Divide and Conquer

Algorithm Analysis

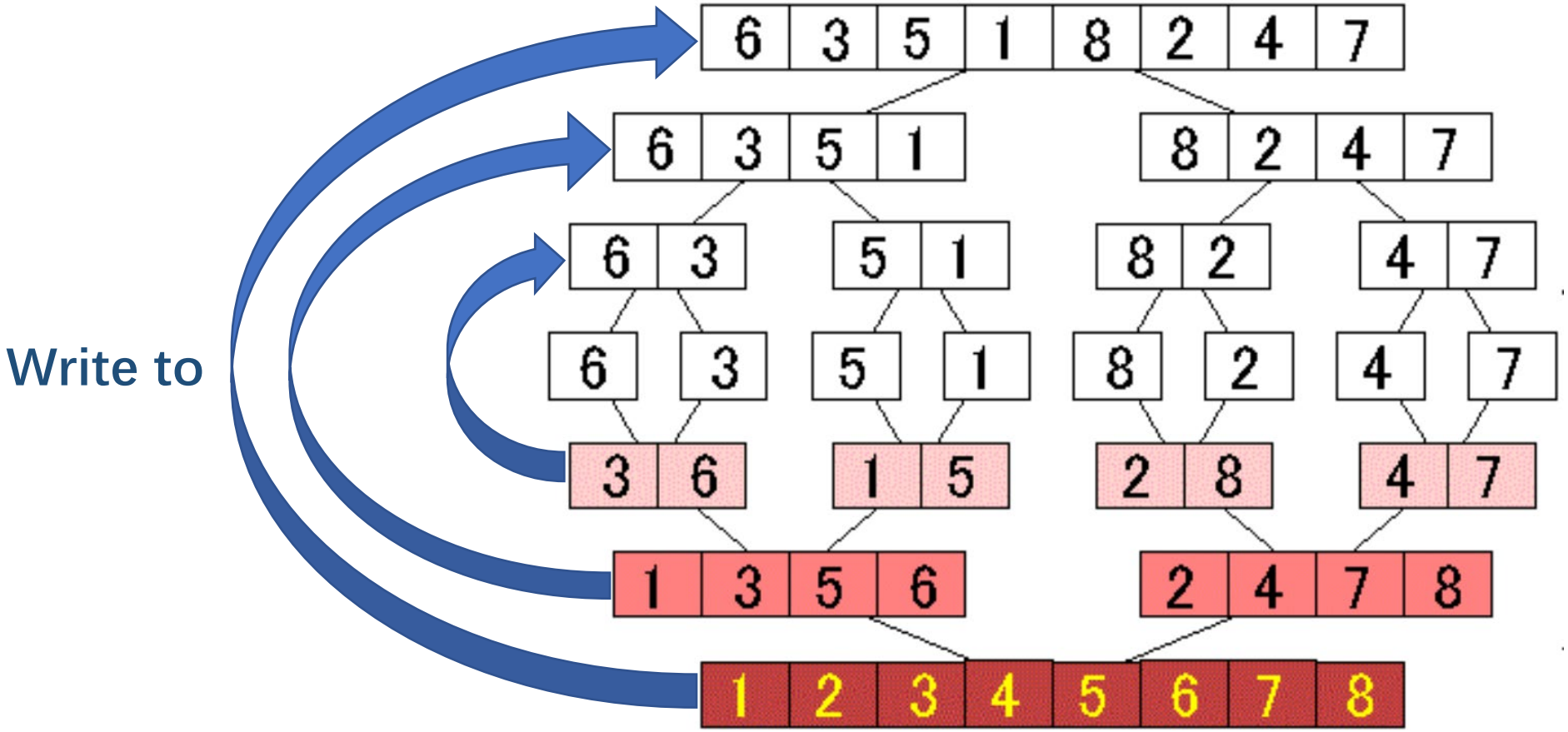
- Basic Idea
- Problems
  - MergeSort
  - QuickSort
    - Partition algorithms
    - K-th element
  - Closest Pair
  - Convex Hull

Basic Idea

Divide – Conquer – Combine

- 1. Divide into subproblems of the same type
- 2. The subproblems are solved
- 3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem

Merge Sort


Write to
6 3 5 1 8 2 4 7
6 3 5 1
8 2 4 7
6 3
5 1
8 2
4 7
6
3
5
1
8
2
4
7
3 6
1 5
2 8
4 7
1 3 5 6
2 4 7 8
1 2 3 4 5 6 7 8

Merge Sort – Merging

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1								

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2							

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2	3						

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2	3	4					

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2	3	4	5				

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2	3	4	5	6			

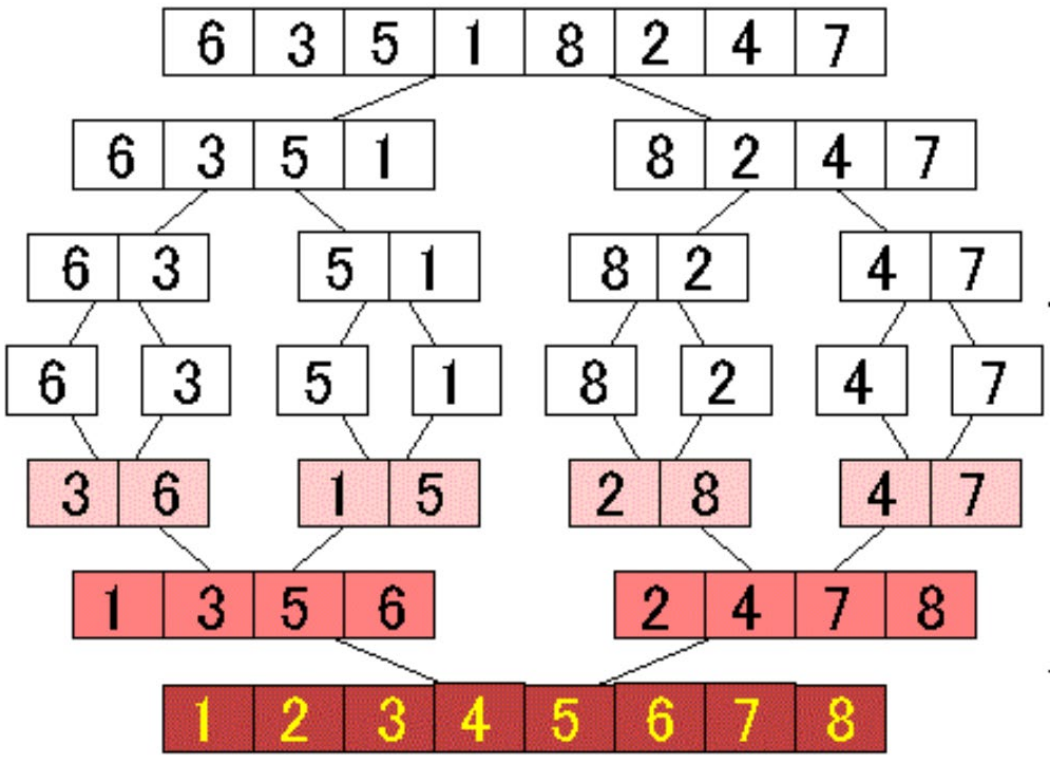
(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2	3	4	5	6	7	8	

(0)	(1)	(2)	(3)		(0)	(1)	(2)	(3)
1	3	5	6		2	4	7	8
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	
1	2	3	4	5	6	7	8	

