The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400CE), translated by Lionel Giles (1910)

Our life is frittered away by detail.... Simplify, simplify.

— Henry David Thoreau, *Walden* (1854)

Now, don't ask me what Voom is. I never will know.
But, boy! Let me tell you, it DOES clean up snow!

— Dr. Seuss [Theodor Seuss Geisel], *The Cat in the Hat Comes Back* (1958)

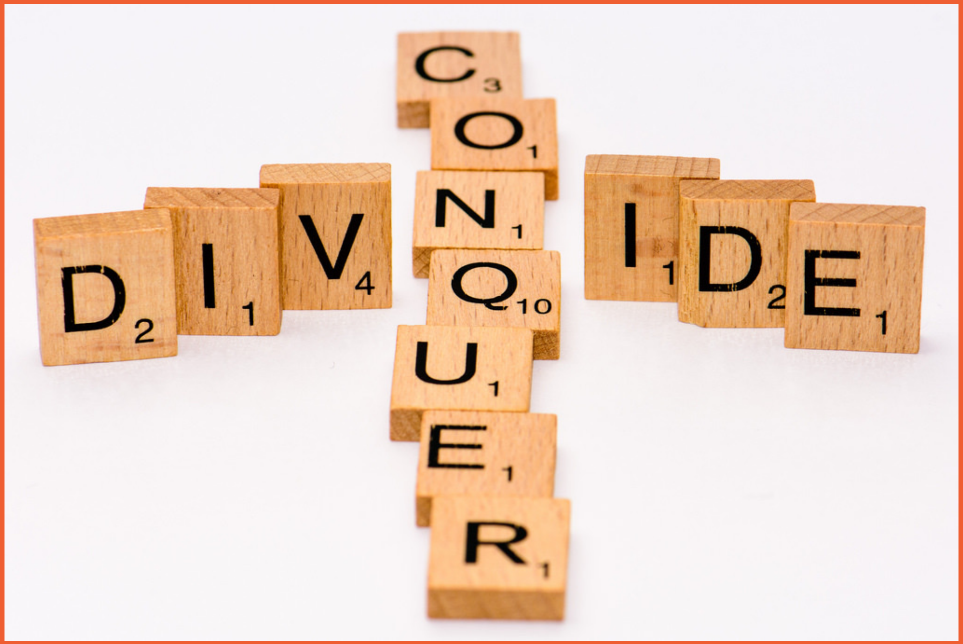
Do the hard jobs first. The easy jobs will take care of themselves.

— attributed to Dale Carnegie

From Jeff Erickson, alogrithms.wtf

Lecture 3:
Divide & Conquer
[Recursion, Erickson;
Divide & Conquer, KT]

William Umboh
School of Computer Science

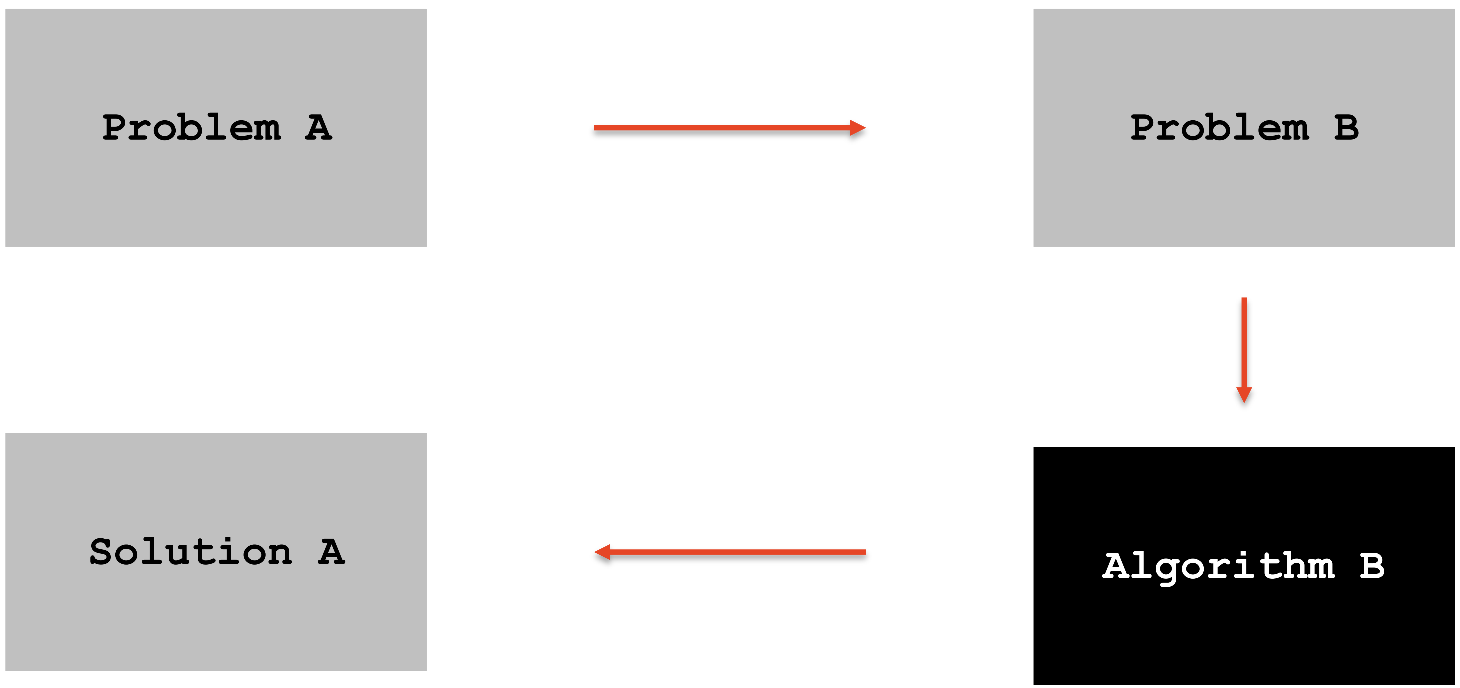


THE UNIVERSITY OF SYDNEY

General techniques in this course

- Greedy algorithms [W2]
- Divide & Conquer algorithms [today]
- Dynamic programming [W4-5]
- Network flow algorithms [W6-7]

Reduction: Powerful Idea in Computer Science

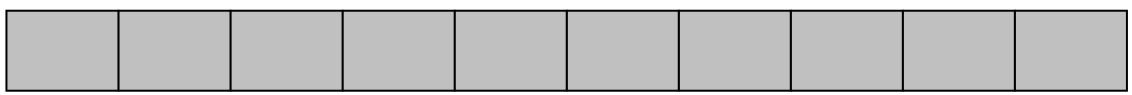


Problem B is a smaller instance of Problem A: Divide-and-Conquer, Dynamic programming OR

Problem B is easier than Problem A: Network Flows, NP-hardness

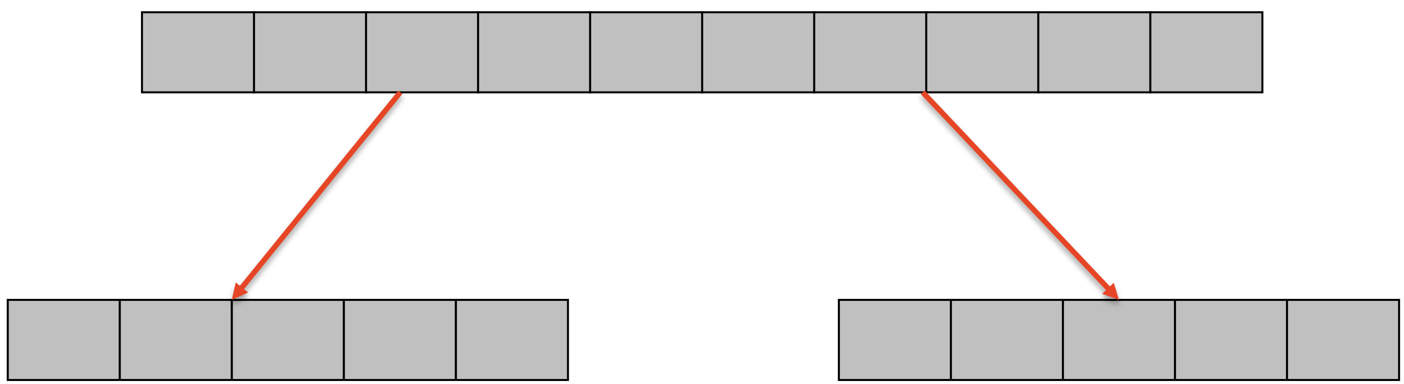
Divide-and-Conquer

- **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.



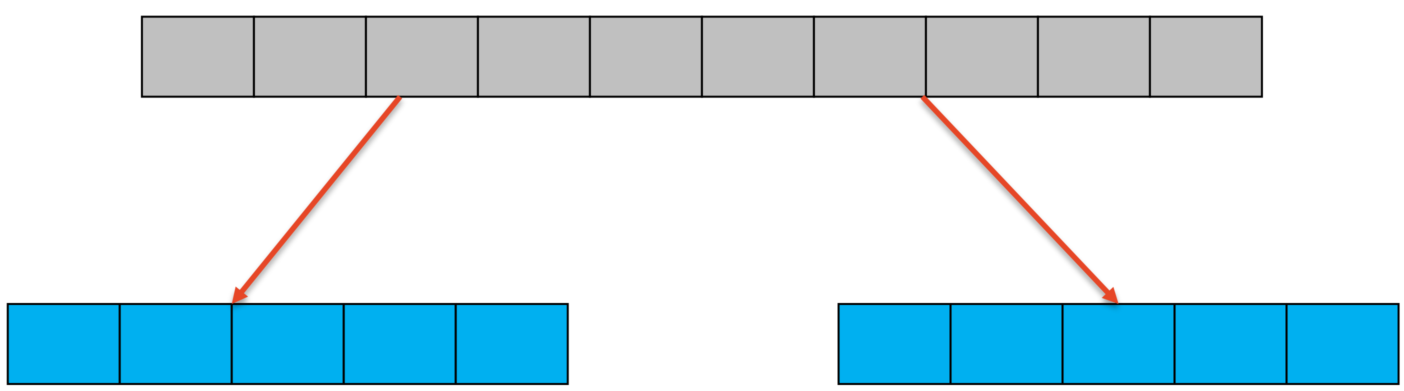
Divide-and-Conquer

– **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.



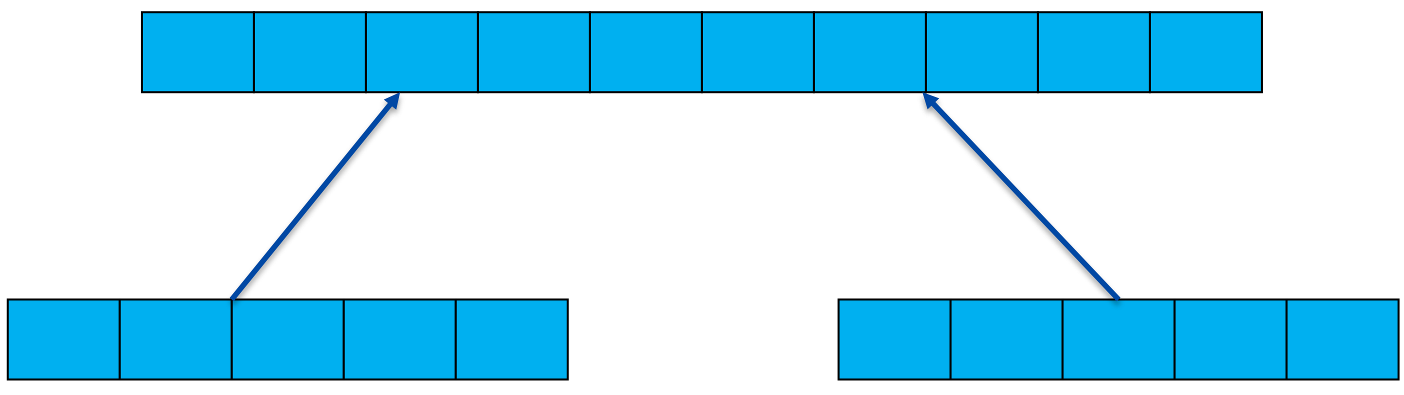
Divide-and-Conquer

- **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.



Divide-and-Conquer

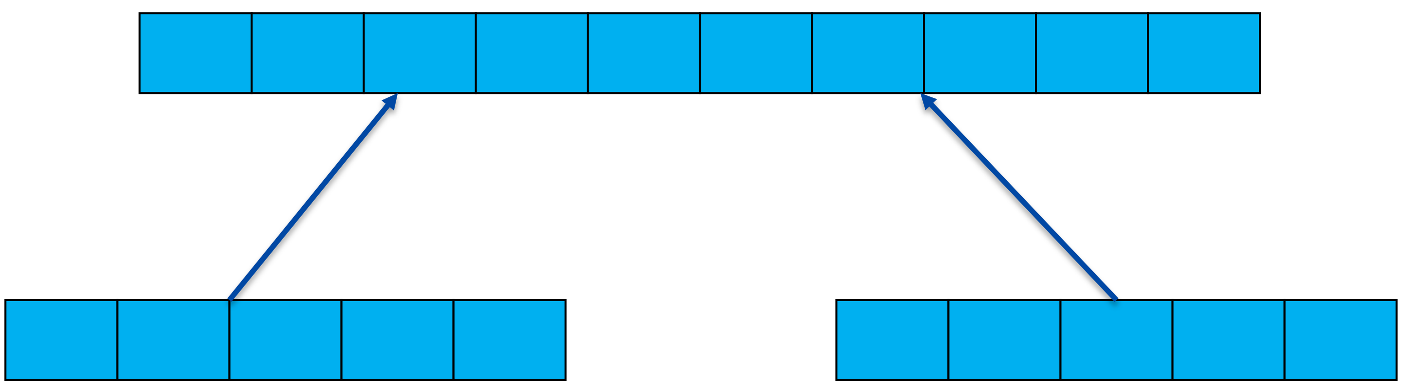
- **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.



