Designing Algorithms with Divide-and-Conquer

Lecture 06.03

by Marina Barsky

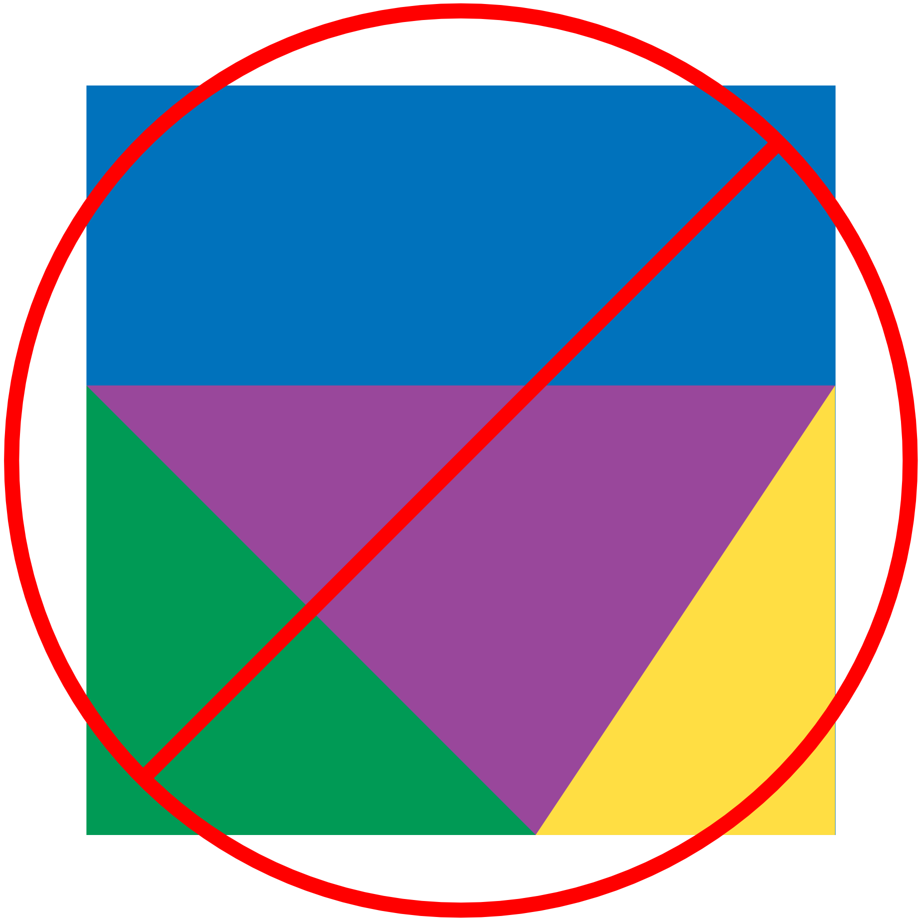
Main algorithm design strategies

- ✓ ***Exhaustive Computation*.** Generate every possible candidate solution and select an optimal solution.
- ✓ ***Greedy. Create next candidate solution one step at a time by using some greedy choice.***
- ***Divide and Conquer*.** Divide the problem into non-overlapping subproblems of the same type, solve each subproblem with the same algorithm, and combine sub-solutions into a solution to the entire problem.
- ***Dynamic Programming*.** Start with the smallest subproblem and combine optimal solutions to smaller subproblems into optimal solution for larger subproblems, until the optimal solution for the entire problem is constructed.
- ***Iterative Improvement.*** Perform multiple iterations of the algorithm, at each iteration moving closer to the optimal solution, until no further improvement is possible.

Big problem to be solved

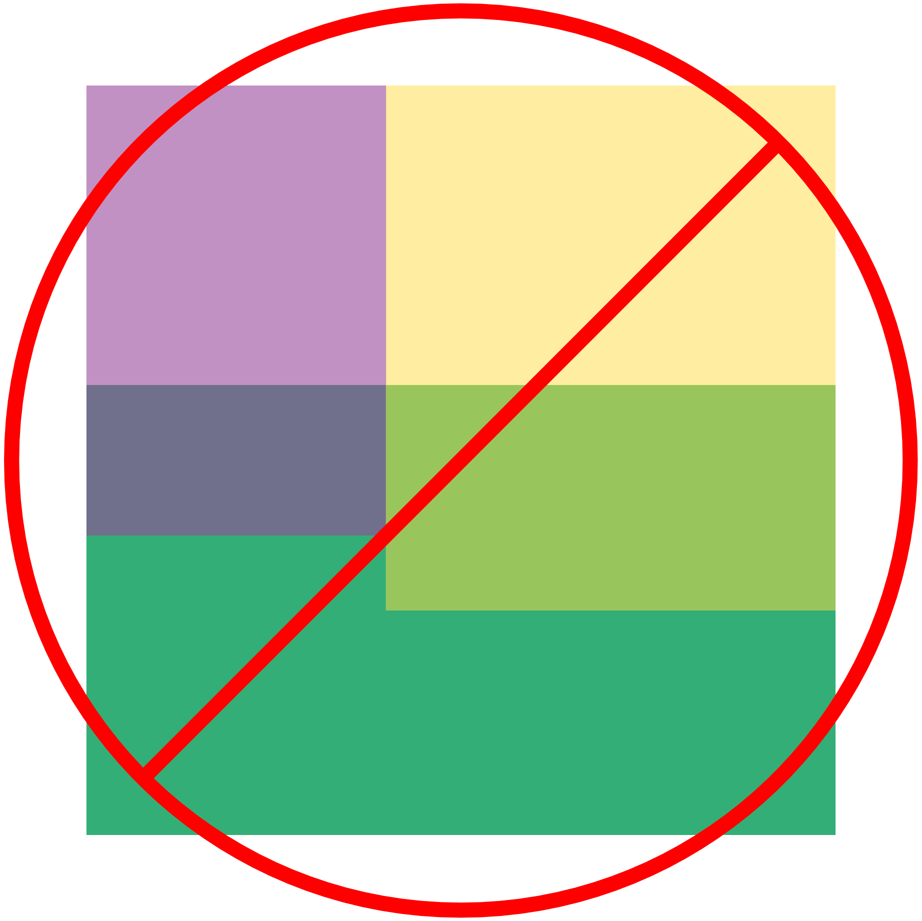
Divide: Break into non-overlapping subproblems of the same type