Note: the last step is to append the rest of array B to array C

Array A	Array B
Array C	

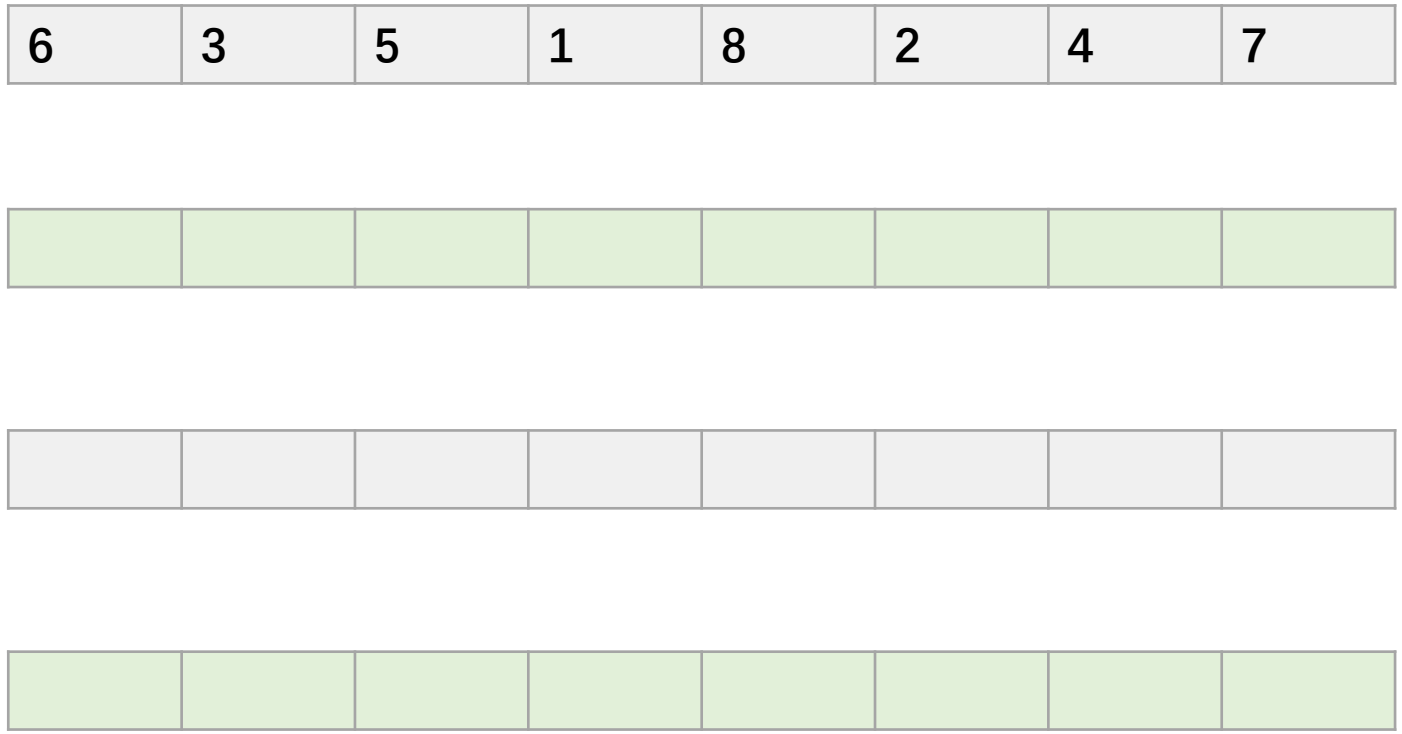
Merge Sort - Space need



- Each level – 'new' a size n auxiliary array
- How many levels ? logn+1

nlogn extra space for aux array

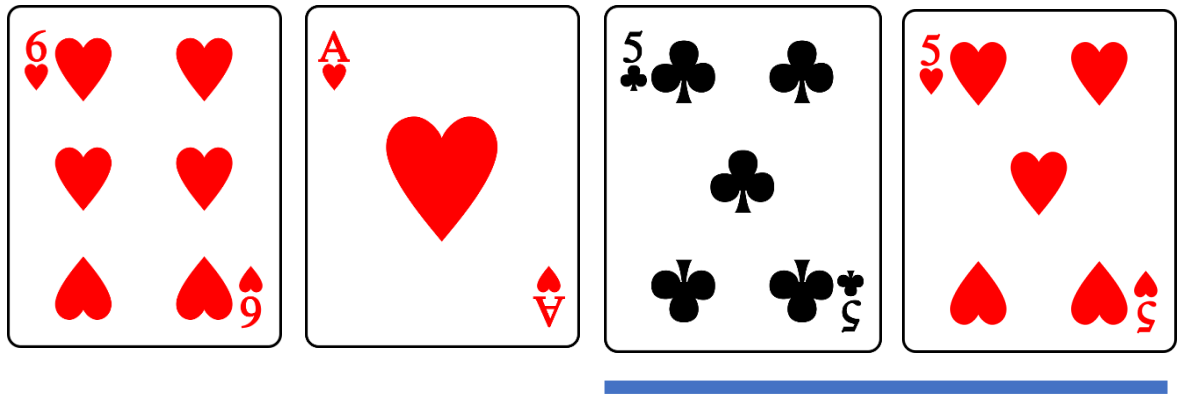
Merge Sort - Speed up in implementing

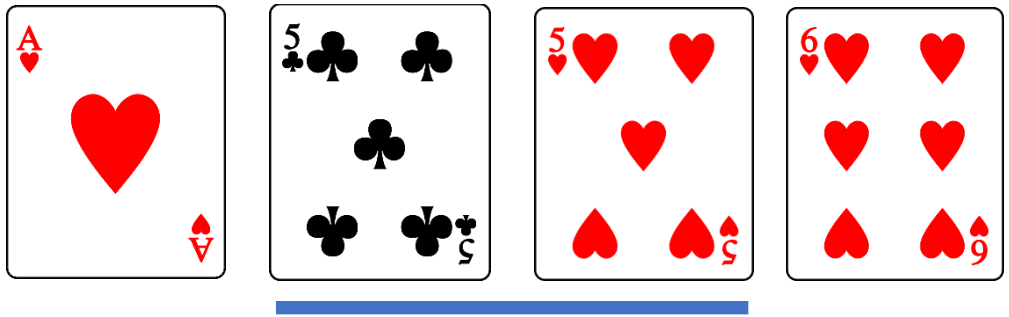


Merge Sort - Conclusion

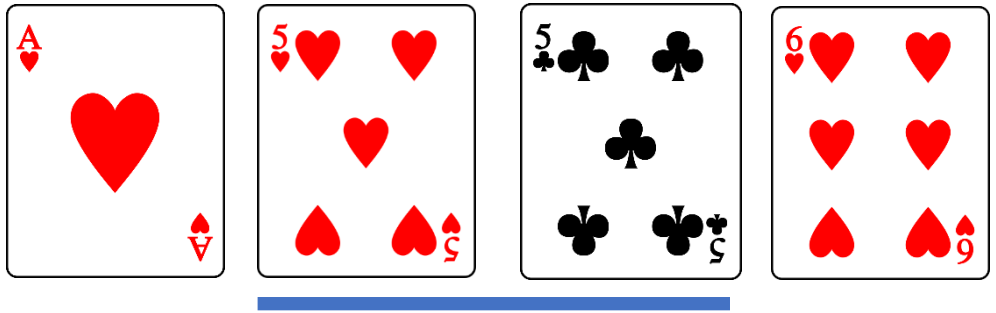
- $T(n) = 2T(n/2) + n$
  - Divided into 2 sub-problems
  - Size of sub-problem: n/2
  - Linear time spent on merging
- O(nlgn)
- Not in-place: need extra space O(n)
- Stable

Stable vs. Unstable



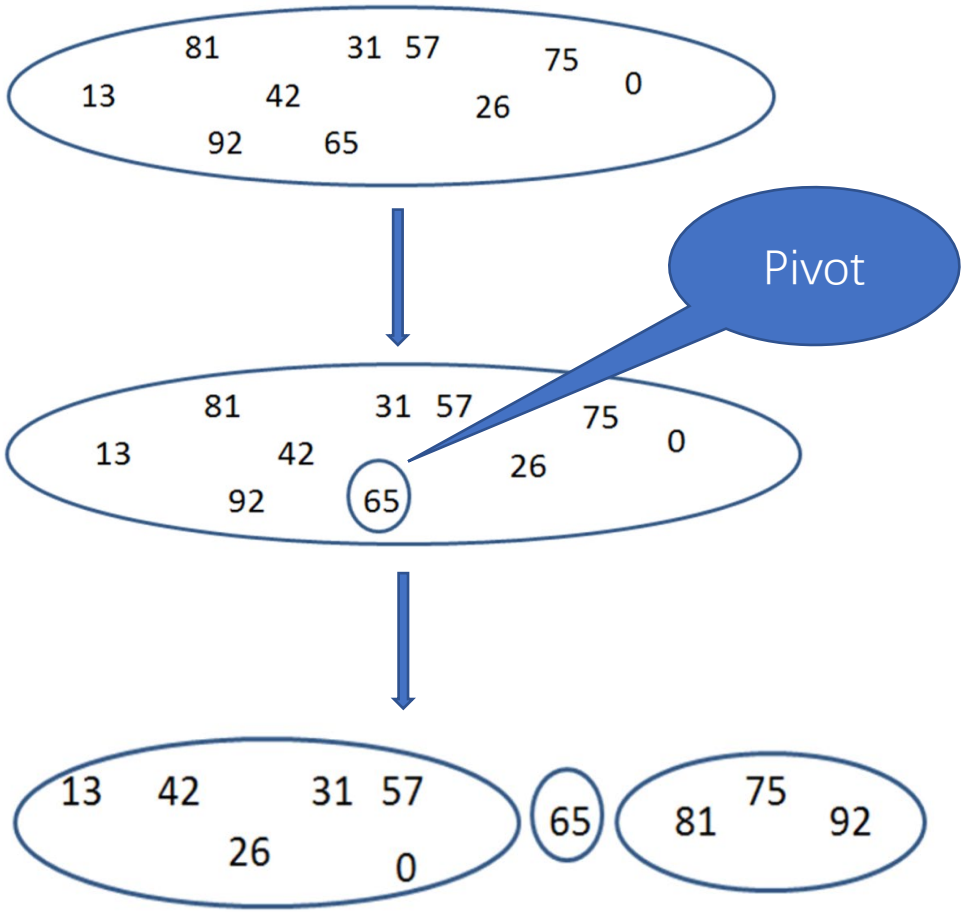


Stable



Unstable

Quick Sort


81 31 57 75 0
13 42 26
92 65
Pivot
81 31 57 75 0
13 42 26
92 65
13 42 31 57
26 0
65
75
81 92

...

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
A:	65	13	0	26	81	92	42	31	57	75	+∞	0	10
A:	65	13	0	26	81	92	42	31	57	75	+∞	4	8
A:	65	13	0	26	57	92	42	31	81	75	+∞	5	7
A:	65	13	0	26	57	31	42	92	81	75	+∞	7	6
A:	65	13	0	26	57	31	42	92	81	75	+∞		
A:	42	13	0	26	57	31	65	92	81	75	+∞		

.....

