CHAPTER 5
DIVIDE AND CONQUER

Iris Hui-Ru Jiang
Fall 2017

Department of Electrical Engineering
National Taiwan University

Preliminaries: Mathematical Induction

Weak Induction

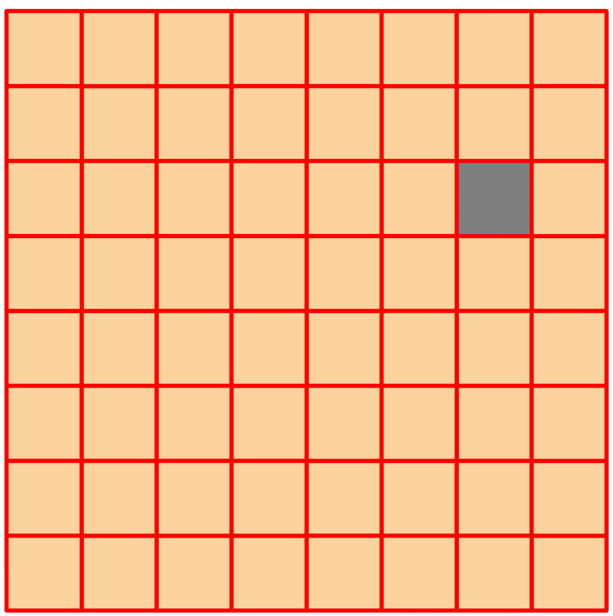
- Given the propositional $P(n)$ where $n \in \aleph$, a proof by mathematical induction is of the form:
  - Basis step: The proposition $P(0)$ is shown to be true
  - Inductive step: The implication $P(k) \rightarrow P(k + 1)$ is shown to be true for every $k \in \aleph$
    - In the inductive step, statement $P(k)$ is called the induction hypothesis

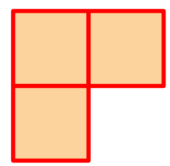
Strong Induction

- Given the propositional $P(n)$ where $n \in \aleph$, a proof by second principle of mathematical induction (or strong induction) is of the form:
  - Basis step: The proposition $P(0)$ is shown to be true
  - Inductive step: The implication $P(0) \wedge P(1) \wedge \ldots \wedge P(k) \rightarrow P(k + 1)$ is shown to be true for every $k \in \aleph$

Example: A Defective Chessboard

- Any 8×8 defective chessboard can be covered with twenty-one triominoes
- Q: How?





Triomino

Example: A Defective Chessboard

- Any 8×8 defective chessboard can be covered with twenty-one triominoes
- Any $2^n \times 2^n$ defective chessboard can be covered with $1/3(2^n \times 2^n - 1)$ triominoes
- Prove by mathematical induction!

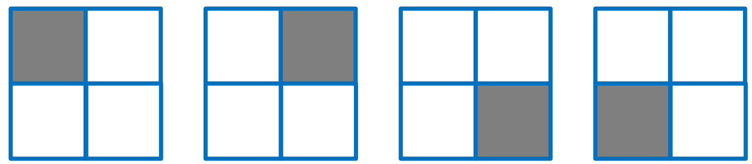
Mathematical Induction

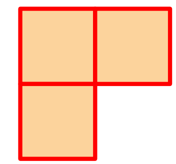
- The first domino falls.
- If a domino falls,
  so will the next domino.
- All dominoes will fall!

 Courtesy of Prof. C.L. Liu

Proof by Mathematical Induction

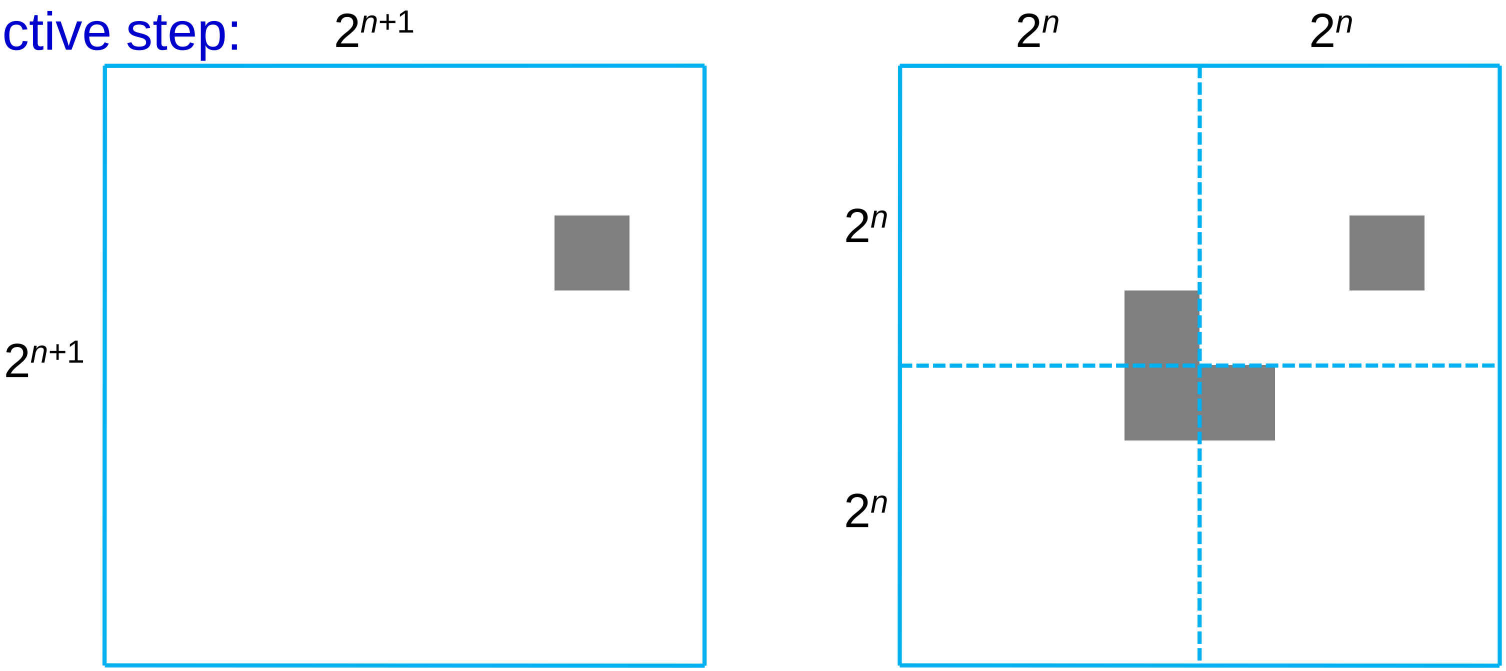
- Any $2^n \times 2^n$ defective chessboard can be covered with $1/3(2^n \times 2^n - 1)$ triominoes
  - Basis step:
    - $n=1$





