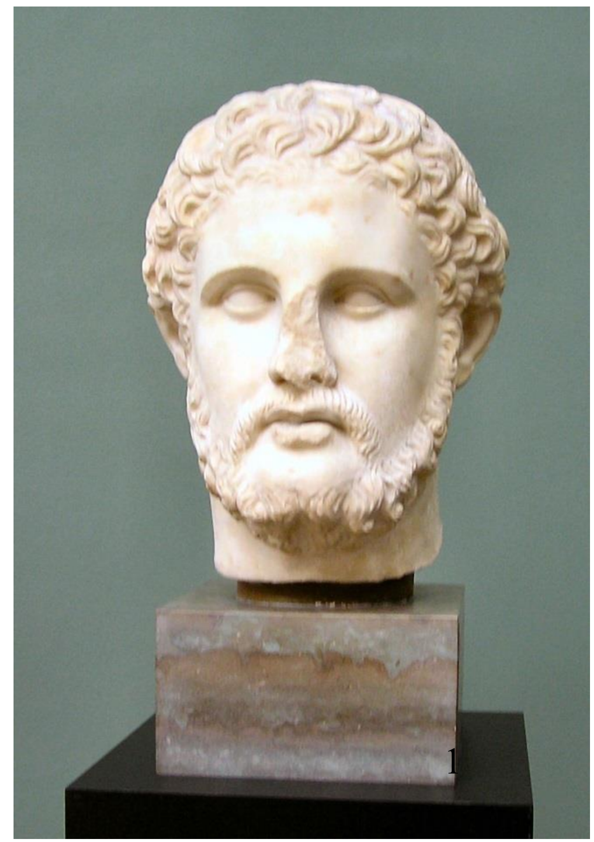
Advanced Algorithms

Problem solving Techniques.

Divide and Conquer

הפרד ומשול



Divide and Conquer

- A method of designing algorithms that (informally) proceeds as follows:
- Given an instance of the problem to be solved, split it into several, smaller, sub-instances (*of the same problem*);
  independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance.

Divide and Conquer

Question: By what methods the sub-instances are independently solved ?

Answer: By the same method, till we have a constant size problem that can be solved in constant time.

This simple answer is central to the concept of *Divide-&-Conquer* algorithms, *and* is a key factor in measuring their efficiency.

Divide and Conquer: Outline

- **Divide** the problem into a number of sub-problems (similar to the original problem but smaller);
- **Conquer** the sub-problems by solving them recursively (if a sub-problem is small enough, just solve it in a straightforward manner).
- **Combine** the solutions for the sub-problems into a solution for the original problem

Example 1: Binary Search

- A directory contains a set of *names* and a telephone *number* is associated with each name.
- The directory is sorted by alphabetical order of names. It contains n entries each having the form [name, number]
- Given a *name* and the value n, the problem is to find the *number* associated with the name
- We assume that any given input name actually *does occur* in the directory.

Binary Search

The Divide & Conquer algorithm for this problem is based on the following:

Given a name, say X, there are 3 possibilities:

X occurs in the *middle* of the *names* array

Or

X occurs in the *first* half of the *names* array.

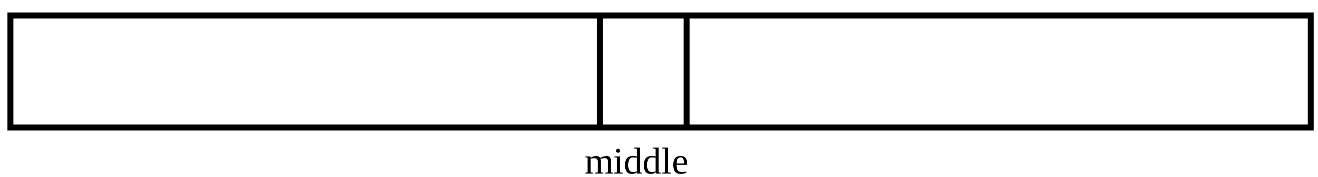
Or

X occurs in the *second* half of the *names* array.

Binary Search

region of answer

```
function binsearch (X : name; start, finish : int)
begin middle := (start+finish)/2;
 if name(middle)=x  return  number(middle);
 else  if X < name(middle) return
   binsearch(X,start,middle-1);
      else [X > name(middle)] return
   binsearch(X,middle+1,finish);
 end  if;
end search;
```



Binary Search

- **Divide** the n-element array into a middle element and two sub-arrays of n/2 (-1) elements.
- **Conquer:** Consider the middle element, if name not found, ignore one sub-array, and solve the problem for the other sub-array **using Binary search**
- **Combine:** Empty.

Binary Search - Performance Analysis

- $T(1) = c_1$ (constant time)
- for n > 1, we have

$$T(n) = T(n/2) + c_2$$

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ T(n/2) + c_2 & \text{if } n > 1 \end{cases}$$

Example 2: Merge Sort

- Sorting problem: Given an array, order the elements according to some order (say increasing value)
- Merge sort: A *sort* algorithm that splits the elements to be sorted into two groups, *recursively* sorts each group, and *merges* them into a final, sorted sequence.

Merge Sort

- **Divide** the n-element sequence to be sorted into two subsequences of n/2 elements each
- **Conquer:** Sort the two subsequences recursively **using merge sort**
- **Combine:** merge the two sorted subsequences to produce the sorted answer
- recursion base case: if the subsequence has only one element, then do nothing.

Merge-Sort(A,p,r)

sorts the elements in the sub-array A[p..r] using divide and conquer

- Merge-Sort(A,p,r)
  - if $p \geq r$, do nothing
  - if $p < r$ then $q \leftarrow \lfloor (p+r)/2 \rfloor$
    - Merge-Sort(A,p,q)
    - Merge-Sort(A,q+1,r)
    - Merge(A,p,q,r)

- Start by calling Merge-Sort($A,1,n$)
- Do we need an example?

Performance Analysis

Known: two sorted arrays of sizes n_1 and n_2 can be merged in time $c(n_1+n_2)$.

Let T(n) denote the time it takes to sort an n-elements array.

- $T(1) = O(1)$
- for $n > 1$, $T(n) = 2T(n/2) + cn$ ← Merging time

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Example 3: 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.

Songs

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

Inversions

3-2, 4-2

  - My rank: 1, 2, …, n. Your rank: $a_1, a_2, \ldots, a_n$.
  - Songs i and j inverted if $i < j$, but $a_i > a_j$.
- Brute force: check all $\Theta(n^2)$ pairs i and j.

Counting Inversions: Divide-and-Conquer

- Divide: separate 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

Divide: O(1).

1	5	4	8	10	2

5 blue-blue inversions

6	9	12	11	3	7

8 green-green inversions

Conquer: 2T(n / 2)

9 blue-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.

Counting Inversions: Combine

- Combine: count blue-green inversions
  - Assume each half is sorted.
  - Count inversions where a_i and a_j are in different halves.
  - Merge two sorted halves into sorted whole.

} Merge-and-Count

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

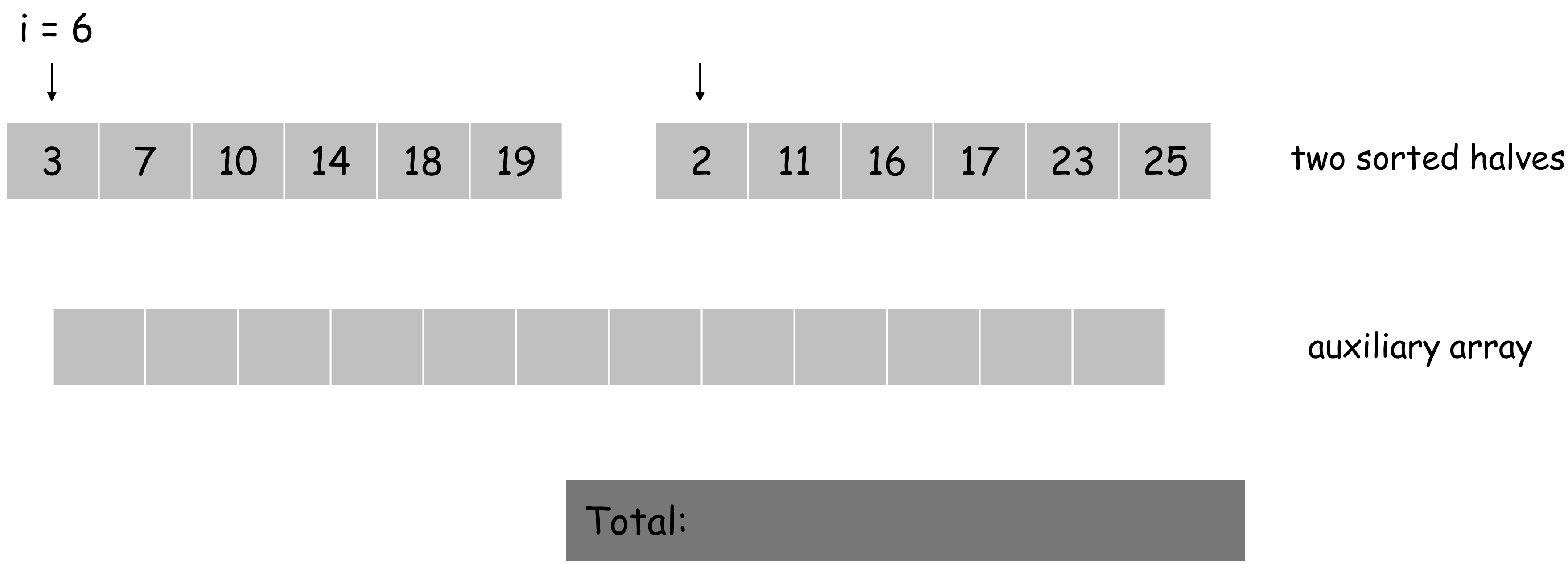
Merge: O(n)

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

Merge and Count

Merge and count step.