Quick Sort - Conclusion

- Best case: the size of sub-problems is n/2
  - $T(n) = 2T(n/2) + n$
  - $O(nlgn)$
- Worst case: the size of one sub-problem is (n-1), another is 0
  - $T(n) = T(n-1) + n$
  - $O(n^2)$
- In-place: no extra aux-list
- Unstable

k'th element

- 65 13 0 26 81 92 42 31 57 75
- Find the 5-th element （Ascending)

- Use partition
  - (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
  - 42 13 0 26 57 31 65 92 81 75 7-th
  - 31 13 0 26 42 57 65 92 81 75 5-th

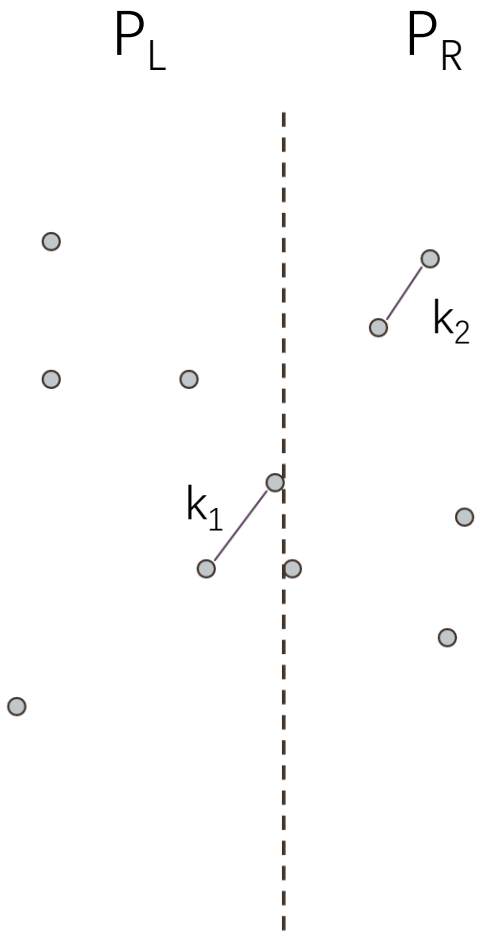
O(n) on average

Closest Pair

Divide

- Find a vertical line split the space
- roughly n/2 points on each side

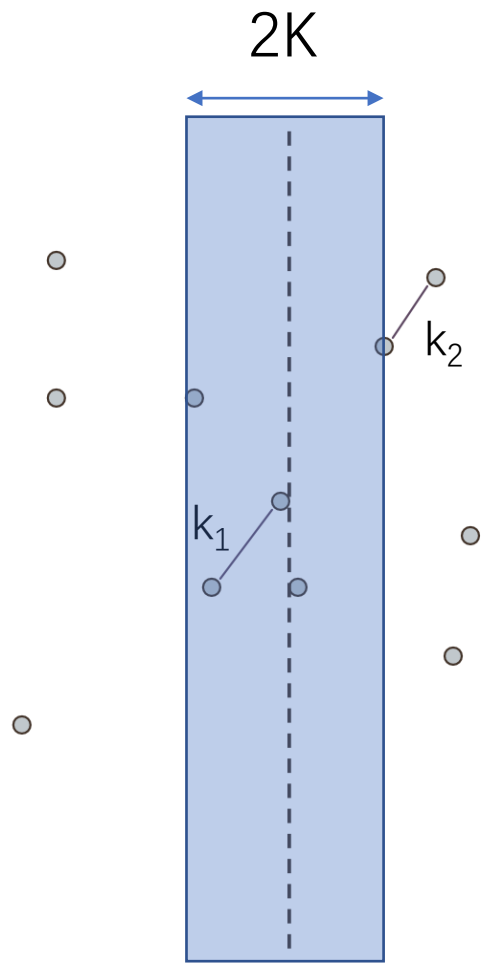
- K_1: closed distance in P_L
- K_2: closed distance in P_R



Combine

- $K = \min(K_1 , K_2)$

- check only pairs which cross the dividing line and
- Lie in the 2k-width strip (shaded region)



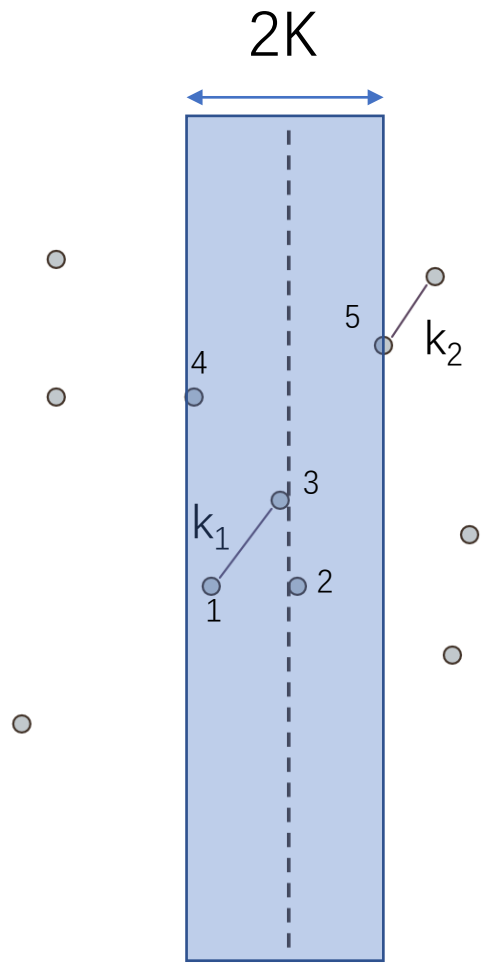
Check the strip

- For every point p in P_L, we inspect points in P_R that may be closer to P than k.

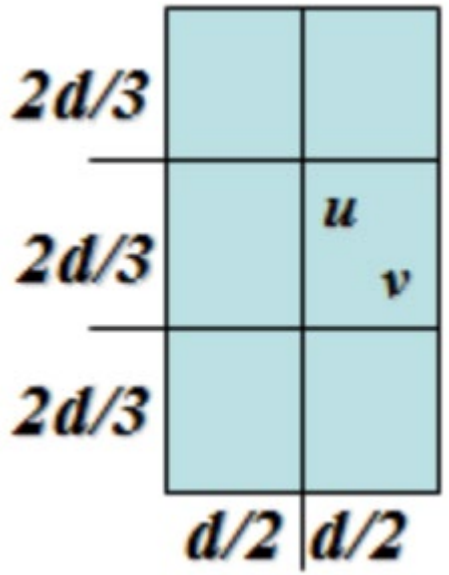
- Order all points in the strip by their y-coordinates
- From bottom to top
  - p_1 p_2 p_3 p_4 p_5

- For each p_i
- $j=i+1$
- For each p_j in the right strip
- IF (p_i.y- p_j.y < K) // r: # of points in strip
- {check d(p_i, p_j) < K, ++j}

Note: Proof by Pigeonhole principle:
For each p_i, there are at most 6 p_j such that p_i.y- p_j.y < K

