Divide and Conquer (Ch 5)

- **Divide and conquer approach**
- **Merge sort**
- **Recursion analysis**
- **Counting inversions**
- **Closest pairs**

The lecture notes/slides are adapted from those associated with the text book by J. Kleinberg and E. Tardos.

Divide and Conquer Approach

- **Divide and conquer approach**

  Divide, partition a problem into (independent) subproblems.

  Conquer, solve each subproblem recursively (algorithms call themselves on subproblems).

  Combine solutions of subproblems into a solution of original problem.

- **A typical usage**

  Divide a problem of size n into two subproblems of size $n/2$.

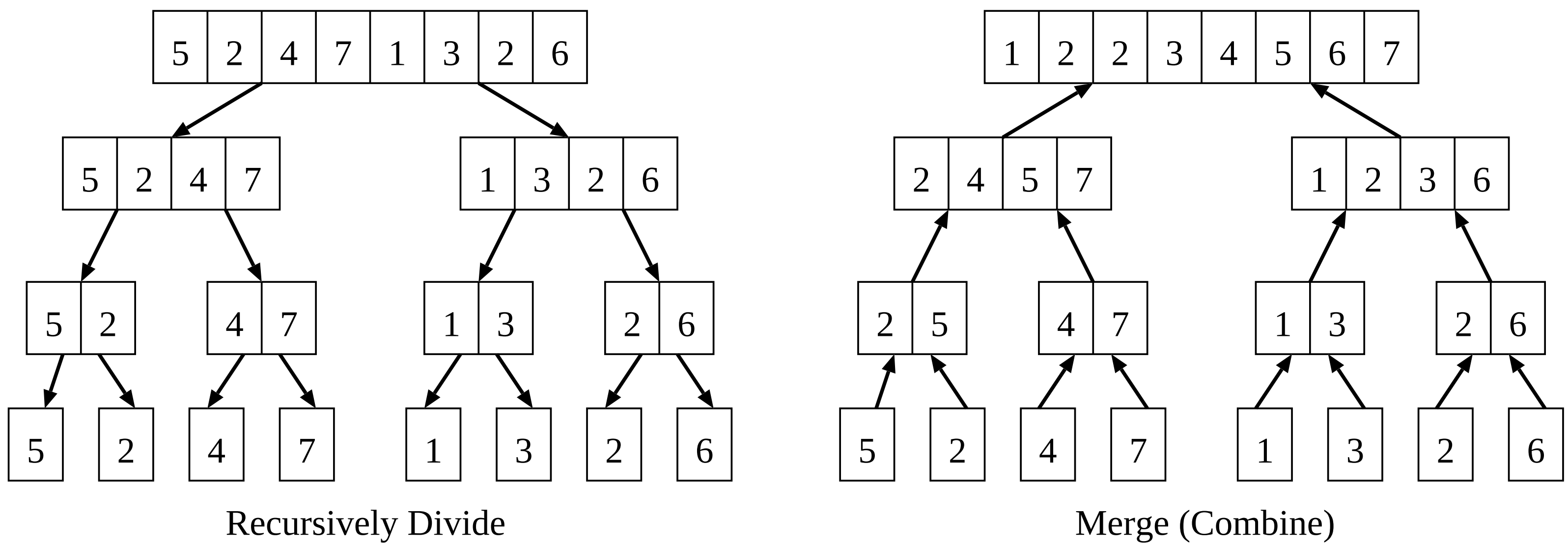
  Solve two subproblems recursively.

  Combine two solutions to subproblems into a solution in $O(n)$ time.

  Reduce the running time from $O(n^2)$ to $O(n \log n)$.

Merge Sort Algorithm

- **Divide a sequence of n numbers into two sub-sequences of $n/2$ numbers.**
- **Call the algorithm itself to sort each of the sub-sequence (conquer).**
- **Merge the two sorted sub-sequences into one sorted sequence (combine).**



MergeSort(A, l, r)

Input: Array A and positions l, r.

