Advanced Algorithms Analysis and Design

By

Nazir Ahmad Zafar

Lecture No 14

Designing Algorithms using Divide & Conquer Approach

Today Covered

Divide and Conquer?

- A General Divide and Conquer Approach
- Merge Sort algorithm
- Finding Maxima in 1-D, and 2-D
- Finding Closest Pair in 2-D

Divide and Conquer Approach

A General Divide and Conquer Algorithm

Step 1:

- If the problem size is small, solve this problem directly
- Otherwise, split the original problem into 2 or more sub-problems with almost equal sizes.

Step 2:

- Recursively solve these sub-problems by applying this algorithm.

Step 3:

- Merge the solutions of the sub- problems into a solution of the original problem.

Time Complexity of General Algorithms

- Time complexity:

$$T(n) = \begin{cases} 2T(n/2) + S(n) + M(n) & , n \geq c \\ b & , n < c \end{cases}$$

  - where S(n) is time for splitting
  - M(n) is time for merging
  - b and c are constants

Example

- Binary search
- Quick sort
- Merge sort

Merge-sort

Merge-sort

Merge-sort is based on divide-and-conquer approach and can be described by the following three steps:

Divide Step:

- If given array A has zero or one element, return S.
- Otherwise, divide A into two arrays, A1 and A2,
- Each containing about half of the elements of A.

Recursion Step:

- Recursively sort array A1, A2

Conquer Step:

- Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence.

Visualization of Merge-sort as Binary Tree

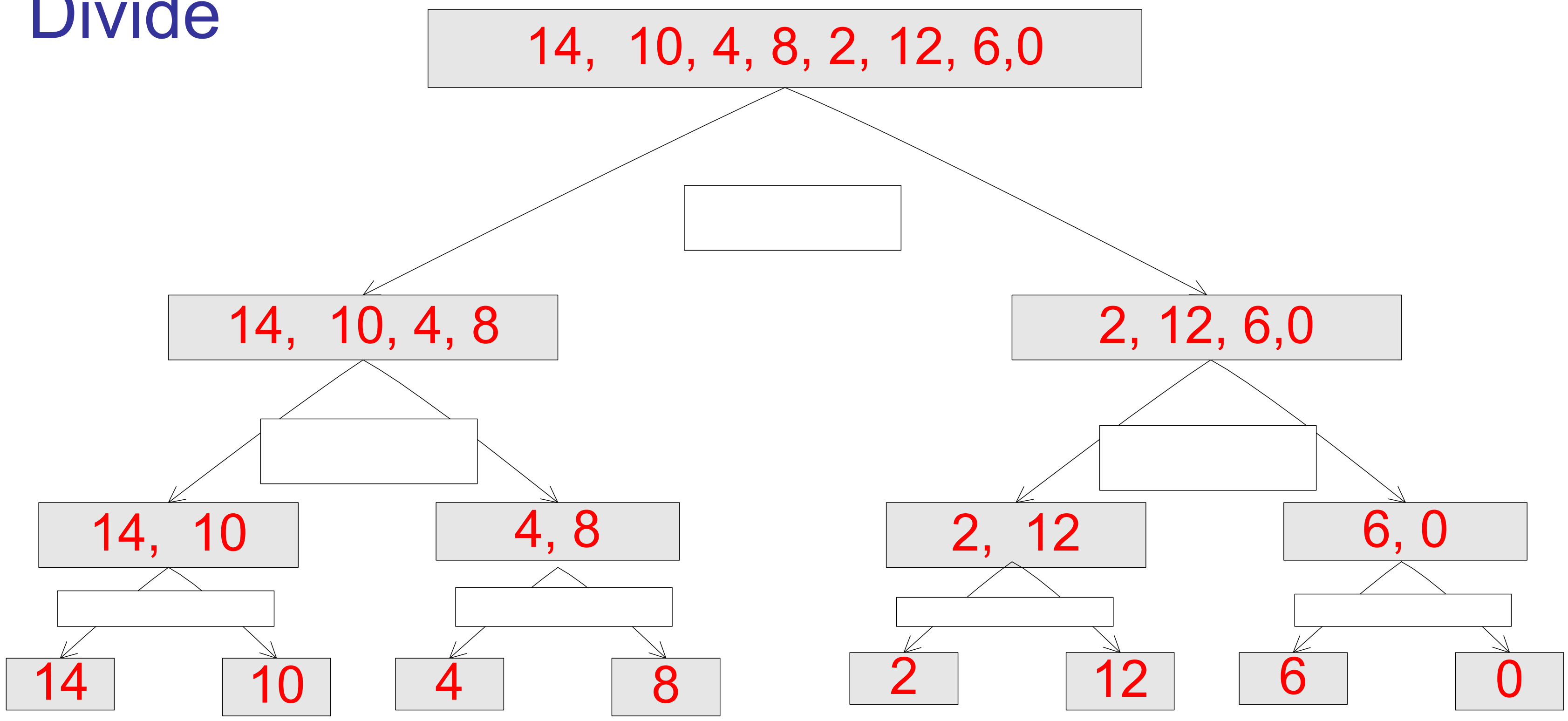
- We can visualize Merge-sort by means of binary tree where each node of the tree represents a recursive call
- Each external node represents individual elements of given array A.
- Such a tree is called Merge-sort tree.
- The heart of the Merge-sort algorithm is conquer step, which merge two sorted sequences into a single sorted sequence
- The merge algorithm is explained in the next

Sorting Example: Divide and Conquer Rule

- Sort the array [14, 10, 4, 8, 2, 12, 6, 0] in the ascending order

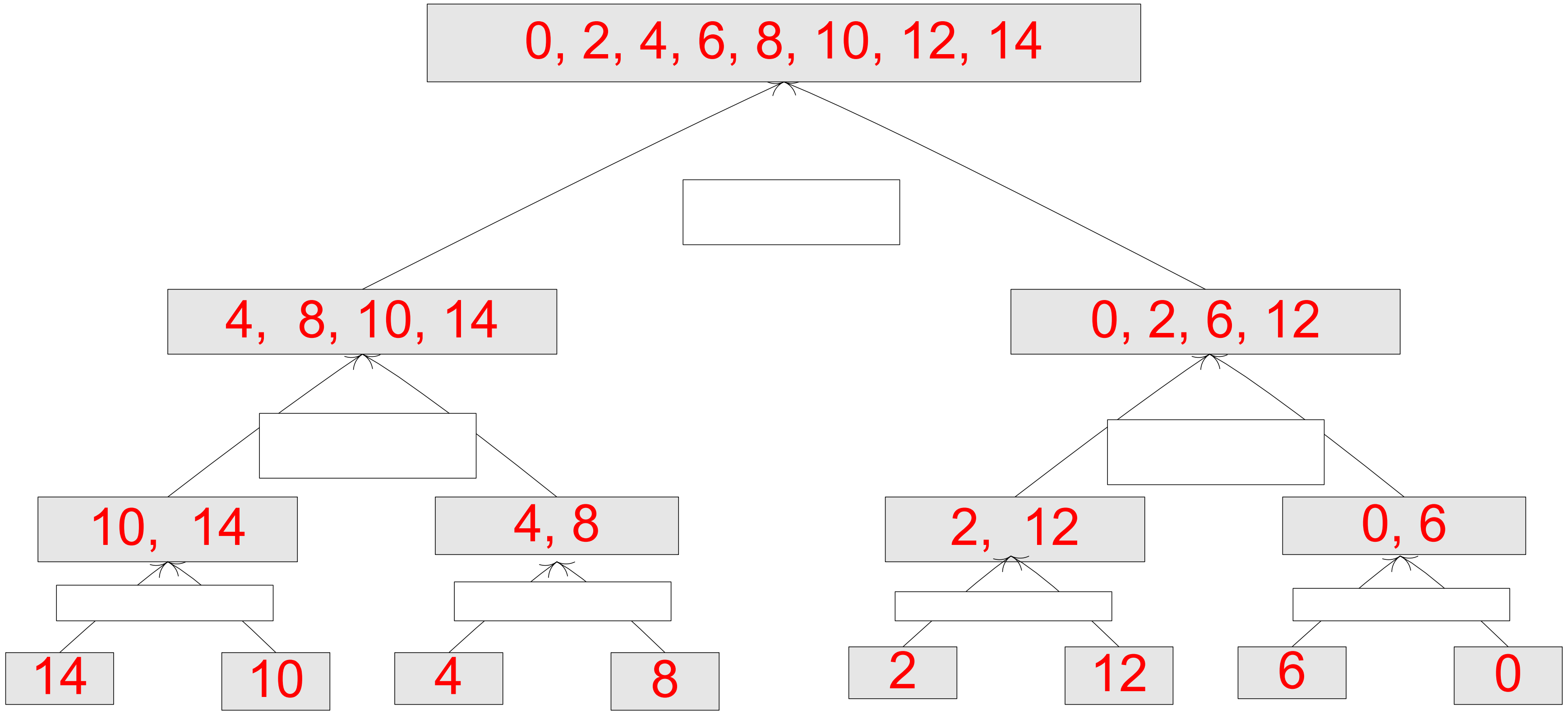
Solution:

- Divide



Contd..

- Recursion and Conquer



Merge-sort Algorithm

Merge-sort(A, f, l)

1. **if** f < l
2. **then** m = (f + l)/2
3. Merge-sort(A, f, m)
4. Merge-sort(A, m + 1, l)
5. Merge(A, f, m, l)

Merge-sort Algorithm

Merge(A, f, m, l)

1. T[f..l] \\declare temporary array of same size
2. i ← f; k ← f; j ← m + 1 \\initialize integers i, j, and k
3. **while** (i ≤ m) and (j ≤ l)
4. **do if** (A[i] ≤ A[j]) \\comparison of elements
5. **then** T[k++] ← A[i++]
6. **else** T[k++] ← A[j++]
7. **while** (i ≤ m)
8. **do** T[k++] ← A[i++] \\copy from A to T
9. **while** (j ≤ l)
10. **do** T[k++] ← A[j++] \\copy from A to T
11. **for** i ← p to r
12. **do** A[i] ← T[i] \\copy from T to A

Analysis of Merge-sort Algorithm

- Let $T(n)$ be the time taken by this algorithm to sort an array of n elements dividing A into sub-arrays A_1 and A_2.
- It is easy to see that the Merge (A_1, A_2, A) takes the linear time. Consequently,

$$T(n) = T(n/2) + T(n/2) + \theta(n)$$
$$T(n) = 2T(n/2) + \theta(n)$$

- The above recurrence relation is non-homogenous and can be solved by any of the methods
  - Defining characteristics polynomial
  - Substitution
  - recursion tree or
  - master method

Analysis: Substitution Method

$$T(n) = 2.T(\frac{n}{2}) + n$$

$$T(\frac{n}{2}) = 2.T(\frac{n}{2^2}) + \frac{n}{2}$$

$$T(\frac{n}{2^2}) = 2.T(\frac{n}{2^3}) + \frac{n}{2^2}$$

$$T(\frac{n}{2^3}) = 2.T(\frac{n}{2^4}) + \frac{n}{2^3} \quad \ldots$$

$$T(\frac{n}{2^{k-1}}) = 2.T(\frac{n}{2^k}) + \frac{n}{2^{k-1}}$$

Analysis of Merge-sort Algorithm

$$T(n) = 2.T(\frac{n}{2}) + \Theta(n) = 2^2.T(\frac{n}{2^2}) + n + n$$

$$T(n) = 2^2.T(\frac{n}{2^2}) + n + n$$

$$T(n) = 2^3.T(\frac{n}{2^3}) + n + n + n$$

$$\ldots$$

$$T(n) = 2^k.T(\frac{n}{2^k}) + \underbrace{n + n + \ldots + \mathrm{n}}_{k-times}$$

Analysis of Merge-sort Algorithm

$$T(n) = 2^k . T(\frac{n}{2^k}) + \underbrace{n + n + \ldots + \mathrm{n}}_{k-times}$$

$$T(n) = 2^k . T(\frac{n}{2^k}) + k.n$$

$$\text{Let us suppose that}: n = 2^k \Rightarrow \log_2 n = k$$

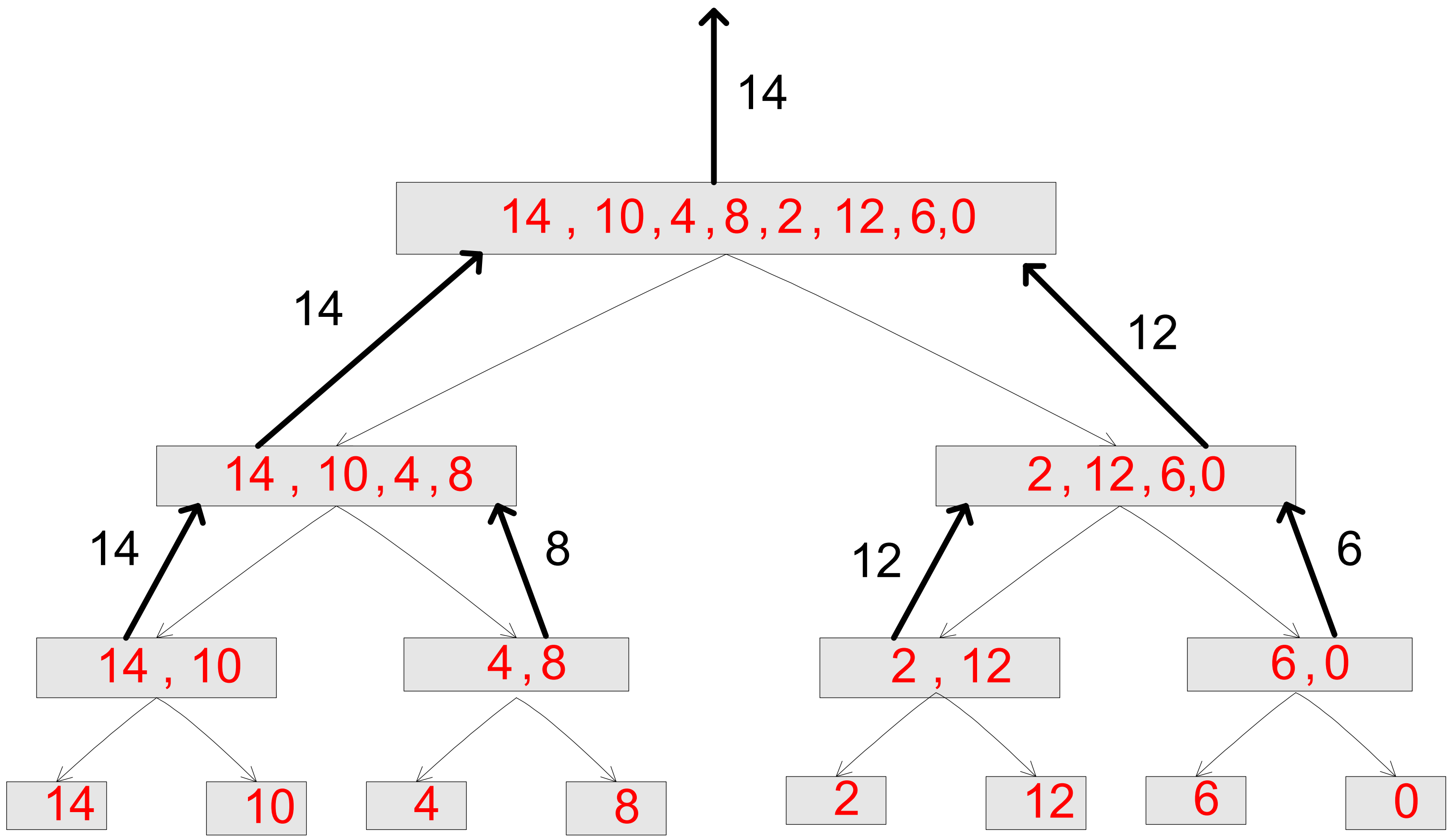
$$\text{Hence}, T(n) = n.T(1) + n.\log_2 n = n + n.\log_2 n$$