Problem



not the
same type

Problem



overlapping

Divide-and-conquer steps

1. Break into *non-overlapping* subproblems *of the same type*
2. Solve subproblems
3. **Combine results**

Most difficult!

Two examples:

- Counting inversions
- Closest pair

Counting inversions

Motivation

- Music site tries to match user song preferences with others.
- I rank *n* songs.
- Music site consults database to find people with similar tastes.

songs

	A	B	C	D	E	F
me	1	2	3	4	5	6

you	1	3	4	2	5	6

How similar are me and you?

SImilarity of rankings

- Similarity metric:
  number of *inversions* between two rankings.
- My rank: 1,2,3,4,5,6
- Your rank: 1,3,4,2,5,6
  - for the same songs

songs

	A	B	C	D	E	F
me	1	2	3	4	5	6
you	1	3	4	2	5	6

For a perfect match you should have ranked D at 4, but you ranked it at 2

Definition

An *inversion* is a pair (A[i], A[j]) of array elements such that index i<j and A[i] > A[j]

i	1	2	3	4	5	6
A[i]	1	3	4	2	5	6

2 inversions in total:
(3,2) and (4,2)

Problem: counting inversions

Input: an array A of length n with numbers $1,2,\ldots n$ in some order

Output: number of *inversions*: number of pairs $A[i],A[j]$ of array elements with $i<j$ and $A[i] > A[j]$

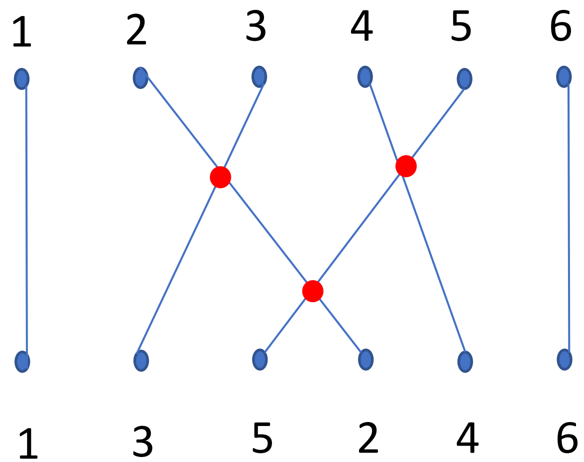
- If A is sorted – what is the number of inversions?
- What is the number of inversions if A is reversed?
- What is the number of inversions in A=[1,3,5,2,4,6]?

Example

- A=[1, 3, 5, 2, 4, 6]

- Inversions:

(3,2), (5,2), (5,4)



What is the largest-possible number of inversions that a 6-element array can have?

Brute-force algorithm for counting inversions

Algorithm count_naive (array A of n integers)

```
count:= 0
for i from 1 to n-1:
          for j from i+1 to n:
                    if A[j] < A[i]
                              count:= count + 1
return count
```

Complexity?

Can we do better?

But how can we do better if total number of inversions is $O(n^2)$???

Idea 1: Divide + Conquer

After dividing array into 2 halves, n/2 each:

For each (i,j) recursively determine if (A[i],A[j]) is an inversion

There are 3 possible cases (3 types of inversions):

Left inversions : if $i,j \le n/2$

Right inversions: if $i,j > n/2$

These two can be computed recursively

Split inversions : if $i \le n/2$ and $j > n/2$

But how to compute these?

5, 3 $\quad \frac{n}{2} \quad$ 2, 1

Developing recursive algorithm

count (array A of length n)

```
if n=1
        return 0
Else
        x = count (1st half of A, n/2)
        y = count (2nd half of A, n/2)
        z = count_split_inv(A, n)
return  x+y+z
```

We do not know how to do that

If we manage to do *CountSplitInv* in O(n) time

then *Count* will run in O(n log n) - just like Merge Sort

Idea 2. What if we use *merge* from merge sort?

- Have recursive calls both *count inversions and sort*

- It turns out that the *merge* subroutine **automatically** recovers inversions!