Output: Array A s.t. $A[l], A[l+1], .., A[r]$ are sorted.

if $l < r$ **then**

$p := \lfloor (l+r)/2 \rfloor$; MergeSort($A, l, p$); MergeSort($A, p+1, r$); Merge($A, l, p, r$);

Merge(A, l, p, r)

$n_l = p - l + 1; n_r = r - p; L[n_l + 1] = \infty; R[n_r + 1] = \infty;$

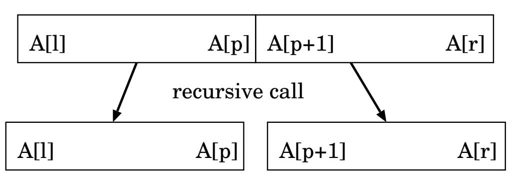
for $i = 1$ **to** n_l **do** $L[i] = A[l + i - 1];$

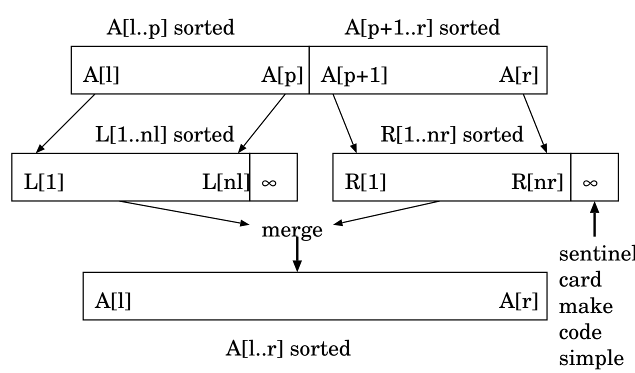
for $j = 1$ **to** n_r **do** $R[j] = A[p + j];$

$i = 1; j = 1;$

for $k = l$ **to** r **do**

if $L[i] \leq R[j]$ **then** $\{A[k] = L[i]; i = i + 1\}$ **else** $\{A[k] = R[j]; j = j + 1\}$





Running Time of MergeSort

- **Let $T(n)$ be the running time of MergeSort. $T(1) = c$, $c > 0$ a constant.**
- **Divide array of size n takes $c_1 n$ time, $c_1 > 0$ a constant.**
- **Recursion on an array of size $n/2$ takes $T(n/2)$ time.**
- **Merge two arrays of size $n/2$ takes $c_2 n$ time, $c_2 > 0$ a constant.**
- $T(n) = c_1 n + T(n/2) + T(n/2) + c_2 n = 2T(n/2) + O(n)$**.**

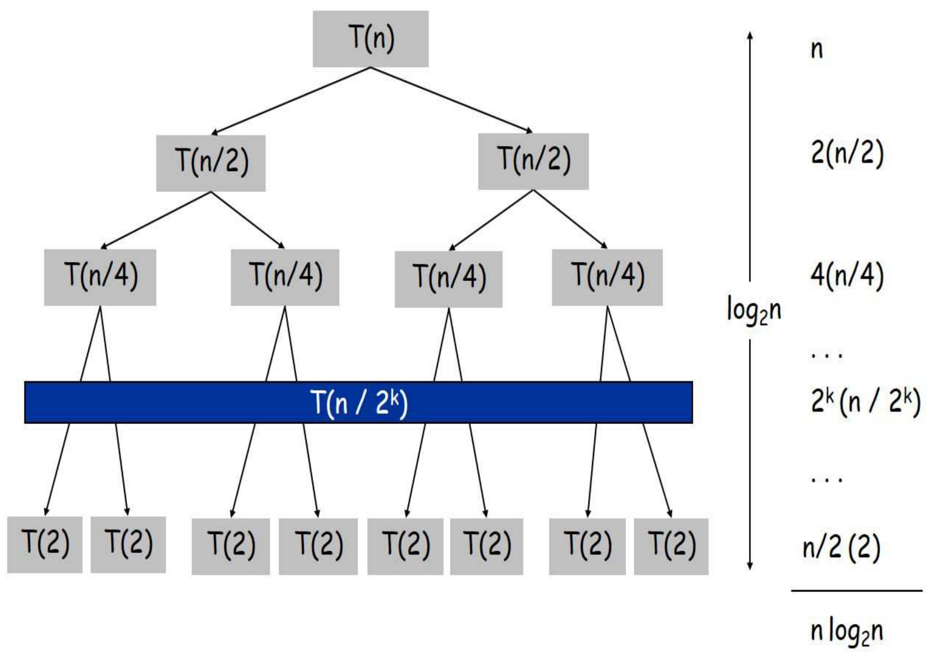
- **Another analysis, recursion tree, $\log n$ levels**

  Level 0, 1 node, the array of size n, $(c_1 + c_2)n$ time.