Divide-and-Conquer

- **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several *smaller* parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.

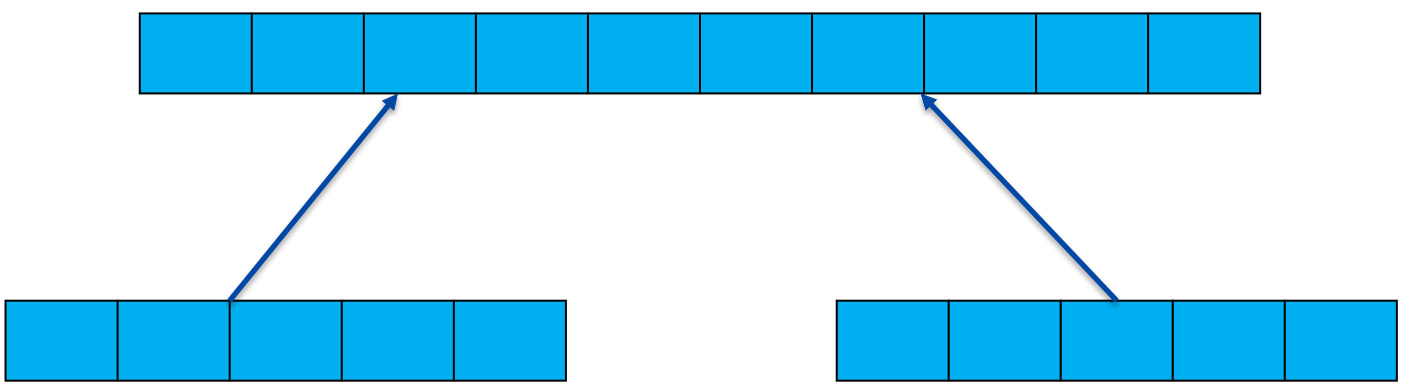
- **Most common usage.**
  - Break up problem of size n into two equal parts of size ½n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.



Divide-and-Conquer

- **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several *smaller* parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.

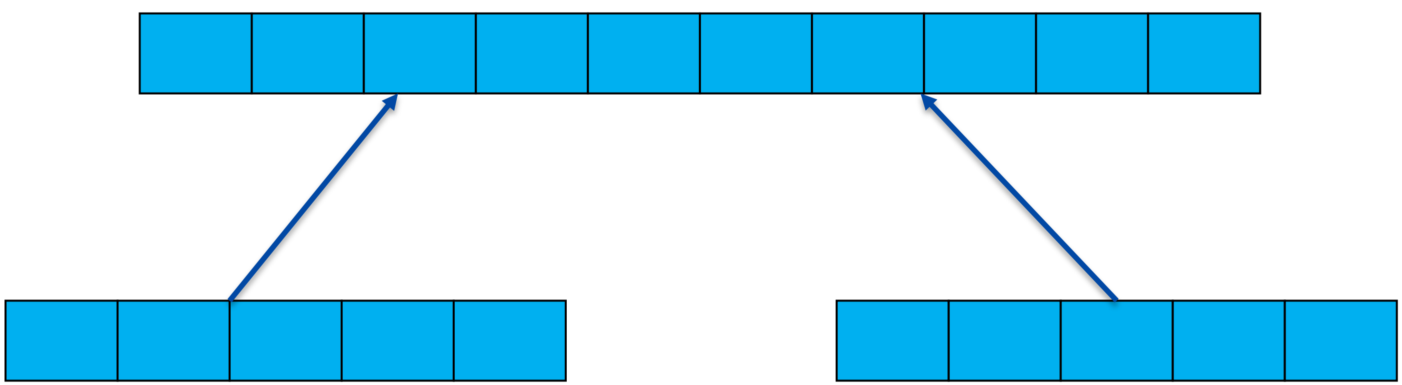
- **Proof of Correctness**
  - By induction on n.
  - Base case. Typically, but not always, n = 1.
  - Inductive case. Prove correctness of combine step assuming correctness of solutions to sub-problems (inductive hypothesis)



Divide-and-Conquer

- **Divide-and-conquer** [usually 3 steps]
  1. Divide: Break up problem into several *smaller* parts.
  2. Conquer: Solve each part recursively.
  3. Combine solutions to sub-problems into overall solution.

- **Time complexity**
  - Solve recurrence relation
  - T(n) = divide step + combine step + $\overbrace{\text{subproblems}}^{2T(n/2)}$



Warmup: Searching

Input: A sorted sequence S of n numbers $a_1, a_2, \ldots, a_n$, stored in an array A[1..n].

Question: Given a number x, is x in S?

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- $m = \text{ceil}(n/2)$
- Compare x to the middle element of the array (A[m]).
- If A[m] = x then "Yes"
- Otherwise, if A[m] > x then recursively Search A[1…m-1].
- Otherwise, if A[m] < x then recursively Search A[m+1...n]

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- $m = \text{ceil}(n/2)$
- Compare x to the middle element of the array (A[m]).
- If A[m] = x then "Yes"
- Otherwise, if A[m] > x then recursively Search A[1…m-1].
- Otherwise, if A[m] < x then recursively Search A[m+1...n]

Proof of correctness by induction on n:

- Base case (n = 1). Trivial.
- Inductive case. Assume correct on input sizes < n.
  - x is in A if and only if it is in the subarray in the recursive call
  - Apply inductive hypothesis

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then “Yes”
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example of inductive case: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example of inductive case: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1…n]

Example of inductive case: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

recurse

Do not unroll recursion! Our job is to reduce to smaller instances and apply correctness on smaller instances.

See also "Recursion Fairy" in Erickson's textbook.

Binary Search

- Compare x to the middle element of the array (A[n/2]).
- If A[n/2] = x then "Yes"
- Otherwise, if A[n/2] > x then recursively Search A[1…n/2-1].
- Otherwise, if A[n/2] < x then recursively Search A[n/2+1...n]

Example of inductive case: x=1 (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

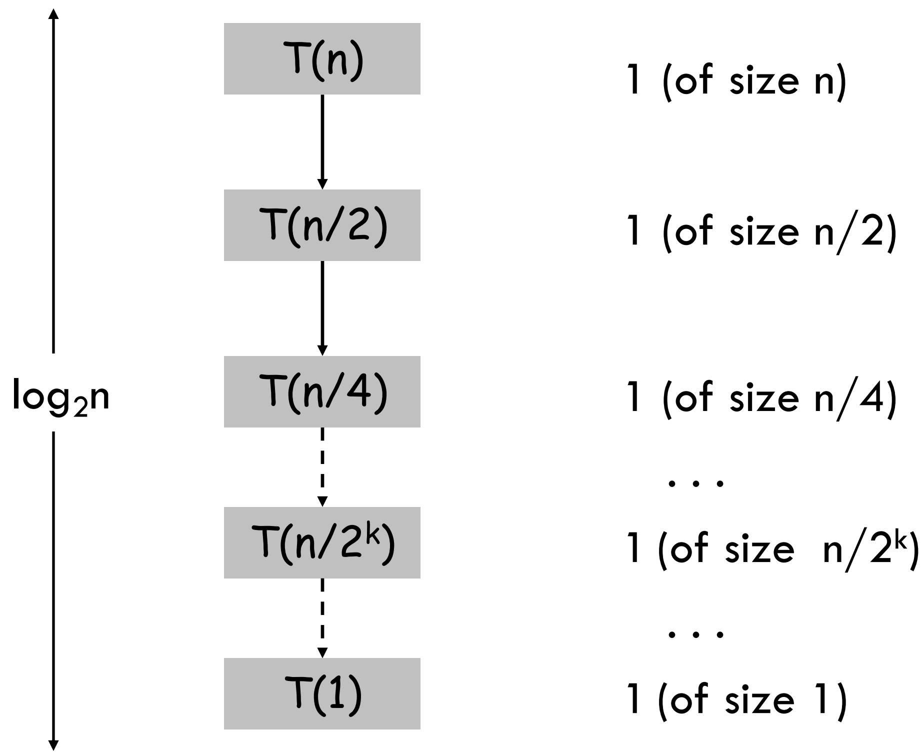
Analysis:

- $T(n) = \overbrace{\text{divide step} + \text{combine step}}^{O(1)} + \overbrace{\text{subproblems}}^{T(n/2)}$

  $= 1 + T(n/2)$

Analyze recurrence via recursion tree

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



Binary Search

- Compare x to the middle element of the array ($A[n/2]$).
- If $A[n/2] = x$ then "Yes"
- Otherwise, if $A[n/2] > x$ then recursively Search $A[1\ldots n/2-1]$.
- Otherwise, if $A[n/2] < x$ then recursively Search $A[n/2+1\ldots n]$

Example of inductive case: $x=1$ (non-integers are rounded up)

0	1	3	4	5	7	10	13	15	18	19	23

Analysis:

- $T(n) = 1 + T(n/2) = O(\log n)$

Maximum-sum contiguous subarray

Given an array A[ ] of n numbers, find the maximum sum found in any contiguous subarray

A zero-length subarray has maximum 0

Example:

1	-2	7	5	6	-5	5	8	1	-6

Maximum-sum contiguous subarray

Given an array A[ ] of n numbers, find the maximum sum found in any contiguous subarray

A zero-length subarray has maximum 0

Example:

1	-2	7	5	6	-5	5	8	1	-6

Divide-and-conquer algorithm (first try)

Maximum contiguous subarray (MCS) in A[1..n]

If $n > 1$, return $\max\{MCS(A[1..n/2]), MCS(A[n/2+1..n])\}$

If $n = 1$,

  If $A[1] < 0$, return 0

  Else return A[1]

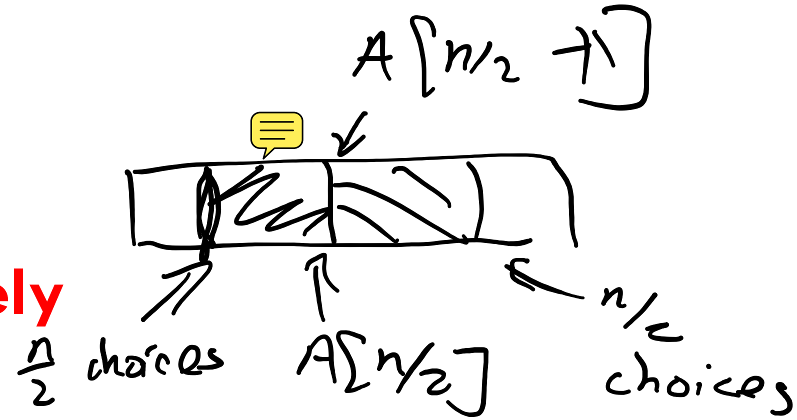
1	-2	7	5	6	-5	5	8	1	-6

Problem: what if optimal subarray contains A[n/2, n/2+1]?

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

- Three cases:
  a) MCS in A[1..n/2]
  b) MCS in A[n/2+1..n]
  c) MCS that spans A[n/2, n/2 + 1]



- (a) & (b) can be found **recursively**

- (c) can be found in two steps
  - Consider MCS in A[1..n/2] ending in A[n/2].
  - Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  - Sum these two maximum

Idea of divide-and-conquer

Example 1:		10	15	-3	-4	-2	-1	8	5
max on L (recursion)	25	10	15						
max on R (recursion)	13							8	5
mid extend to L	28	10	15	-3	-4				
			18						
mid extend to R						-2	-1	8	5
							10		

– Possible candidates:
  – 25, 13, 28 (=18+10)
  – overall maximum **28**.

Idea of divide-and-conquer

Example 2:		-2	5	-1 |	-5	2	-1	2
max on L (recursion)	5		<u>5</u>					
max on R (recursion)	3					<u>2</u>	<u>-1</u>	<u>2</u>
mid extend to L	4		<u>5</u>	<u>-1</u>				
mid extend to R	0				*<u>not take any</u>*			

$\left.\begin{matrix}4\\0\end{matrix}\right\}4$

- Possible candidates:
  - 5, 3, 4 (=4+0)
  - overall maximum **5**

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

a. MCS in A[1..n/2]
b. MCS in A[n/2+1..n]
c. MCS that spans across A[n/2]

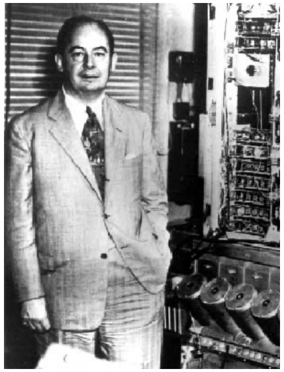
$2 \cdot T(n/2)$

- (a) & (b) can be found **recursively**
- (c) can be found in two steps
  - Consider MCS in A[1..n/2] ending in A[n/2].
  - Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  - Sum these two maximum

$O(n)$

Total time: $T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$

Mergesort - Recap



John von Neumann (1945)

1. Divide array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge two halves to make sorted whole.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S	divide	O(1)
A	G	L	O	R	H	I	M	S	T	sort	2T(n/2)

A	G	H	I	L	M	O	R	S	T	merge	O(n)

Merging

- Merging. Combine two pre-sorted lists into a sorted whole.