Recursive Algorithm (in progress)

sort_count (array A of length n)

if n=1
    return (A,0)
Else
    (B, x) = sort_count (1^{st} half of A, n/2)
    (C, y) = sort_count (2^{nd} half of A, n/2)
    (D, z) = **count_split_inv**(B,C)
return (D, x+y+z)

B- sorted 1^{st} half of A

C- sorted 2^{nd} half of A

We still do not know how to do that

If we manage to do *count_split_inv* in O(n) time then *sort_count* will run in O(*n log n*) - just like Merge Sort

merge subroutine: from Merge Sort

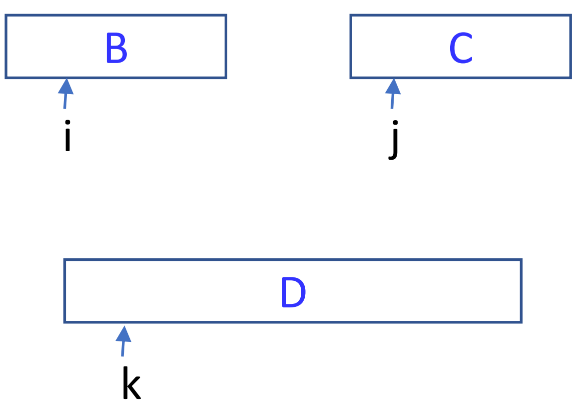
D = will contain sorted array

B = 1st sorted subarray [1:n/2]

C = 2nd sorted subarray [n/2:n]

i = 1

j = 1




```
for k: = 1 to n
        if B[i]< C[j]
                D[k]: = B[i]
                i:= i+1
        else if C[j] < B[i]
                D[k]: = C[j]
                j:= j+1

…
```

Stop and think

Suppose the input array A **has no split inversions**.

B C

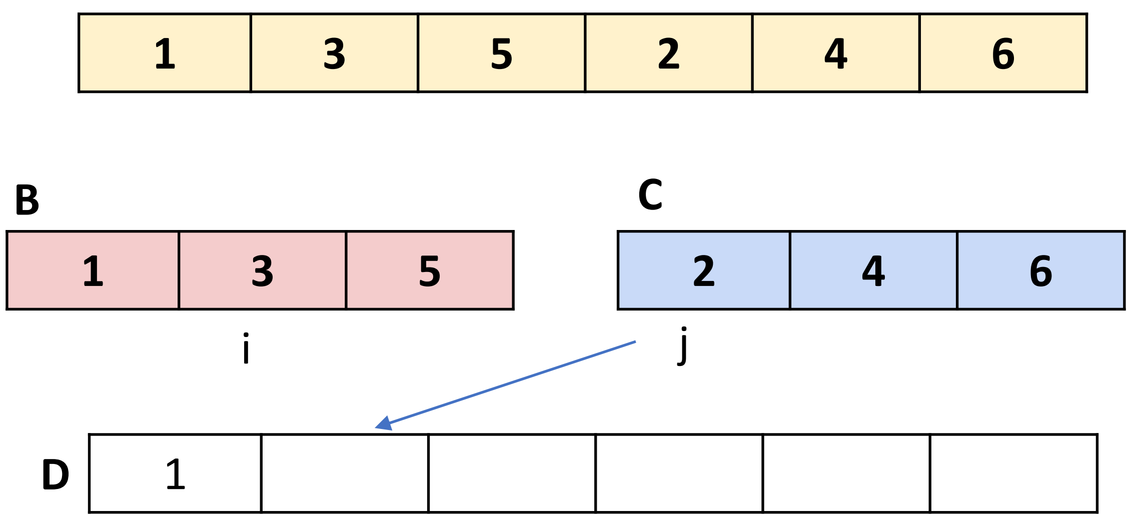
What is the relationship between the sorted subarrays B and C?

A. B has the smallest element of A, C has the second--smallest, B has the third- smallest, and so on.

A. All elements of B are less than all elements of C.

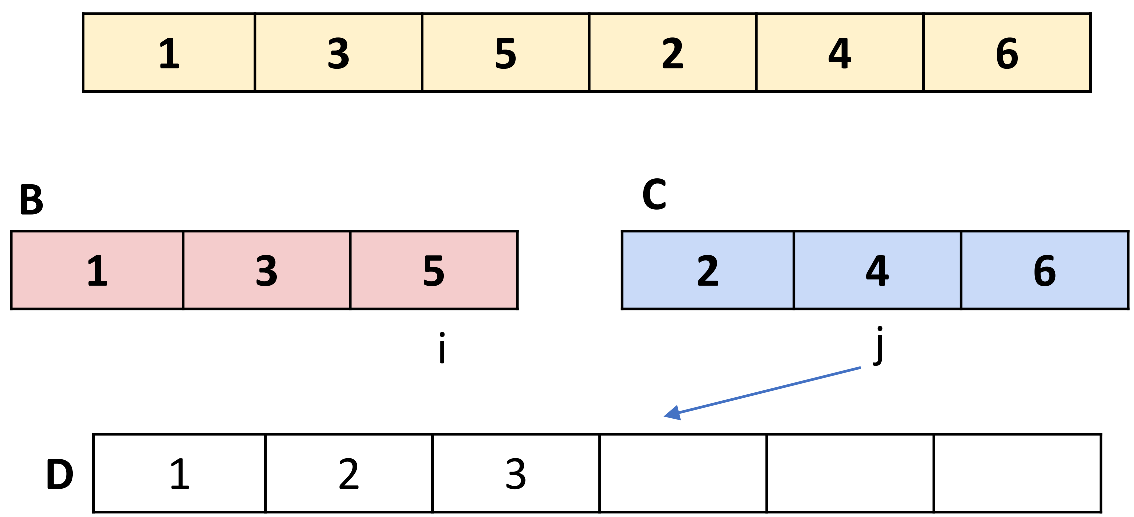
A. There is not enough information to answer this question.