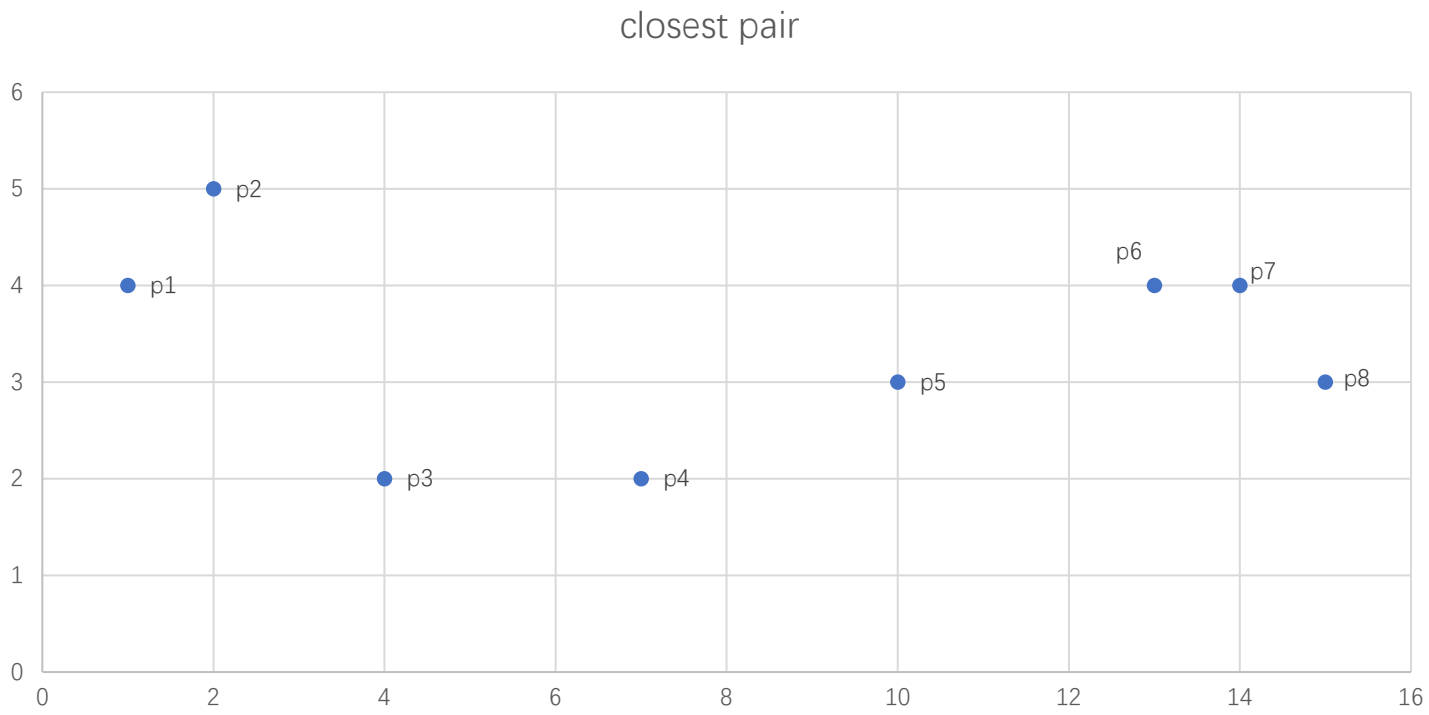


Pigeonhole principle



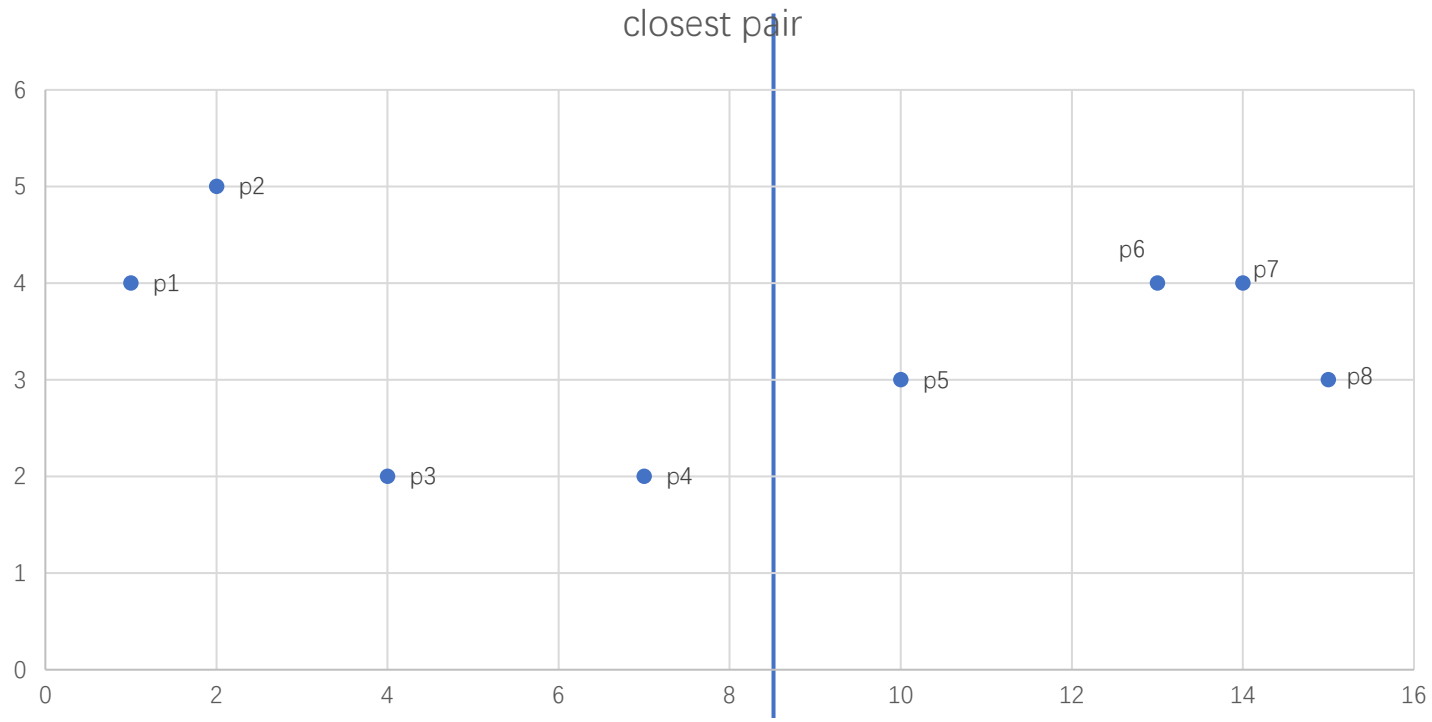
An example - Input

P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



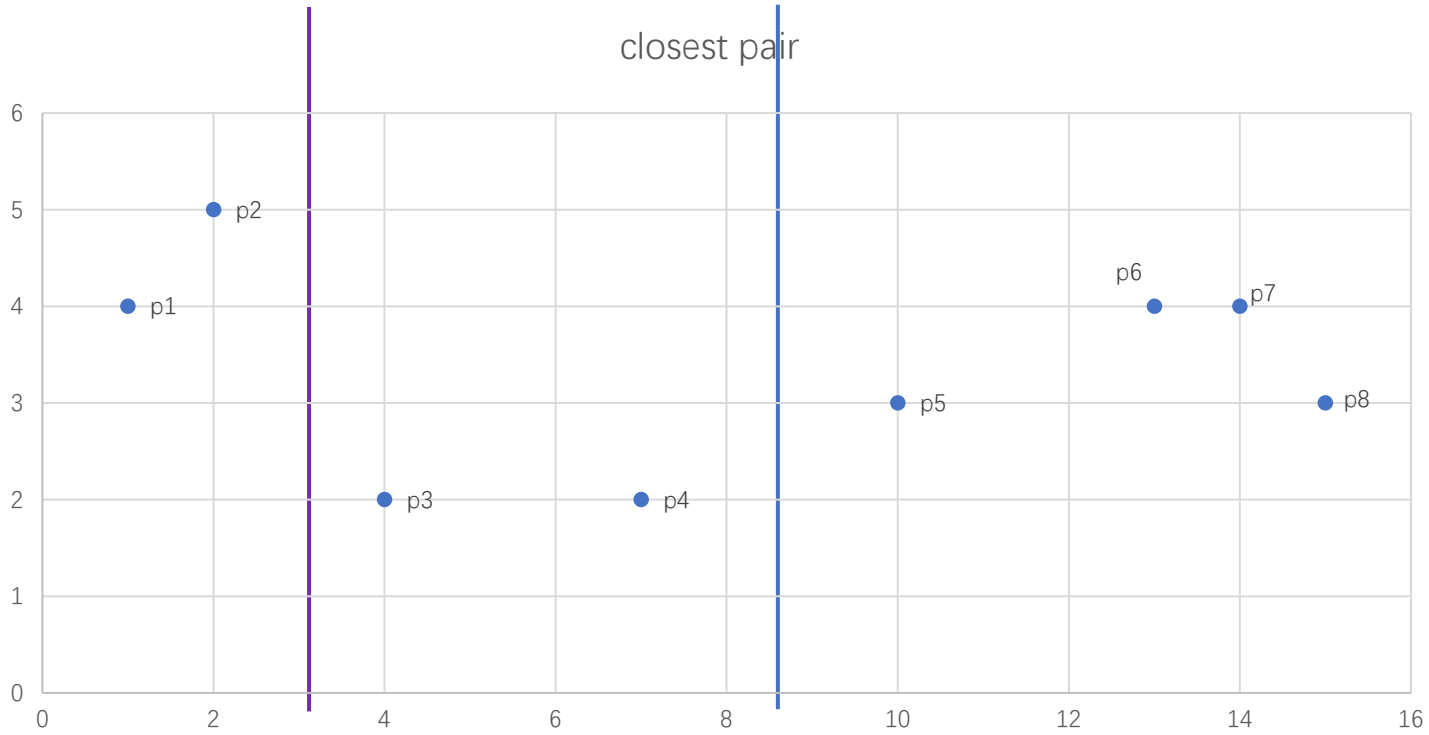
An example - divide

P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



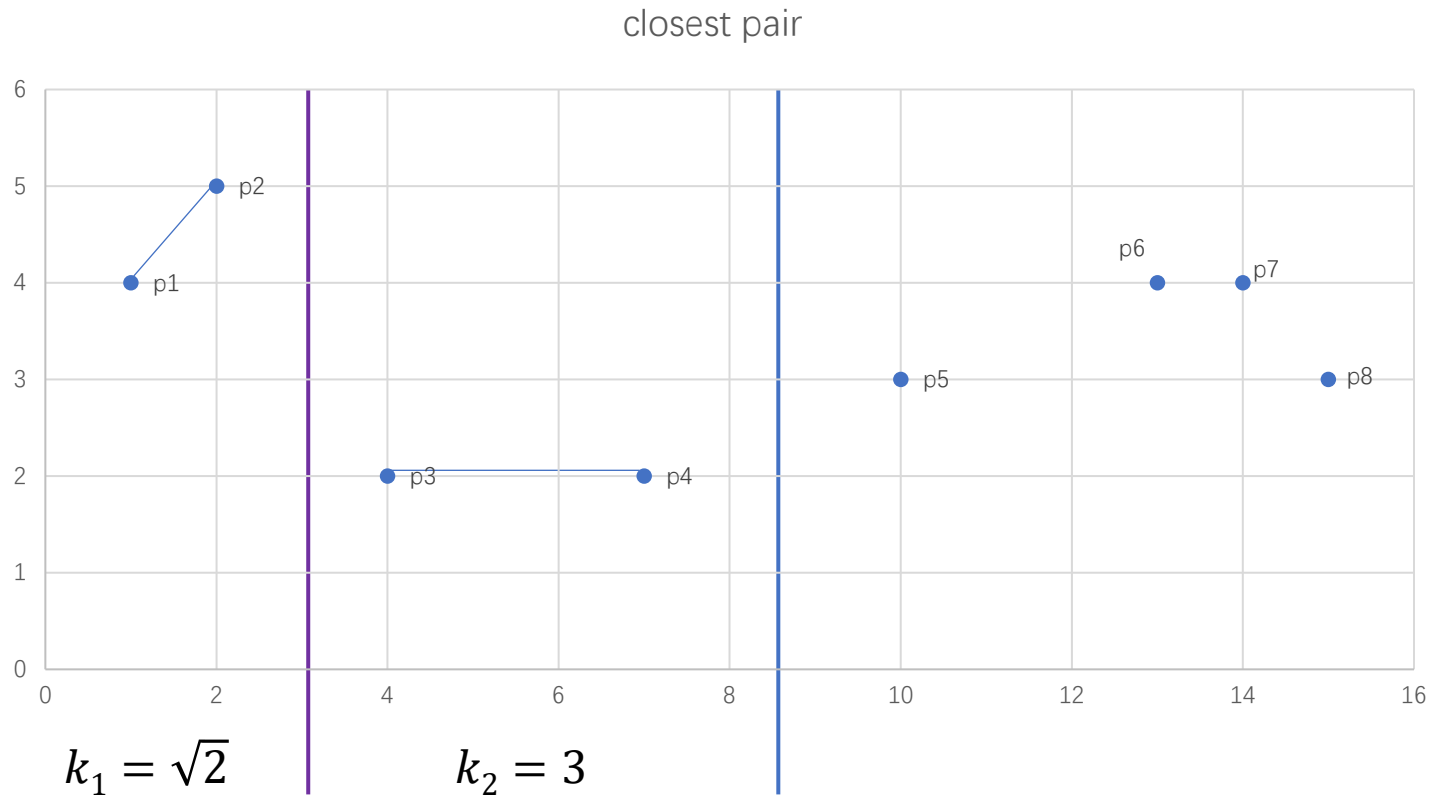