  Level 1, 2 nodes, each is an array of size $n/2$, $2\frac{(c_1+c_2)n}{2}$ time.

  Level i, 2^i nodes, each is an array of size $n/2^i$, $2^i\frac{(c_1+c_2)n}{2^i}$ time.

- $T(n) = \Theta(n \log n)$.



Recurrence analysis

Master Theorem

- **Let** $a \geq 1, b > 1$ **be constants,** $f(n)$ **be a function, and** $T(n)$ **be defined on** $n \geq 0$ **by recurrence**

$$T(n) = aT(n/b) + f(n).$$

- **Master Theorem compares** $f(n)$ **(called *driving function*) with** $n^{\log_b a}$ **(called *watershed function*) to bound** $T(n)$ **asymptotically:**

Case 1 **If** $f(n)$ **is** $O(n^{(\log_b a)-\epsilon})$ **(**$f(n) = O(\frac{n^{\log_b a}}{n^{\epsilon}})$**) for some constant** $\epsilon > 0$**, then** $T(n)$ **is** $\Theta(n^{\log_b a})$**.**

Case 2 **If** $f(n)$ **is** $\Theta(n^{(\log_b a)} \log^k n)$ **(**$k > -1$**), then** $T(n)$ **is** $\Theta(n^{\log_b a} \log^{k+1} n)$ **(**$\Theta(n^{\log_b a} \log^k n) \log n)$**).**

Case 3 **If** $f(n)$ **is** $\Omega(n^{(\log_b a)+\epsilon})$ **(**$f(n) = \Omega(n^{\log_b a} \cdot n^{\epsilon})$**) for some constant** $\epsilon > 0$**, and if** $af(n/b) \leq cf(n)$ **for some constant** $c > 0$ **and all sufficiently large** n**, then** $T(n)$ **is** $\Theta(f(n))$**.**

Exension of Case 2 of Master Theorem

Case 2a For $k > -1$**, if** $f(n)$ **is** $\Theta(n^{(\log_b a)} \log^k n)$**, then** $T(n)$ **is** $\Theta(n^{\log_b a} \log^{k+1} n)$**.**

Case 2b For $k = -1$**, if** $f(n)$ **is** $\Theta(n^{(\log_b a)} \log^k n)$**, then** $T(n)$ **is** $\Theta(n^{\log_b a} \log \log n)$**.**

Case 2c For $k < -1$**, if** $f(n)$ **is** $\Theta(n^{(\log_b a)} \log^k n)$**, then** $T(n)$ **is** $\Theta(n^{\log_b a})$**.**