Sample merge



Discovered 2 inversions:
(3,2) and (5,2)

Sample merge



Discovered inversion
(5,4)

General claim

The split inversions involving an element *y* of the 2nd array *C* are precisely the numbers left in the 1st array B when *y* is copied to the output *D*.

Proof:

Let *x* be an element of the 1st array B.

- If x copied to output D before y, then x < y
  => no inversions involving x and y

- If y copied to output D before x, then y < x
  ⇒ x and all elements after it are (split) inversions. ■

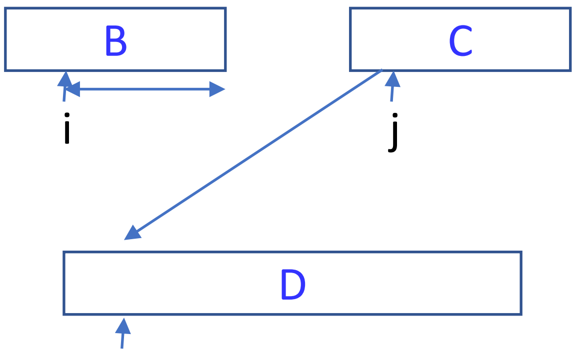
Recursive Algorithm (revised)

sort_count_inv (array A of length n)

if n=1
        return (A, 0)
Else
        (B, x) = sort_count_inv(1st half of A)
        (C, y) = sort_count_inv(2nd half of A)
        (D, z) = merge_**count_split_inv**(B,C)
return (D, x+y+z)

Split inversions are recovered during the merge of the sorted sub-arrays

Merge and count



- While merging the two sorted subarrays, keep running total of number of split inversions

- When element of 2nd array *C* gets copied to output *D*, increment *total* by number of elements remaining in 1st array *B*

merge running total

Runtime of *merge_count_split_inv*: O(n) + O(n) = O(n)

sort_count_inv **runs in O(n log n) time**

just like Merge Sort

Closest pair

Motivation

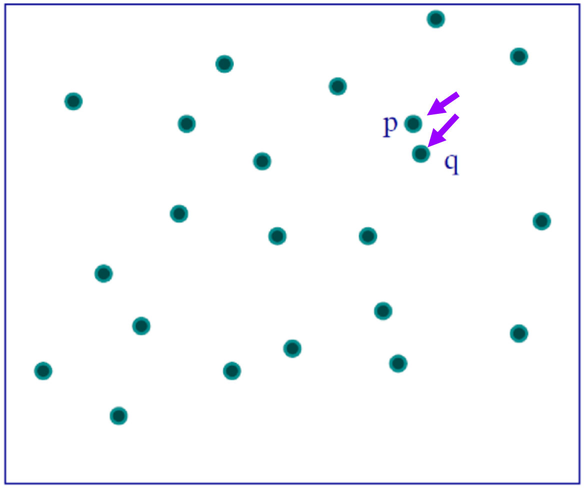
The closest-pair is a subroutine for:

- Dynamic minimum spanning trees
- Straight skeletons and roof design
- Ray-intersection diagram
- Collision detection applications
- Hierarchical clustering
- Traveling salesman heuristics
- Greedy matching
- …

"A pair of the closest points, the one lying on a robot and the other on its obstacles, yields the most important information for generation of obstacle-avoiding robot motions. " *ref*

Closest Pair Problem

- **Input**: *n* points in *d* dimensions
- **Output**: two points *p* and *q* whose mutual distance is smallest



A naive algorithm takes $O(dn^2)$ time.

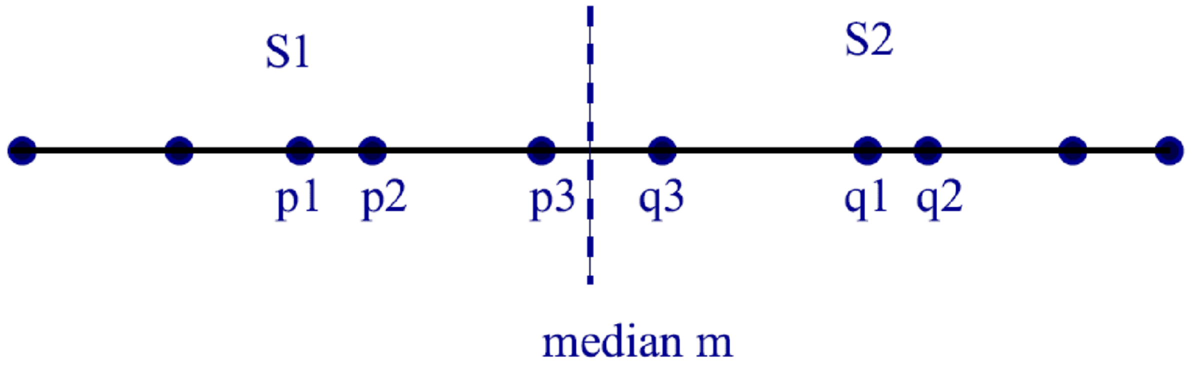
(Number of dimensions *d* can be assumed a constant for a given problem)

Can we do better?