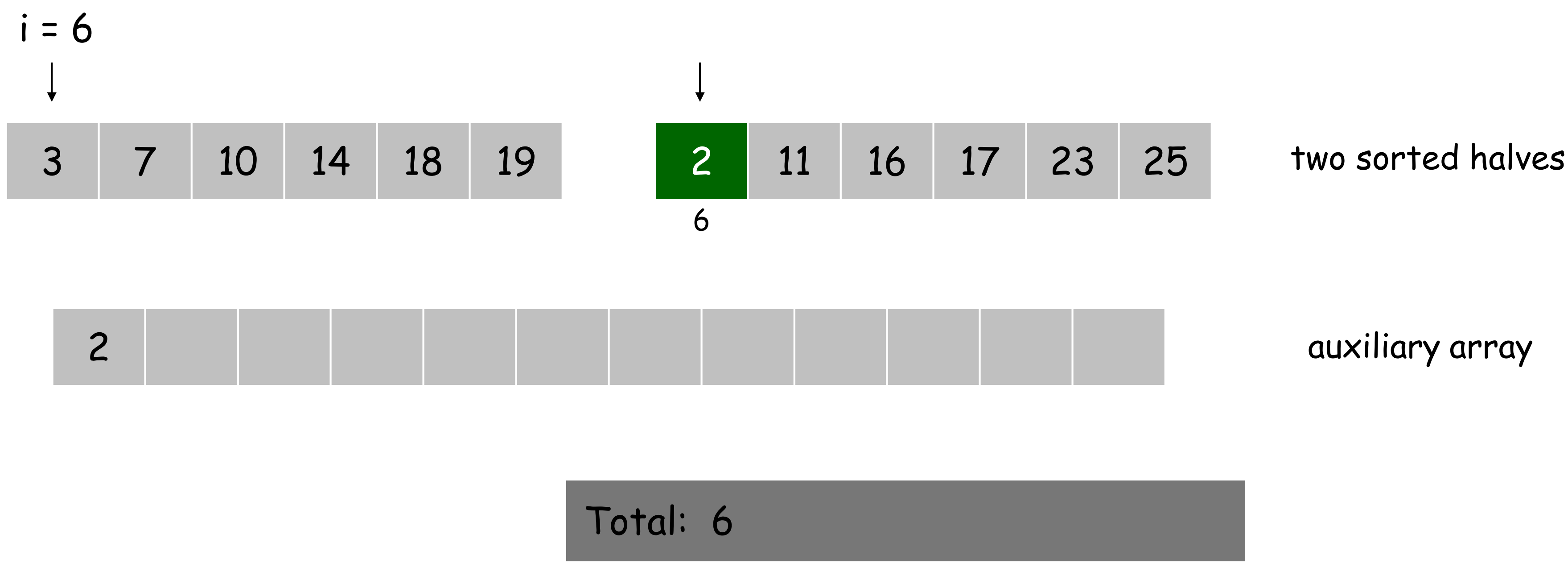
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

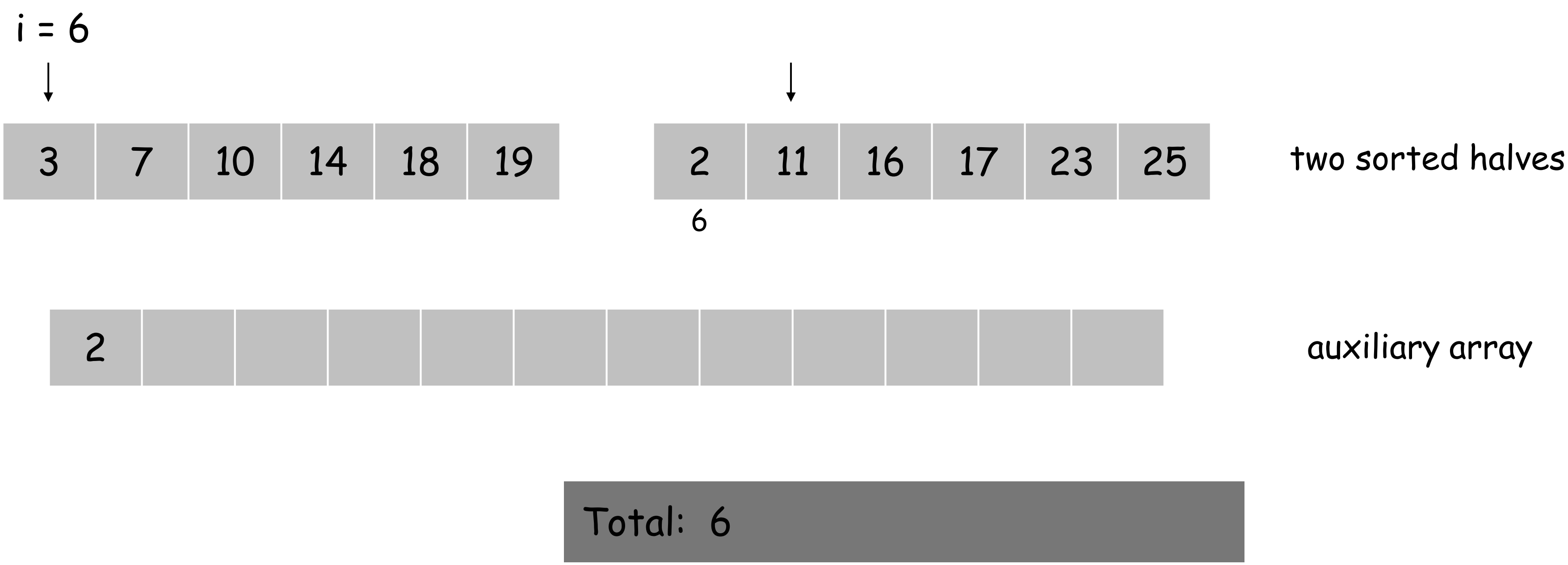
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

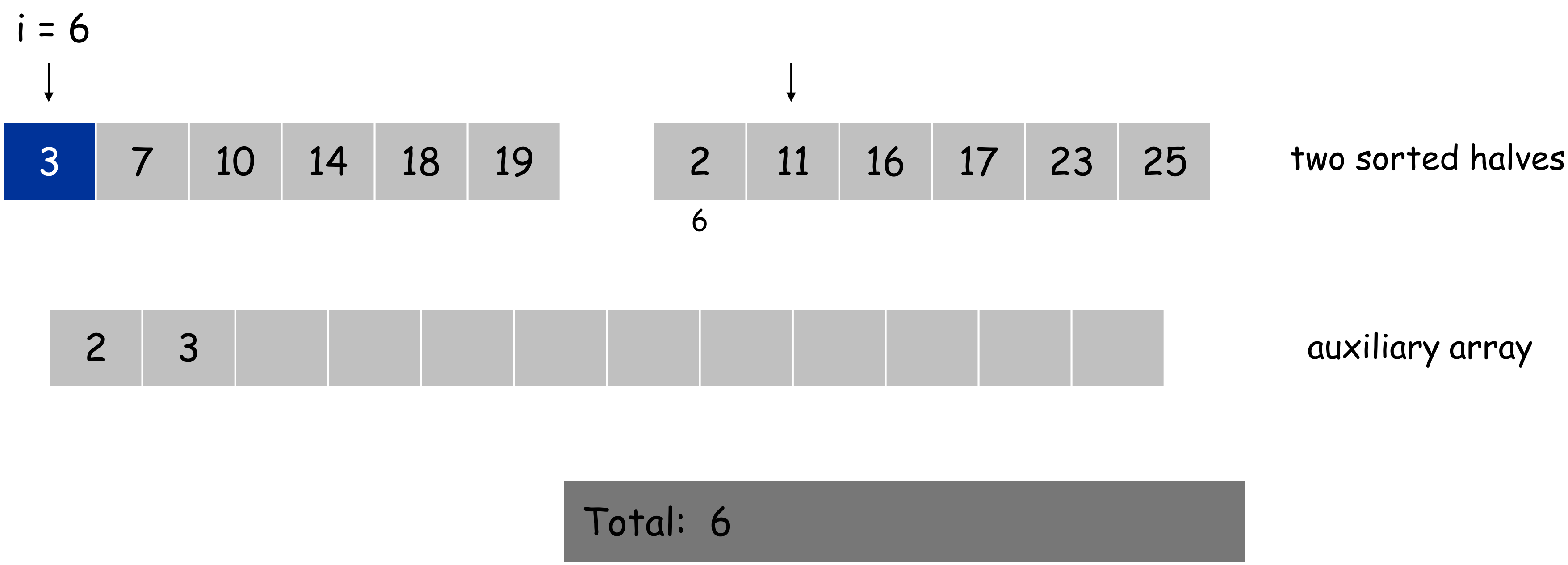
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

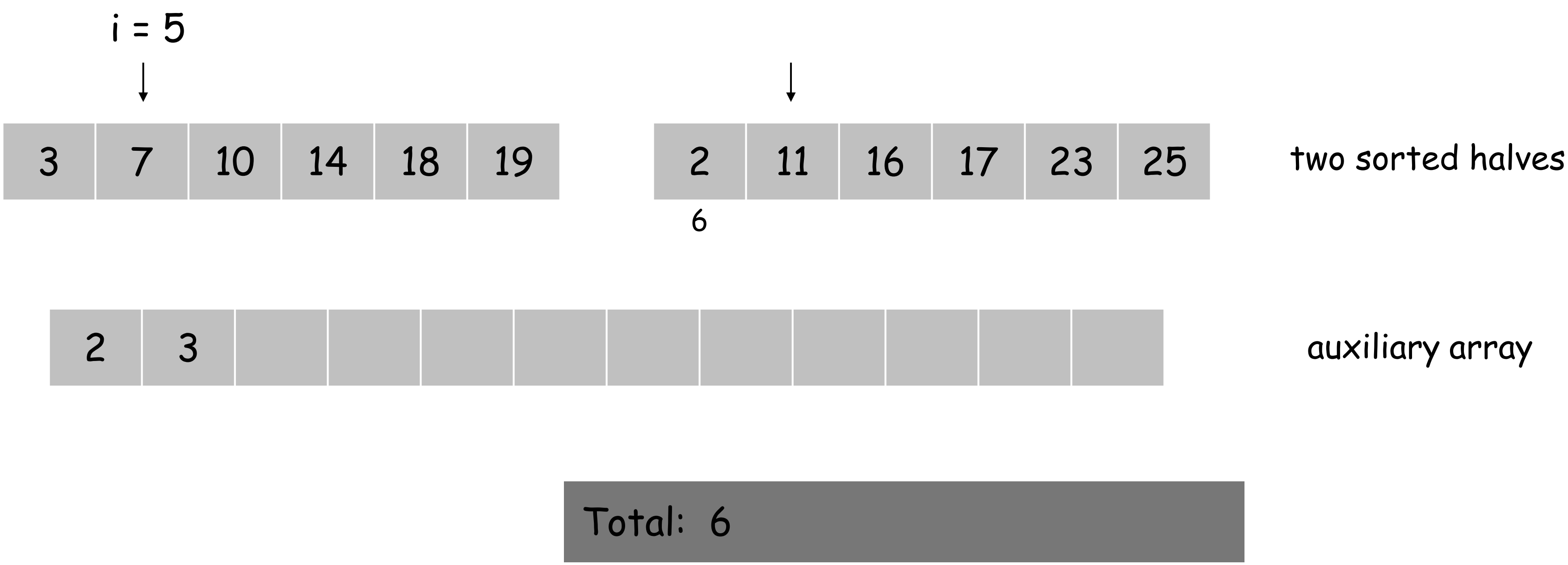
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