$$T(n) = \Theta(n.\log_2 n)$$

Searching: Finding Maxima in 1-D

A Simple Example in 1-D

Finding the maximum of a set S of n numbers



Time Complexity

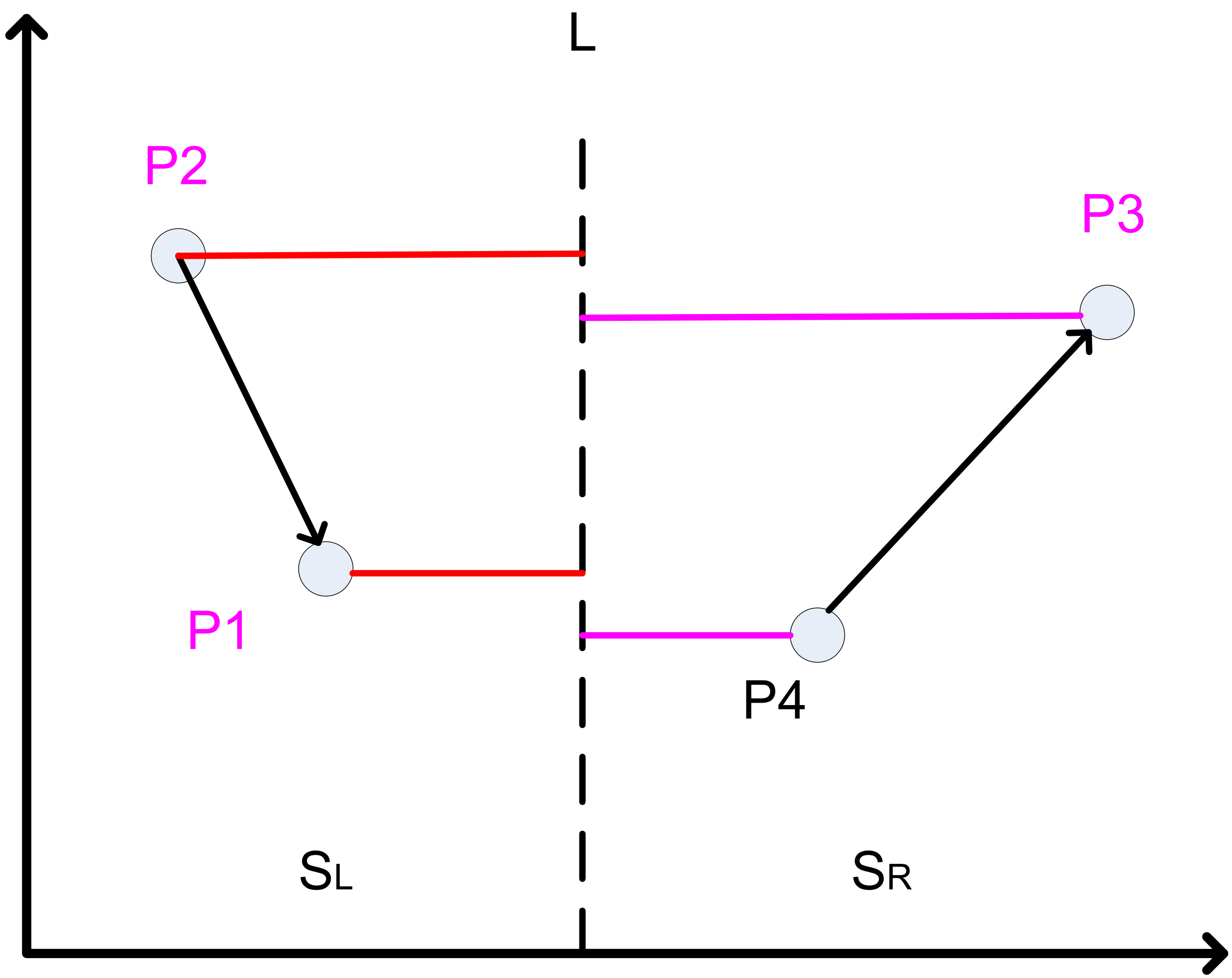
$$T(n) = \begin{cases} 2T(n/2) + 1, & n > 2 \\ 1, & n \leq 2 \end{cases}$$

- Assume $n = 2^k$, then

$$
\begin{aligned}
T(n) &= 2T(n/2) + 1 = 2(2T(n/4) + 1) + 1 \\
&= 2^2T(n/2^2) + 2 + 1 \\
&= 2^2(2T(n/2^3) + 1) + 2 + 1 \\
&= 2^3T(n/2^3) + 2^2 + 2^1 + 1 \\
&\quad \vdots \\
&= 2^{k-1}T(n/2^{k-1}) + 2^{k-2} + \ldots + 2^2 + 2^1 + 1 \\
&= 2^{k-1}T(2) + 2^{k-2} + \ldots + 2^2 + 2^1 + 1 \\
&= 2^{k-1} + 2^{k-2} + \ldots + 4 + 2 + 1 = 2^k - 1 = n - 1 = \Theta(n)
\end{aligned}
$$