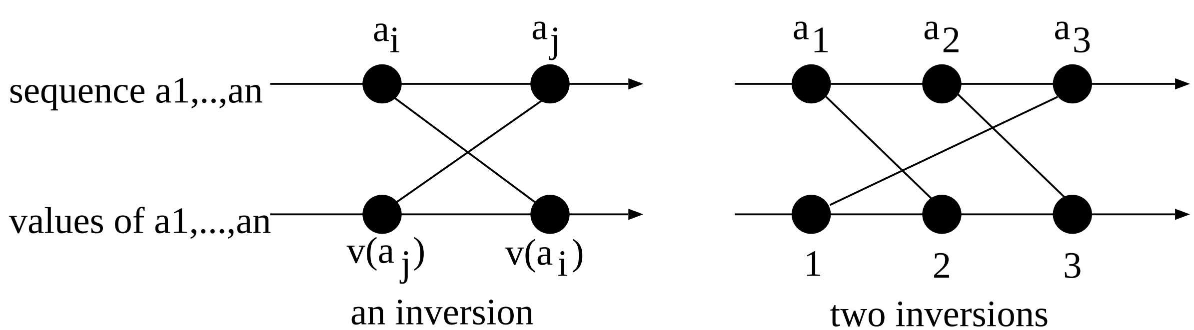
Master theorem examples

- $T(n) = 9T(n/3) + n$. **We have** $a = 9$, $b = 3$, $f(n) = n$. **So,** $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. **Since** $f(n) = O(n^{\log_3 9-\epsilon})$, $\epsilon = 1$ **(**$0 < \epsilon \leq 1$**),** $T(n) = \Theta(n^2)$ **(Case 1).**
- $T(n) = T(2n/3) + 1$. **We have** $a = 1$, $b = 3/2$, $f(n) = 1$. **So,** $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. **Since** $f(n) = \Theta(n^{\log_{3/2} 1}) = \Theta(1)$, $T(n) = \Theta(\log n)$ **(Case 2).**
- $T(n) = 3T(n/4) + n \log n$. **We have** $a = 3$, $b = 4$, $f(n) = n \log n$. **So,** $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. **Since** $f(n) = \Omega(n^{\log_4 3+\epsilon})$, $\epsilon \approx 0.2$, $T(n) = \Theta(n \log n)$ **(Case 3).**
- $T(n) = 2T(n/2) + n \log n$. **We have** $a = 2$, $b = 2$, $f(n) = n \log n$. **So,** $n^{\log_b a} = n$, $f(n) = O(n^{\log_b a} \log^k n)$ **for** $k = 1$. **By Case 2,** $T(n) = O(n \log^{k+1} n) = O(n \log^2 n)$.

Counting Inversions

- **Given a sequence $a_1, .., a_n$ of numbers, a pair a_i and a_j is called an inversion if $i < j$ and $a_i > a_j$.**
- **Counting inversions problem: Given a sequence $a_1, .., a_n$ of numbers, find the number of inversions.**

  Example, for $a_1, a_2, a_3 = 2, 3, 1$, $a_1 > a_3$, $a_2 > a_3$, total 2 inversions.
- **Brute force algorithm, for $i = 1$ to n, check a_i with every a_j with $i < j$, $O(n^2)$ time.**
- **Improvement, use divide and conquer**



Divide and Conquer for Counting Inversions

- **Divide: partition the sequence into two sub-sequences A and B.**

  Conquer: recursively count inversions in each of A and B.

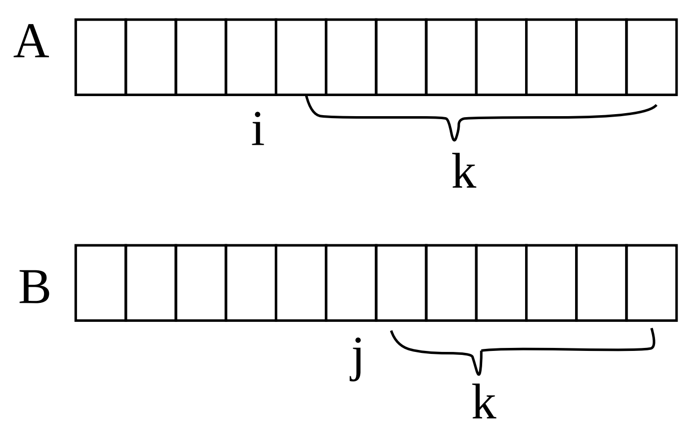
  Combine: count inversions (a, b) with $a \in A$ and $b \in B$, and sum up the three counts (in A, B and between A and B).

- **Count inversions (a, b) with $a \in A$ and $b \in B$ efficiently**

  Sort A and B, compare a_i (from $i = 1$) with b_j (from $j = 1$).

  If $a_i < b_j$, then $a_i < b_k$ for every $k > j$. Stop check a_i.

  If $a_i > b_j$, then $a_k > b_j$ for every $k > i$. When check a_k, start from b_{j+1}.



Sort-and-Count(L)

Input: A sequece L of n numbers.

Output: Number of inversions in L.

if $|L| \leq 1$ **then** return 0

else

Divide L into A and B of size $n/2$;