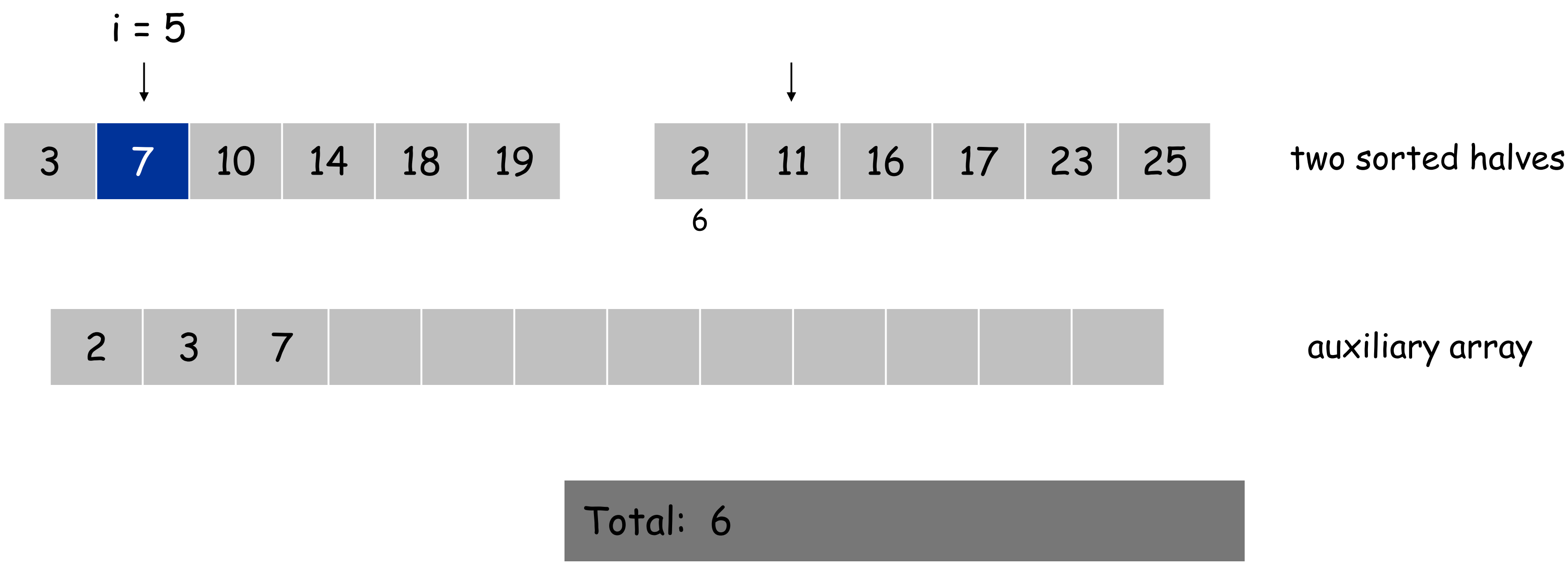
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

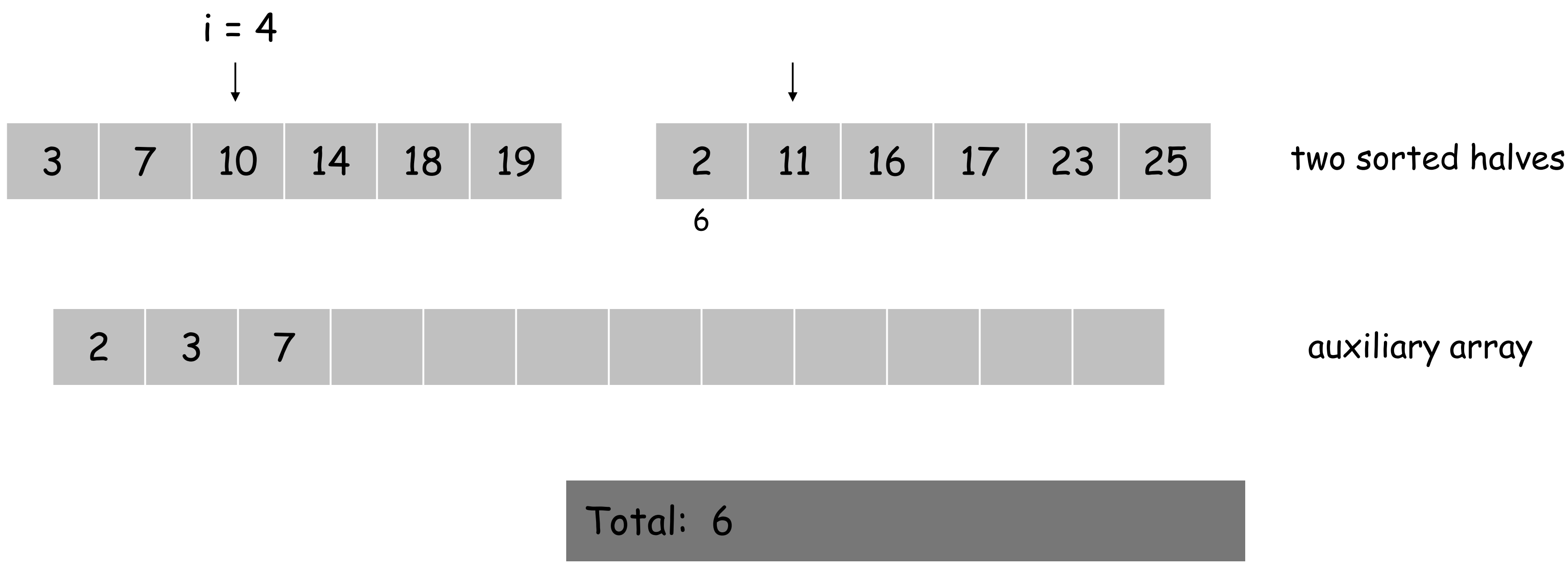
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

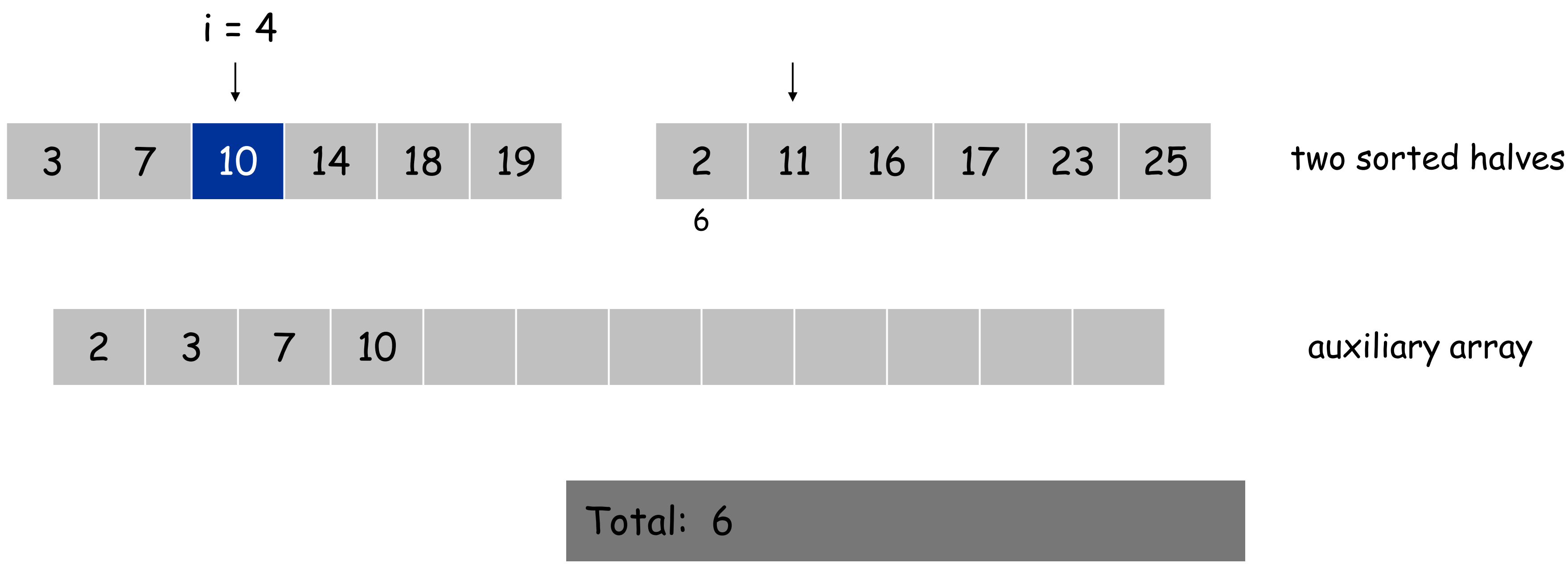
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