An example – divide further

P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



An example - conquer

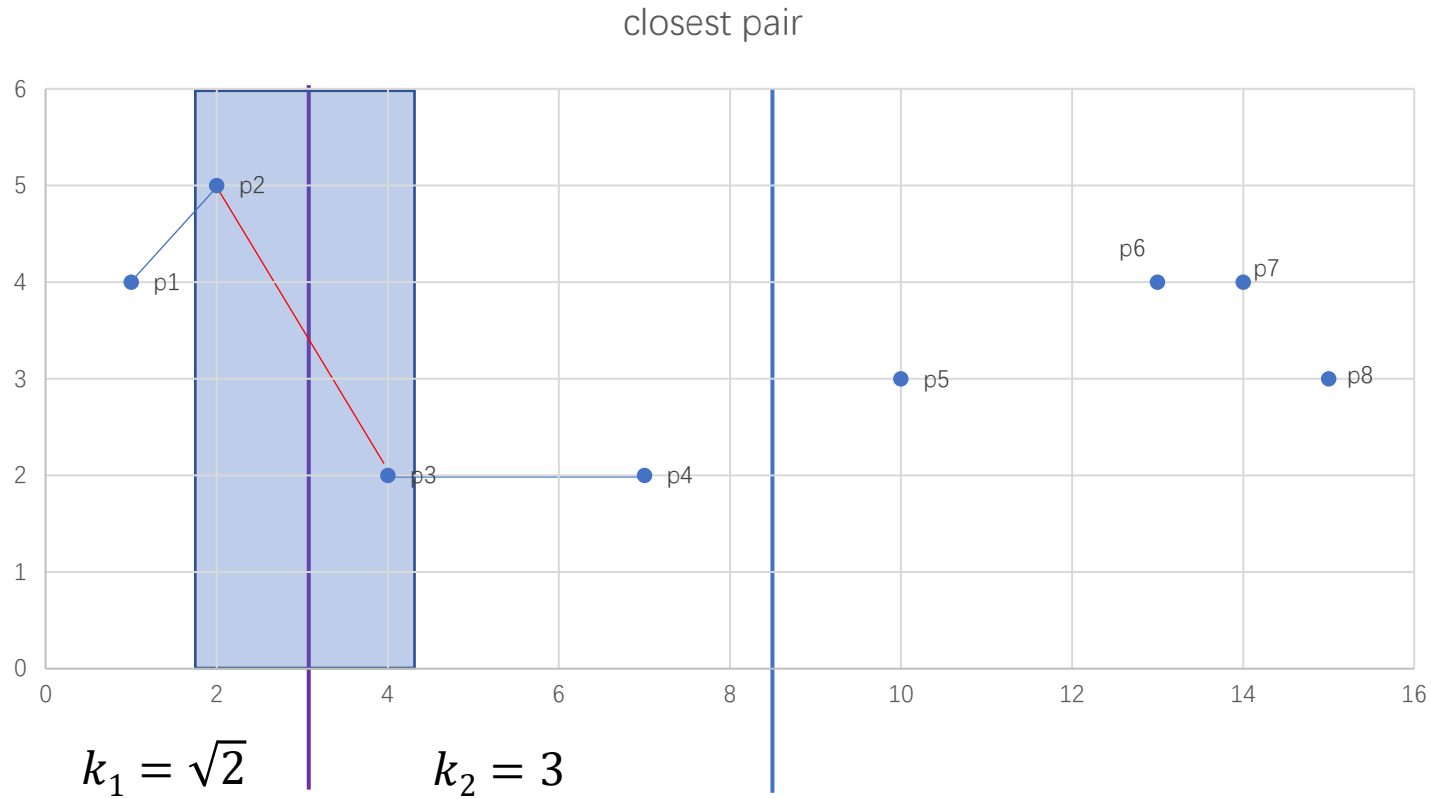
P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



$$k = \sqrt{2}$$

An example - combine

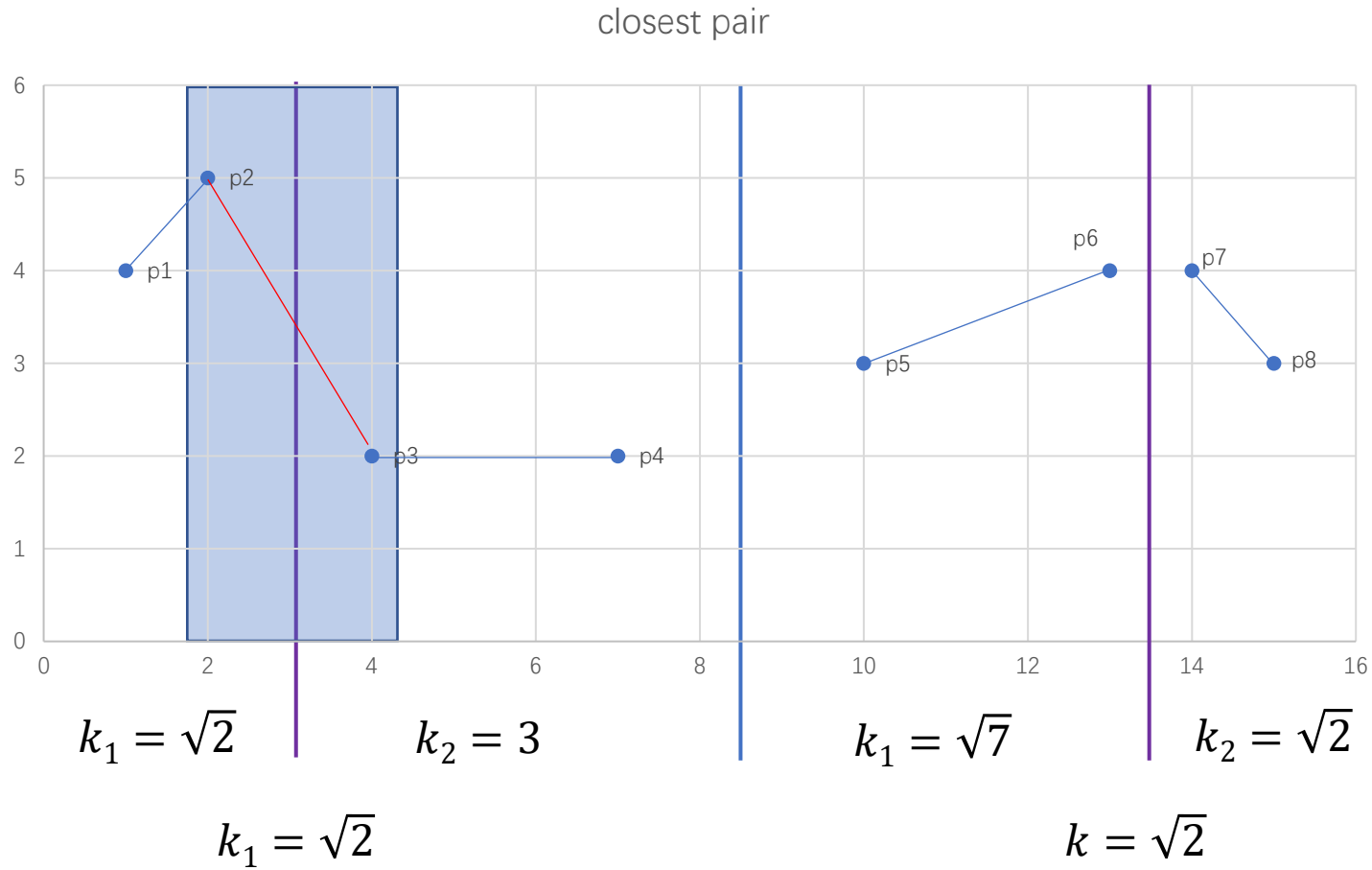
P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



