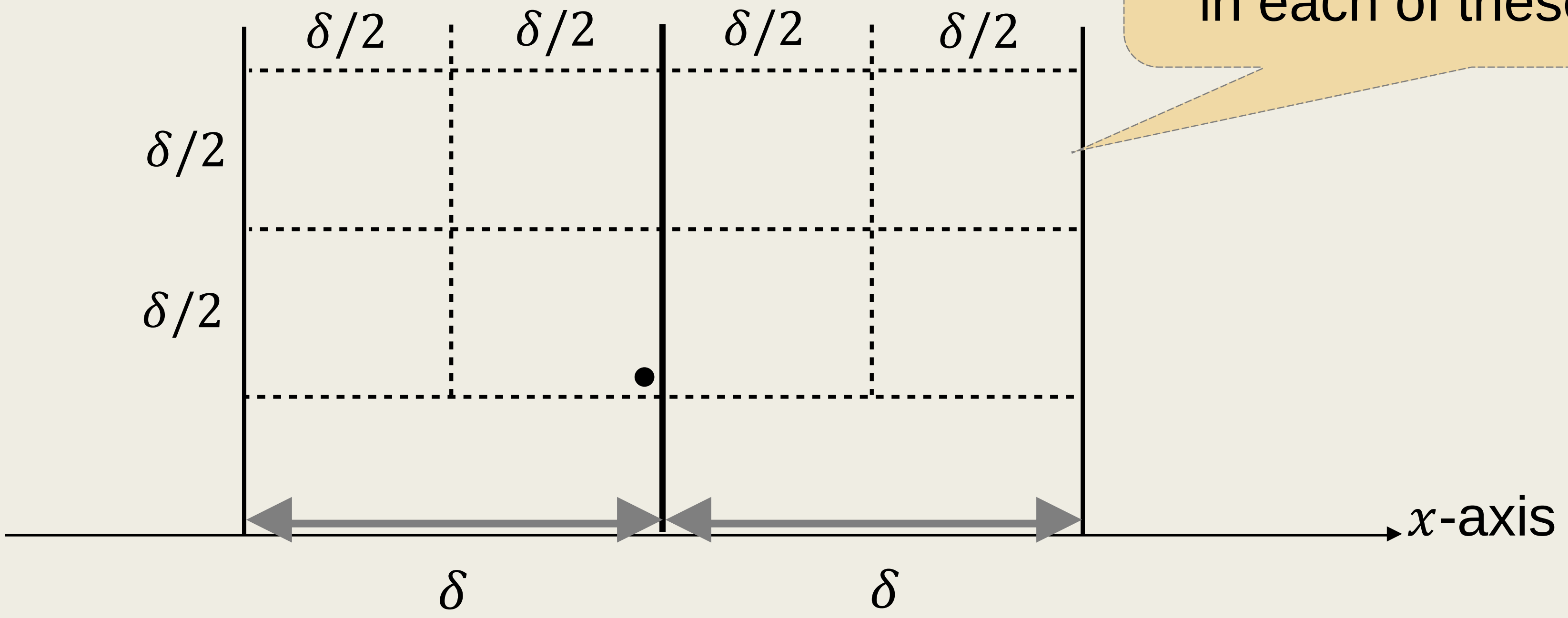
- Only ***points that are within a distance δ to the bisector*** need to be considered.



- Let $\delta := \min(\delta_L, \delta_R)$.

Observation 2

- For each point in the strip,
  at most 7 points above it are relevant.



At most one point can exist in each of these cells.

The Algorithm

Let $P = \{ p_1, p_2, \dots, p_n \}$ be the input points, and $\boldsymbol{P_x}$**,** $\boldsymbol{P_y}$ be the *sorted orders* of P according to the x*-coordinates* and y*-coordinates* separately.

1. ***Partition*** the input into two *equal-sized* subsets L and R.

   $O(n)$ time.

2. ***Recursively solve*** L and R.
   Let δ be the min-distance within L and R.

3. Consider the points ***within the strip with width*** $\boldsymbol{2\delta}$ centered at any bisector separating L and R *according to their* y*-coordinates*.
   - For each points considered, compare δ to ***its distance to the previous 7 points*** considered.

   $O(n)$ time.