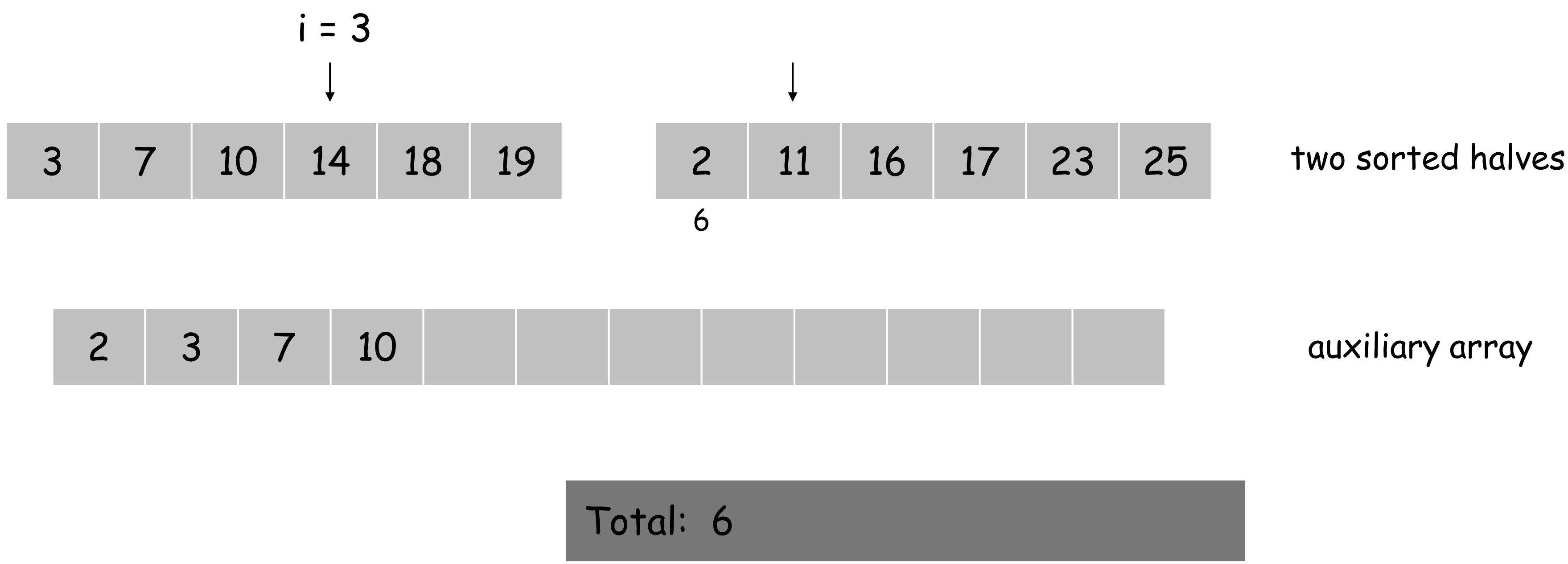
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

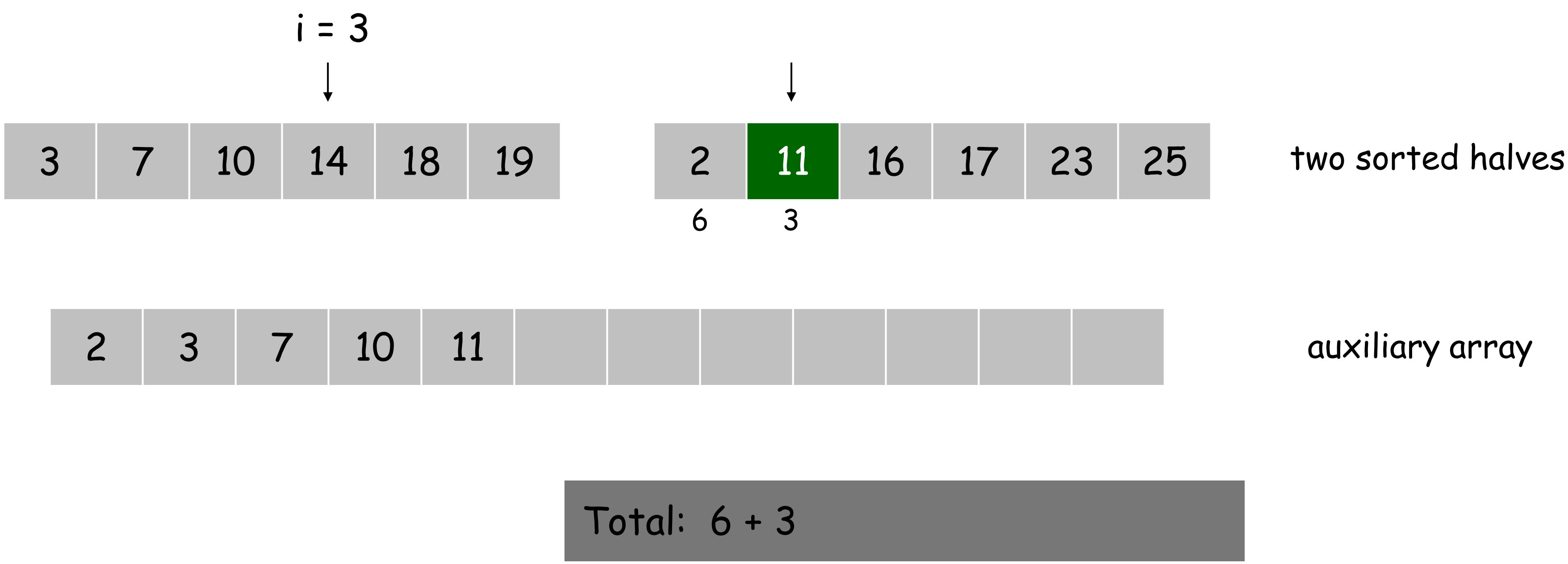
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

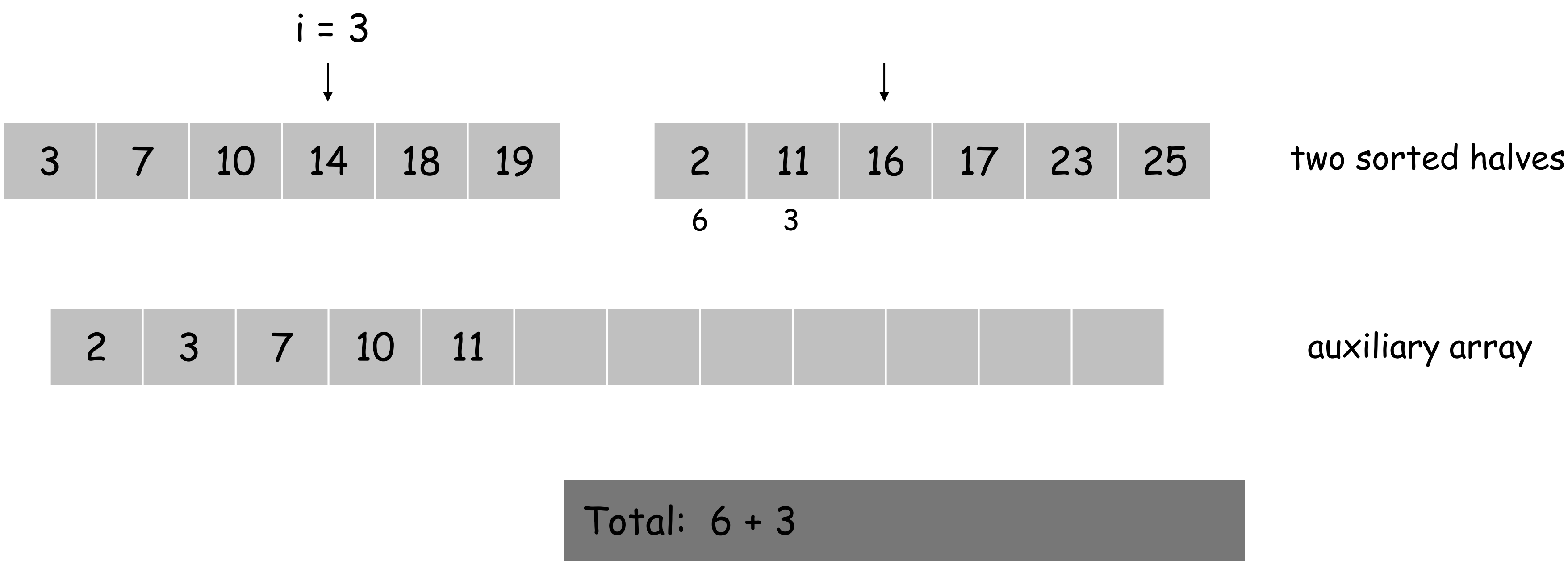
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

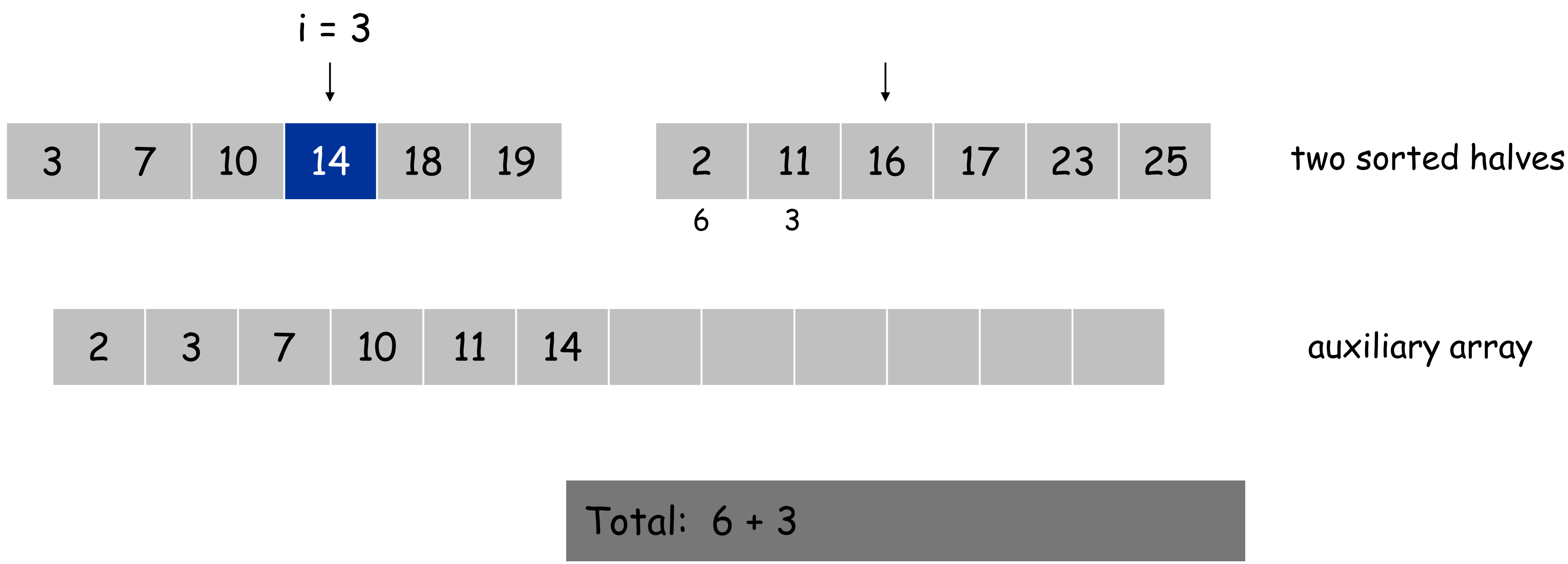
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