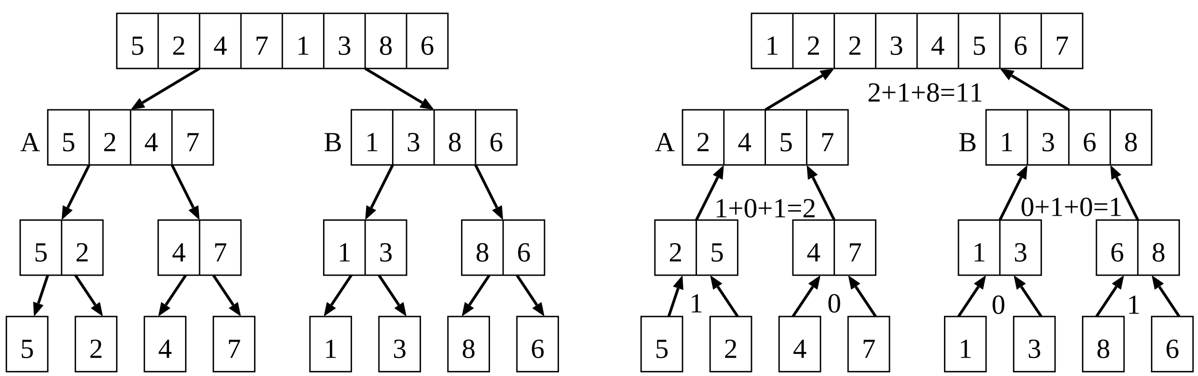
c_A =Sort-and-Count(A);

c_B =Sort-and-Count(B);

c_L =Merge-and-Count(A, B);

end if

Return $c_A + c_B + c_L$ and the sorted L;



Merge-and-Count(A, B)

Input: A sequence A of n_a numbers and a sequence B of n_b numbers.

Output: Number of inversions (a, b) with $a \in A$ and $b \in B$.

$i = 1; j = 1; k = 1;$ count=0;

while $i \neq n_a + 1$ and $j \neq n_b + 1$ **do**

if $A[i] \leq B[j]$ **then**

$L[k] = A[i]; k = k + 1; i = i + 1$

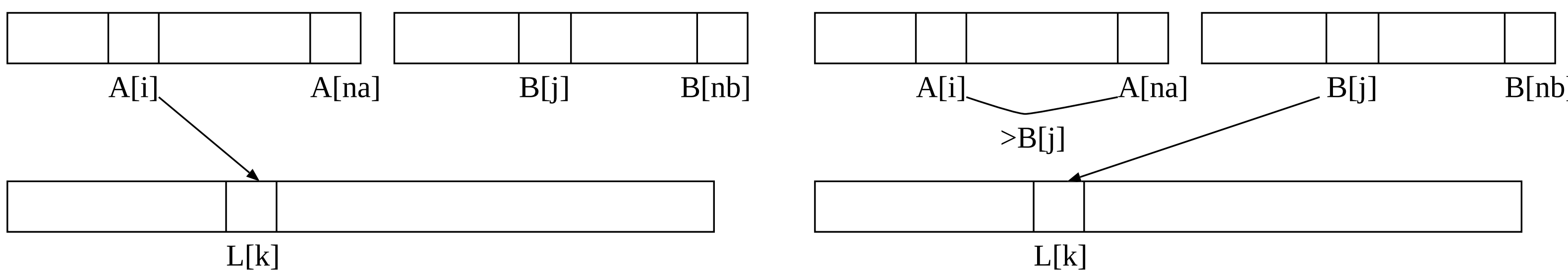
else

$L[k] = B[j]; k = k + 1; j = j + 1;$ count=count$+(n_a - i + 1);$

end if

end while

Return count and L;



A[i]≤B[j], move A[i] to L[k],
next compare A[i+1] with B[j]

A[i]>B[j], move B[j] to L[k],
na-i+1 elementsA[i],..,A[na]>B[j],
next compare A[i] with B[j+1]

- **Sort-and-Count algorithm counts the number of inversions in a permutation of $1, 2, .., n$ in $O(n \log n)$ time.**