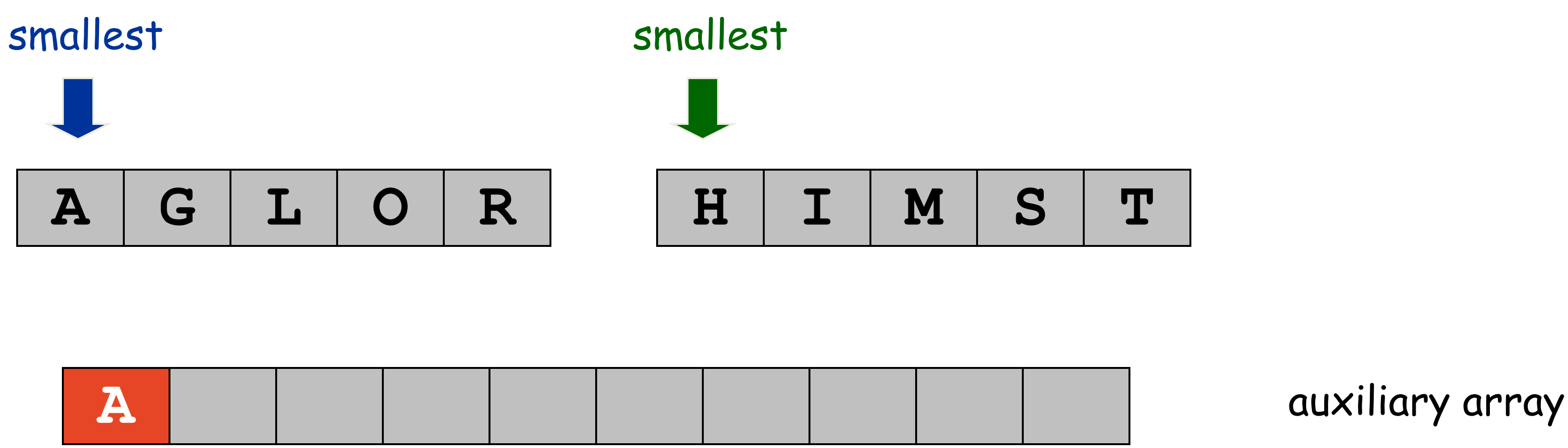
- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.

A	G	L	O	R	H	I	M	S	T

A	G	H	I						

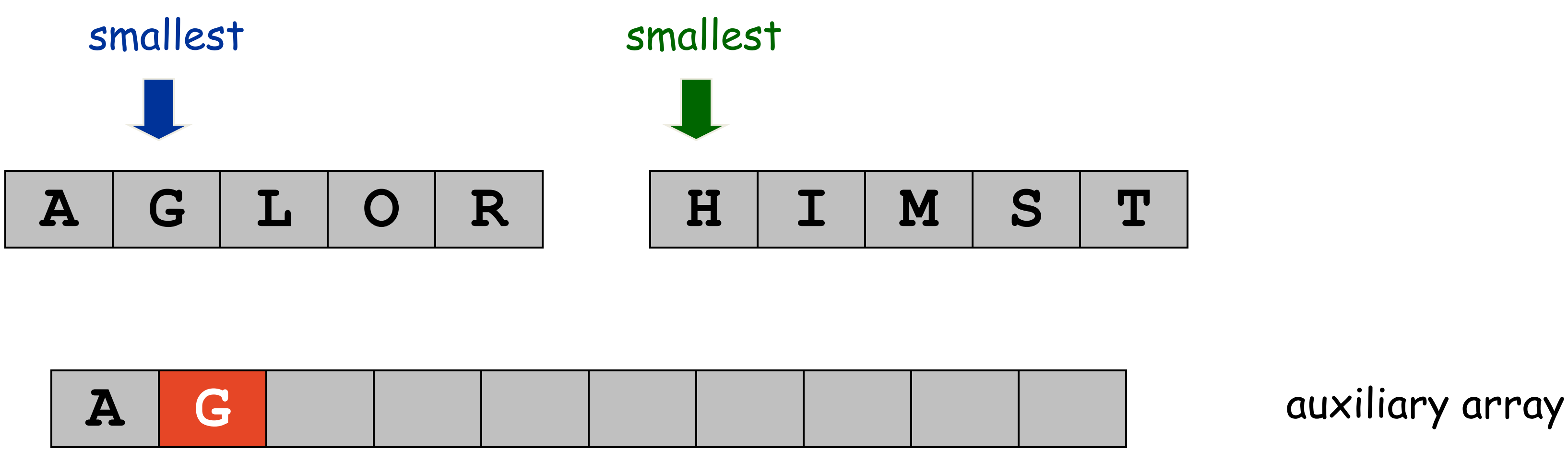
Merging

- Merge.
  - Keep track of smallest unprocessed element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



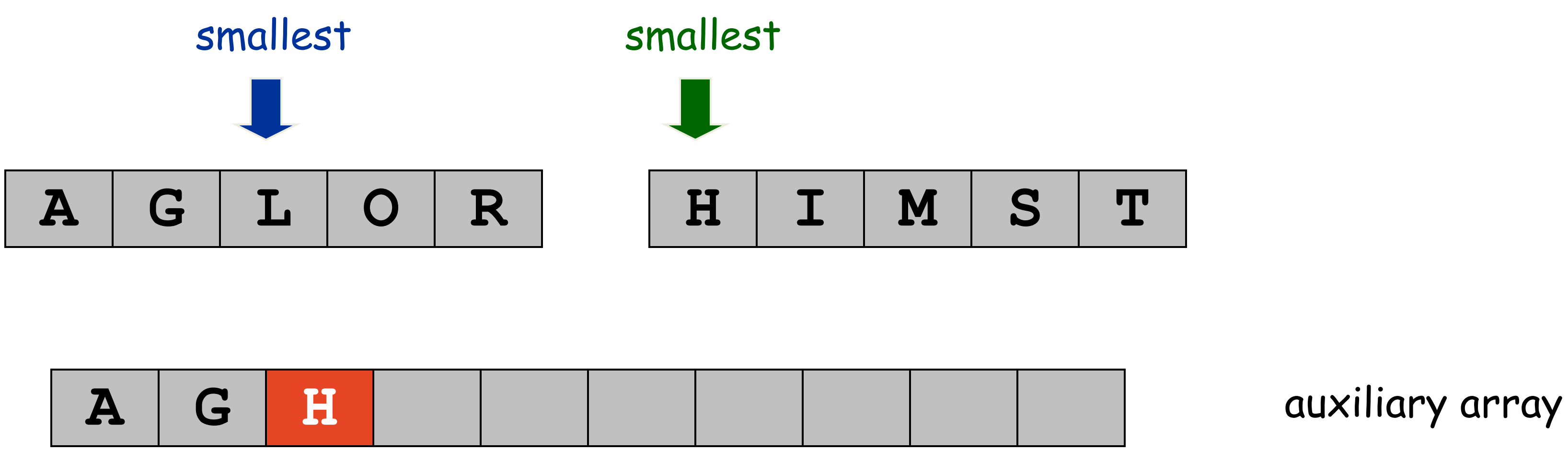
Merging

- Merge.
  - Keep track of smallest unprocessed element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



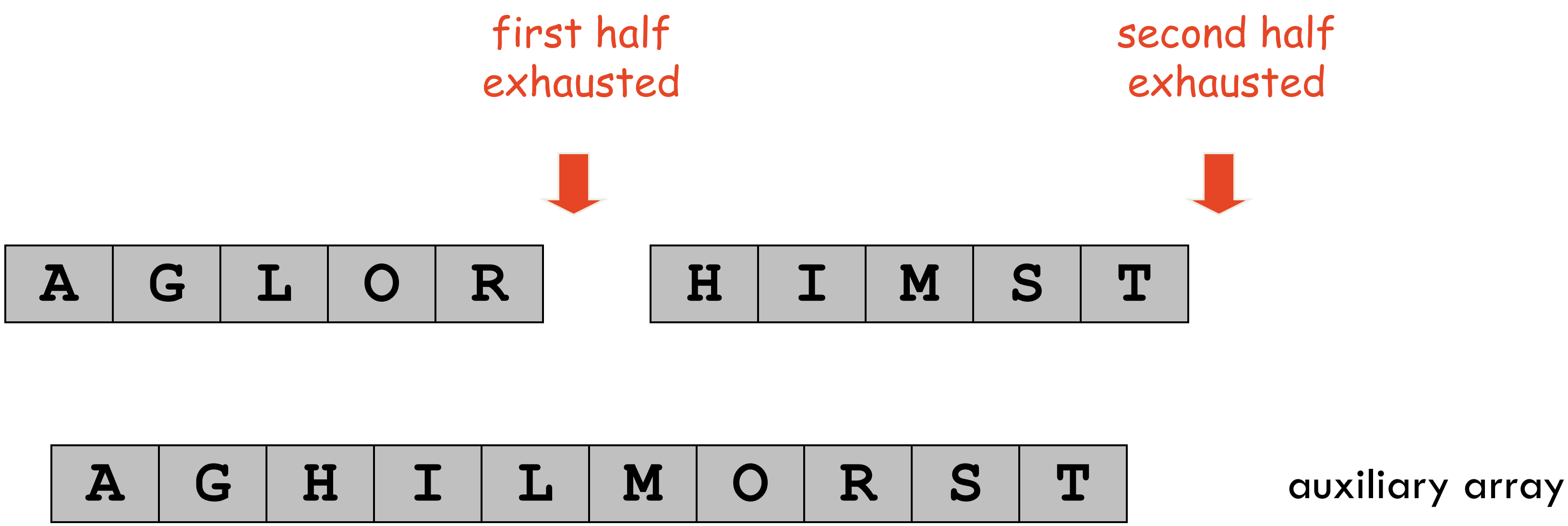
Merging

- Merge.
  - Keep track of smallest unprocessed element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Merging

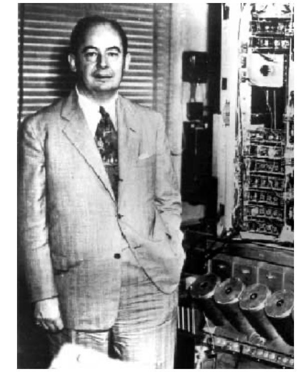
- Merge.
  - Keep track of smallest unprocessed element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Total # comparisons: O(n)

Note: runtime dominated by # comparisons

Mergesort



John von Neumann (1945)

1. Divide array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge two halves to make sorted whole.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S	divide	O(1)
A	G	L	O	R	H	I	M	S	T	sort	2T(n/2)

A	G	H	I	L	M	O	R	S	T	merge	O(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 k inverted if $i < k$, but $a_i > a_k$.

Songs

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

Inversions

3-2, 4-2

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

Applications

- Applications.
  - Voting theory.
  - Collaborative filtering.
  - Measuring the "sortedness" of an array.
  - Sensitivity analysis of Google's ranking function.
  - Rank aggregation for meta-searching on the Web.
  - Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions: Divide-and-Conquer

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

3	14	10	18	19	7	11	17	25	23	2	16

Counting Inversions: Divide-and-Conquer

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

3	14	10	18	19	7	11	17	25	23	2	16

Divide: O(1).

3	14	10	18	19	7

11	17	25	23	2	16

Counting Inversions: Divide-and-Conquer

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

3	14	10	18	19	7	11	17	25	23	2	16

Divide: O(1).

3	14	10	18	19	7

11	17	25	23	2	16

Conquer: 2T(n/2)

5 blue-blue inversions

14-10, 14-7, 10-7, 18-7, 19-7

8 green-green inversions

11-2, 17-2, 17-16, 25-2,
25-16, 25-23, 23-2, 23-16

Counting Inversions: Divide-and-Conquer

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

3	14	10	18	19	7	11	17	25	23	2	16

Divide: O(1).

3	14	10	18	19	7

11	17	25	23	2	16

Conquer: 2T(n/2)

5 blue-blue inversions

8 green-green inversions

13 blue-green inversions

11-14, 11-18, 11-19, 17-18, 17-19, 2-3, 2-14, 2-10, 2-18, 2-19, 2-7, 16-18, 16-19

Combine: ???

Total = 5 + 8 + 13 = 26.

Counting Inversions: Divide-and-Conquer

Key Observation:

- For each a_k on the right half, the number of blue-green inversions it is involved in is exactly the number of elements on the left half that is larger than a_k
- Computing this is much easier if the two halves are sorted.

3	14	10	18	19	7	11	17	25	23	2	16

Divide: O(1).

3	14	10	18	19	7

5 blue-blue inversions

11	17	25	23	2	16

8 green-green inversions

Conquer: 2T(n/2)

13 blue-green inversions

11-14, 11-18, 11-19, 17-18, 17-19, 2-3, 2-14, 2-10, 2-18, 2-19, 2-7, 16-18, 16-19

Combine: ???

Total = 5 + 8 + 13 = 26.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Merge two sorted halves into sorted whole.
- Simultaneously, count inversions where a_i and a_k are in different halves.

3	7	10	14	18	19

5 blue-blue inversions

2	11	16	17	23	25
6	3	2	2	0	0

8 green-green inversions

How many blue-green inversions?