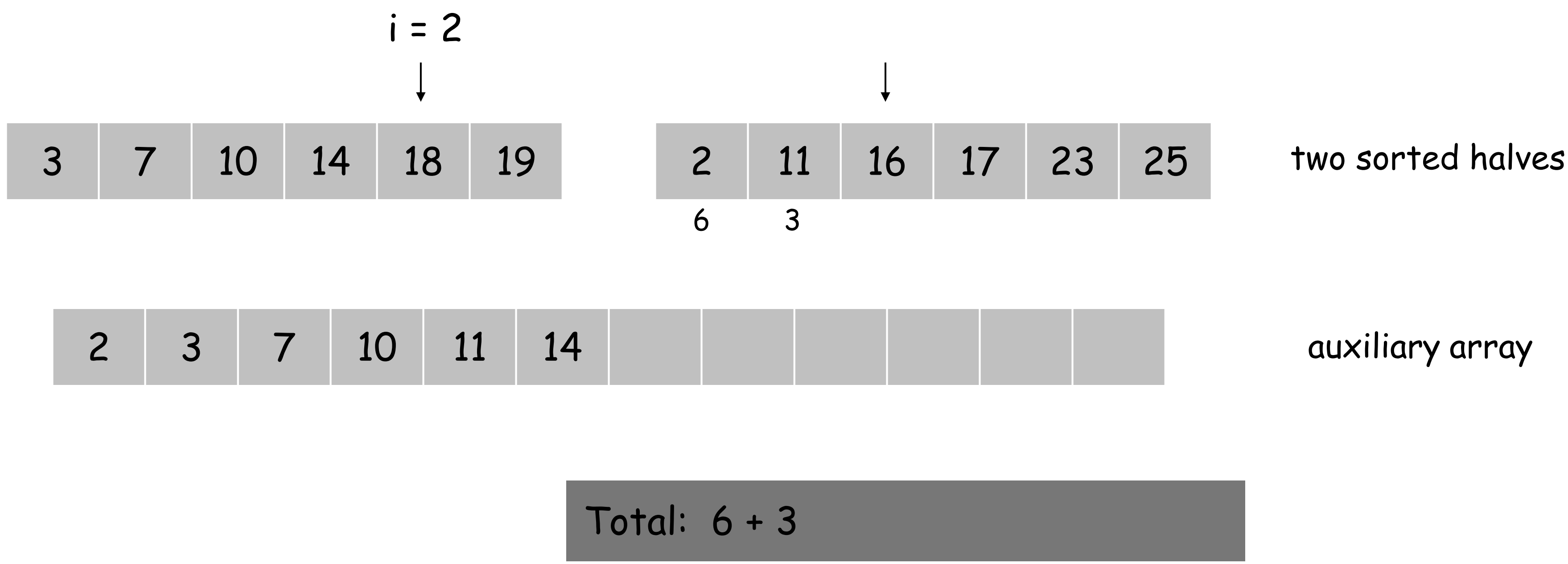
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

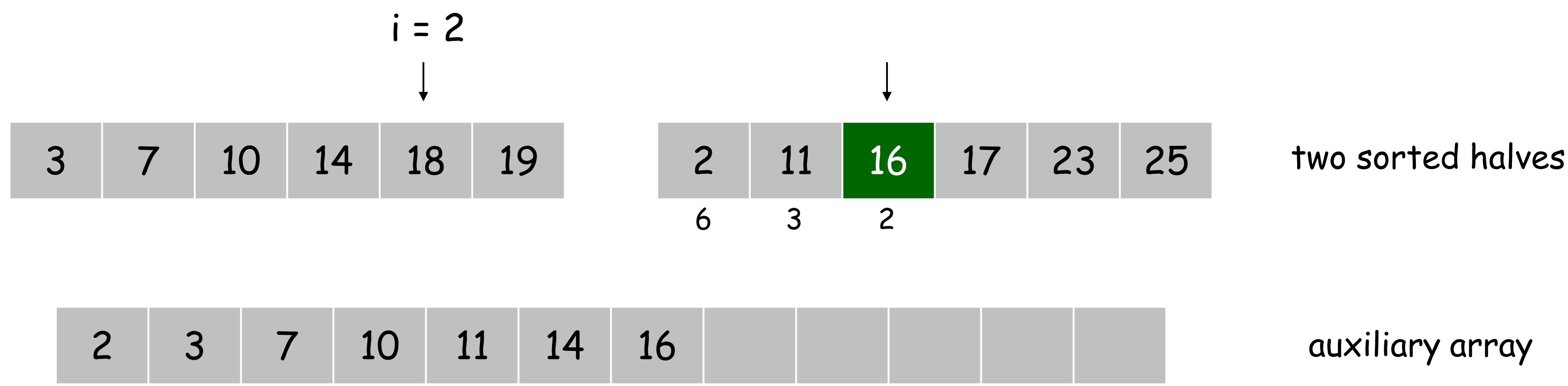
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

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

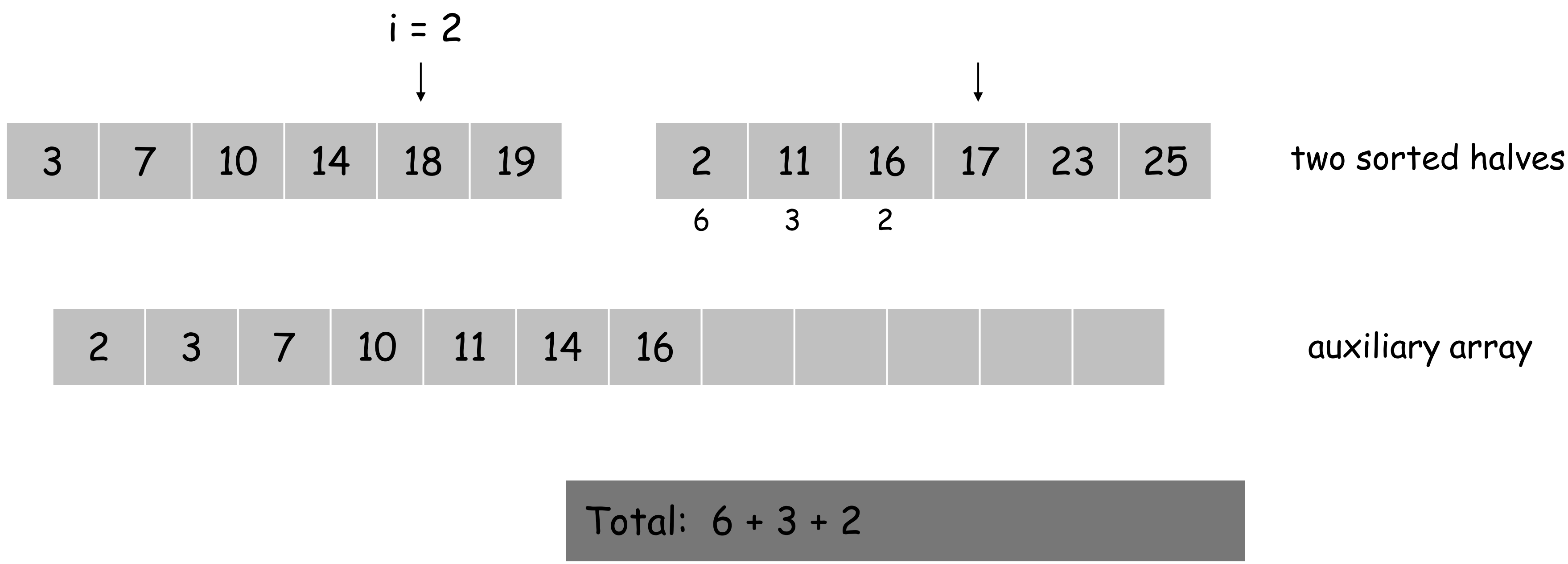


Total: 6 + 3 + 2

Merge and Count

Merge and count step.

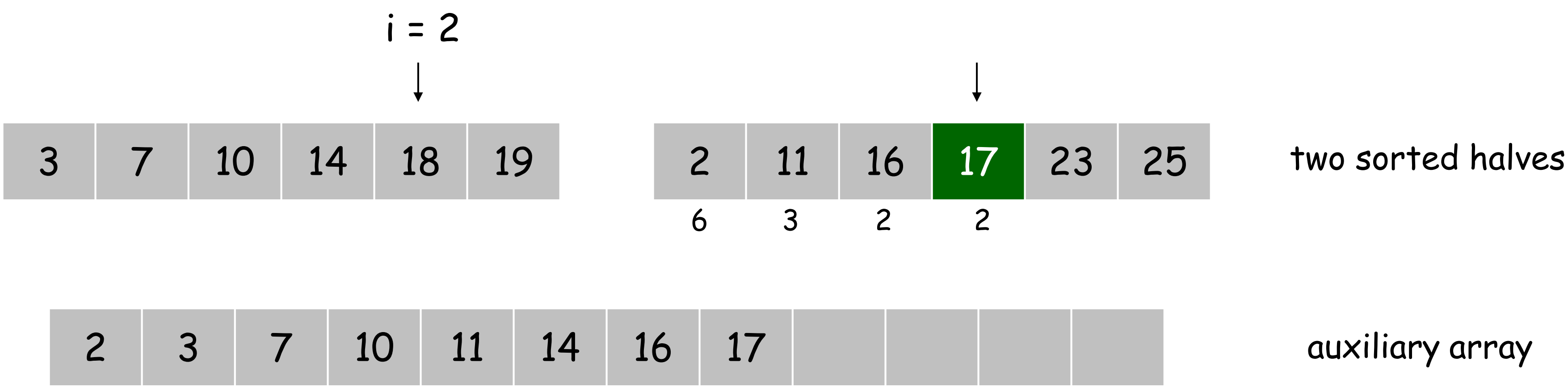
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