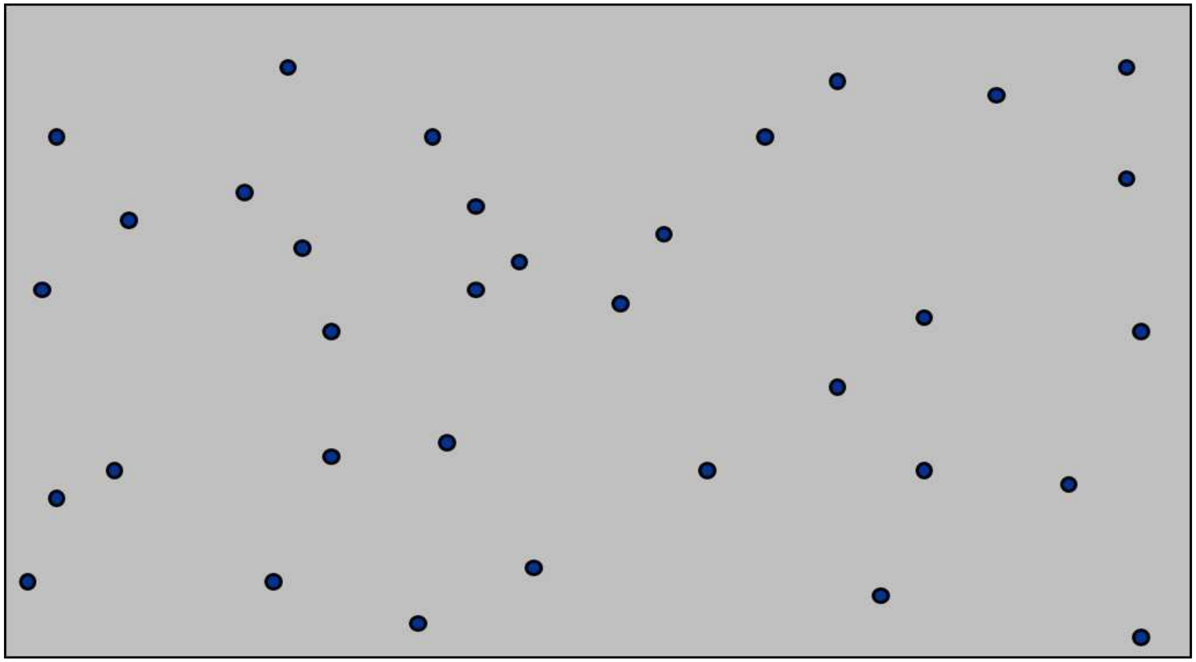
  Proof. It is obvious that the algorithm counts the number of inversions correctly. Let $T(n)$ be the running time of the algorithm. Then

$$T(n) = \begin{cases} \Theta(1) & \text{for } n = 1 \\ 2T(n/2) + \Theta(n) & \text{for } n > 1 \end{cases}$$

  By the master theorem, $T(n) = O(n \log n)$. $\square$

Closest Pair Problem

- **Given n points in the 2-dimensional Euclidean plane, find a pair of points with the smallest Euclidean distance between them.**



Algorithms for Closest Pair Problem

- **Brute force, check all pairs with $\Theta(n^2)$ distance calculations.**
- **Divide-and-conquer,**

  Divide, partition the set of points into two subsets of equal size.

  Conquer, find the closest pair recursively in each subset.

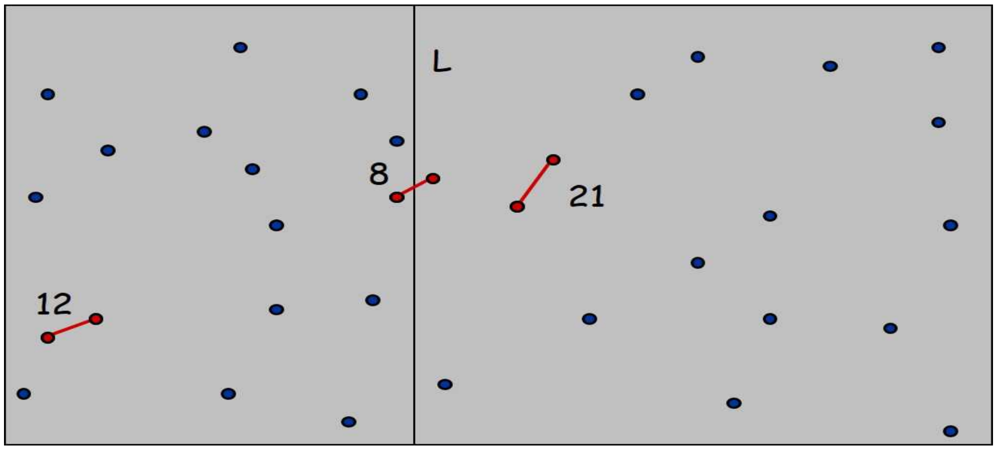
  Combine, check if there is a closer pair with one point in each subset.
- **Partition the point set is a key**

  Assumption, no two points have the same x-coordinate nor the same y-coordinate.