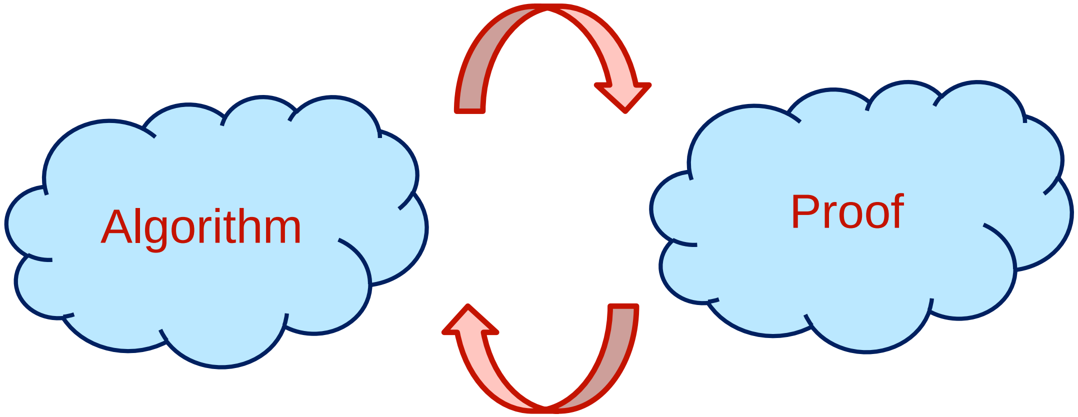
Triomino

  - Inductive step:



Proof vs. Algorithm

- Based on the defective chessboard, we can see



Outline

- Content:
  - A first recurrence: the mergesort algorithm
  - Counting inversions
  - Finding the closest pair of points
- Reading:
  - Chapter 5

Warm Up: Searching

- Problem: Searching
- Given
  - A sorted list of *n* distinct integers
  - integer *x*
- Find
  - *j* if *x* equals some integer of index *j*
- Solution:
  - Naïve idea: compare one by one
    - Correct but slow: O(*n*)
  - Better idea?
    - Hint: input is sorted

Use known information
to improve your solution

Binary Search

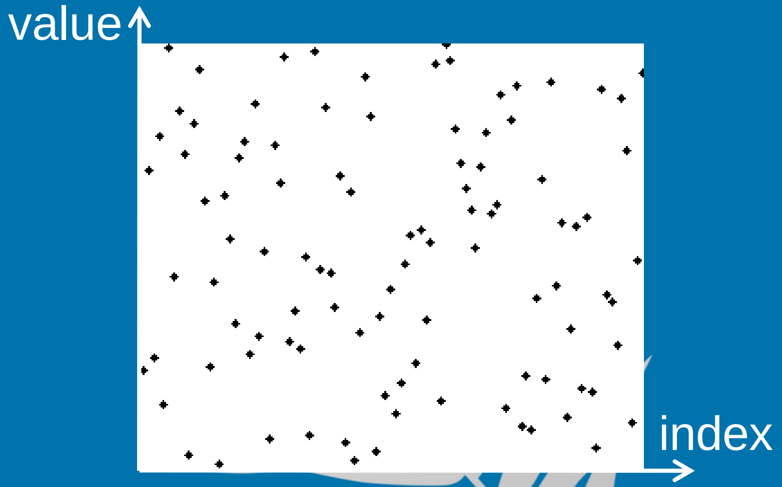
- **D&C paradigm**
- Divide: Break the input into several parts of the same type.
- Conquer: Solve the problem in each part recursively.
- Combine: Combine the solutions to sub-problems into an overall solution

- **Search a sorted array**
- Divide: check the middle element
- Conquer: search the subarray recursively
- Combine: trivial

0	5	13	19	22	41	55	68	72	81	98	<	55
						55	68	72	81	98	>	55
						55	68				=	55

Mergesort

John von Neumann, 1945



http://en.wikipedia.org/wiki/File:Merge_sort_animation2.gif

Divide and Conquer

- Divide-and-conquer
  - Divide: Break the input into several parts of the same type.
  - Conquer: Solve the problem in each part recursively.
  - Combine: Combine the solutions to sub-problems into an overall solution.
- Complexity: recurrence relation
  - A divide and conquer algorithm is naturally implemented by a recursive procedure.
  - The running time of a divide and conquer algorithm is generally represented by a recurrence relation that bounds the running time recursively in terms of the running time on smaller instances.
- Correctness: mathematical induction
  - The basic idea is mathematical induction!