$k_1 = \sqrt{2}$

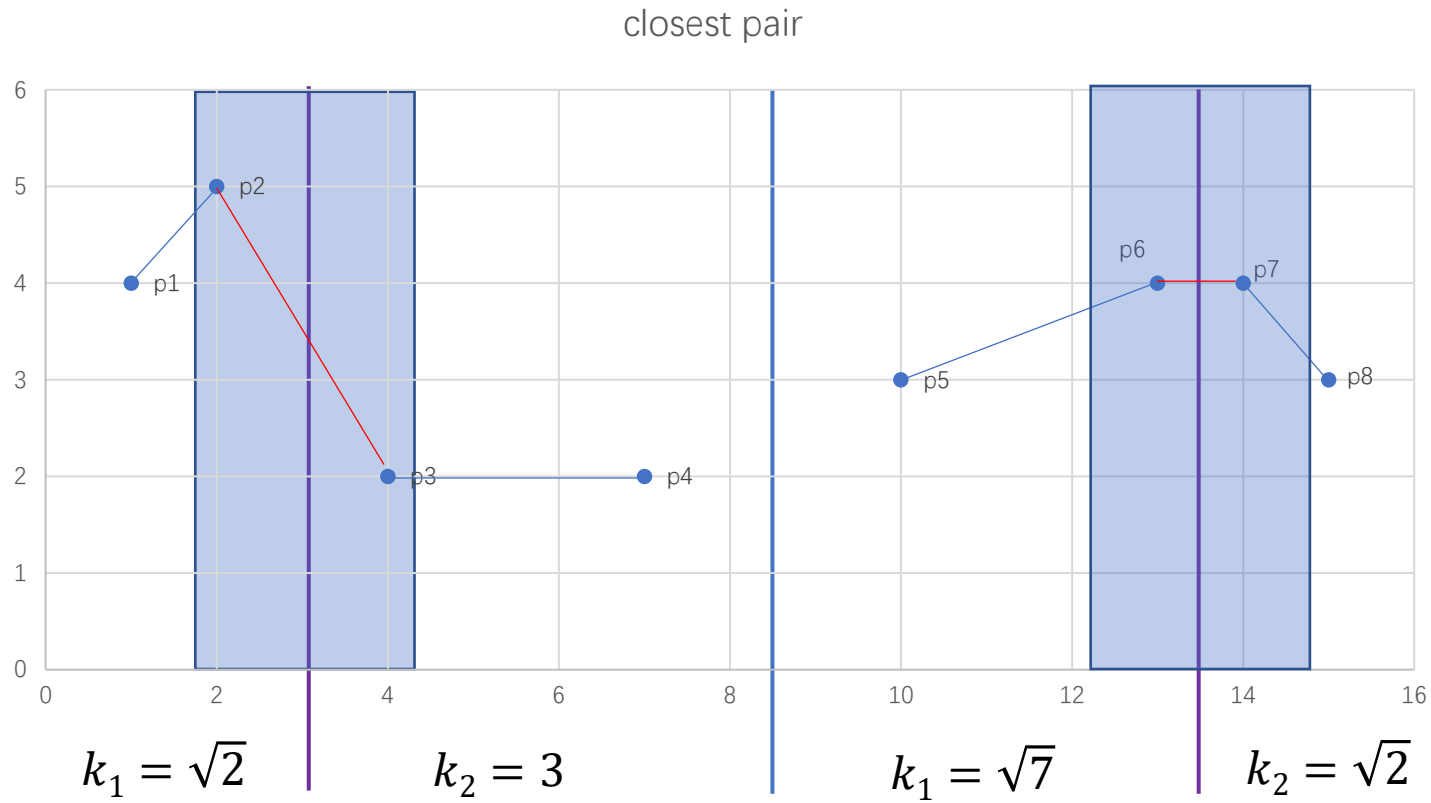
An example – divide (right) and conquer

P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



An example - combine

P_i	(x,y)
P_1	(1,4)
P_2	(2,5)
P_3	(4,2)
P_4	(7,2)
P_5	(10,3)
P_6	(13,4)
P_7	(14,4)
P_8	(15,3)



$k_1 = \sqrt{2}$

$k_2 = 1$

$\boldsymbol{k = 1}$

Closest Pair Algorithm

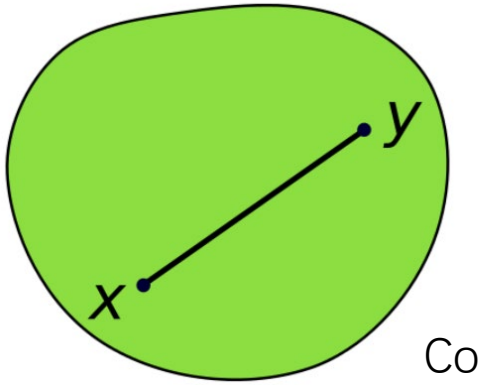
- Closest-pair(p_1, ⋯, p_n) {
  - Sort the list so that we can find a vertical line to split the points // O(nlogn)
  - K_1 = Closest-pair(left half) // T(n/2)
  - K_2 = Closest-part(right half)) // T(n/2)
  - K = min(K_1, K_2)
  - Sort points in the strip by their y-coordinates // O(nlogn)
  - Check the points in the strip to determine if there is closer pair // O(n)
- }

$O(nlog^2n)$

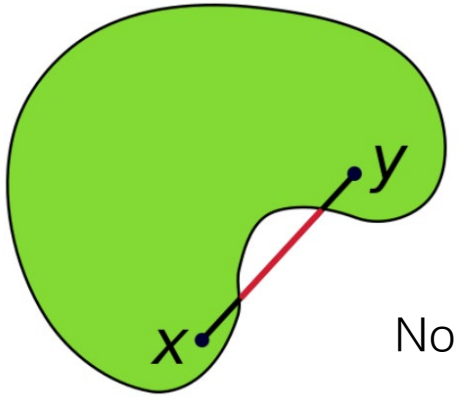
Convex Hull

Definition

- In mathematics, the **convex hull** or **convex envelope** of a set X of points in the Euclidean plane or in a Euclidean space is the smallest convex set that contains X.
- In convex geometry, a **convex set** is a subset of an affine space that is closed under convex combinations.
- More specifically, in a Euclidean space, a **convex region** is a region where, for every pair of points within the region, every point on the straight line segment that joins the pair of points is also within the region



Convex set



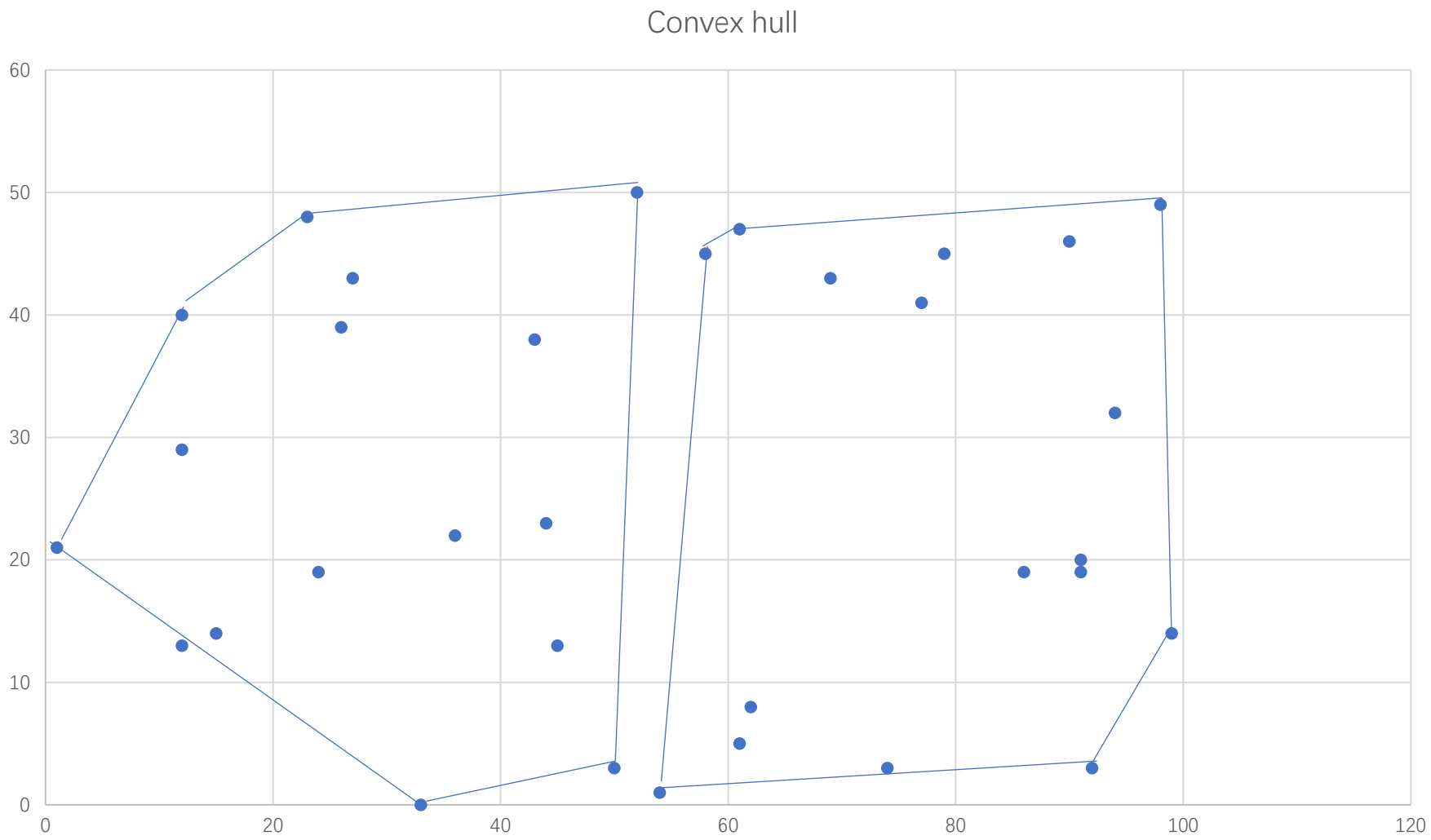
Non-Convex set

Divide and Conquer

- Sorting the points by their X coordinate O(nlogn)

- Hull(S)
  - If |S| <= 5, then compute the convex hull by brute force in O(1) time and return
  - Otherwise, partition the point set S into two sets A and B, where A consists of half the points with the lowest x coordinates and B consists of half of the points with the highest x coordinates
  - Recursively compute HA = Hull(A) and HB = Hull(B)
  - Merge the two hulls into a common convex hull, H, by computing the upper and lower tangents for HA and HB and discarding all the points lying between these two tangents