Divide-and-Conquer Algorithm for Closest Pair Problem [Shamos 1975]

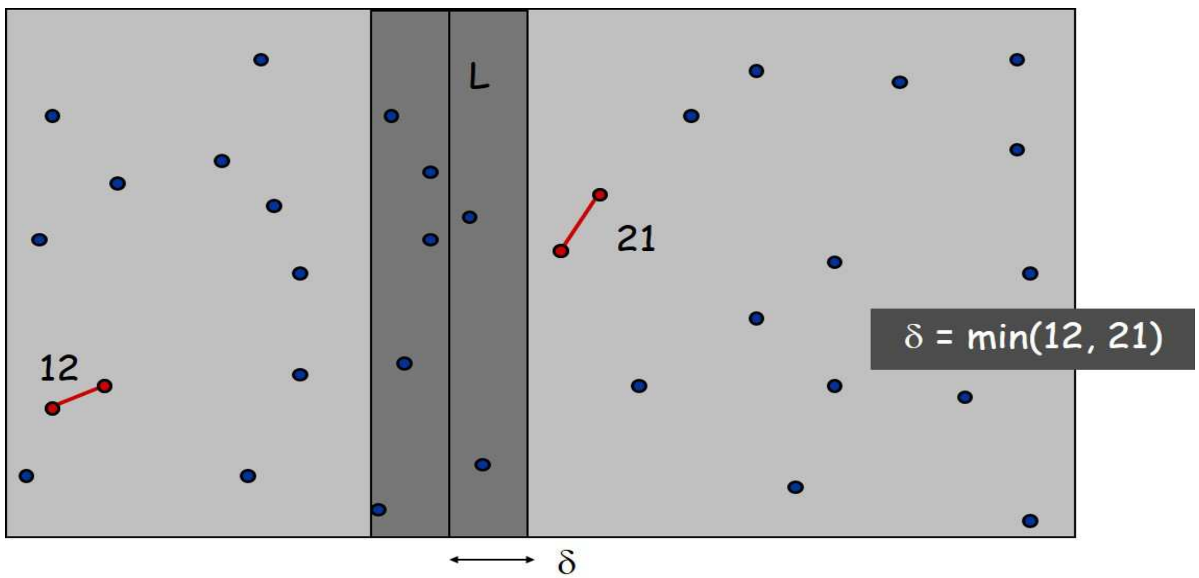
Let P be the set of points.

- **Partition P into Q and R of $n/2$ points by the x-coordinate: draw a vertical line L s.t. points of Q are on one side of L and points of R are on the other side.**
- **Find the closest pair in each of Q and R.**
- **Find the closest pair with one point in each of Q and R.**
- **Return the closest pair from the three solutions.**



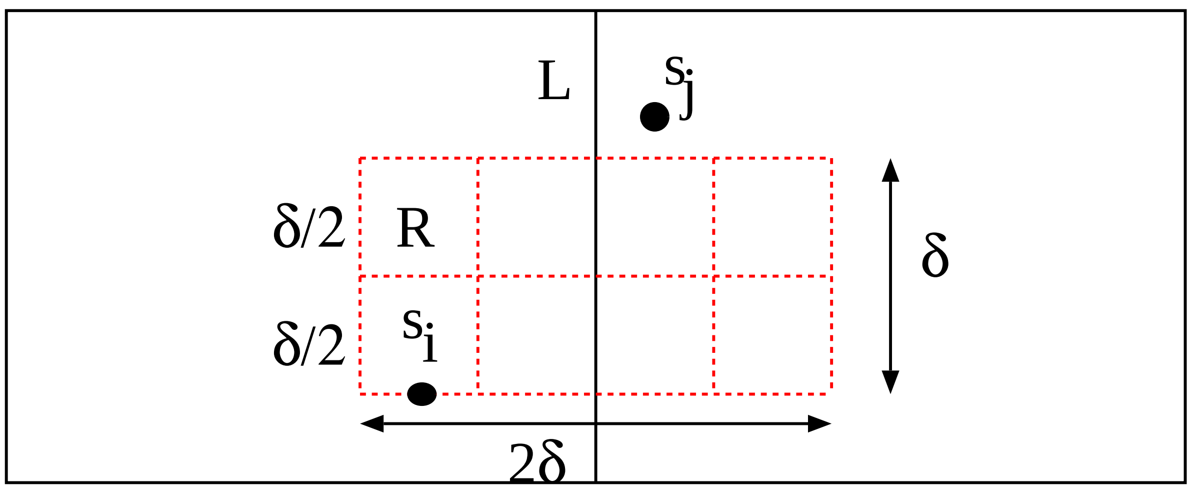
Find the closest pair with one point in each side (1)