A Divide-and-Conquer Template

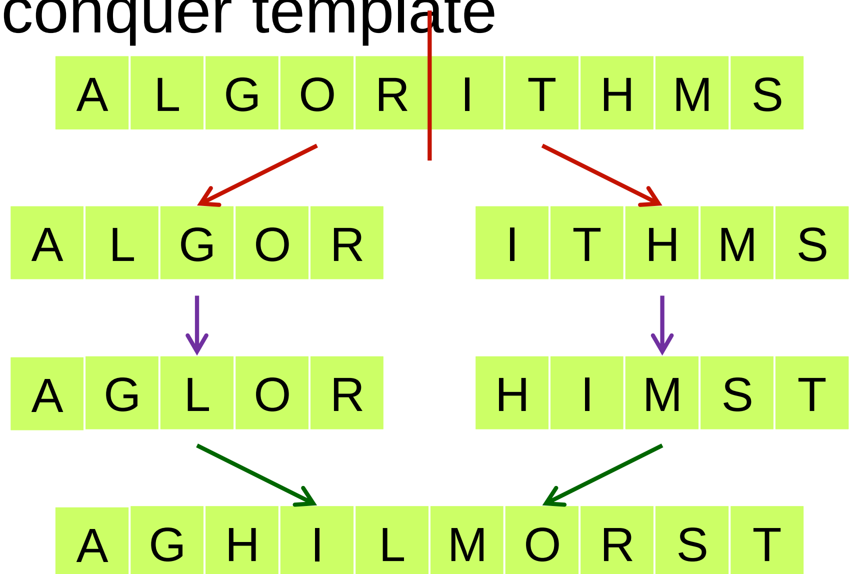
- Divide: divide the input into two pieces of equal size
- Conquer: solve the two subproblems on these pieces separately by recursion
- Combine: combine the two results into an overall solution

- Spend only linear time for the initial division and final recombining

Mergesort (1/2)

- Problem: Sorting
- Given
  - A set of *n* numbers
- Find
  - Sorted list in ascending order
- Solution: many!
- Mergesort fits the divide-and-conquer template
  - Divide the input into two halves.
  - Sort each half recursively.
    - Need base case
  - Merge two halves into one.

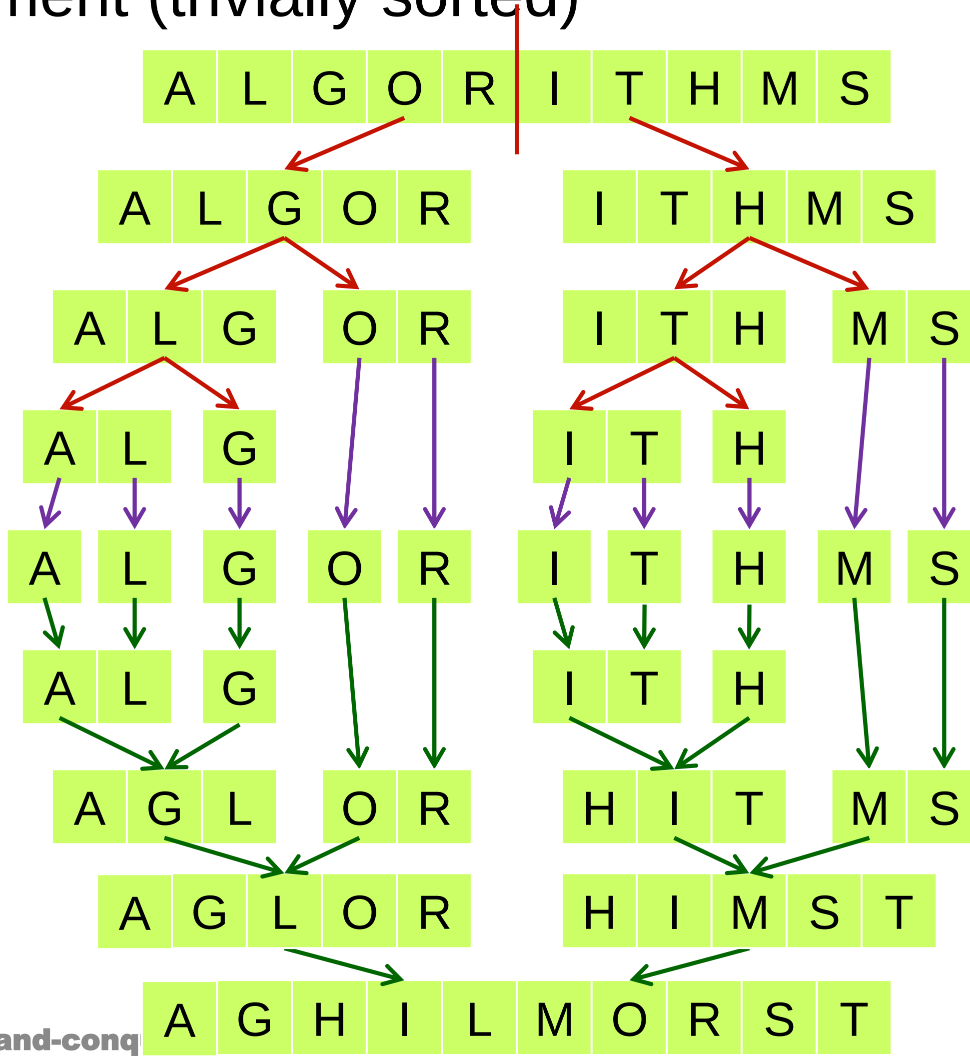
Stop recursion



Mergesort (2/2)

- The base case: single element (trivially sorted)

```
Mergesort(A, p, r)
// A[p..r]: initially unsorted
1. if (p < r) then
2.      q = ⌊(p+r)/2⌋
3.      Mergesort(A, p, q)
4.      Mergesort(A, q+1, r)
5.      Merge(A, p, q, r)
```



- Running time:
  - $T(n)$ for input size n
  - Divide: lines 1-2, $D(n)$
  - Conquer: lines 3-4, $2T(n/2)$
  - Combine: line 5, $C(n)$
  - $T(n) = 2T(n/2) + D(n) + C(n)$