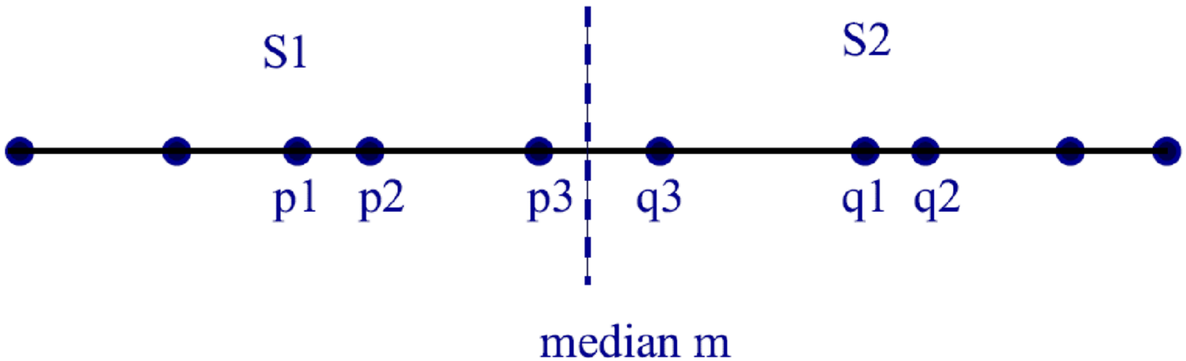
Closest pair in one dimension

Can be solved in O(n log$n$) via sorting, and then linear scanning.
Let's develop a **recursive** solution to find the closest pair

- If the points are sorted by their coordinate:
- Divide the points set S into 2 sets S_1, S_2, by median x-coordinate m such that $p<q$ for all $p \in S_1$ and $q \in S_2$
- Recursively compute closest pair (p_1,p_2) in S_1 and (q_1,q_2) in S_2



Closest pair in one dimension: *combine* step



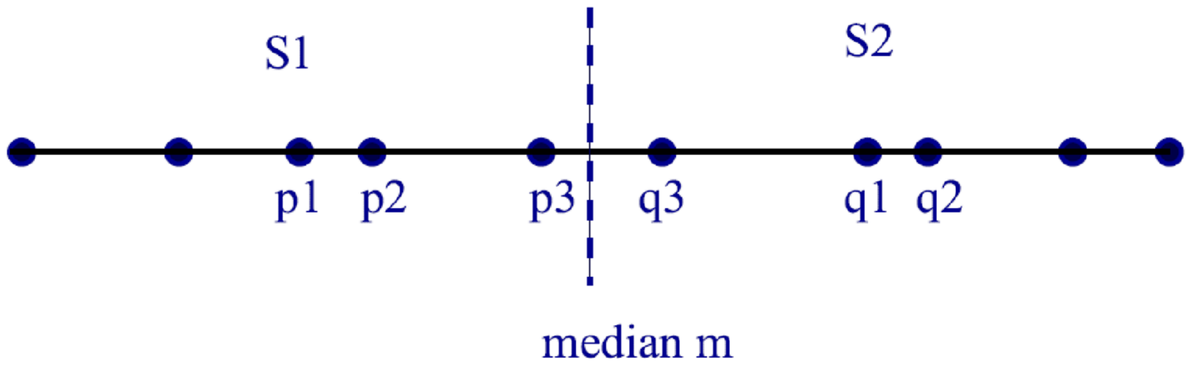
- Let δ be the smallest pairwise distance found in 2 partitions

δ = min($|p_2 - p_1|$, $|q_2 - q_1|$

- The closest pair is either (p_1,p_2), or (q_1,q_2), or some (p_3,q_3) where $p_3 \in S_1$ and $q_3 \in S_2$

- Can we find (p_3,q_3) in a constant time?

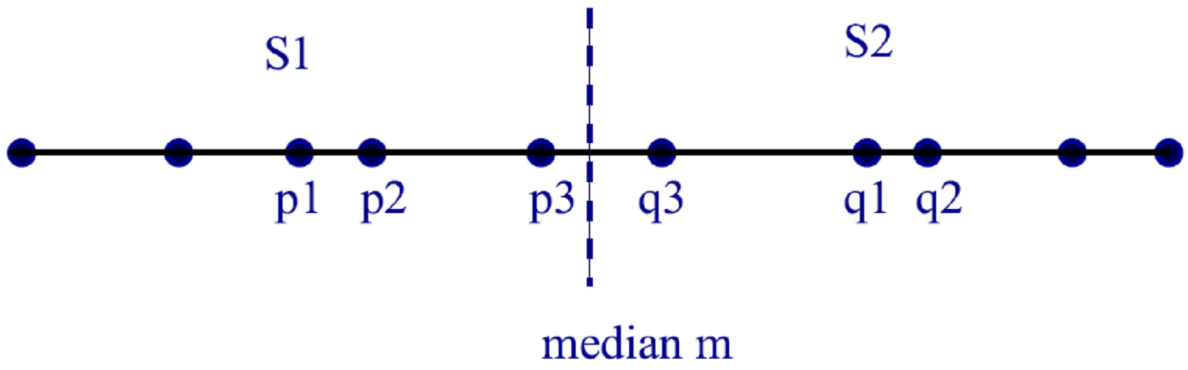
Closest pair in 1 dimension



- The closest pair is either (p_1,p_2), or (q_1,q_2), or some (p_3,q_3) where $p_3 \in S_1$ and $q_3 \in S_2$