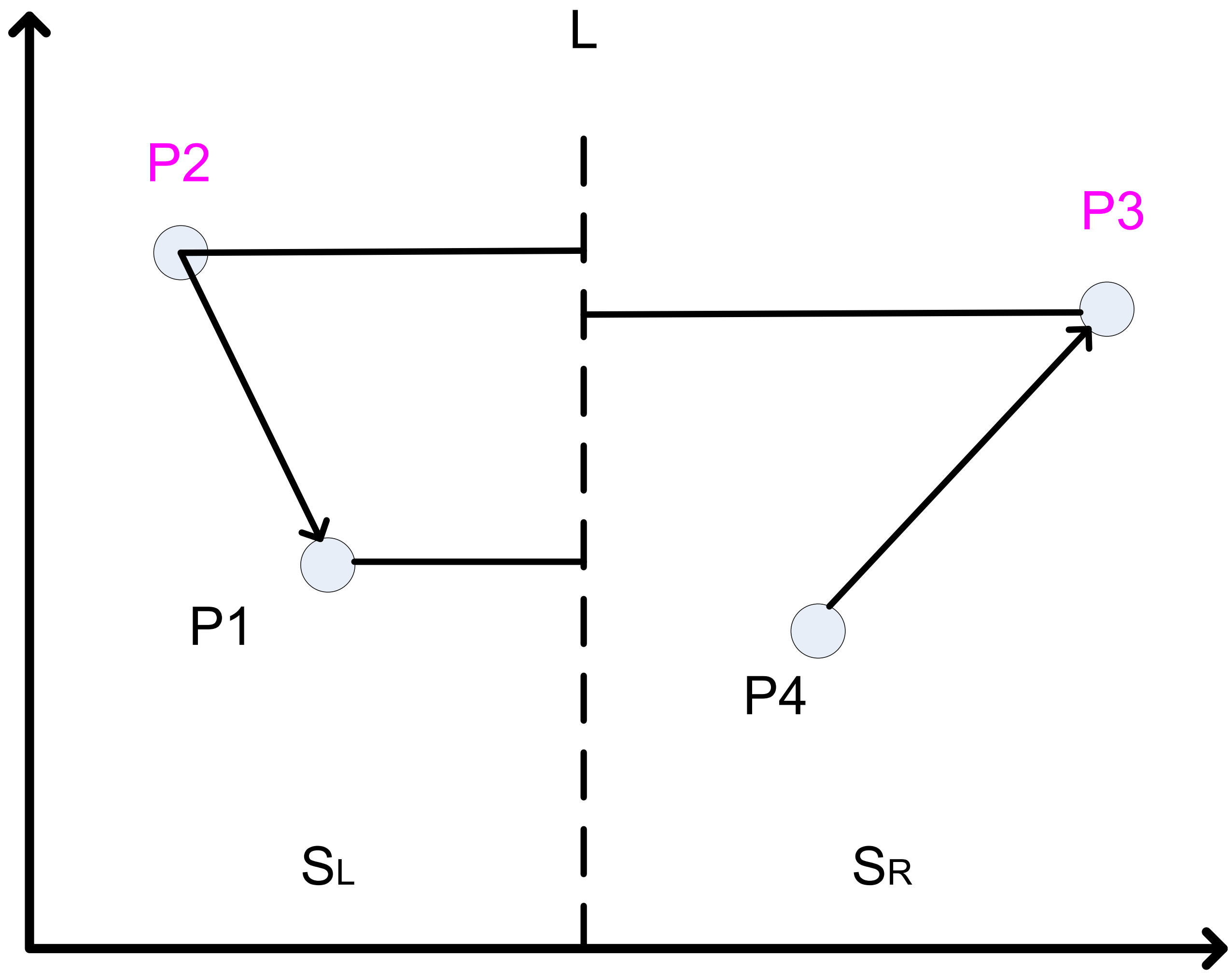
Finding Maxima in 2-D using Divide and Conquer

How to Find Maxima in 2-D



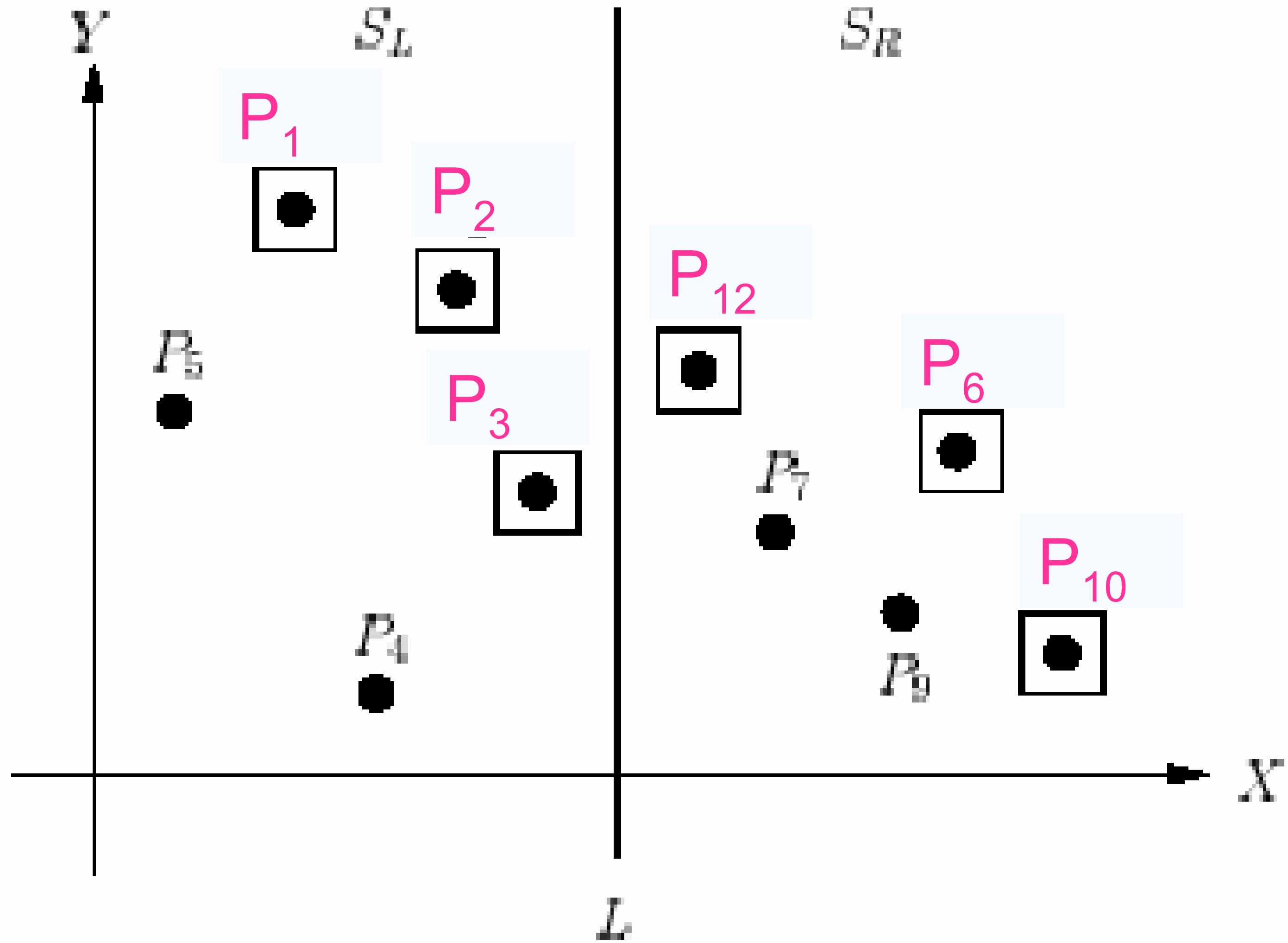
$\{P_1, P_2\}$ both maximal in S_L and $\{P_3\}$ only maxima in S_R