Implementation: Division and Merging

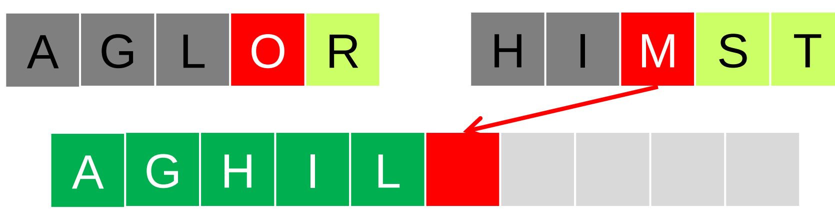
- Running time: $T(n)$
  - $T(n)$ for input size n
  - Divide: lines 1-2, $D(n)$
    - O(1) for array
  - Combine: line 5, $C(n)$
  - Q: Linear time $O(n)$? How?
- Efficient merging
  - See the demonstration of Merge

```
Mergesort(A, p, r)
// A[p..r]: initially unsorted
1.  if (p < r) then
2.      q = ⌊(p+r)/2⌋
3.      Mergesort(A, p, q)
4.      Mergesort(A, q+1, r)
5.      Merge(A, p, q, r)
```

Implementation: Division and Merging

Mergesort(A, p, r)
// $A[p..r]$: initially unsorted
1. **if** ($p < r$) **then**
2. $q = \lfloor (p+r)/2 \rfloor$
3. Mergesort(A, p, q)
4. Mergesort(A, $q+1$, r)
5. Merge(A, p, q, r)

- Running time: $T(n)$
  - $T(n)$ for input size n
  - Divide: lines 1-2, $D(n)$
    - O(1) for array
  - Combine: line 5, $C(n)$
  - Q: Linear time $O(n)$? How?
- Efficient merging
  - Linear number of comparisons
  - Use an auxiliary array



  - $O(n)$
- Merge sort is often the best choice for sorting a linked list
  - Q: Why? How efficient on running time and storage?

Recurrence Relation

- Running time: $T(n)$
  1. Base case: for $n \leq 2$, $T(n) \leq c$
  2. $T(n) = 2T(n/2) + D(n) + C(n)$
  - $T(n) = 2T(n/2) + O(1) + O(n)$
  - $T(n) \leq 2T(n/2) + cn$

Mergesort(A, p, r)
// $A[p..r]$: initially unsorted
1. **if** ($p < r$) **then**
2. $q = \lfloor (p+r)/2 \rfloor$
3. Mergesort(A, p, q)
4. Mergesort(A, q+1, r)
5. Merge(A, p, q, r)

- A recursion corresponds to a recurrence relation
  - Recursion is a function defined by itself

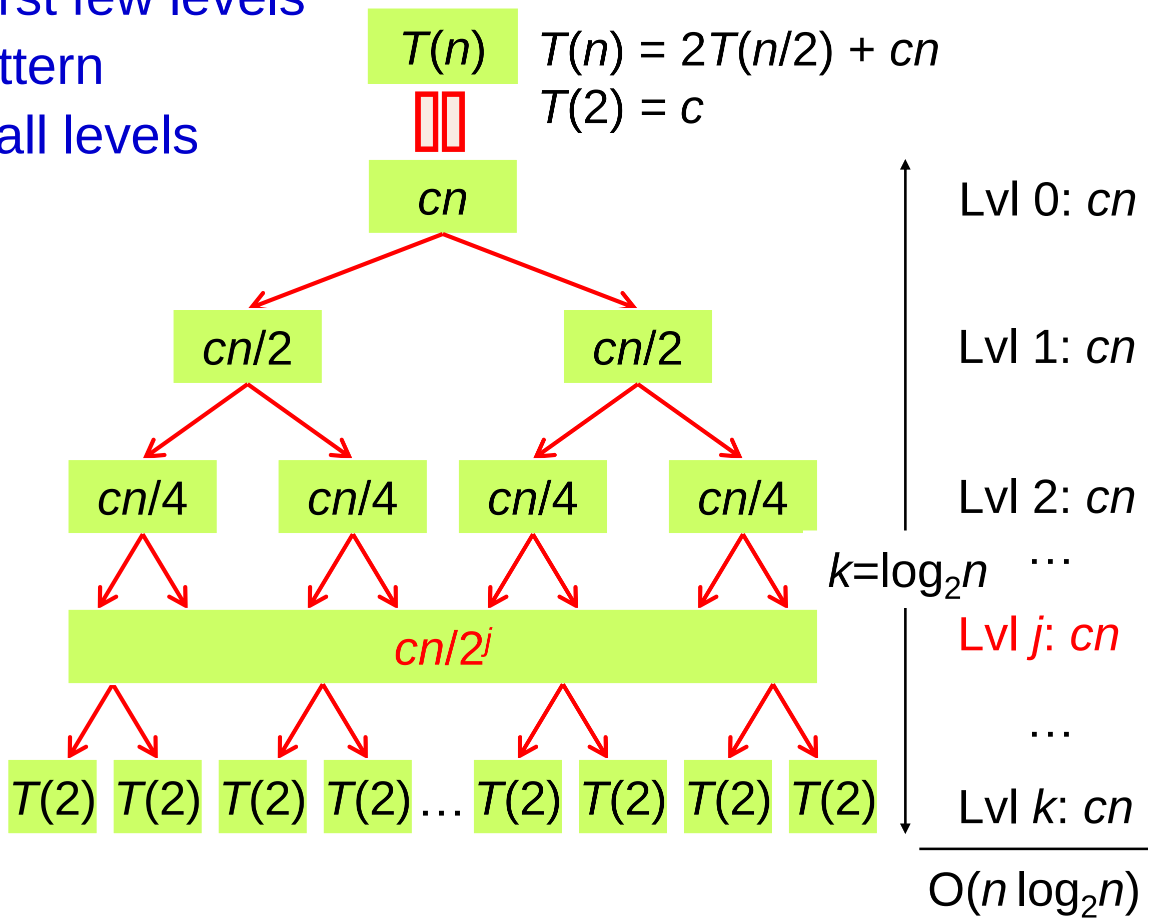
- Q: Why not $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$?
- A: Asymptotic bounds are not affected by ignoring $\lfloor\rfloor$ & $\lceil\rceil$.