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

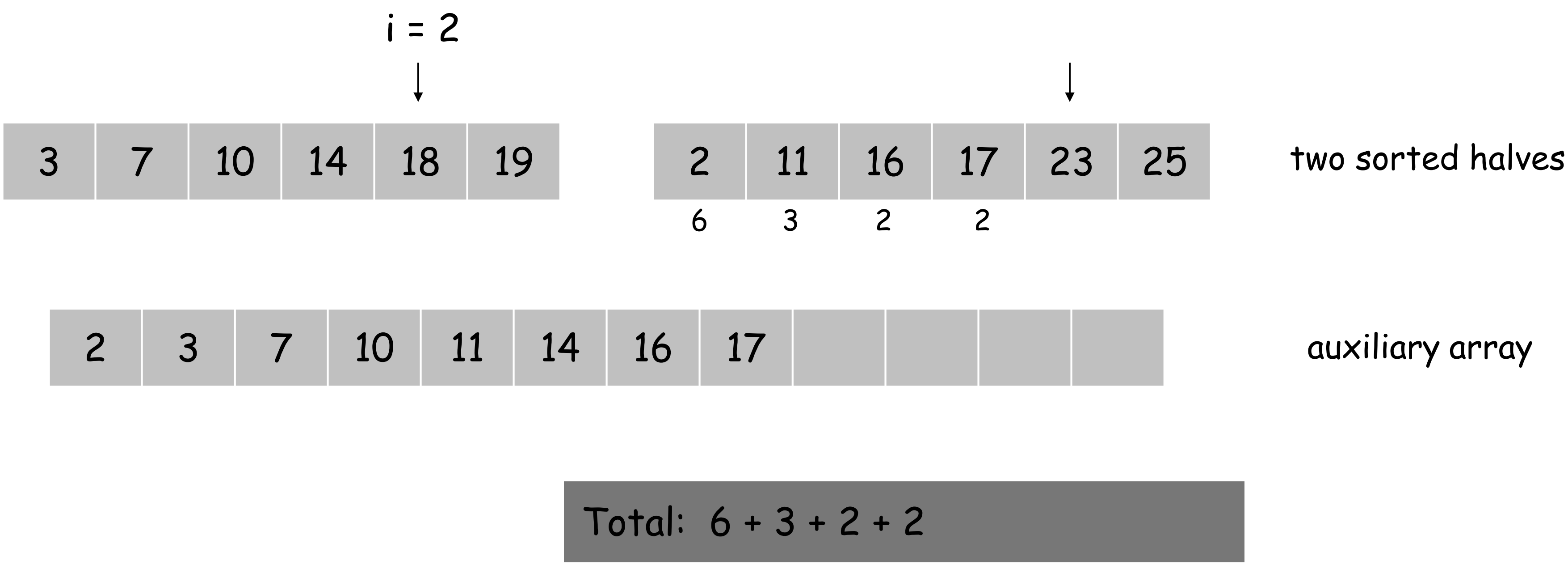


Total: 6 + 3 + 2 + 2

Merge and Count

Merge and count step.

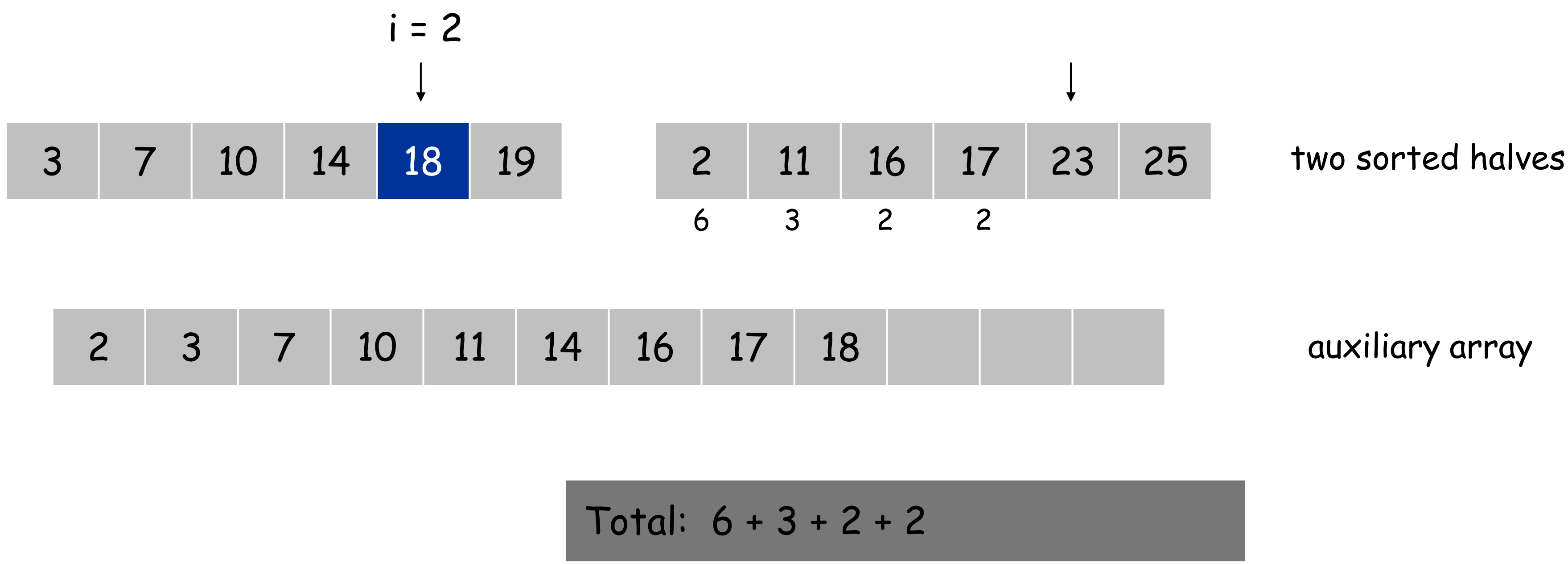
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

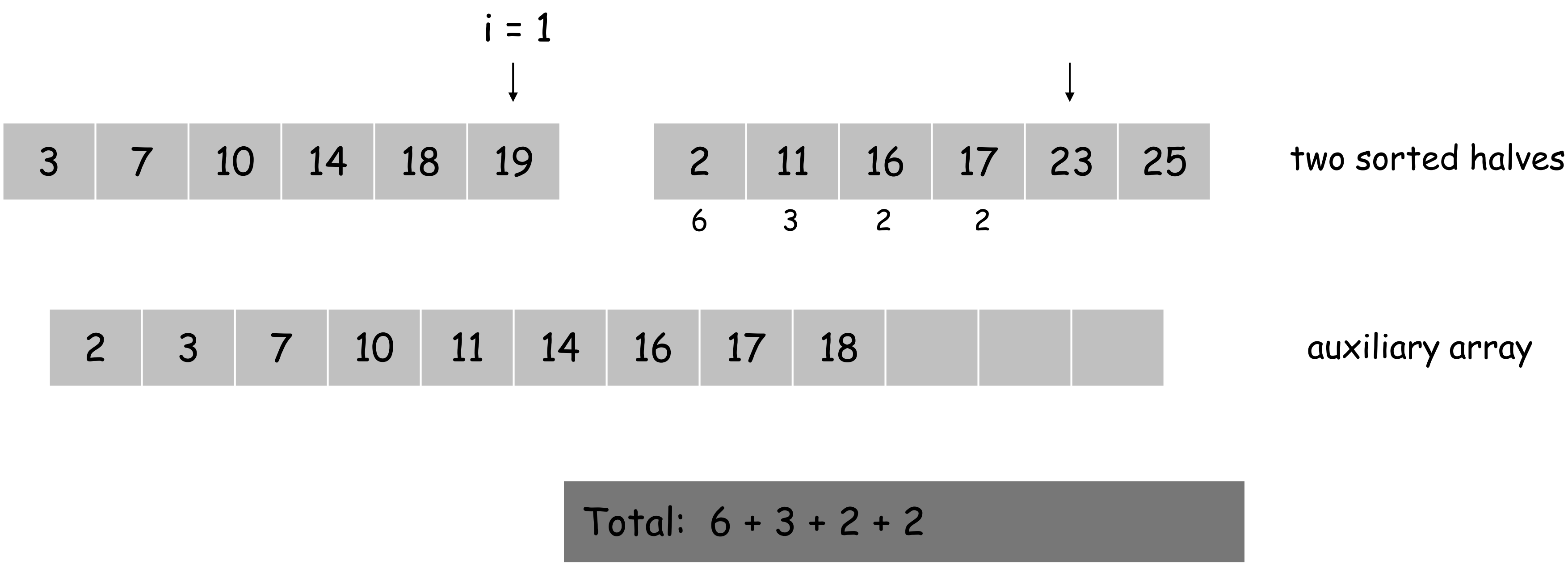
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j 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_j 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

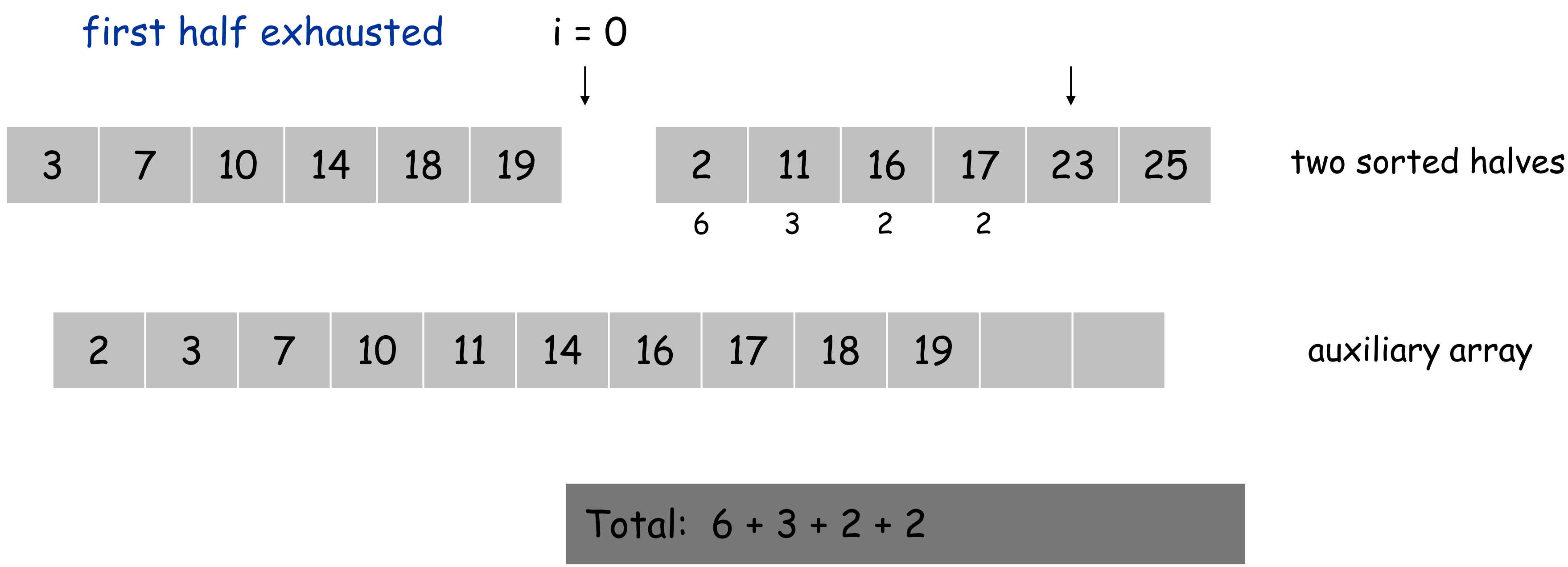
2	3	7	10	11	14	16	17	18	19		

Total: 6 + 3 + 2 + 2

Merge and Count

Merge and count step.