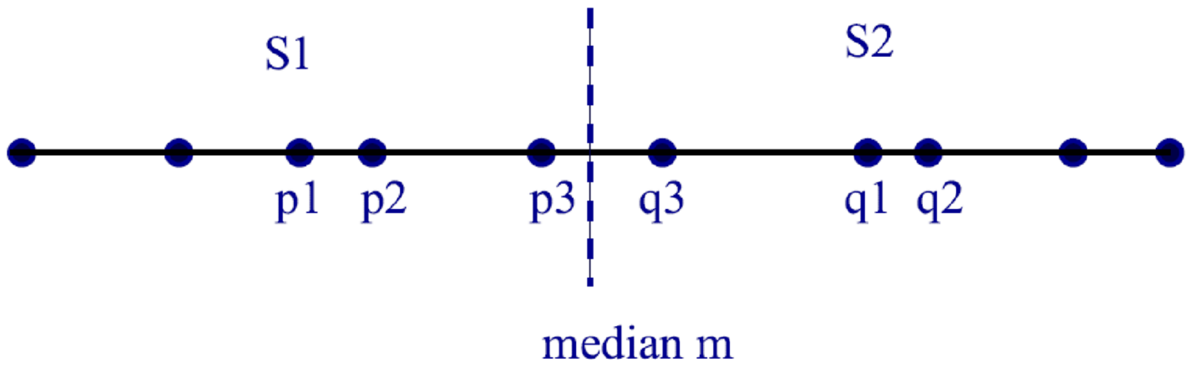
- Key observation: If m is the dividing coordinate, then both p_3 and q_3 have to be within δ of m

Closest pair in 1 dimension



- Key observation: If *m* is the dividing coordinate, then both p_3 and q_3 have to be within δ of *m*
- How many such pairs exist?

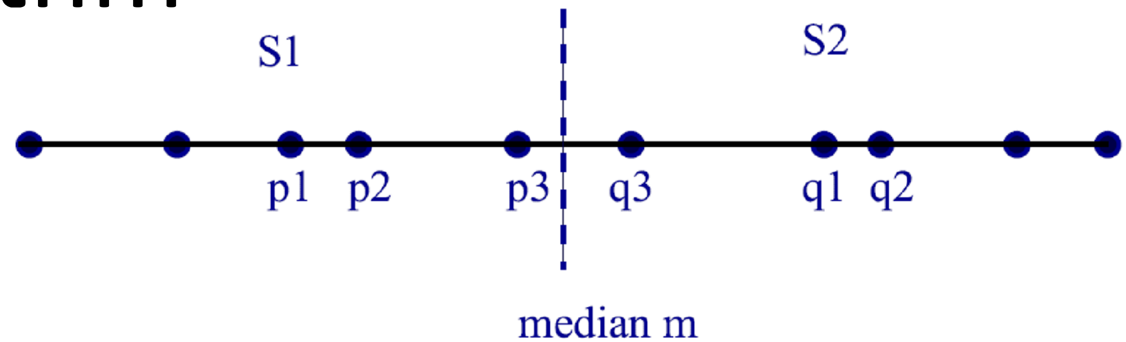
Closest pair in 1 dimension



- Key observation: If *m* is the dividing coordinate, then both p_3 and q_3 have to be within δ of *m*

- How many points of S1 can lie in the interval (m – δ, m]?
- So we need to check one pair only - constant time

Closest pair 1D: recursive algorithm



closest_pair (S – set of sorted points $p_i \ldots p_n$, n>=2)

if $|S| = 2$

return $\delta = |p_2 - p_1|$

Here we only compute the shortest distance, but it is easy to modify to return 2 points which produced this distance

Divide S into S_1 and S_2 at m = value[n/2]

δ_1 = closest_pair (S_1)

δ_2 = closest_pair (S_2)

δ_3 = **closest_pair_across** (S_1, S_2, min(δ_1, δ_2))

Constant time

return $\delta = \min(\delta_1, \delta_2, \delta_3)$

Closest pair in 1 dimension: time complexity

closest_pair (S – set of sorted points $p_i \ldots p_n$, n>=2)

if $|S| = 2$

return $\delta = |p_2 - p_1|$

Divide S into S_1 and S_2 at m = value[n/2]

δ_1 = closest_pair (S_1)

δ_2 = closest_pair (S_2)

δ_3 = **closest_pair_across** (S_1, S_2, min(δ_1, δ_2))

return δ = min(δ_1, δ_2, δ_3)

Constant time

T(n) = 2T(n/2) + O(1)

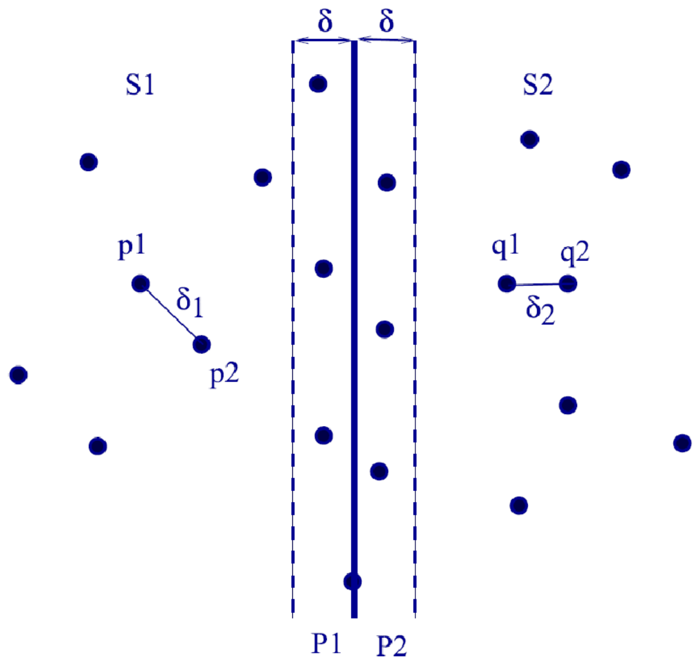
Which solves into O(n)