Example

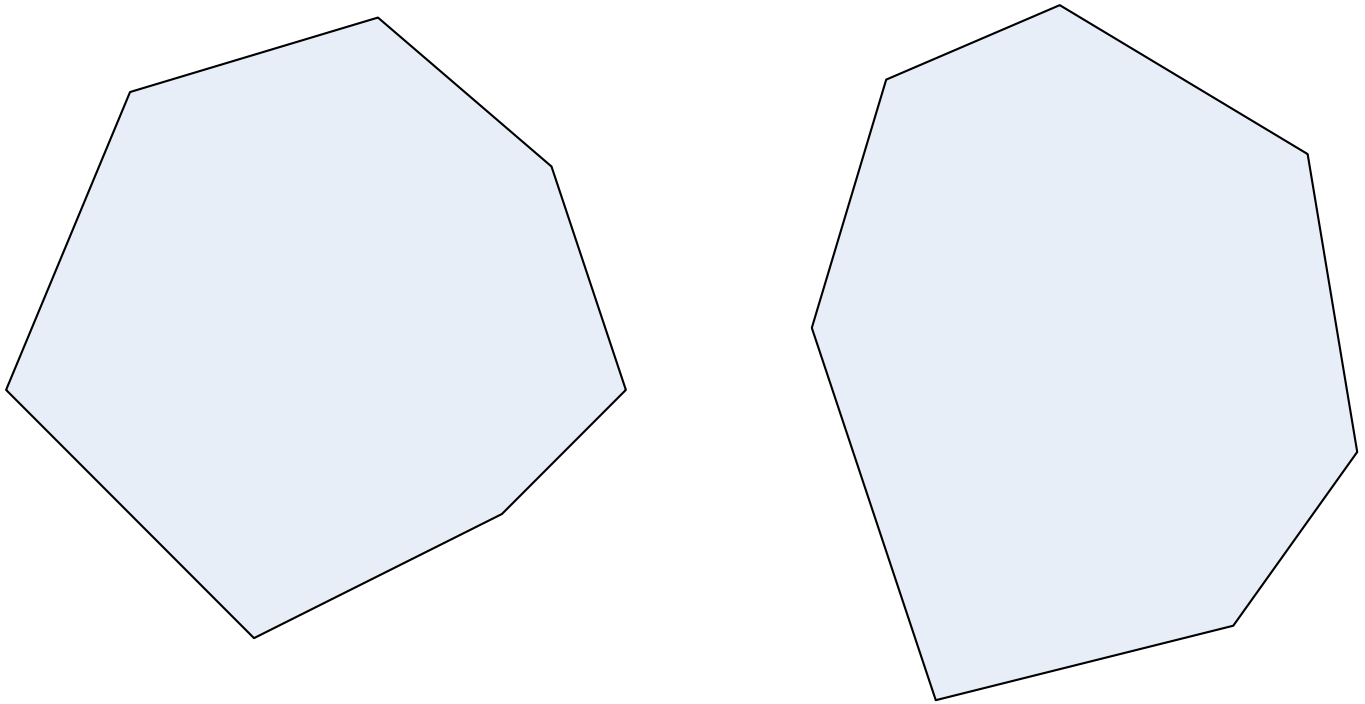


Finding the lower tangent

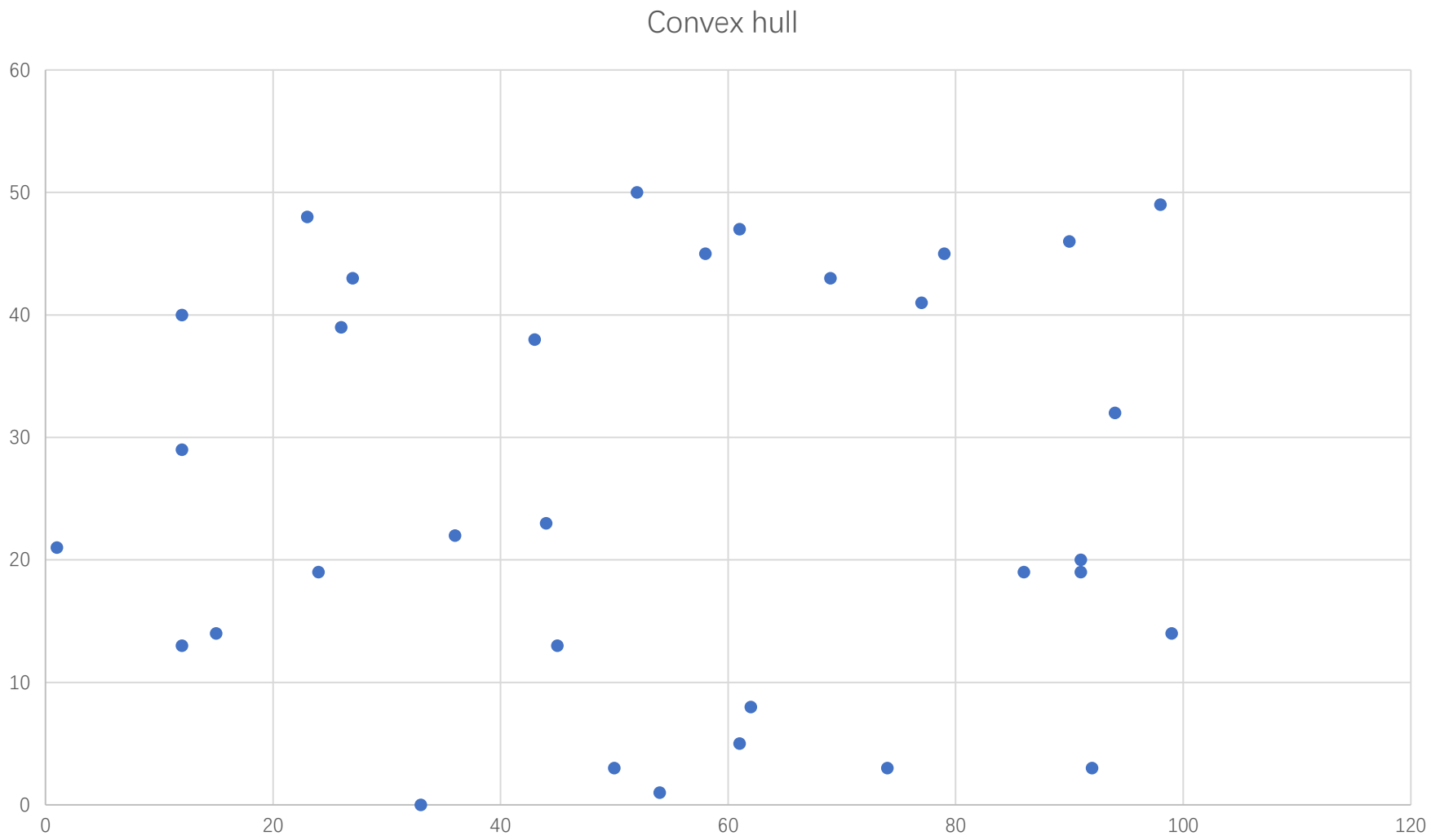
- LowerTangent(HA ; HB)
- Let a be the rightmost point of HA .
- Let b be the leftmost point of HB .
- While ab is not a lower tangent for HA and HB do

    While ab is not a lower tangent to HA do a = a - 1 (move a clockwise).

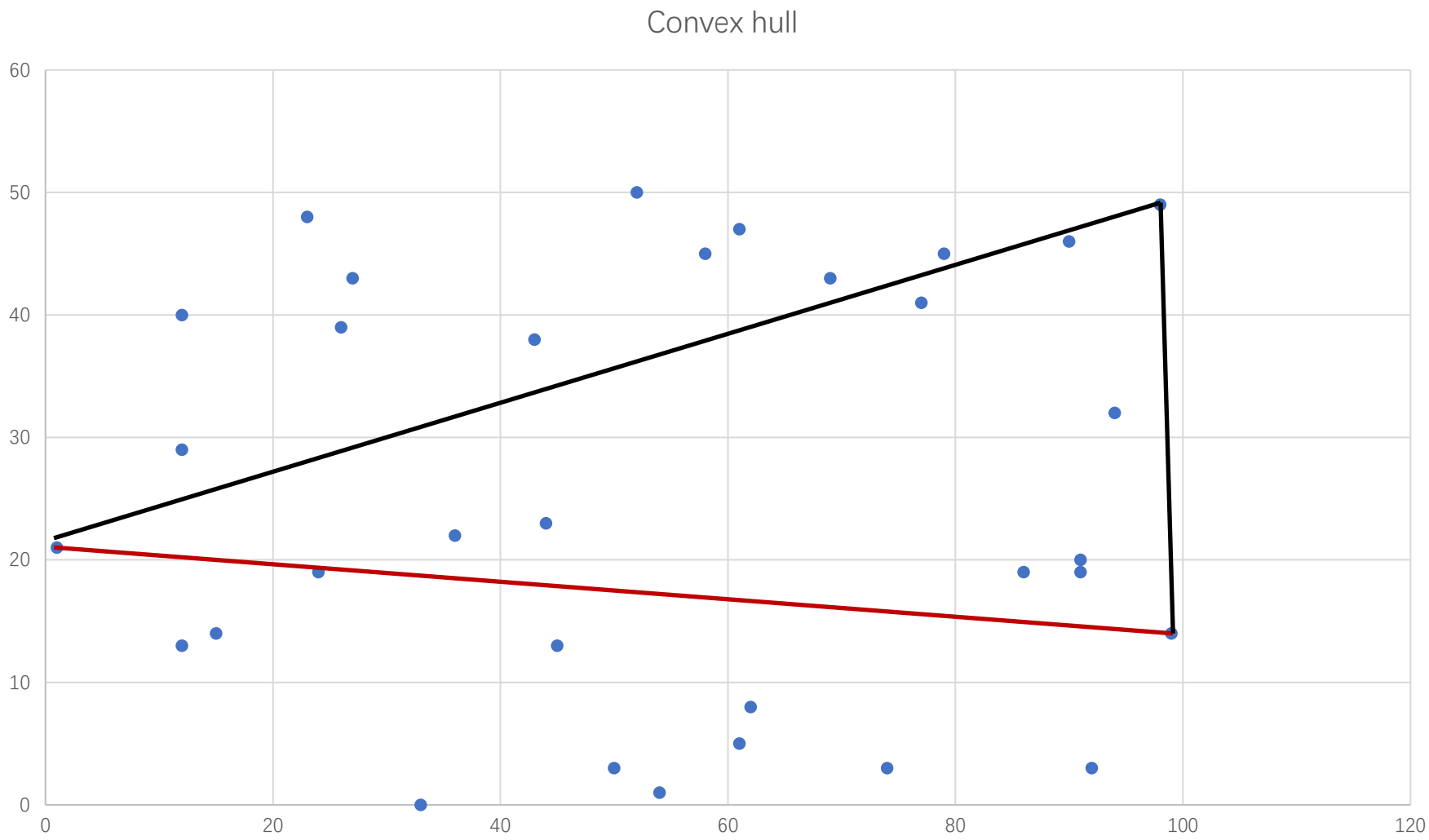
    While ab is not a lower tangent to HB do b = b + 1 (move b counterclockwise).
- Return ab.