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 6

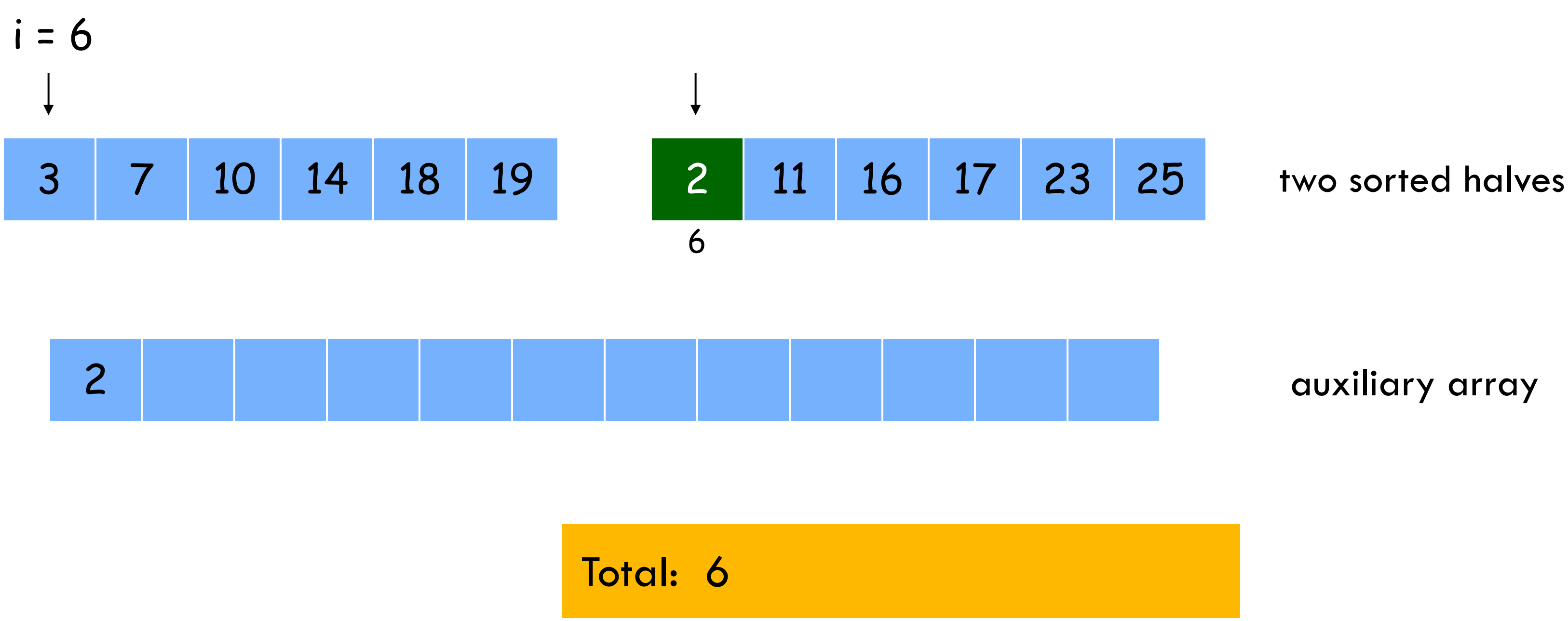
3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves

												auxiliary array

Total:

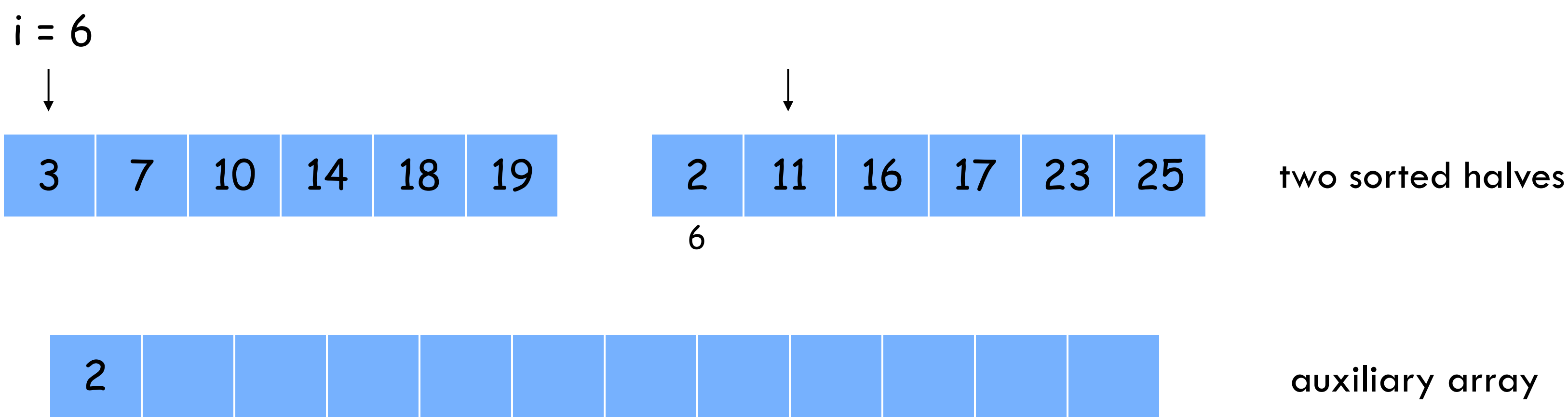
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 6

3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves
						6						

2	3											auxiliary array

Total: 6

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 5

3	7	10	14	18	19		2	11	16	17	23	25
							6					

two sorted halves

2	3										

auxiliary array

Total: 6

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 5

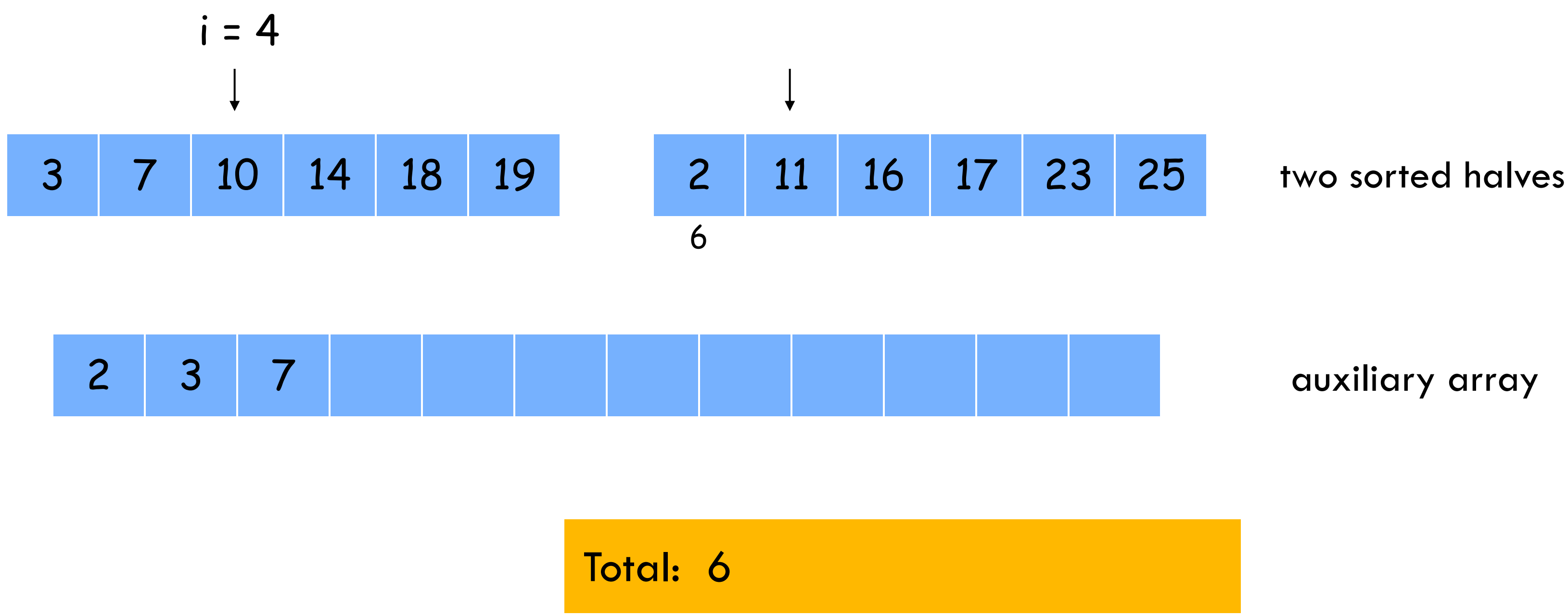
3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves
						6						

2	3	7										auxiliary array

Total: 6

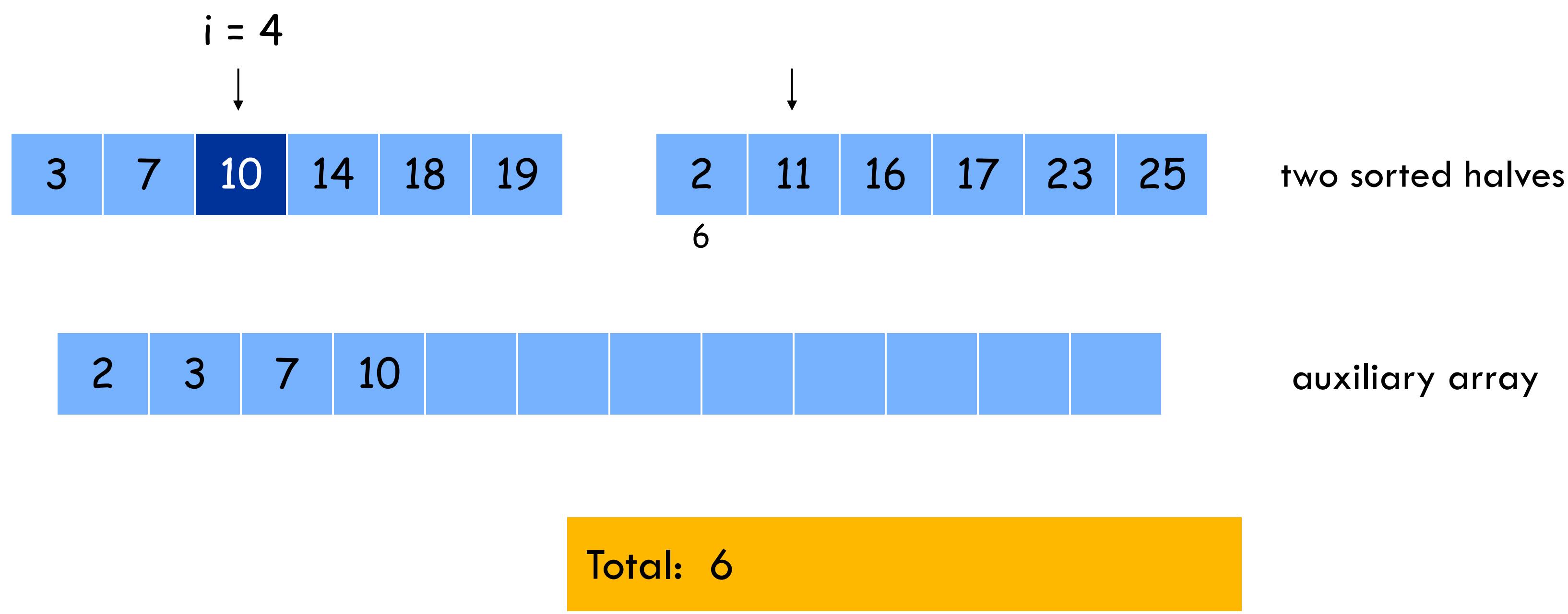
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



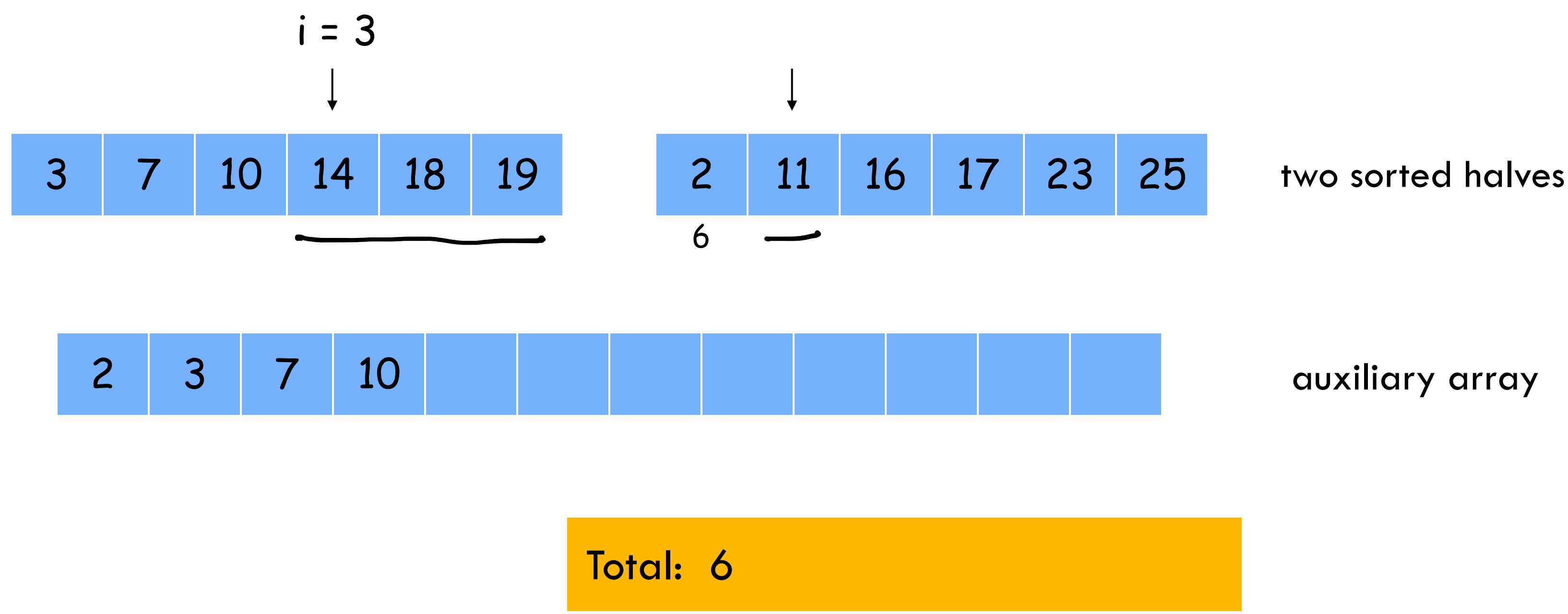
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