Merging S_L and S_R



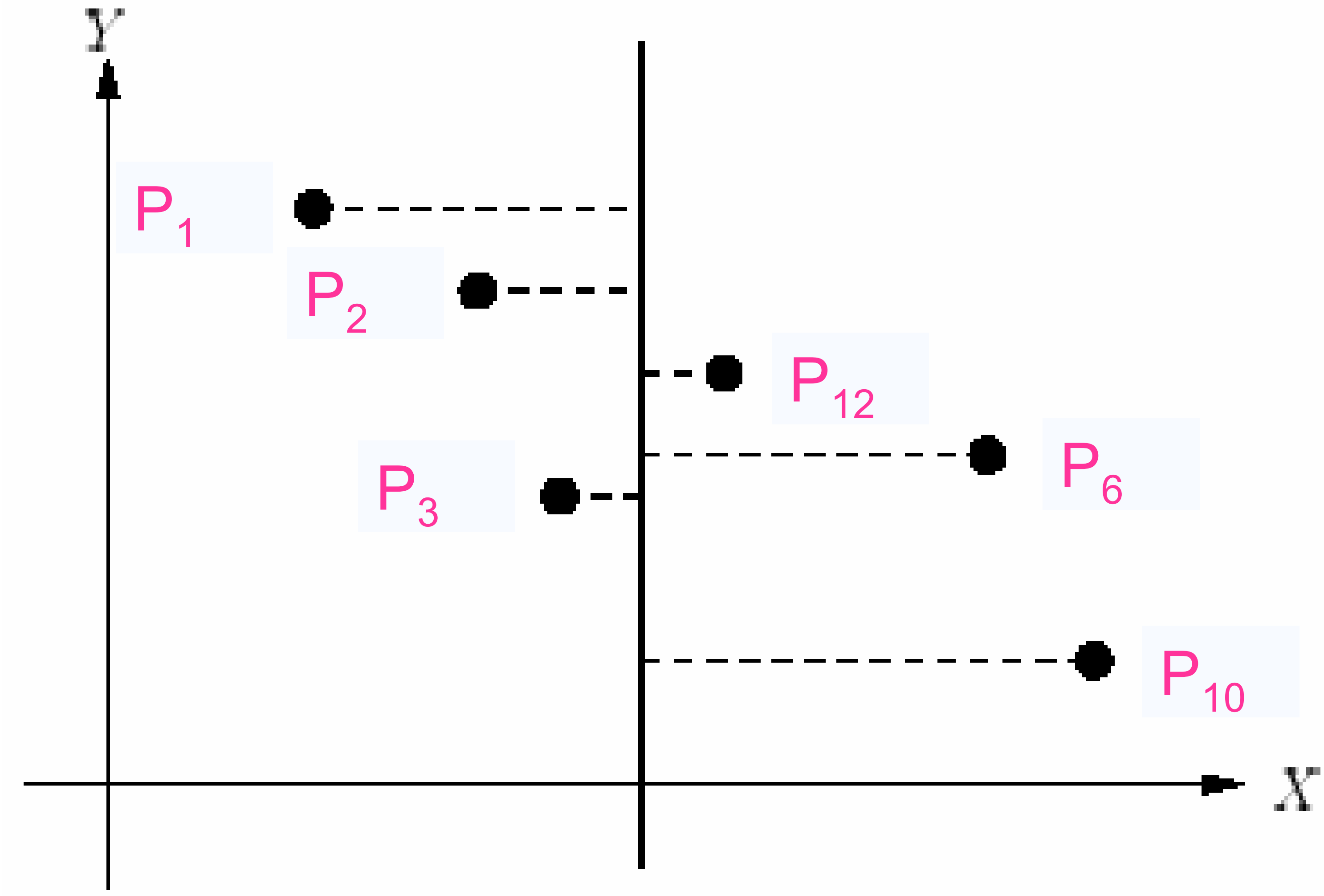
After Merging Maximal in S_L and S_R we get $\{P_2, P_3\}$ only maximal

Divide and Conquer for Maxima Finding Problem



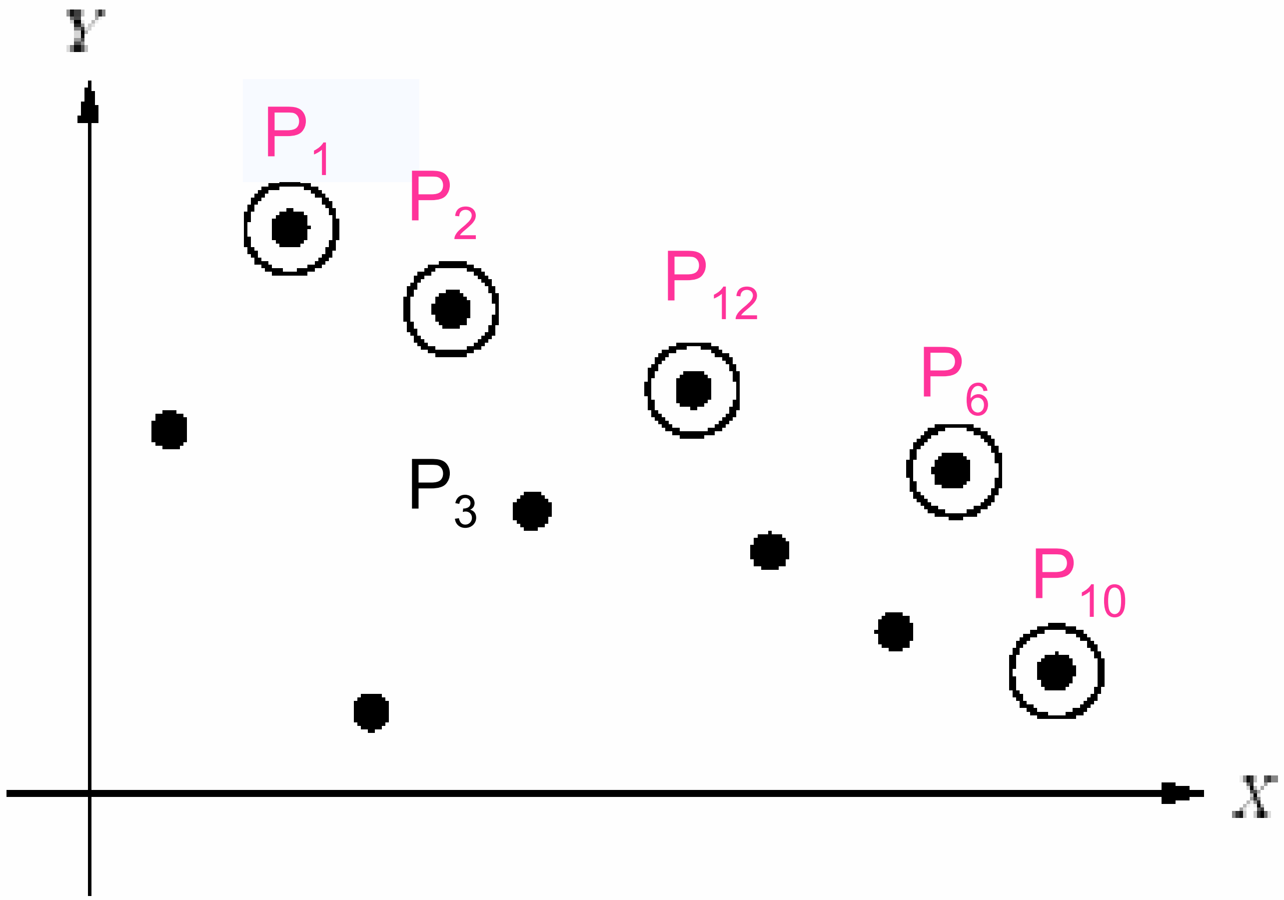
The maximal points of S_L and S_R

Divide and Conquer for Maxima Finding Problem



P3 is not maximal point of S_L

2-D Maxima Finding Problem



Algorithm: Maxima Finding Problem

Input: A set S of 2-dimensional points.

Output: The maximal set of S.

Maxima(P[1..n])

1. Sort the points in ascending order w. r .t. X axis
2. If |S| = 1, then return it, else
   find a line perpendicular to X-axis which separates S into S_L and S_R, each of which consisting of n/2 points.
3. Recursively find the maxima's S_L and S_R
4. Project the maxima's of S_L and S_R onto L and sort these points according to their y-values.
5. Conduct a linear scan on the projections and discard each of maxima of S_L if its y-value is less than the y-value of some maxima's of S_R .