Solving Recurrences

- Two basic ways to solve a recurrence
  - Unrolling the recurrence (recursion tree)
  - Substituting a guess
- Initially, we assume n is a power of 2 and replace $\leq$ with $=$.
  - $T(n) = 2T(n/2) + cn$
  - Simplify the problem by omitting floors and ceilings
  - Solve the worst case

Unrolling – Recursion Tree

- Procedure
  1. Analyzing the first few levels
  2. Identifying a pattern
  3. Summing over all levels



Substituting

- Any function $T(.)$ satisfying this recurrence

  $$T(n) \le 2T(n/2) + cn \text{ when } n > 2, \text{ and } T(2) \le c$$

  is bounded by $O(n \log_2 n)$, when $n > 1$.
- Pf: Guess and proof by induction

  assume n is a power of 2
- Suppose we believe that $T(n) \le cn \log_2 n$ for all $n \ge 2$
  1. Base case: $n = 2$, $T(2) \le c \le 2c$. Indeed true.
  2. Inductive step:
  - Inductive hypothesis: $T(m) \le cm \log_2 m$ for all $m < n$.
  - $T(n/2) \le c(n/2) \log_2(n/2)$; $\log_2(n/2) = (\log_2 n) - 1$
  - $T(n) \le 2T(n/2) + cn$

    $\le 2c(n/2) \log_2(n/2) + cn$

    $= cn [(\log_2 n) - 1] + cn$

    $= (cn\log_2 n) - cn + cn$

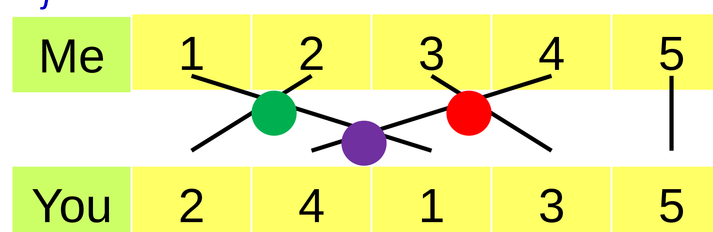
    $= cn\log_2 n.$

Counting Inversions

Counting Inversions

- Music site tries to match your song preferences with others.
  - You rank *n* songs.
  - Music site consults database to find people with similar tastes.
- Similarity metric: number of inversions between two rankings.
  - My rank: 1, 2, …, n.
  - Your rank: $a_1, a_2, \ldots, a_n$.
  - Songs i and j inverted if $i < j$, but $a_i > a_j$.

	A	B	C	D	E
Me	1	2	3	4	5
You	2	4	1	3	5



inversion = crossing

(2, 1), (4, 1), (4, 3)

- Brute force: check all $\Theta(n^2)$ pairs i and j.

Divide and Conquer

- Counting inversions
  - Divide: separate the list into two pieces.
  - Conquer: recursively count inversions in each half.
  - Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

1	5	4	8	10	2

6	9	12	11	3	7

Divide: O(1)

5 red-red inversions
(5,4), (5,2), (4,2),
(8,2), (10,2)

8 green-green inversions
(6,3), (9,3), (9,7), (12,11),
(12,3), (12,7), (11,3), (11,7)

Conquer: $2T(n/2)$

9 red-green inversions
(5,3), (4,3), (8,6), (8,3), (8,7), (10,6), (10,9), (10,3), (10,7)

Combine: ??

Total: 5 + 8 + 9 = 22 inversions

Combine?

- Inversions: inter and intra
  - Intra: inversions within each half – done by conquer
  - Inter: inversions between two halves – done by combine
    - The “combine” in Mergesort is done in $O(n)$; goal: $O(n \log_2 n)$
    - Assume each half is sorted. ← to maintain sorted invariant
    - Count inversions where a_i and a_j are in different halves.
    - Merge two sorted halves into sorted whole.

1	2	4	5	8	10	3	6	7	9	11	12	
						4	2	2	1	0	0	Count: $O(n)$

(4,3), (5,3), (8,6), (10,3), (8,6), (10,6), (8,7), (10,7), (10,9)

9 red-green inversions

1	2	3	4	5	6	7	8	9	10	11	12	Merge: $O(n)$

Combine: $O(n)$

Total: $O(n \log_2 n)$

Implementation: Counting Inversions

- Similar to Mergesort, extra effort on counting inter-inversions

Sort-and-Count(L, p, q)
// $L[p..q]$: initially unsorted
1. **if** ($p = q$) **then return** 0
2. **else**
3. $m = \lfloor (p+q)/2 \rfloor$
4. r_p = Sort-and-Count(L, p, m)
5. r_q = Sort-and-Count(L, m+1, q)
6. r_m = Merge-and-Count(L, p, m, q)
7. **return** $r = r_p + r_q + r_m$

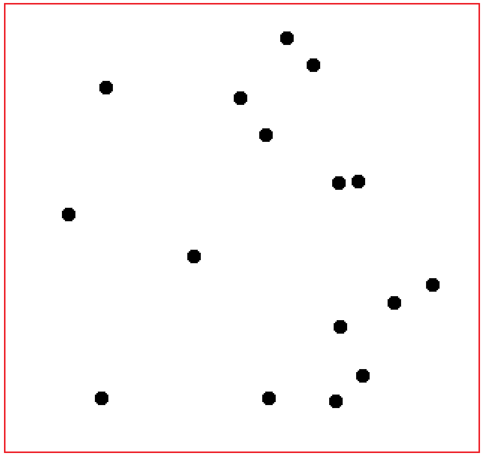
Closest Pair of Points

M. I. Shamos and D. Hoey, 1975

M. I. Shamos and D. Hoey. "Closest-point problems." In *Proc. 16th Annual IEEE Symposium on Foundations of Computer Science* (FOCS), pp. 151—162, 1975

Closest Pair of Points

- The closest pair of points problem
- Given
  - A set of n points on a plane, p_i is located at $(x_i, \ y_i)$.
- Find
  - A pair with the smallest Euclidean distance between them
    - Euclidean distance between p_i and p_j = $[(x_i - x_j)^2 + (y_i - y_j)^2]^{1/2}$



Closest Pair of Points: First Attempt

- Q: How?
- A: Brute-force? $\Theta(n^2)$ comparisons.