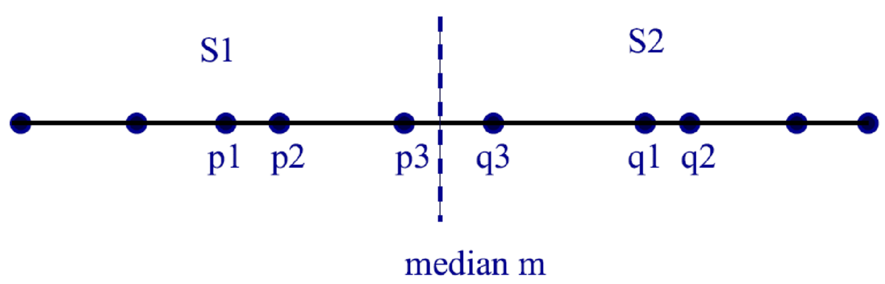
We will learn why later

Together with sorting: O(n log n)

Closest pair in 2 dimensions

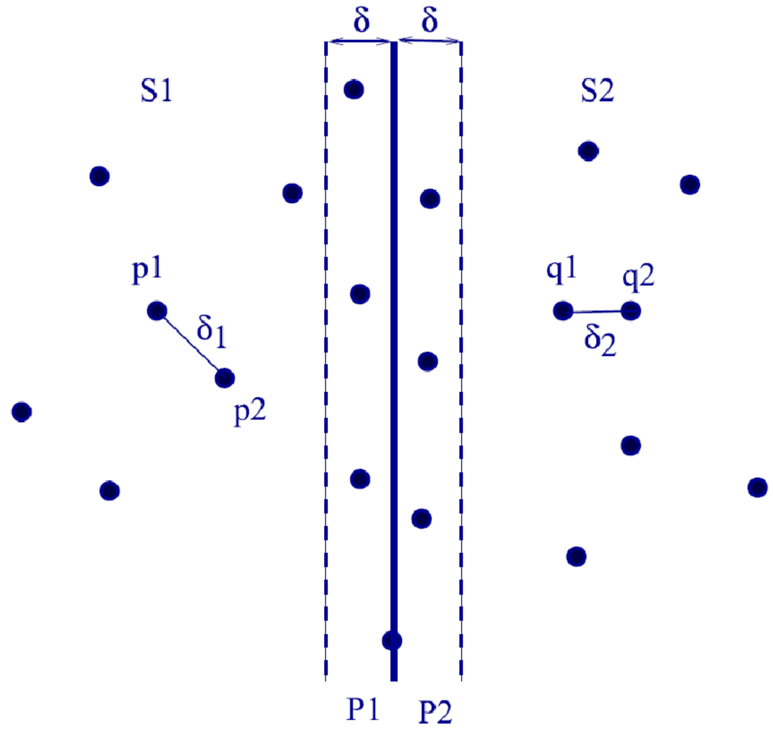
The previous algorithm does not generalize to higher dimensions, **or does it**?





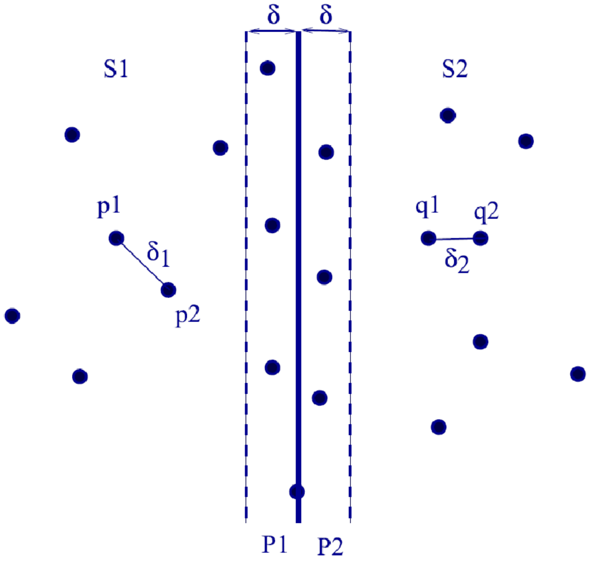
2D closest pair: divide

- Taking sorting as a free O(n log n) invariant, we sort all points in S by x coordinate
- Partition S into S_1, S_2 by vertical line l defined by median x-coordinate in S