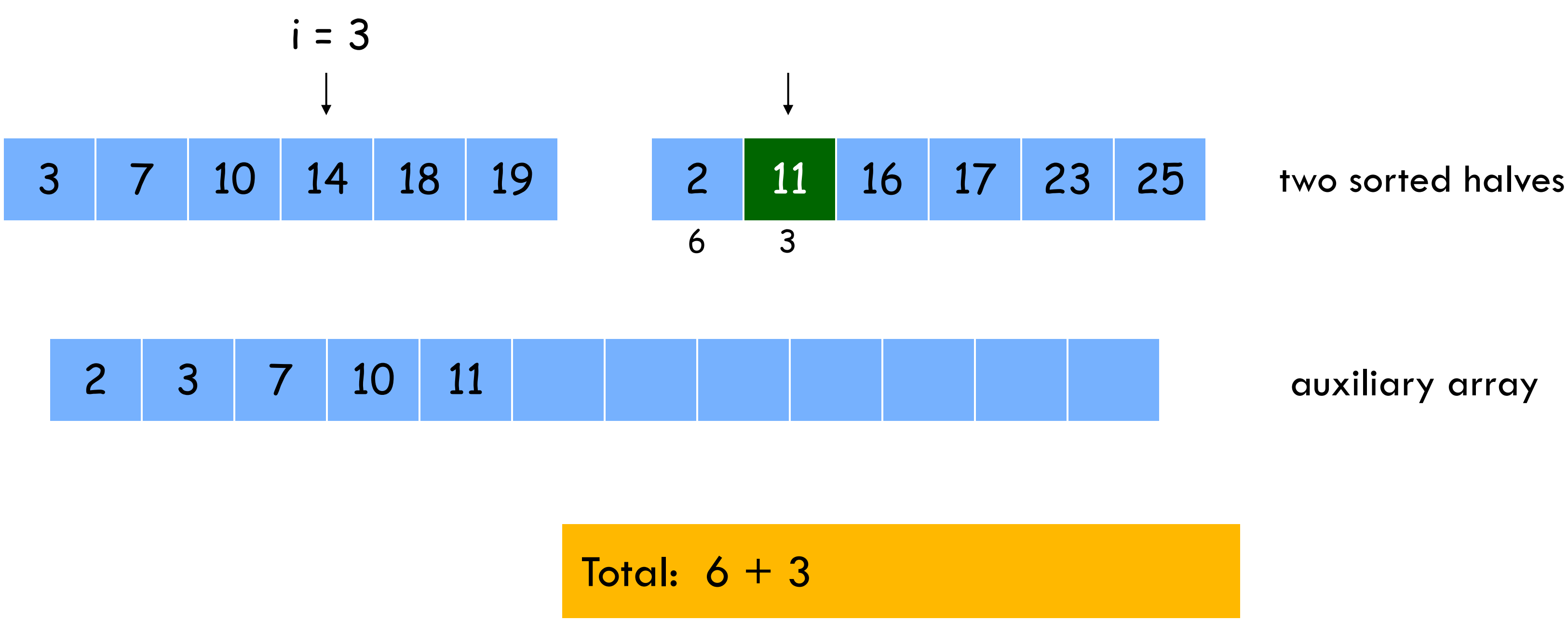
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



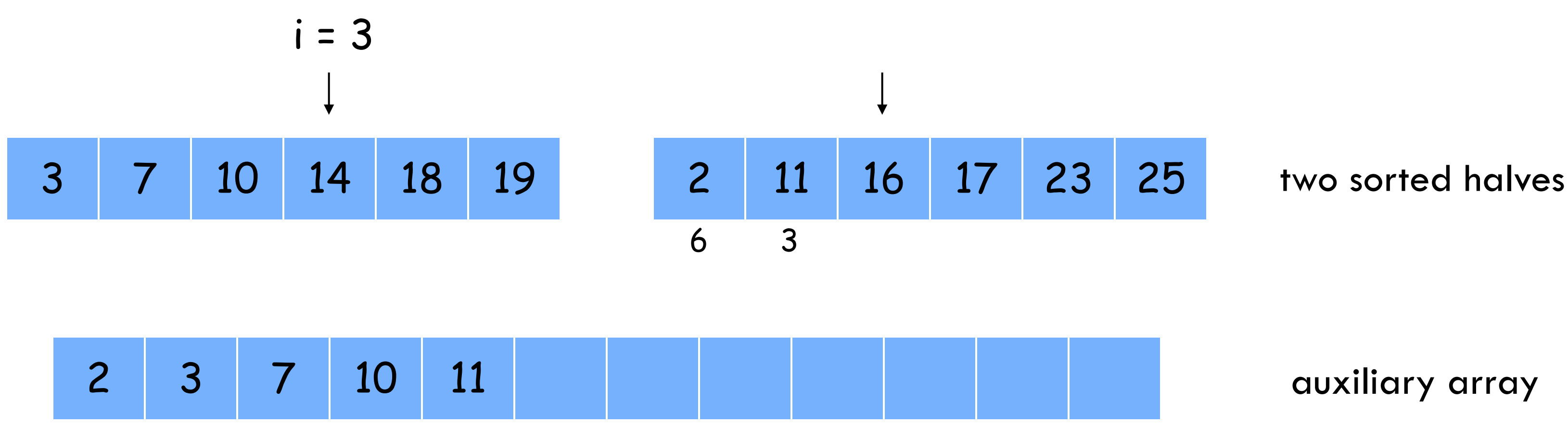
Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 3

3	7	10	14	18	19		2	11	16	17	23	25	two sorted halves
							6	3					

2	3	7	10	11	14							auxiliary array

Total: 6 + 3

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 2

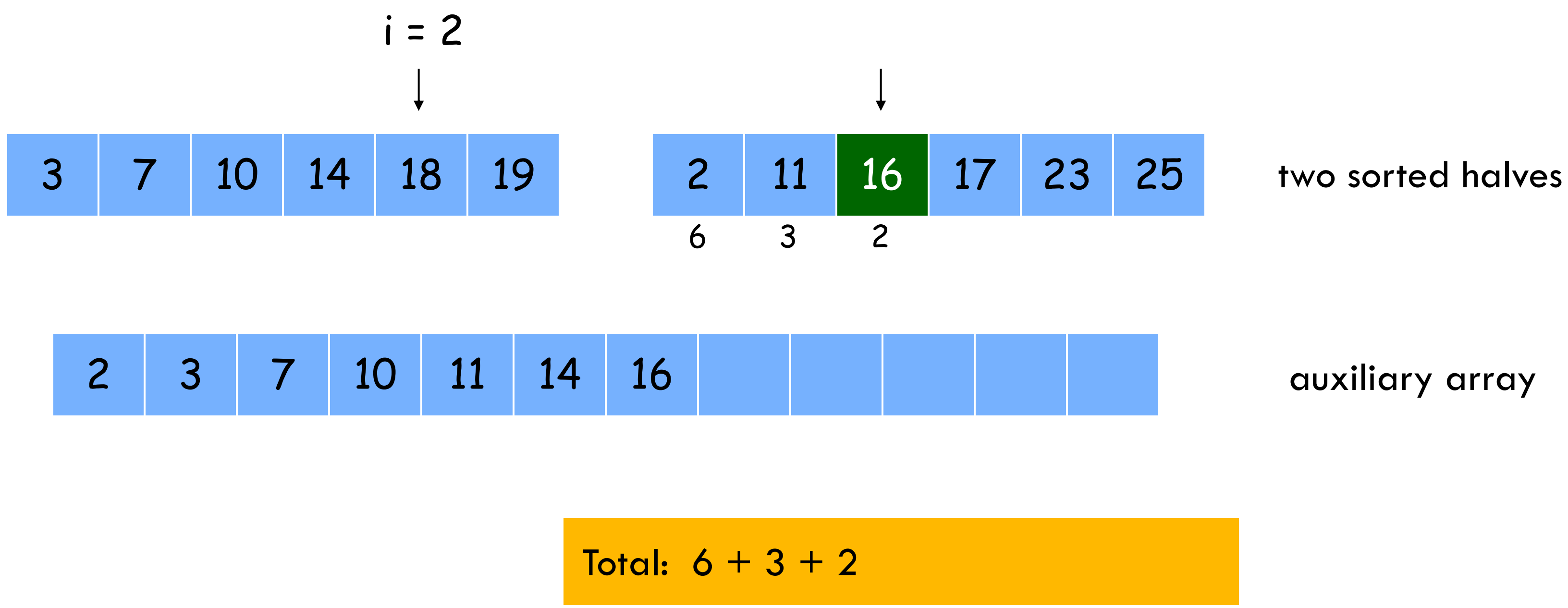
3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves
						6	3					

2	3	7	10	11	14							auxiliary array

Total: 6 + 3

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 2

3	7	10	14	18	19		2	11	16	17	23	25
							6	3	2			

two sorted halves

2	3	7	10	11	14	16					

auxiliary array

Total: 6 + 3 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 2

3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves
						6	3	2	2			

2	3	7	10	11	14	16	17					auxiliary array

Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 2

two sorted halves

3	7	10	14	18	19		2	11	16	17	23	25
							6	3	2	2		

auxiliary array

2	3	7	10	11	14	16	17				

Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 2

two sorted halves

3	7	10	14	18	19	2	11	16	17	23	25
						6	3	2	2		

auxiliary array

2	3	7	10	11	14	16	17	18			

Total: $6 + 3 + 2 + 2$

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 1

two sorted halves

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2		

auxiliary array

2	3	7	10	11	14	16	17	18			

Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 1

two sorted halves

3	7	10	14	18	**19**		2	11	16	17	23	25
							6	3	2	2		

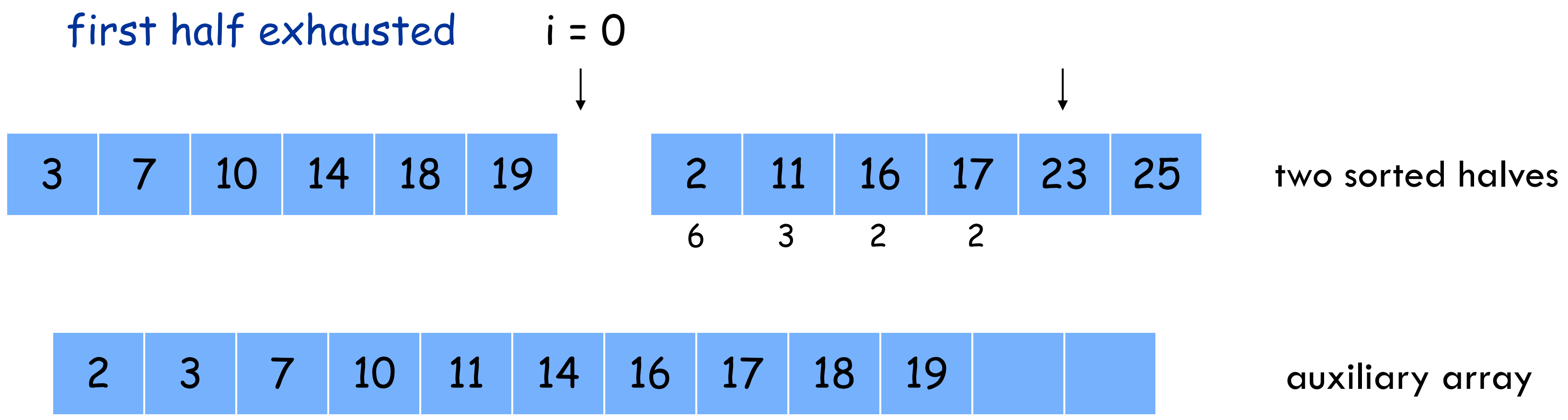
auxiliary array

2	3	7	10	11	14	16	17	18	19		

Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.