- Q: What if 1-D version?

- A: If points are all on a line, easy! O(n log n) for sorting.

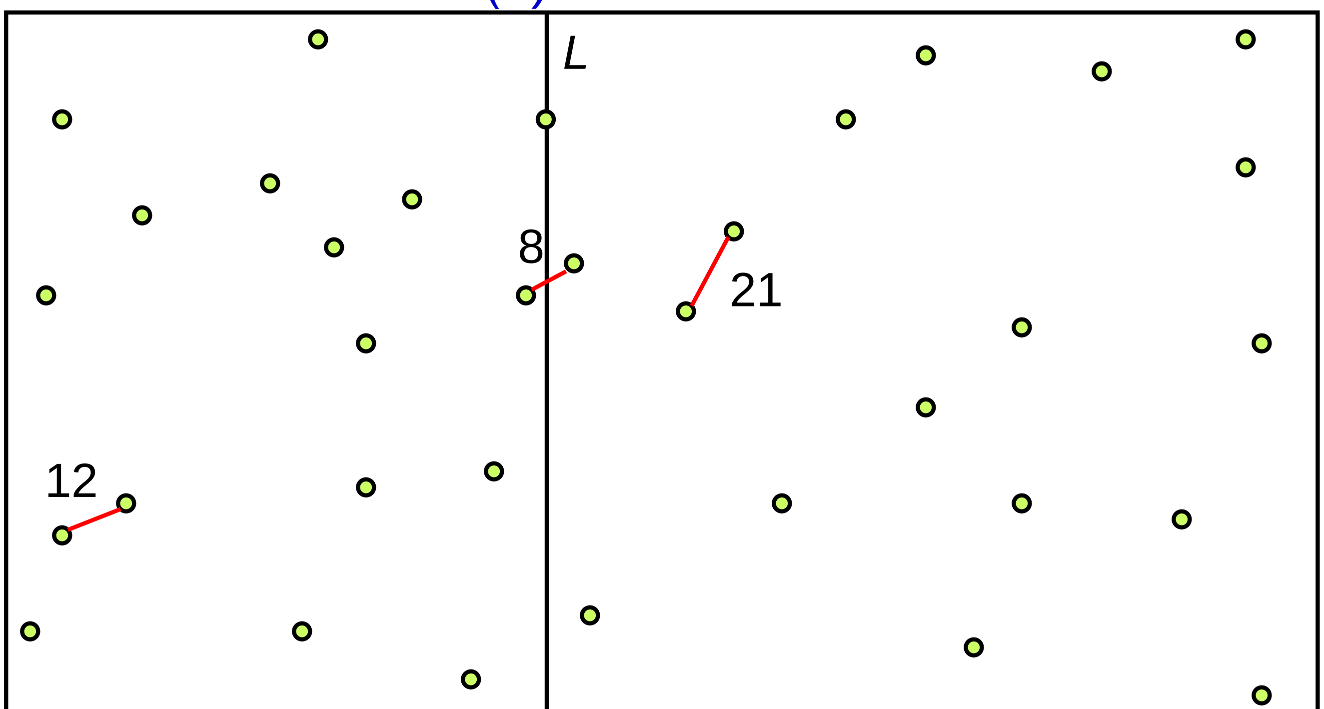
- Q: What if 2-D version?
- A: Non-trivial.

to make presentation cleaner

- Assumption: No two points have same x-coordinate.