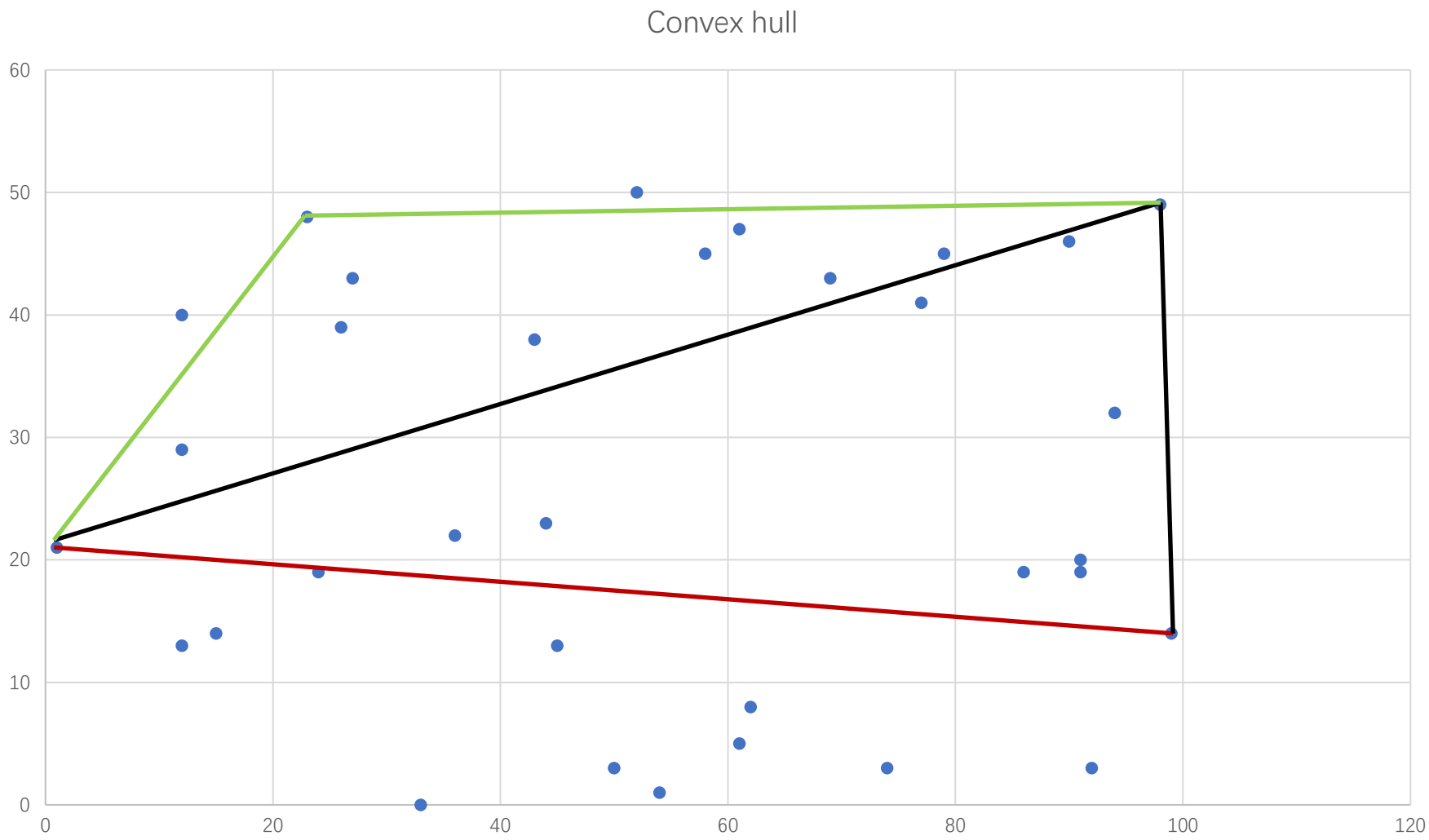
Example



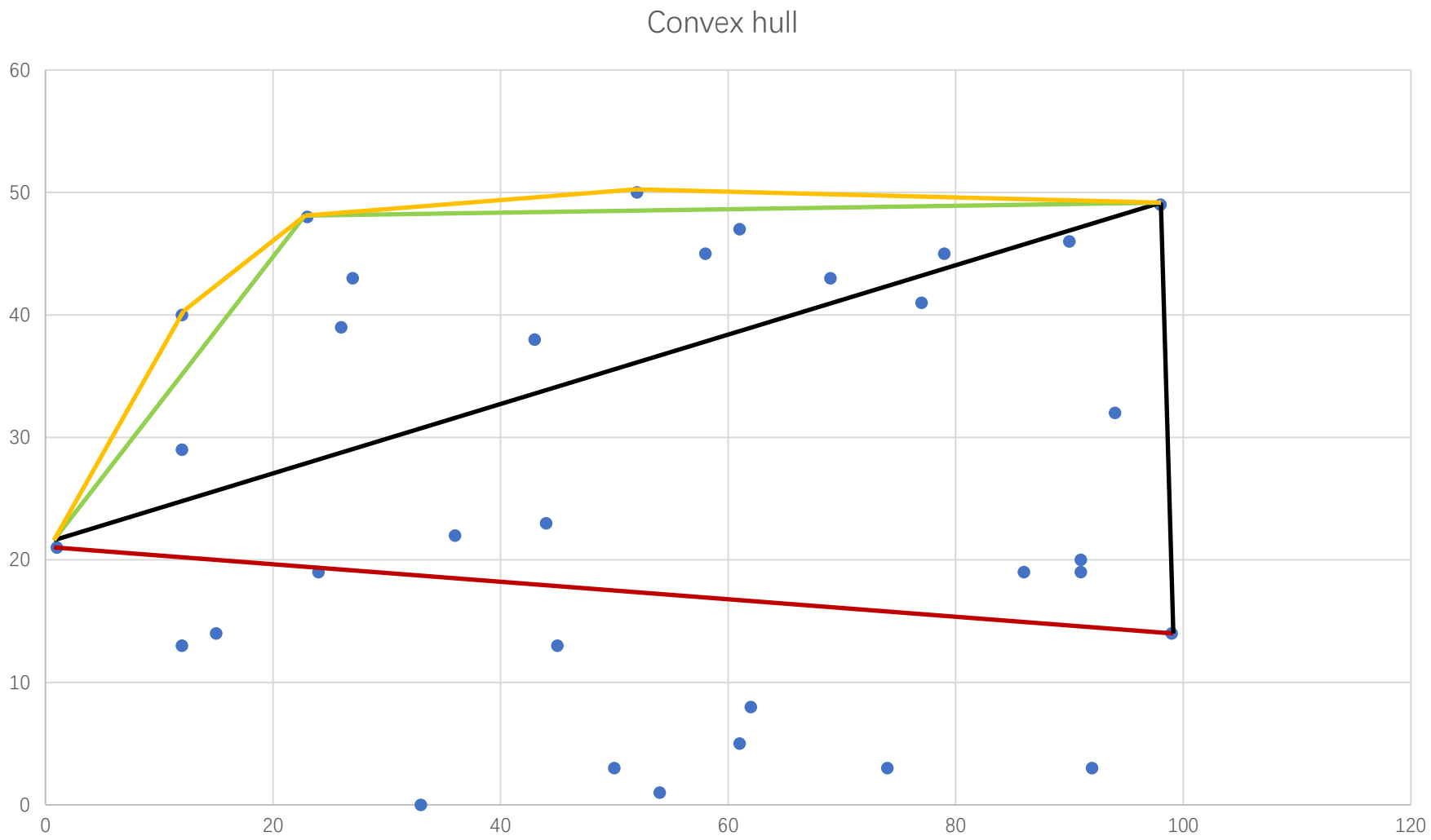
Quickhull



Quickhull



Quickhull



Quick Hull Algorithm

- 1) Find the points with minimum and maximum x coordinates
- 2) Use the line joined by the above two points to divide the set into two subsets
- 3) On one side of the line, determine the point with the maximum distance from the line. This point forms a triangle with those of the line
- 4) The points lying inside of that triangle cannot be part of the convex hull and therefore can be ignored
- 5) Repeat the steps 3) and 4) on the two lines formed by the triangle until no more points are left

Quickhull - Summary

- Best case: the size of sub-problems is n/2
  - $T(n) = 2T(n/2) + n$
  - $O(n\lg n)$
- Worst case: the size of one sub-problem is (n-1), another is 0
  - $T(n) = T(n-1) + n$
  - $O(n^2)$
- Assumption: data are chosen from a uniform distribution over convex region. Average-case: $O(n\lg n)$

Convex Hull – Applications

- **Collision avoidance**: If the convex hull of a car avoids collision with obstacles then so does the car.
- **Smallest box**: The smallest area rectangle that encloses a polygon has at least one side flush with the convex hull of the polygon, and so the hull is computed at the first step of minimum rectangle algorithms. Similarly, finding the smallest three-dimensional box surrounding an object depends on the 3D-convex hull.
- **Shape analysis**: Shapes may be classified for the purposes of matching by their "convex deficiency trees", structures that depend for their computation on convex hulls.