Time Complexity

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n) & , n \geq 2 \\ 1 & , n < 2 \end{cases}$$

Assume $n = 2^k$, then

$$\begin{aligned} T(n) &= 2T(n/2) + n + n \\ &= 2(2T(n/4) + n/2 + n/2) + n + n \\ &= 2^2T(n/2^2) + n + n + n + n \\ &= 2^2T(n/2^2) + 4n \\ &= 2^2(2T(n/2^3) + n/4 + n/4) + 4n \\ &= 2^3T(n/2^3) + n + n + 6n \end{aligned}$$

Time Complexity

$$T(n) = 2^3T(n/2^3) + n + n + 6n$$

.

.

$$T(n) = 2^kT(n/2^k) + 2kn$$

$$= 2^kT(2^k/2^k) + 2kn \qquad \text{Since } n = 2^k$$

Hence

$$T(n) = 2^k + 2kn$$

$$T(n) = 2^k + 2kn \qquad n = 2^k \Rightarrow k = \log(n)$$

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

Necessary Dividing Problem into two Parts?

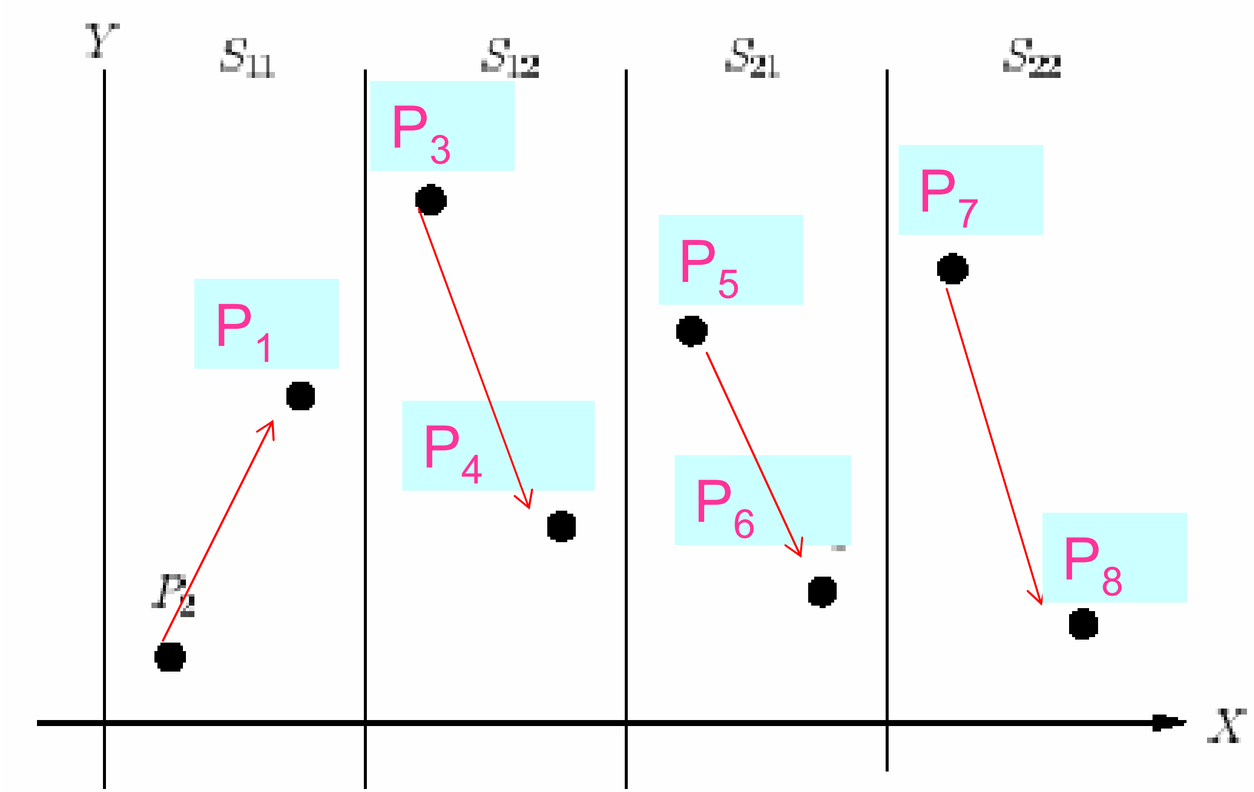
Maximal Points: Dividing Problem into four Parts

Maximal points in $S_{11} = \{P_1\}$

Maximal points in $S_{12} = \{P_3, P_4\}$

Maximal points in $S_{21} = \{P_5, P_6\}$

Maximal points in $S_{22} = \{P_7, P_8\}$



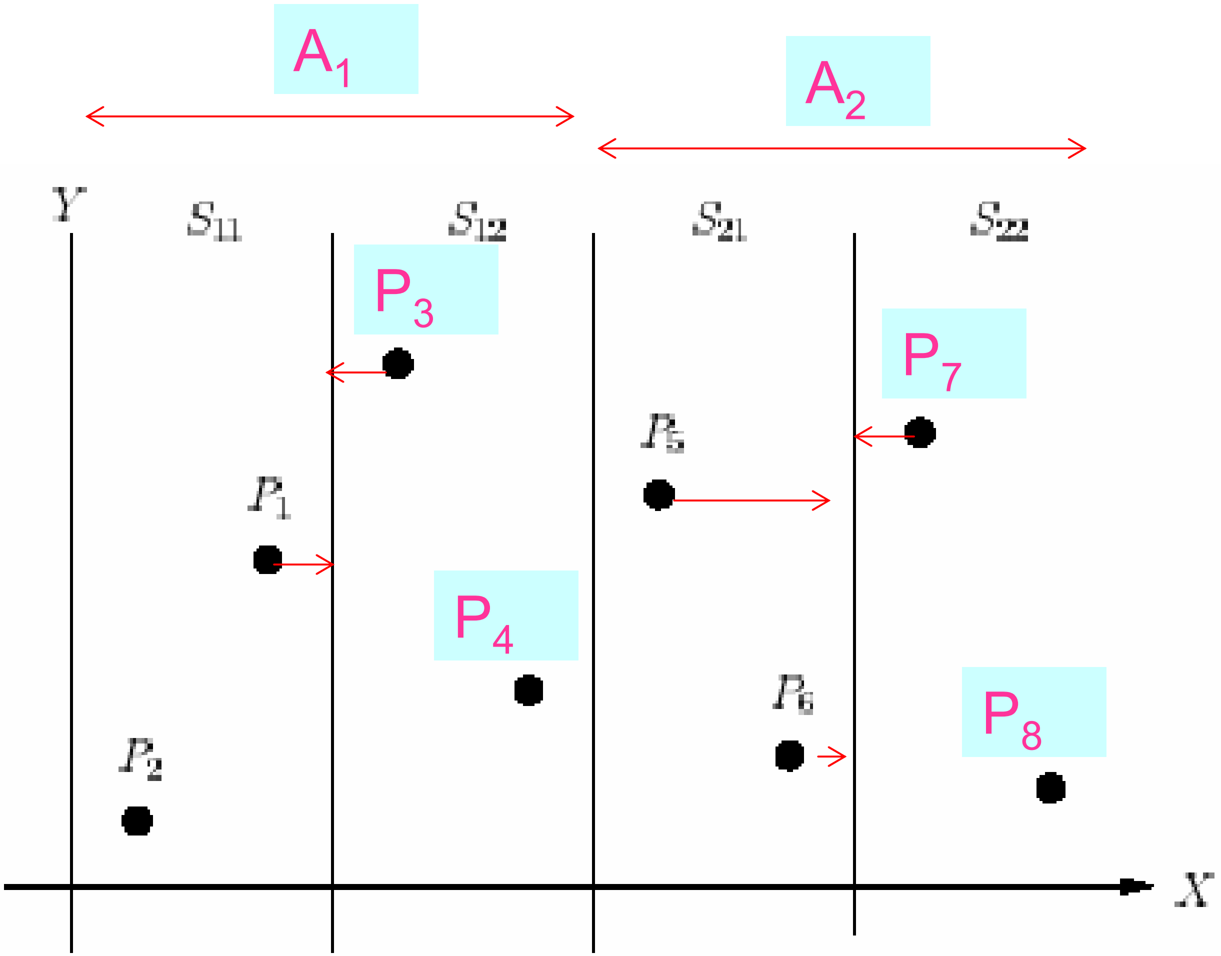
Maximal Points: Dividing Problem into four Parts