Total: 6 + 3 + 2 + 2

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

two sorted halves

3	7	10	14	18	19		2	11	16	17	23	25
							6	3	2	2	0	

auxiliary array

2	3	7	10	11	14	16	17	18	19	23	

Total: 6 + 3 + 2 + 2 + 0

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

3	7	10	14	18	19		2	11	16	17	23	25
							6	3	2	2	0	

two sorted halves

2	3	7	10	11	14	16	17	18	19	23	

auxiliary array

Total: 6 + 3 + 2 + 2 + 0

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

3	7	10	14	18	19	2	11	16	17	23	25	two sorted halves
						6	3	2	2	0	0	

2	3	7	10	11	14	16	17	18	19	23	25	auxiliary array

Total: 6 + 3 + 2 + 2 + 0 + 0

Merge and Count

- Merge and count step.
  - Given two sorted halves, count number of inversions where a_i and a_k are in different halves.
  - Combine two sorted halves into sorted whole.

i = 0

two sorted halves

3	7	10	14	18	19	2	11	16	17	23	25
						6	3	2	2	0	0

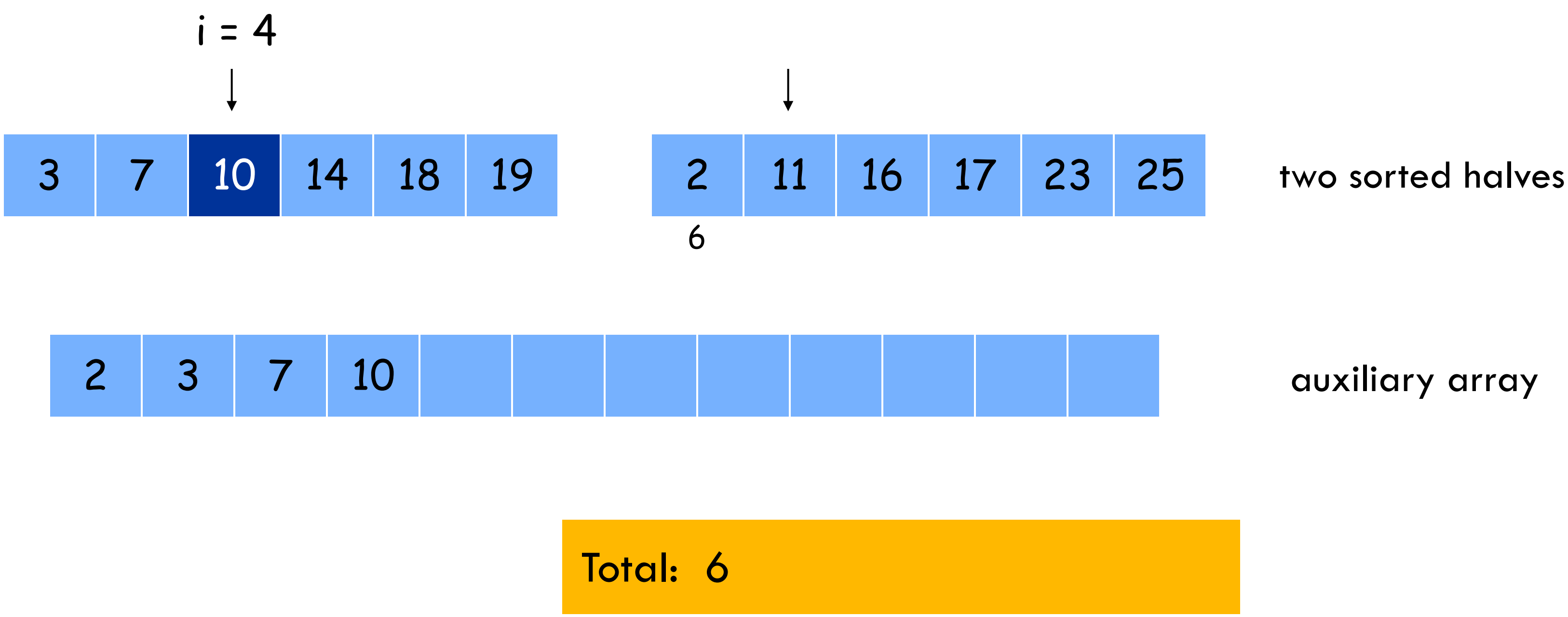
auxiliary array

2	3	7	10	11	14	16	17	18	19	23	25

Total: $6 + 3 + 2 + 2 + 0 + 0 = 13$

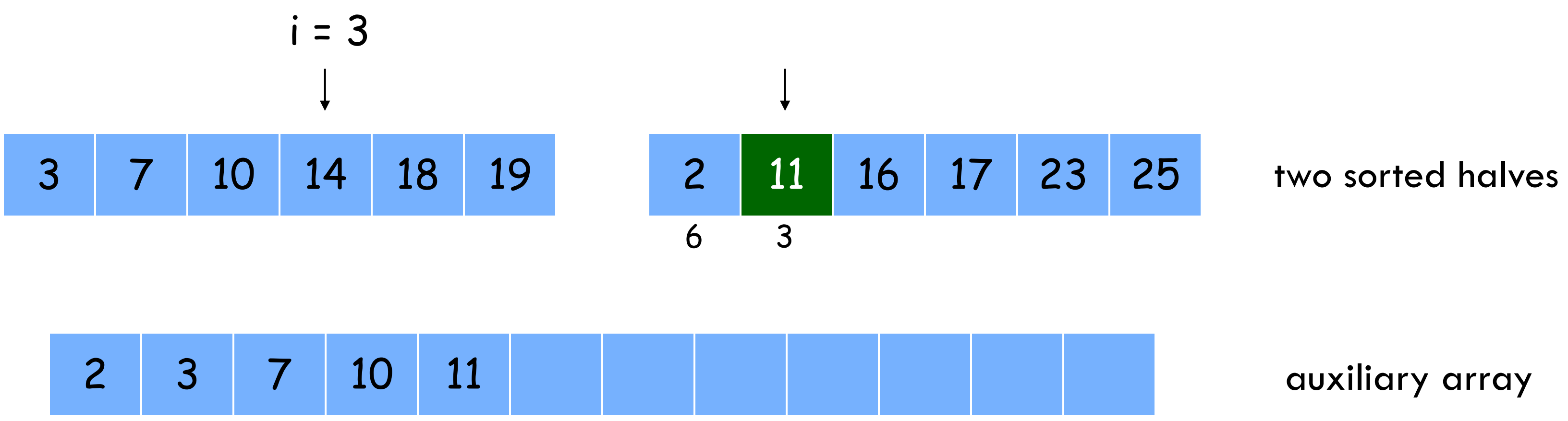
Merge and Count

- Correctness.
  - When we place an element from left half in auxiliary array, it is smaller than remaining elements in right half



Merge and Count

- Correctness.
  - When we place an element from left half in auxiliary array, it is smaller than remaining elements in right half
  - When we place element from right half in auxiliary array, it is larger than remaining elements in left half



Total: 6 + 3

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_k are in different halves.
- Merge two sorted halves into sorted whole.

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: $6 + 3 + 2 + 2 + 0 + 0$

Count: $O(n)$

2	3	7	10	11	14	16	17	18	19	23	25

Merge: $O(n)$

Time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Counting Inversions: Implementation

- Pre-condition. [Merge-and-Count] A and B are sorted.
- Post-condition. [Sort-and-Count] L is sorted.

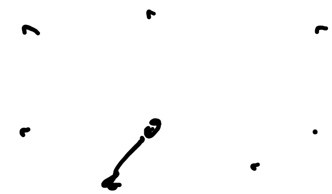
```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (rA, A') ← Sort-and-Count(A)
   (rB, B') ← Sort-and-Count(B)
   (rC, L) ← Merge-and-Count(A', B')

   return r = rA + rB + rC and the sorted list L
}
```

#inversions

Useful strategy: Strengthen inductive hypothesis

Closest Pair of Points

Closest Pair of Points

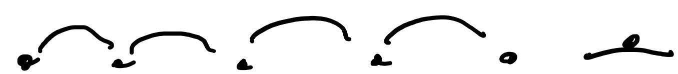


– **Closest pair.** Given n points in the plane, find a pair with smallest Euclidean distance between them.

– **Fundamental geometric primitive.**
  – Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
  – Special case of nearest neighbor, Euclidean MST, Voronoi diagram...

– **Warm up 1: Brute force.** Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

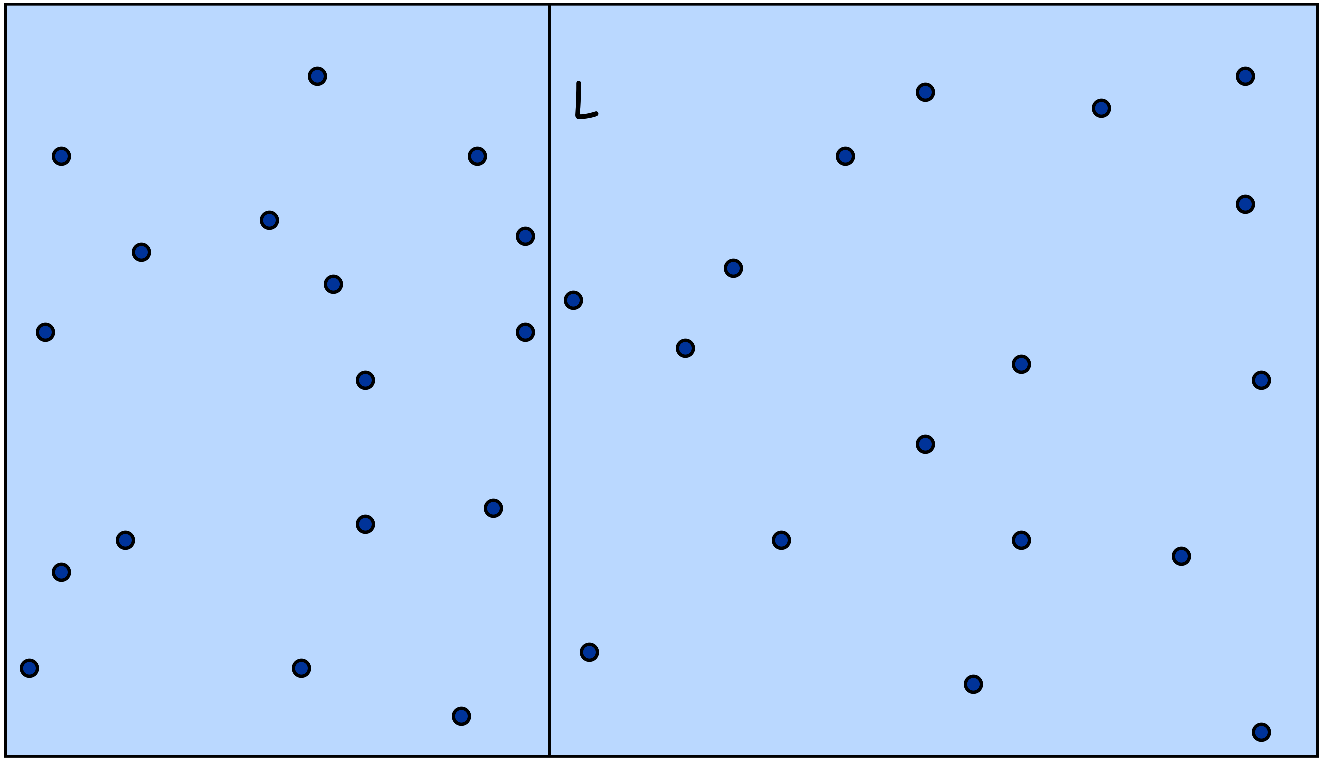
– **Warm up 2: 1-D version.** O(n log n) easy if points are on a line.



– **Assumption.** No two points have same x coordinate.

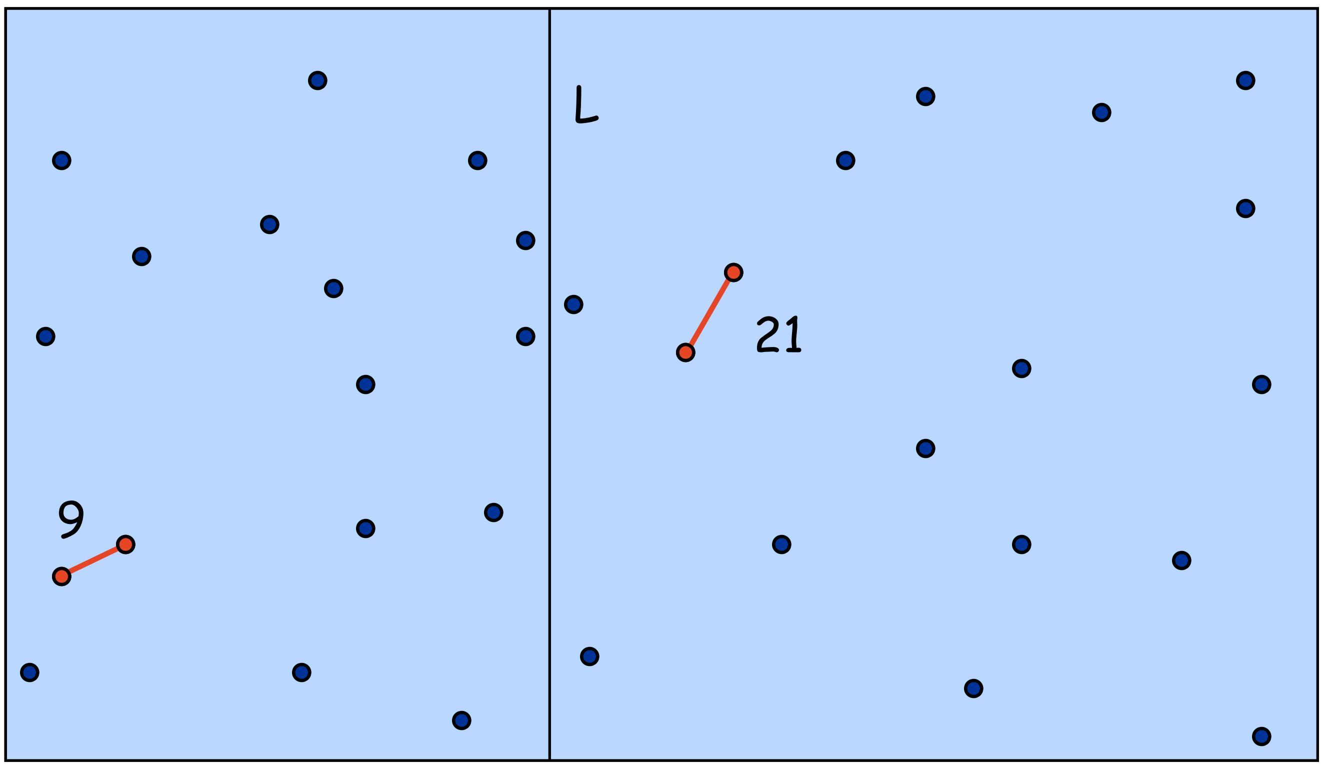
Closest Pair of Points

- **Algorithm.**
  - Divide: draw vertical line L so that exactly ½n points on each side.