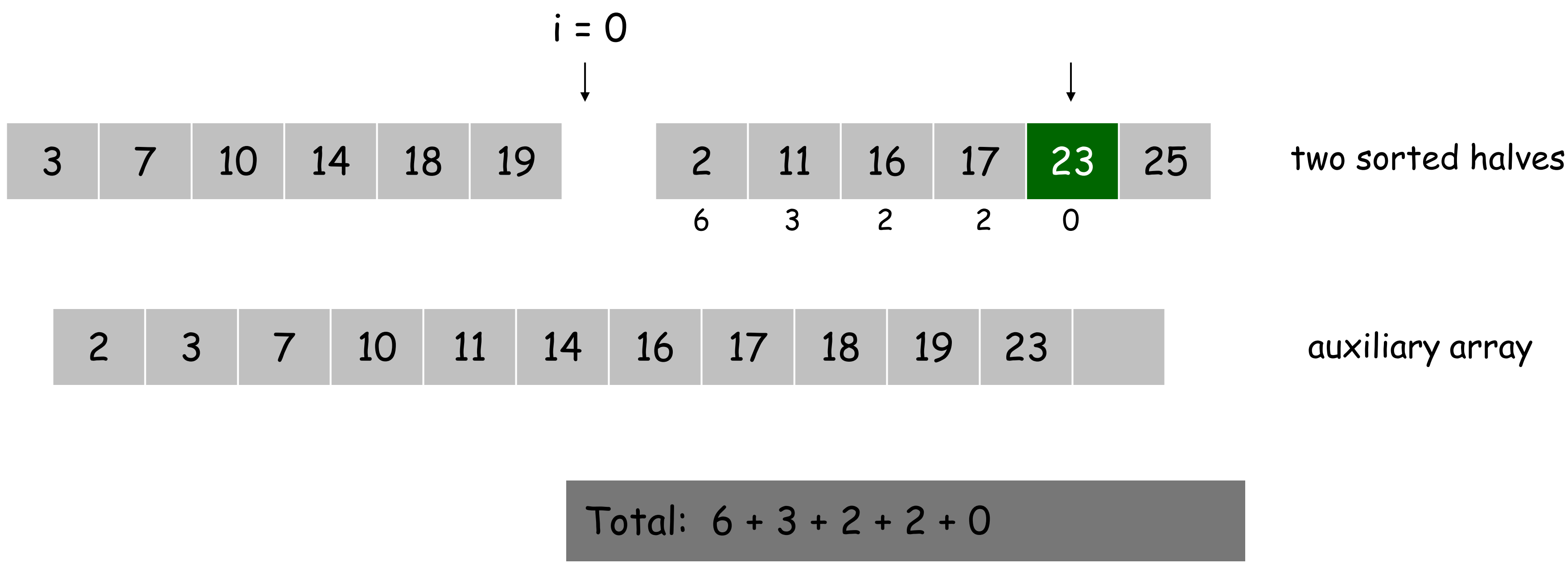
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

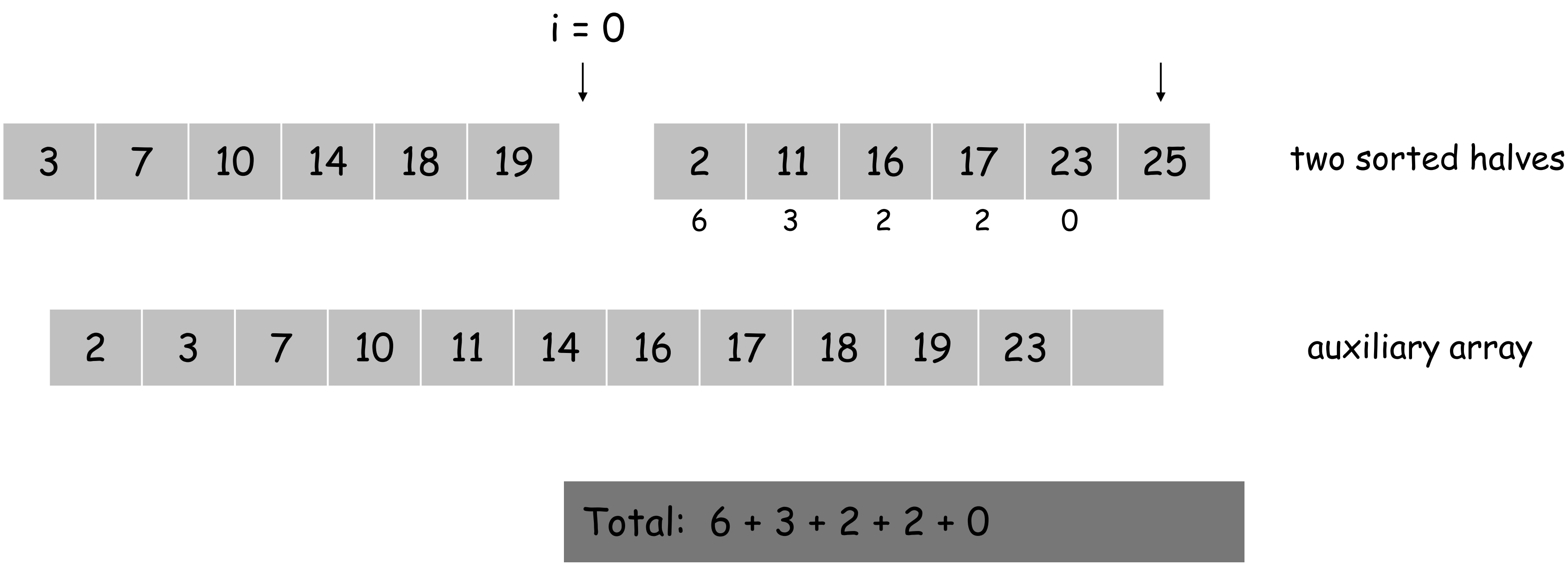
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

- Given two sorted halves, count number of inversions where a_i and a_j 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_j 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

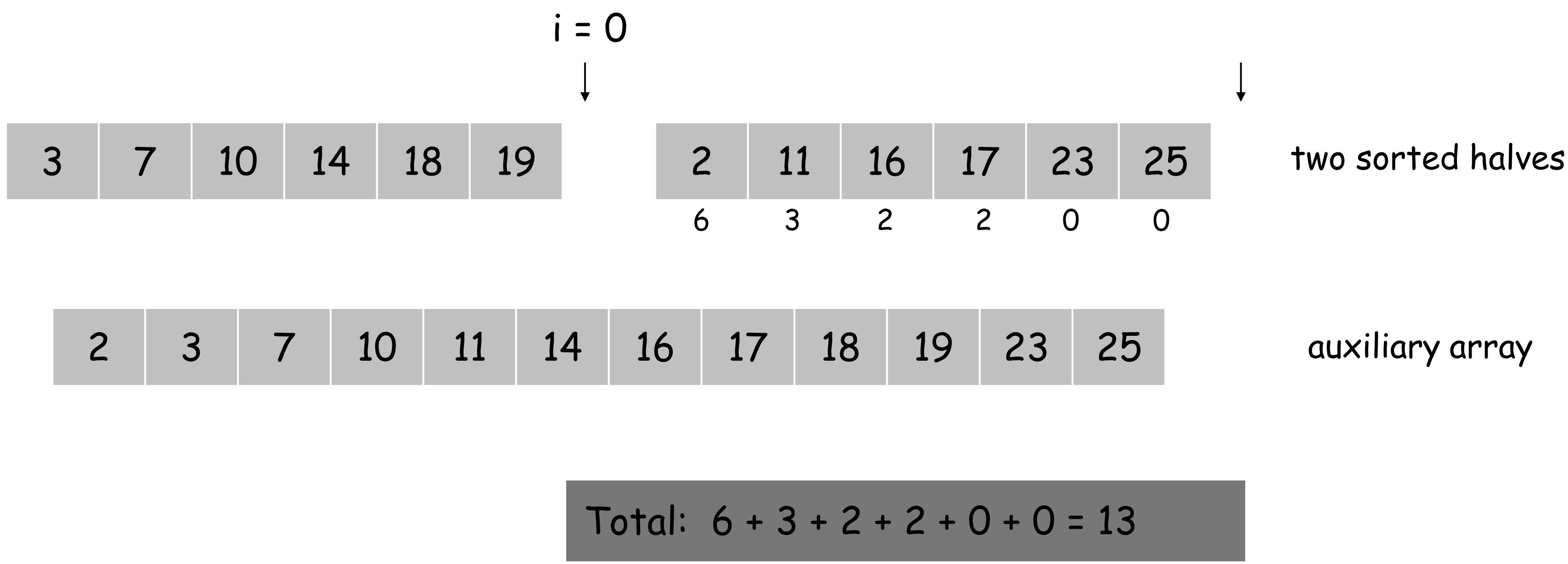
2	3	7	10	11	14	16	17	18	19	23	25

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

Merge and Count

Merge and count step.