Merging S_{12}, S_{12}

$A_1 = \{P_3, P_4\}$

Merging S_{21}, S_{22}

$A_2 = \{P_7, P_8\}$



Maximal Points: Dividing Problem into four Parts

Merging S_{12}, S_{12}

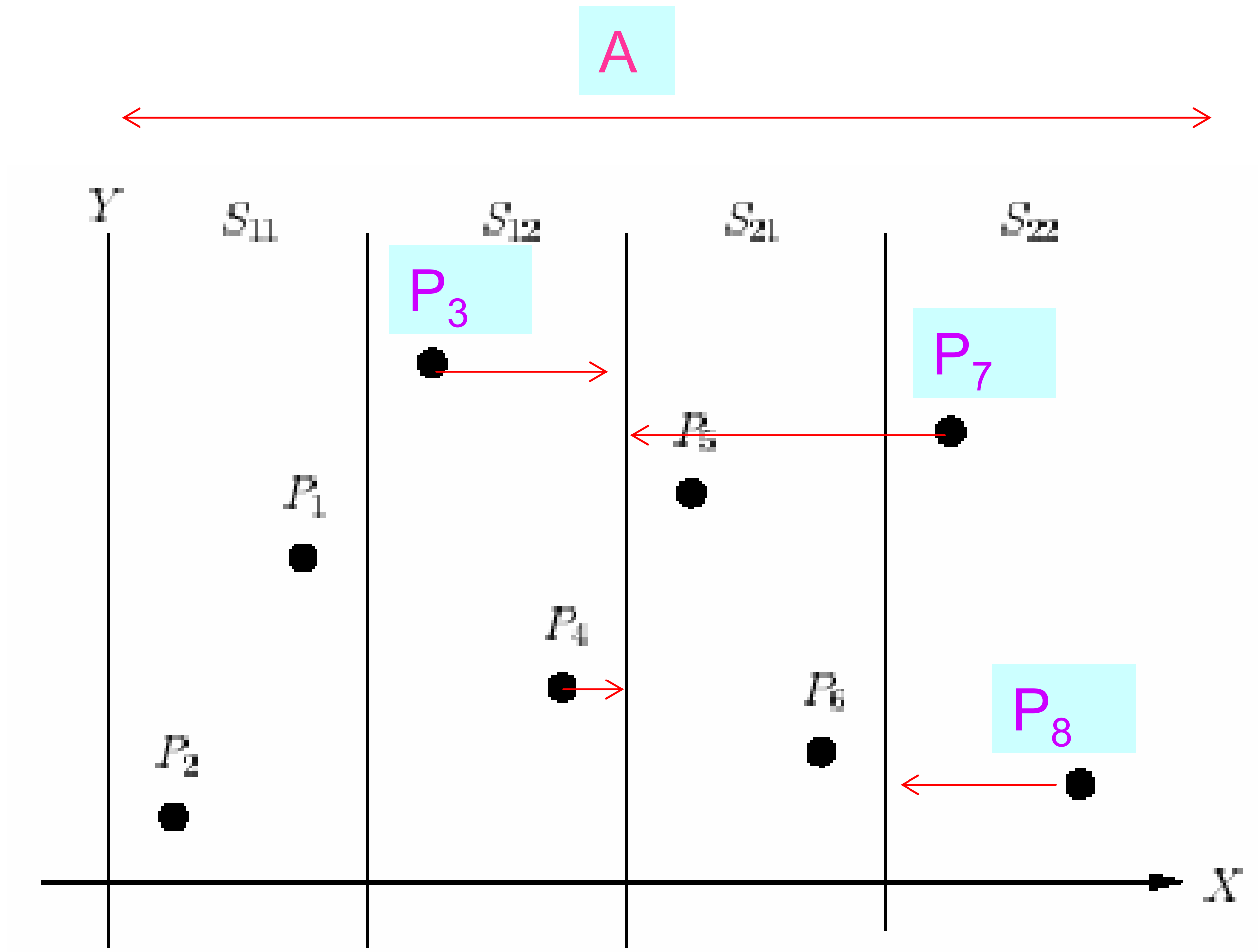
$A_1 = \{P_3, P_4\}$

Merging S_{21}, S_{22}

$A_2 = \{P_7, P_8\}$

Merging A_1, A_2

$A = \{P_3, P_7, P_8\}$



Finding Closest Pair in 2-D

Problem

The closest pair problem is defined as follows:

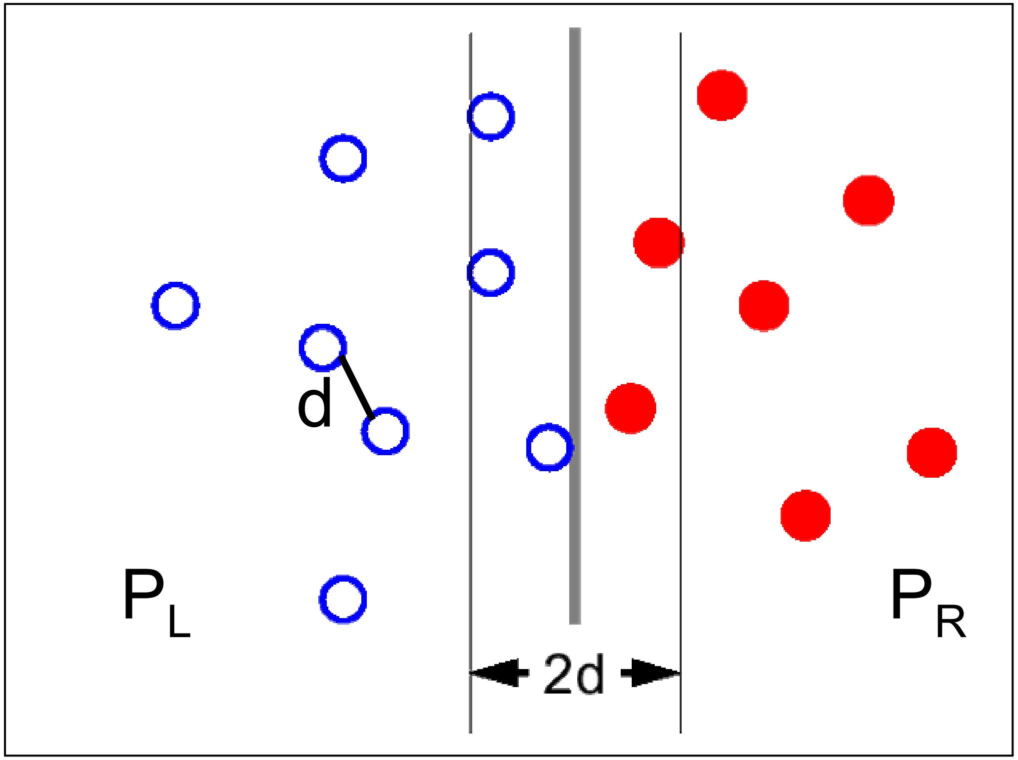
- Given a set of n points
- Determine the two points that are closest to each other in terms of distance.
- Furthermore, if there are more than one pair of points with the closest distances, all such pairs should be identified.