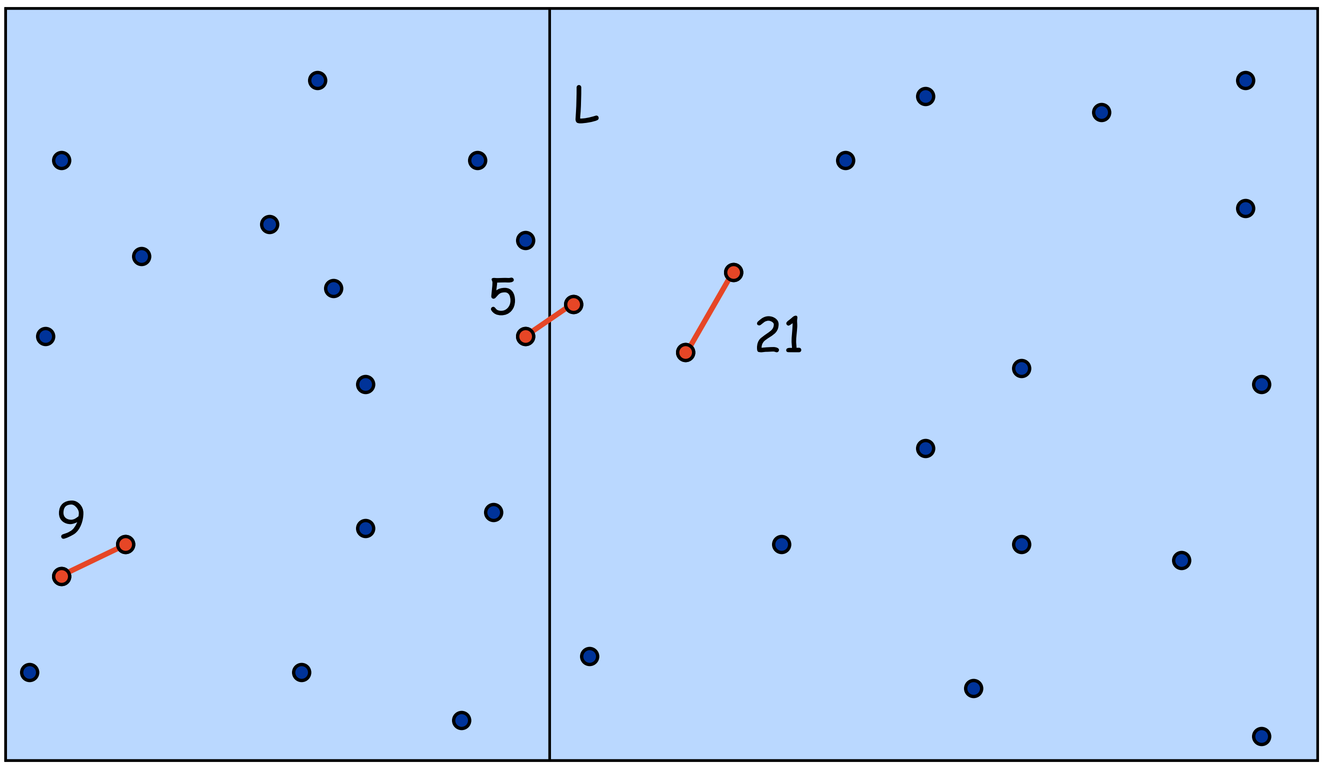
Closest Pair of Points

- **Algorithm.**
  - Divide: draw vertical line L so that roughly ½n points on each side.
  - Conquer: find closest pair in each side recursively.



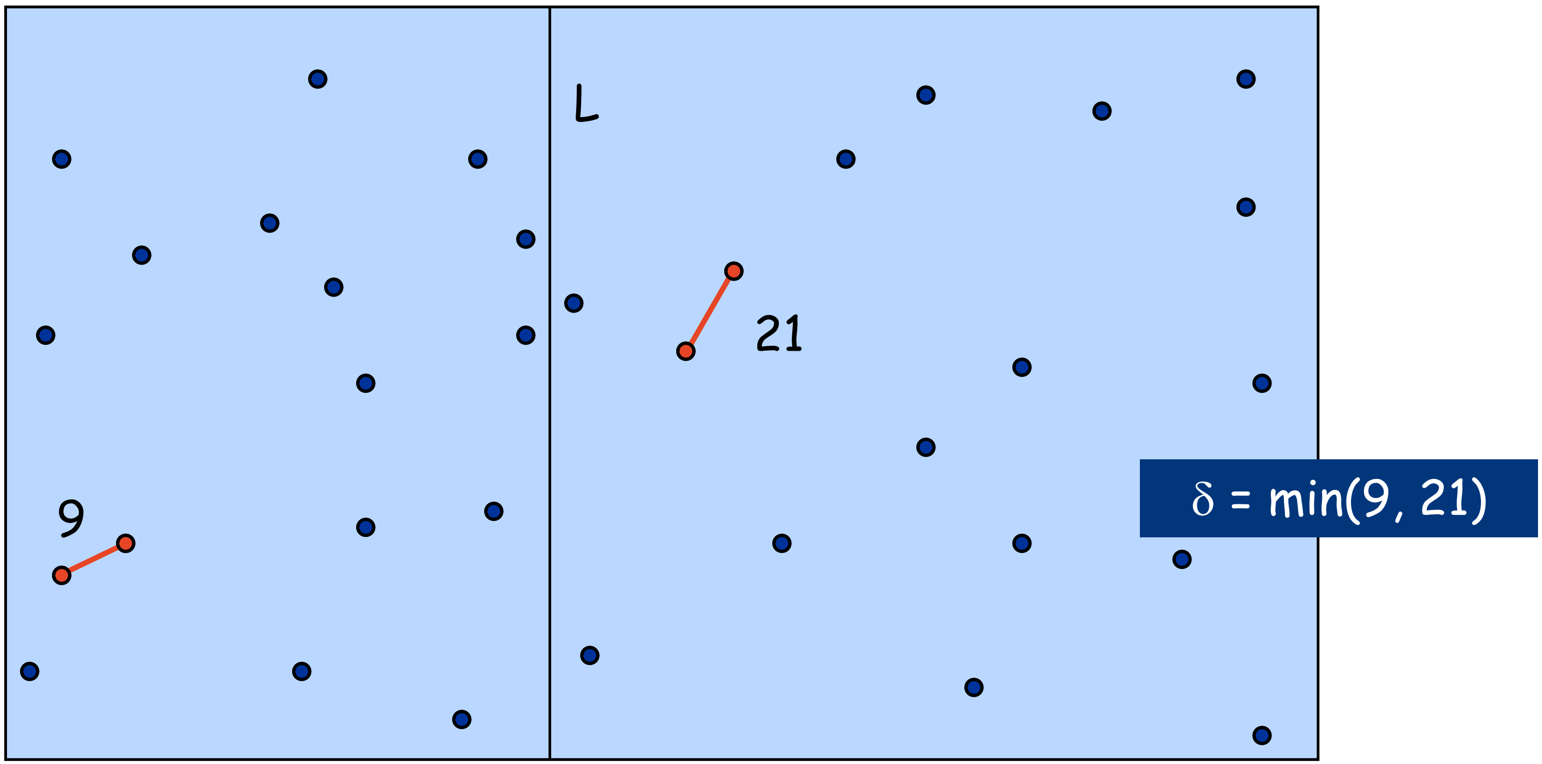
Closest Pair of Points

- **Algorithm.**
  - Divide: draw vertical line L so that roughly ½n 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.



Closest Pair of Points

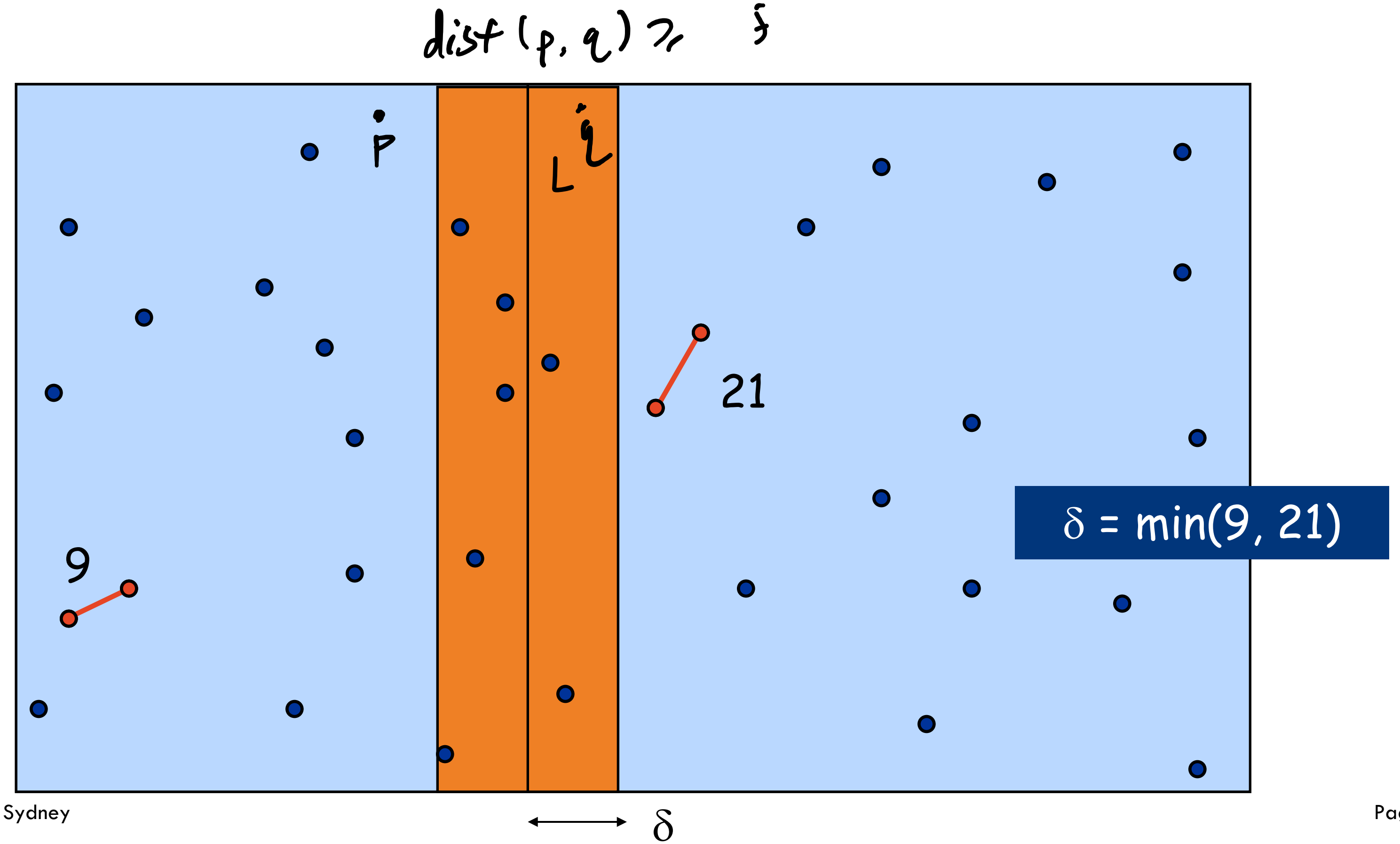
– Find closest pair with one point in each side, assuming that δ = min(closest pair in left half, closest pair in right half).



Closest Pair of Points

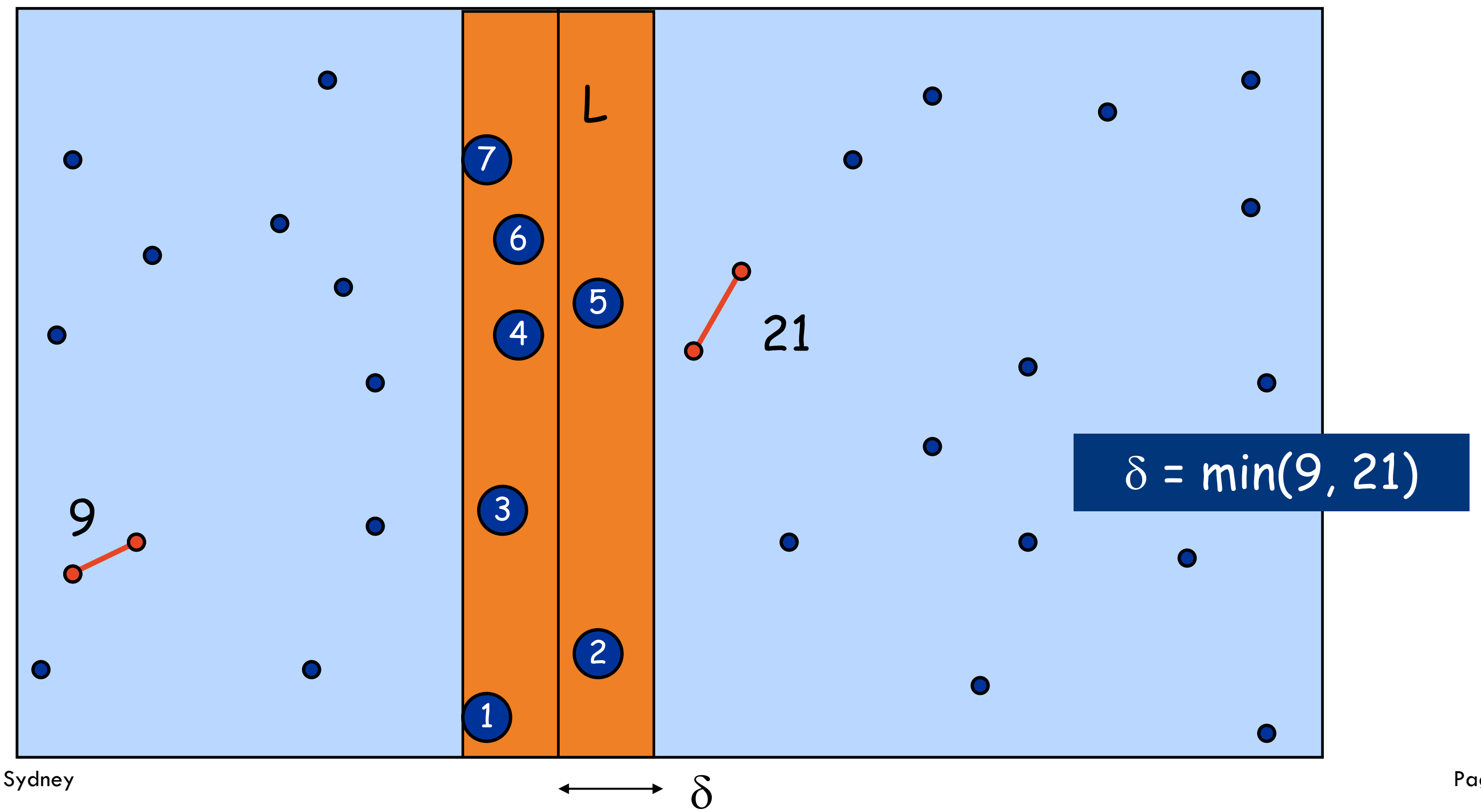
- Find closest pair with one point in each side, assuming that δ = min(closest pair in left half, closest pair in right half).
  - **Observation:** only need to consider points within δ of line L.