- **Let d_q and d_r be the distances of the closest pairs in Q and R, respectively, and let $\delta = \min\{d_q, d_r\}$.**
- **To find the closest pair with one point in each of Q and R, suffice to consider only the points with distance from L smaller than δ.**



Find the closest pair with one point in each side (2)

- **Let S be the set of points in the 2δ-strip around L.**
- **Sort the points of S by the y-coordinate.**
- **If $d(s_i, s_j) < \delta$ for $s_i, s_j \in S$ then $|i - j| \le 7$. So, to check each $s_i \in S$, only $O(1)$ distance calculations are needed.**



Closest-Pair(P)

Input: A set P of n points in plane.

Output: A pair of points with the minimum distance between them.

Construct lists P_x and P_y of points sorted by x-coordinate and y-coordinate, respectively.

$(p, p')=$Closest-Pair-Rec(P, P_x, P_y);

Closest-Pair-Rec(P, P_x, P_y)

if $|P| \leq 3$ **then** return the closest pair by brute force

else

Partition P into Q and R by a vertical line L;

Construct lists Q_x, Q_y and R_x, R_y;

$(q, q')=$Closest-Pair-Rec(Q, Q_x, Q_y); $(r, r')=$Closest-Pair-Rec(R, R_x, R_y);

$\delta = \min\{d(q, q'), d(r, r')\}$;

$S = \{$points within distance δ from $L\}$; construct S_y;

for each $s_i \in S$ **do** compute distance $d(s_i, s_j)$ for $s_j \in S_y$ and $i < j \leq i + 7$;

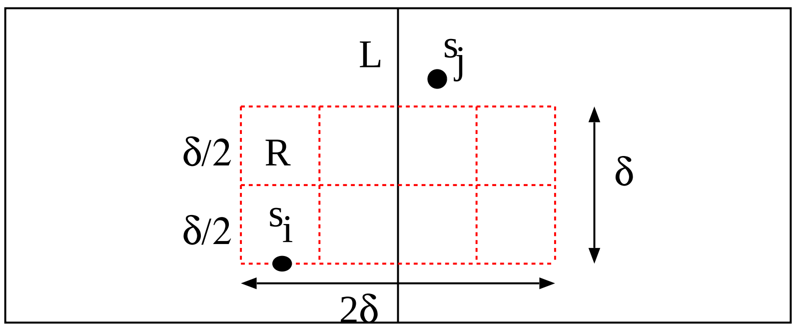
if minimum $d(s_i, s_j) < \delta$ **then** return (s_i, s_j)

else if $d(q, q') < d(r, r')$ **then** return (q, q') **else** return (r, r');

end if

Theorem. [Shamos 1975] Closest-Pair algorithm computes a closest pair of points in $O(n \log n)$ time

Proof. Let $S = \{s_1, .., s_m\}$ s.t. $i < j$ if the y-coordinate of $s_i <$ the y-coordinate of s_j. We claim that if $|j - i| > 7$ then the distance between s_i and $s_j > \delta$. Let R be the 2δ-by-δ rectangle in the strip s.t. the minimum y-coordinate of R equals that of s_i. For any point s_j with y-coordinate $>$ the maximum y-coordinate of R, the distance between s_i and $s_j > \delta$. R can be partitioned into 8 $(\delta/2)$-by-$(\delta/2)$ squares. Each square can have at most one point because the diameter of the square is $\delta/\sqrt{2} < \delta$ and the distance between any pair of points at one side of $L \geq \delta$. So the claim holds. By the claim, the algorithm computes a closest pair.



Let $T(n)$ be the running time of the algorithm. Then $T(n) = 2T(n/2) + O(n) + t(n)$, where $t(n)$ is the time to sort n points by x-coordinate and y-coordinate. If the points are sorted from scratch in each, then $t(n) = O(n \log n)$ and $T(n) = O(n \log^2 n)$. Since sorted lists of subsets of points are available at each recursive call, the sorted lists of points can be obtained by merging the lists from the recursive calls. This takes $O(n)$ time and $T(n) = O(n \log n)$. $\square$