Closest Pair: Divide and Conquer Approach

- First we sort the points on x-coordinate basis, and divide into left and right parts

  $p_1\ p_2\ \ldots\ p_{n/2}$ and $p_{n/2+1}\ \ldots\ P_n$

- Solve recursively the left and right sub-problems
- Let d = min $\{d_l, d_r\}$,
- How do we combine two solutions to sub-problems?

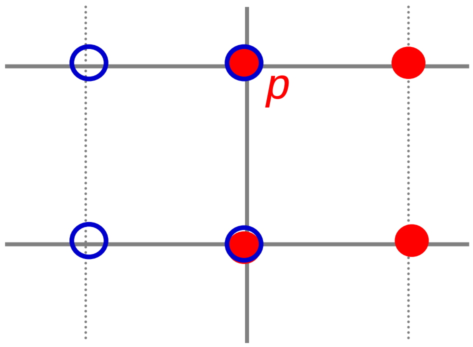


Closest Pair: Divide and Conquer Approach

- How do we combine two solutions?
  - Let d = min $\{d_l, d_r\}$, where d is distance of closest pair where both points are either in left or in right
  - Something is missing. We have to check where one point is from left and the other from the right.
  - Such closest-pair can only be in a strip of width 2d around the dividing line, otherwise the points would be more than d units apart.
- Combining solutions:
- Finding the closest pair in a strip of width 2d, knowing that no one in any two given pairs is closer than d

Closest Pair: Divide and Conquer Approach

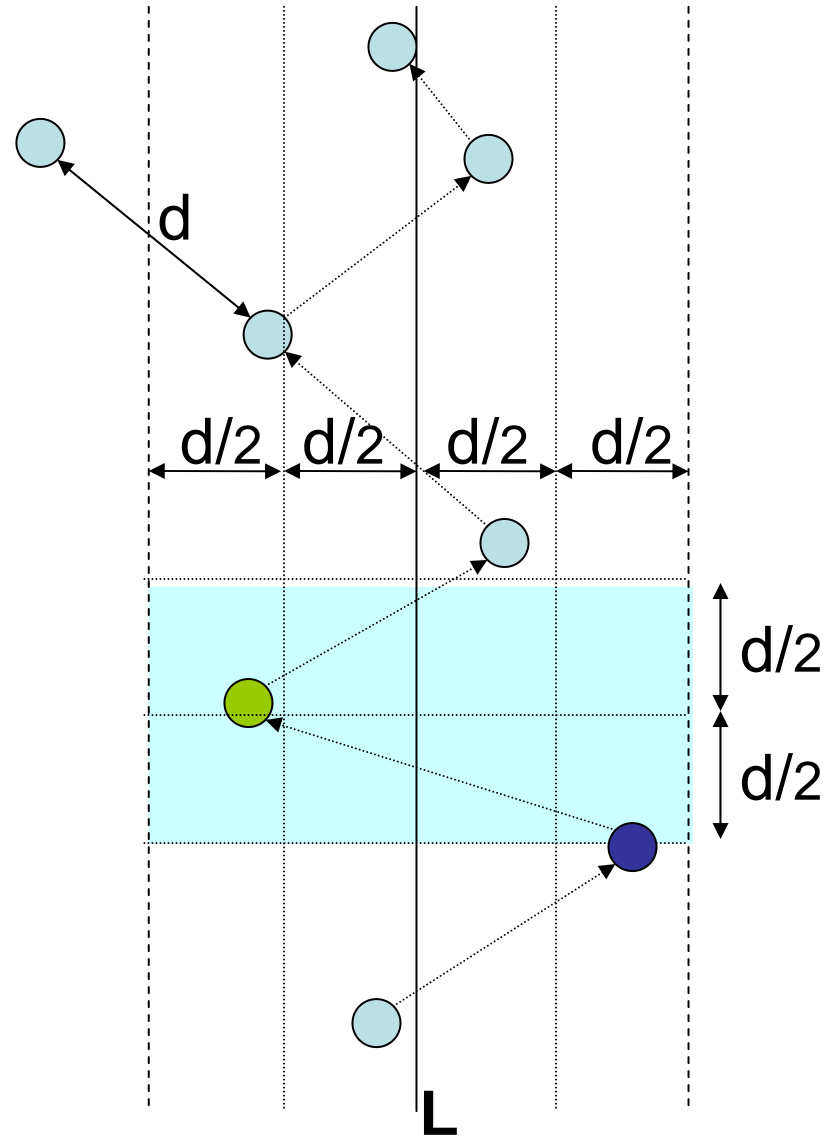
- Combining solutions:
- For a given point p from one partition, where can there be a point q from the other partition, that can form the closest pair with p?
- How many points can there be in this square?
  - At most 4



- Algorithm for checking the strip:
  - Sort all the points in the strip on the y-coordinate
  - For each point p only 7 points ahead of it in the order have to be checked to see if any of them is closer to p than d

Closest Pair: Divide and Conquer Approach

1. Partition the strip into squares of length d/2 as shown in the picture.
2. Each square contains at most 1 point by definition of d.
3. If there are at least 2 squares between points then they can not be the closest points.
4. There are at most 8 squares to check.



Closest Pair: Divide and Conquer Approach

```
Closest-Pair(P, l, r)
if r – l < 3 then return ClosestPairBF(P)
q ← ⌈(l+r)/2⌉
dl ← Closest-Pair(P, l, q-1)
dr ← Closest-Pair(P, q, r)
d ← min(dl, dr)
for i ← l to r do
  if P[q].x - d ≤ P[i].x ≤ P[q].x + d then
      append P[i] to S
Sort S on y-coordinate
for j ← 1 to size_of(S)-1 do
   Check if any of d(S[j],S[j]+1), ...,
   d(S[j],S[j]+7) is smaller than d, if so set
   d to the smallest of them
return d
```

Closest Pair: Divide and Conquer Approach

Running Time

- Running time of a divide-and-conquer algorithm can be described by a recurrence
  - Divide = $O(1)$
  - Combine = $O(n \lg n)$
  - This gives the recurrence given below
  - Total running time: O(n log2 n)

$$T(n) = \begin{cases} n & n \leq 3 \\ 2T(\frac{n}{2}) + n \log n & \text{otherwise} \end{cases}$$

Improved Version: Divide and Conquer Approach

- Sort all the points by x and y coordinate once
- Before recursive calls, partition the sorted lists into two sorted sublists for the left and right halves, it will take simple time $O(n)$
- When combining, run through the y-sorted list once and select all points that are in a $2d$ strip around partition line, again time $O(n)$
- New recurrence:

$$T(n) = \begin{cases} n & \text{n} \leq 3 \\ 2T(\frac{n}{2}) + n & \text{otherwise} \end{cases}$$

Conclusion

- Brute Force approach is discussed, design of some algorithms is also discussed.
- Algorithms computing maximal points is generalization of sorting algorithms
- Maximal points are useful in Computer Sciences and Mathematics in which at least one component of every point is dominated over all points.
- In fact we put elements in a certain order
- For Brute Force, formally, the output of any sorting algorithm must satisfy the following two conditions:
  - Output is in decreasing/increasing order and
  - Output is a permutation, or reordering, of input.