dist(p, q) ≥ δ



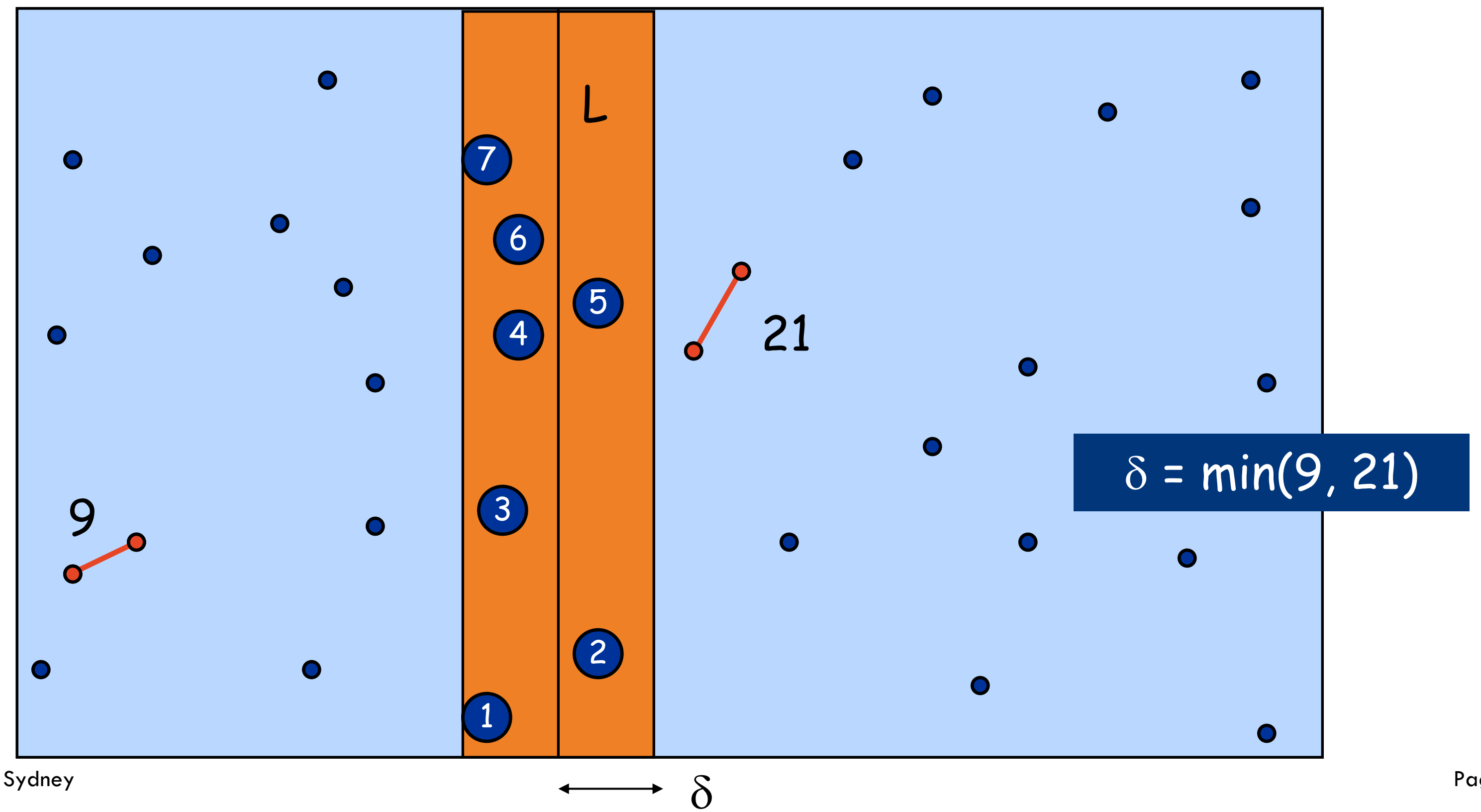
Closest Pair of Points

- Find closest pair with one point in each side, assuming that δ = min(closest pair in left half, closest pair in right half).
  - **Observation:** only need to consider points within δ of line L.
  - Sort points in 2δ-strip by their y-coordinate.



Closest Pair of Points

- Find closest pair with one point in each side, assuming that δ = min(closest pair in left half, closest pair in right half).
  - **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 7 positions in sorted list!

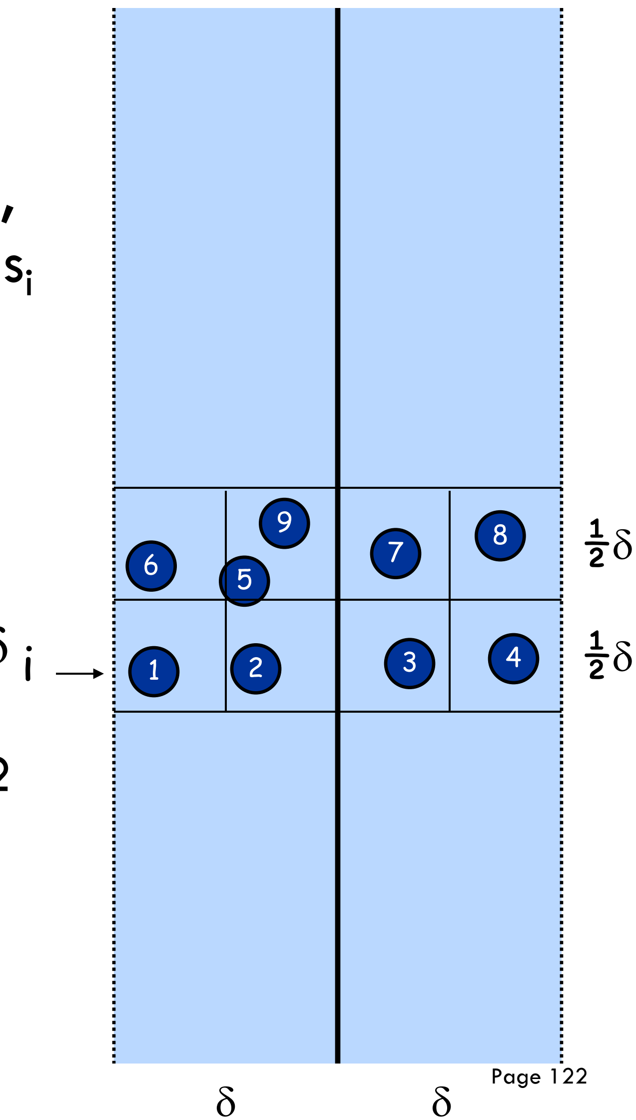


Closest Pair of Points

- **Definition:** Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate y_i.

- **Claim:** For any 9 consecutive points $s_i, \ldots, s_{i+8}$ in the ordering, the distance between s_i and s_k is $> \delta$. In fact, $y_{i+8} - y_i > \delta$.

- **Proof:**
  - Suppose that $y_{i+8} - y_i \leq \delta$.
  - Then $s_i, \ldots, s_{i+8}$ lie in a rectangle with width 2δ and height δ.
  - Partition rectangle into 8 squares of width $\delta/2$
  - At least 2 of $s_i, \ldots, s_{i+8}$ lie in same square.
  - But 2 points in square of width $\delta/2$ have distance $< \delta$!
  - Since each square completely in left or right side, this contradicts definition of δ

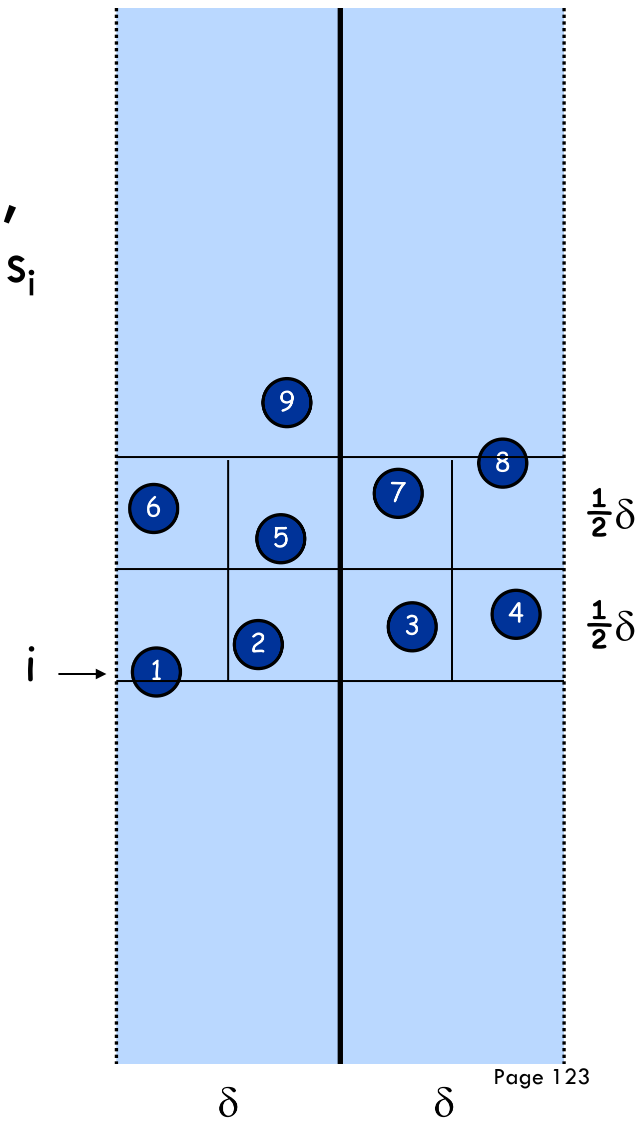


Closest Pair of Points

- **Definition:** Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate y_i.

- **Claim:** For any 9 consecutive points $s_i, \ldots, s_{i+8}$ in the ordering, the distance between s_i and s_k is $> \delta$. In fact, $y_{i+8} - y_i > \delta$.

- **Alternative Proof:**
  - Draw rectangle of height δ and width 2δ such that s_i is on bottom edge of rectangle
  - Divide rectangle into squares of width δ
  - No two points lie in same square, by def of δ
  - Thus, at most 8 points (including s_i) can be in rectangle.
  - So $y_{i+8} - y_i > \delta$



Closest Pair Algorithm

```
Closest-Pair(p₁, …, pₙ) {

 If |P|≤ 3 then compute closest-pair brute force
 else
   Compute separation line L such that half the points     O(n log n)
   are on one side and half on the other side.

   δ₁ = Closest-Pair(left half)                            2T(n / 2)
   δ₂ = Closest-Pair(right half)
   δ  = min(δ₁, δ₂)

   Delete all points further than δ from separation line L O(n)

   Sort remaining points by y-coordinate.                  O(n log n)

   Scan points in y-order and compare distance between     O(n)
   each point and next 7 neighbors. If any of these
   distances is less than δ, update δ.

 return δ.
}
```

Closest Pair of Points: Analysis

- **Running time**

$$T(n) \le 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$$

- **Question:** Can we achieve O(n log n)?

- **Answer:** Yes. Don't sort points in strip from scratch each time.
  - Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
  - Sort by merging two pre-sorted lists.

$$T(n) \le 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$

Sort P by x-coordinates $\Rightarrow P_x$
Sort P by y-coordinates $\Rightarrow P_y$ } preprocessing $O(n \log n)$

Closest-Pair(P_x, P_y) {

  If $|P| \leq 3$ **then** compute closest-pair brute force
  else
    Compute separation line L $O(1)$

Divide step
    $P_{x,left}$ = points to the left of L sorted by x-coordinate
    $P_{y,left}$ = points to the left of L sorted by y-coordinate $O(n)$
    $P_{x,right}$ = points to the right of L sorted by x-coordinate
    $P_{y,right}$ = points to the right of L sorted by y-coordinate

Conquer step
    δ_1 = Closest-Pair($P_{x,left}$, $P_{y,left}$)
    δ_2 = Closest-Pair($P_{x,right}$, $P_{y,right}$) $2T(n / 2)$

Merge step
    δ = min(δ_1, δ_2)

    Delete all points further than δ from separation line L $O(n)$

    Scan points in y-order and compare distance between each point and next 7 neighbors. If any of these distances is less than δ, update δ. $O(n)$

  return δ.
}

Summary: Divide-and-Conquer

- **Divide-and-conquer.**
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- **Master theorem**

- **Problems**
  - Maximum Contiguous Subarray
  - Counting inversions
  - Closest pair

This weeks quiz is all about solving recurrences!