2D closest pair: conquer

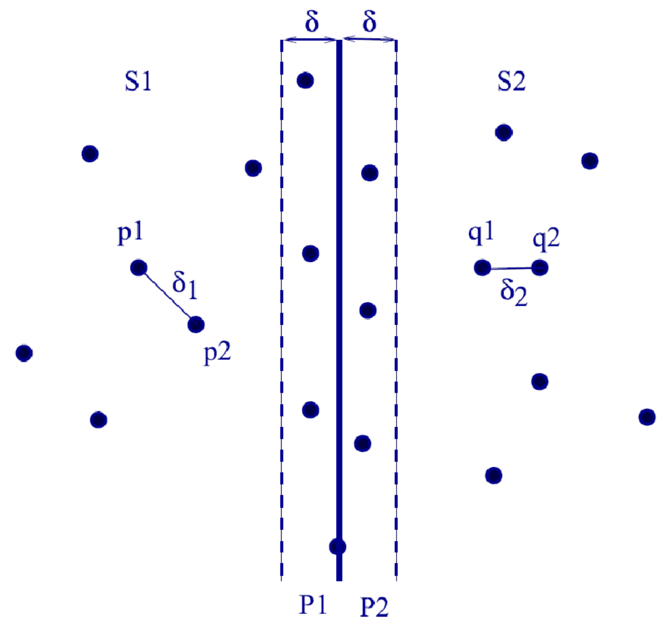
- Recursively compute closest pair distances δ_1 and δ_2 in S_1 and S_2
- Set $\delta=\min(\delta_1, \delta_2)$



2D closest pair: combine

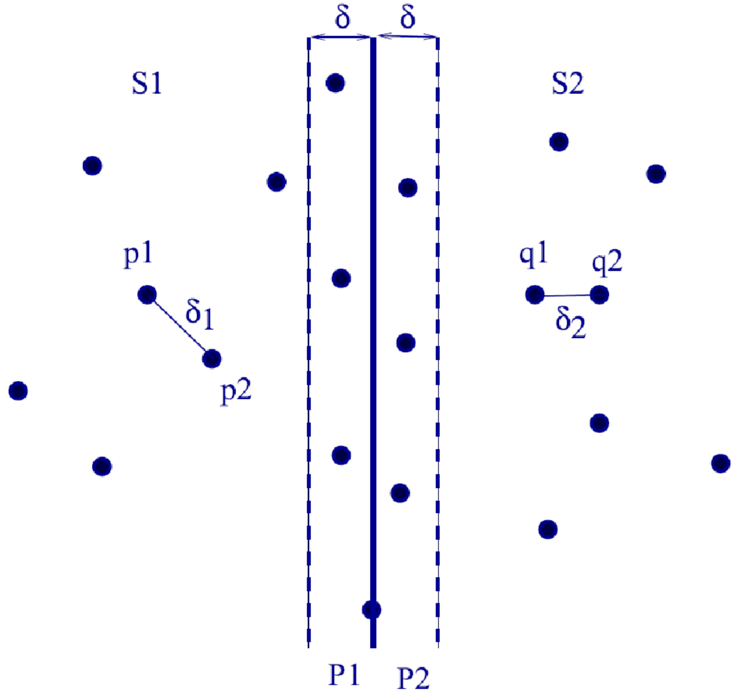
- Closest pair distances in S_1 and S_2 are δ_1 and δ_2. $\delta=\min(\delta_1, \delta_2)$
- Now need to combine: compute the closest pair across dividing line l
- In each candidate pair (p,q), where $p \in S_1$ and $q \in S_2$,

the only candidate points *p, q* must both lie within δ of *l*.



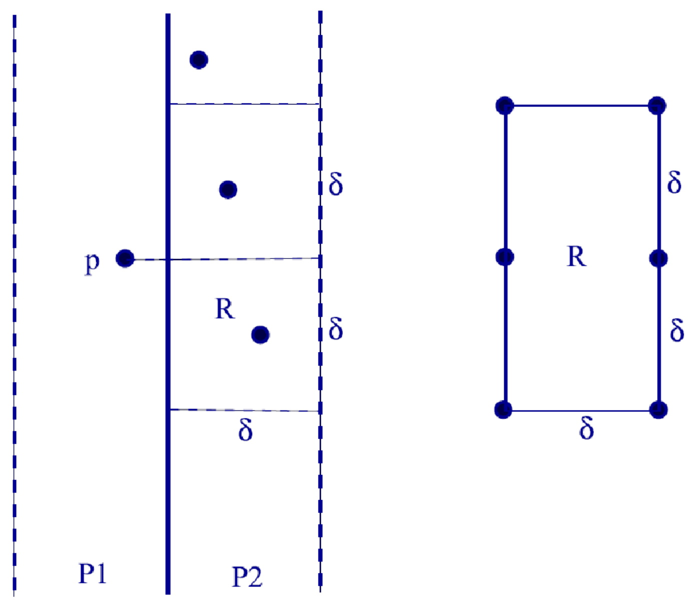
2D closest pair combine: complications

- At this point, complications arise, which were not present in 1D
- It is entirely possible that all $n/2$ points of S_1 (and S_2) lie within δ of l
- Naïvely, this would require $n^2/4$ comparisons



Combining split points

- Consider a point $p \in S_1$.
- All points of S_2 within distance δ of p must lie in a δx2δ rectangle R
- How many points can be inside R if we know that each pair is at least δ apart?
- **In 2D, this number is at most 6!**



So we only need to perform (n/2)*6 distance calculations during the combine step!
We do not have the O(n log n) algorithm yet. Why?

Combine in linear time

- In order to determine at most 6 potential mates of *p*, project *p* and all points of S_2 into *y* axis
- Pick out points whose projection is within δ of *p*: at most 6

- If we pre-sort S_1 and S_2 by the *y* coordinate
- Then we can do our check for all $p \in S_1$, by walking sorted lists S_{1y} and S_{2y}, in total O(*n*) time

The entire solution then runs in **O(*n* log *n*)**

https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/