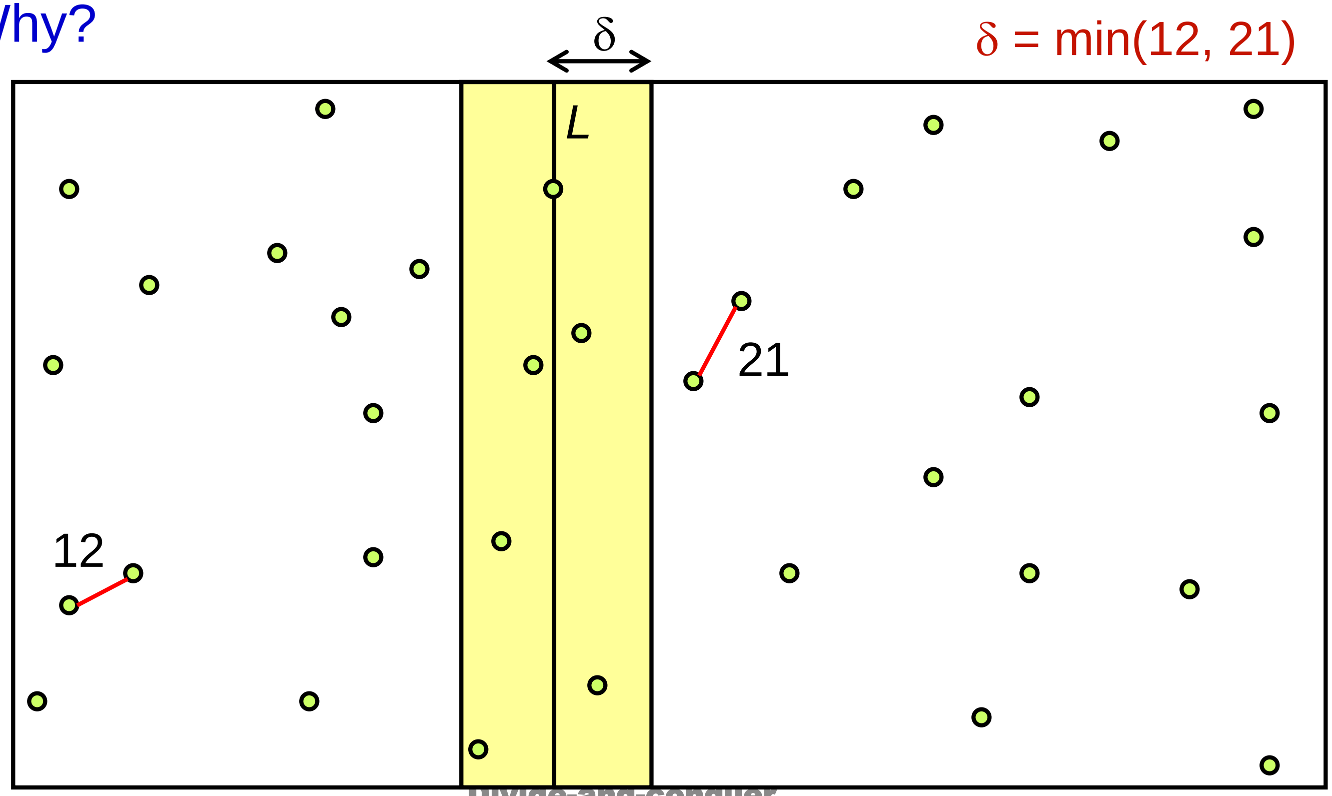
Divide-and-Conquer

- Divide: draw vertical line L so that half points on each side.
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side; return best of 3 solutions.
  - Q: How to "combine" in O(n)?



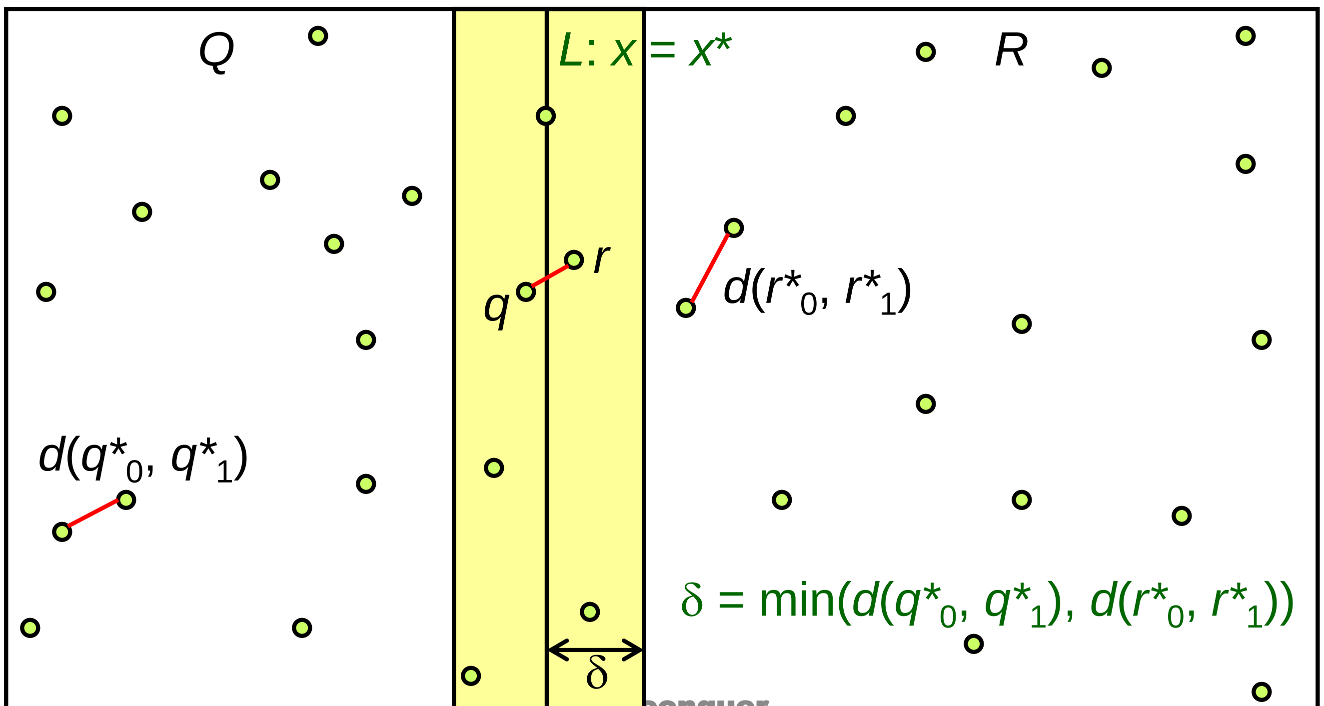
Combining the Solutions (1/4)

- Find closest pair with one point in each side.
  - $L = \{(x, y): x = x^* = x\text{-coordinate of the rightmost point in } Q\}$.
  - δ = the smaller one of these two pairs.
- Observation: only need to consider points within δ of line L.
  - Q: Why?



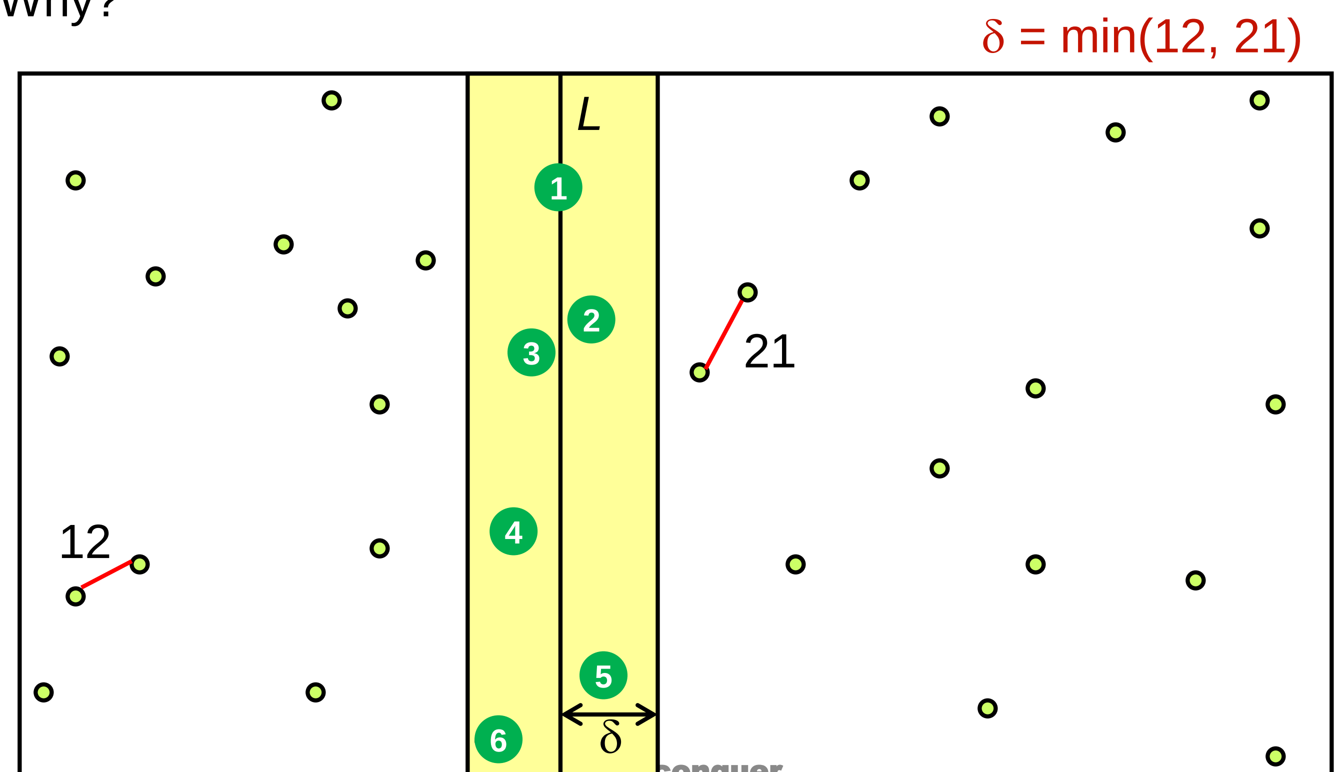
Combining the Solutions (2/4)

- If $\exists\ q \in Q$ and $r \in R$ for which $d(q, r) < \delta$, then each of q and r lies within a distance δ of L.
- Pf: Suppose such q and r exist; let $q = (q_x, q_y)$ and $r = (r_x, r_y)$.
  - By definition, $q_x \le x^* \le r_x$.
  - $x^* - q_x \le r_x - q_x \le d(q, r) < \delta$; $r_x - x^* \le r_x - q_x \le d(q, r) < \delta$.



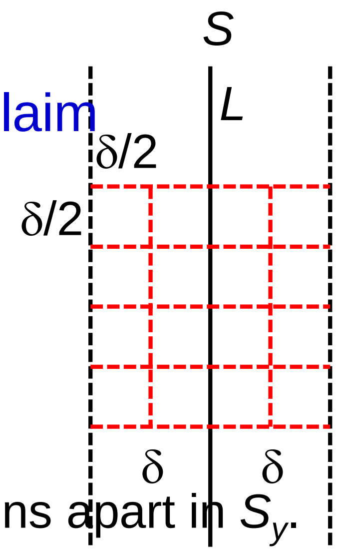
Combining the Solutions (3/4)

- Observation: only need to consider points within δ of line *L*.
  - Sort points in 2δ-strip by their *y*-coordinate.
  - Only check distances of those within 15 positions in sorted list of *y*-coordinates!
    - Q: Why?



Combining the Solutions (4/4)

- If s, $s' \in S$ are of $d(s, s') < \delta$, then s, s' are within 15 positions of each other in the sorted list S_y of y-coordinates of S.
- Pf: S contains all points within δ of line L, we partition S into boxes, each box contains at most one point.
  - Partition the region into boxes, each of area $\delta/2*\delta/2$; we claim
    1. *s* and *s'* lies in different boxes
       - Suppose *s* and *s'* lies in the same box
       - These two points both belong either to *Q* or to *R*
       - $d(s, s') \le 0.5*\delta*(2)^{1/2} < \delta \rightarrow\leftarrow$
    2. *s* and *s'* are within 15 positions
       - Suppose s, $s' \in S$ of $d(s, s') < \delta$ and they are at least 16 positions apart in S_y.
       - Assume WLOG *s* has the smaller *y*-coordinate; since at most one point per box, at least 3 rows of *S* lying between *s* and *s'*.
       - $d(s, s') \ge 3\delta/2 > \delta \rightarrow\leftarrow$



Closest Pair Algorithm

Closest-Pair(P)
1. construct P_x and P_y
2. (p^*_0, p^*_1) = Closest-Pair-Rec(P_x, P_y)

Closest-Pair-Rec(P_x, P_y)
1. **if** $|P| \leq 3$ **then return** closest pair measured by all pair-wise distances
2. $x^* = (\lceil n/2 \rceil)$-th smallest x-coordinate in P_x
3. construct Q_x, Q_y, R_x, R_y
4. (q^*_0, q^*_1) = Closest-Pair-Rec(Q_x, Q_y)
5. (r^*_0, r^*_1) = Closest-Pair-Rec(R_x, R_y)
6. $\delta = \min(d(q^*_0, q^*_1), d(r^*_0, r^*_1))$
7. $L = \{(x, y): x = x^*\}$; S = {points in P within distance δ of L}
8. construct S_y
9. **for each** $s \in S$ **do**
10. compute distance from s to each of next 15 points in S_y
11. $d(s, s')$ = min distance of all computed distances
12. **if** $d(s, s') < \delta$ **then return** (s, s')
13. **else if** $d(q^*_0, q^*_1) < d(r^*_0, r^*_1)$ **then return** (q^*_0, q^*_1)
14. **else return** (r^*_0, r^*_1)