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



Counting Inversions: Implementation

- Pre-condition of [Merge-and-Count]: A and B are sorted.
- Post-condition of [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)
   (r, L) ← Merge-and-Count(A, B)

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

D&C Performance Analysis

The running time of a Divide & Conquer algorithm is affected by 3 criteria:

1. The number of sub-instances (*a*) into which a problem is split.
2. The ratio of initial problem size to sub-problem size (*b*).
3. The number of steps required to divide the instance (D(n)), and to combine sub-solutions (C(n)).

General Recurrence for Divide-and-Conquer

- If a divide and conquer scheme divides a problem of size *n* into *a* sub-problems of size at most *n/b*. Suppose the time for Divide is D(*n*) and time for Combination is C(*n*), then

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < c \\ aT(n/b) + D(n) + C(n) & \text{if } n \geq c \end{cases}$$

- **How do we bound T(*n*)?**

The Master Theorem

- Let

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < c \\ aT(n/b) + f(n) & \text{if } n \geq c \end{cases}$$

where $a \geq 1$ and $b \geq 1$

- we will ignore ceilings and floors (all absorbed in the O or Θ notation)

The Master Theorem: A relaxed version for f(n)= $\Theta(n^k)$

- As special cases, when $f(n)= \Theta(n^k)$ we get the following:

- If $a>b^k$ then $T(n) = \Theta(n^{\log_b a})$
- If $a=b^k$ then $T(n) = \Theta(n^k \log n)$
- If $a<b^k$ then $T(n) = \Theta(n^k)$

The Master Theorem

Examples (in class)

1. $T(n) = T(2n/3) + 1$ then $T(n) = \Theta(\log n)$
2. $T(n) = 9T(n/3) + n$, then $T(n) = \Theta(n^2)$

More examples: Back to merge sort and binary search:

- MS: $T(n) = 2T(n/2) + cn$
- BS: $T(n) = T(n/2) + c$

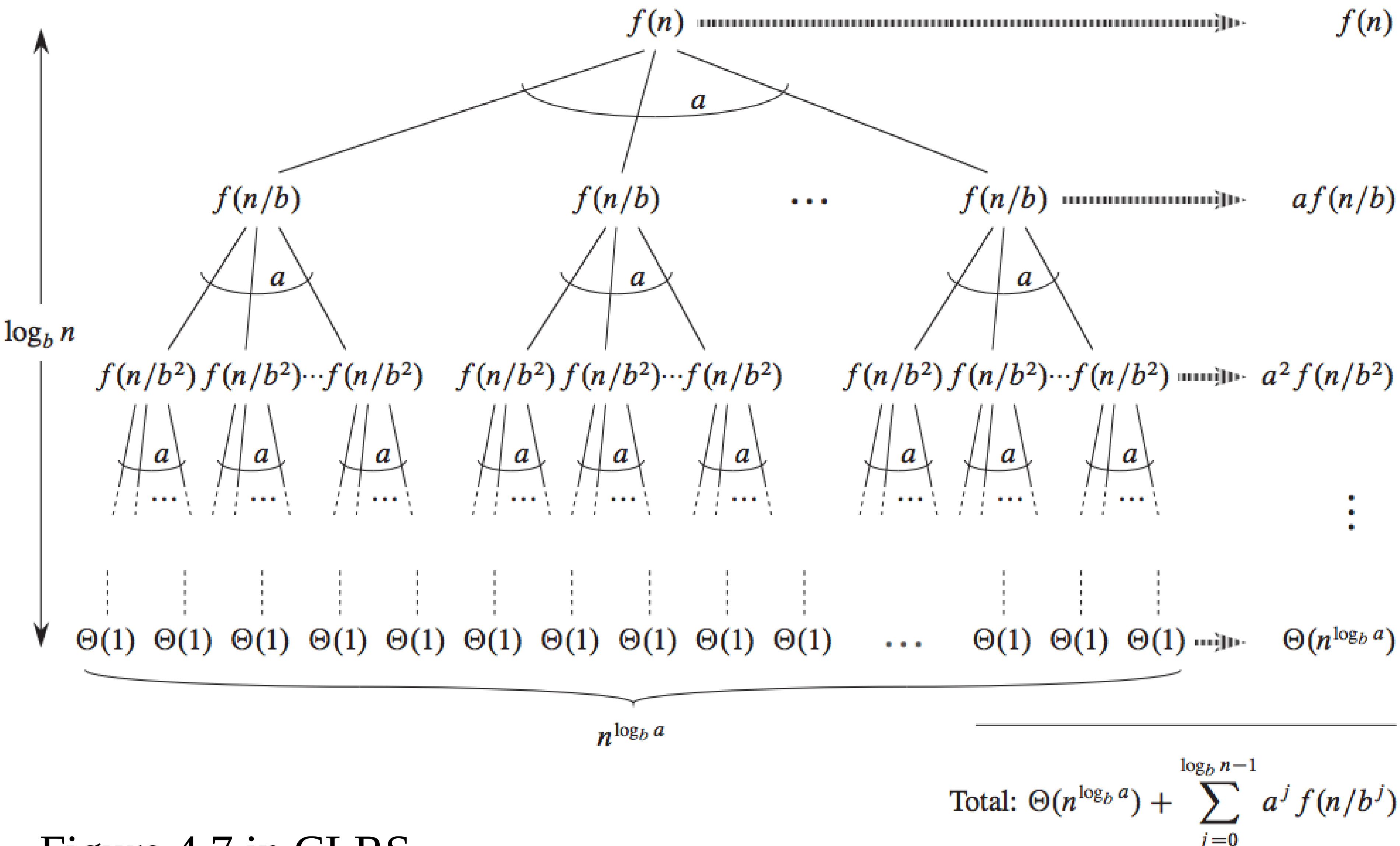


Figure 4.7 in CLRS

Counting.....

- num. of nodes at depth i is a^i
- depth of tree is $\log_b n$
- num. of leaves: $a^{\log_b n} = 2^{\log a \frac{\log n}{\log b}} = n^{\log_b a}$
- so $T(n) = \theta\left(n^{\log_b a}\right) + \sum_j a^j f(n/b^j)$

The Master Theorem (general f)

Intuition: Compare f(n) to $n^{\log_b a}$

- If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large *n*, then $T(n) = \Theta(f(n))$

Example 4: Integer Multiplication

- Used extensively in Cryptography:
  - public-key encryption/decryption uses multiplication of huge numbers

- Standard multiplication on two n-bit numbers takes $\Theta(n^2)$ operations.

- Note: standard addition takes O(n)

```
×             *************
              *************
         ___________________
              *************
             *************
            *************
           *************
          *************
         *************
        *************
       *************
      *************
     *************
    *************
   *************
```

Can We do Better?

- Divide and Conquer (assume X,Y given in binary)

X =	a	b
Y =	c	d

$$X = a2^{n/2} + b, \quad Y = c2^{n/2} + d$$

$$XY = ac2^{n} + (ad + bc)2^{n/2} + bd$$

- MULT(X,Y)
  - if |X| = |Y| = 1 then return XY
  - else return

$$MULT(a,c)2^{n} + \big(MULT(a,d) + MULT(b,c)\big)2^{n/2} + MULT(b,d)$$

Complexity

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 4T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- By the Master Theorem: $T(n) = \Theta(n^2)$
- Not an improvement over standard multiplication.

Can we do better? (Karatsuba 1962)

- Gauss Equation

$$ad + bc = (a+b)(c+d) - ac - bd$$

- MULT(X,Y)
  - **if** $|X| = |Y| = 1$ **then return** XY
  - **else**
    - A_1 = MULT(a,c);
    - A_2 = MULT(b,d);
    - A_3 = MULT((a+b),(c+d));
    - Return $A_1 2^n + (A_3 - A_1 - A_2) 2^{n/2} + A_2$

Recall:

$$XY = ac2^n + (ad + bc)2^{n/2} + bd$$

Complexity

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- By the Master Theorem:

$$T(n) = \Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1.58}\right)$$

Example 5: Matrix Multiplication

- Dot product: Given two length n vectors a and b, compute $c = a \cdot b$.

$$a \cdot b = \sum_{i=1}^{n} a_i \, b_i$$

- Grade-school: $\Theta(n)$ arithmetic operations.

$$a = [.70 \quad .20 \quad .10]$$
$$b = [.30 \quad .40 \quad .30]$$
$$a \cdot b = (.70 \times .30) + (.20 \times .40) + (.10 \times .30) = .32$$

- Remark: Grade-school dot product algorithm is optimal.

Section 28.2 (or 4.2) in CLRS

Matrix Multiplication

- Given two n-by-n matrices *A* and *B*, compute *C = AB*.
- Grade-school: $\Theta(n^3)$ arithmetic operations.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

- Q. Is grade-school matrix multiplication algorithm optimal?

Block Matrix Multiplication

$$\underbrace{\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix}}_{C_{11}\text{: top-left } 2\times 2} = \underbrace{\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}}_{A_{11}\text{, } A_{12}\text{: top-left, top-right } 2\times 2} \times \underbrace{\begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}}_{B_{11}\text{, } B_{21}\text{: top-left, bottom-left } 2\times 2}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Matrix Multiplication: Warmup

- To multiply two n-by-n matrices A and B:
  - Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
  - Conquer: multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
  - Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Fast Matrix Multiplication

- Key idea. multiply 2-by-2 blocks with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

  - 7 multiplications, 14 2-by-2 elements.
  - $18 = 8 + 10$ additions and subtractions.

Fast Matrix Multiplication

- To multiply two n-by-n matrices A and B: [Strassen 1969]
  - Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
  - Compute: 14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via 10 matrix additions.
  - Conquer: multiply 7 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
  - Combine: 7 products into 4 terms using 8 matrix additions.
- Analysis.
  - Assume n is a power of 2.
  - $T(n)$ = # arithmetic operations.

$$T(n) = \underbrace{7\,T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \quad \Rightarrow \quad T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication: Practice

- Implementation issues.
  - Sparsity.
  - Caching effects.
  - Numerical stability.
  - Odd matrix dimensions.
  - Crossover to classical algorithm around $n = 128$.

- Common misperception. *"Strassen is only a theoretical curiosity."*
  - Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2{,}500$.
  - Range of instances where it's useful is a subject of controversy.

Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar mult?

A. Yes! [Strassen 1969] $\Theta(n^{\log_2 7}) = O(n^{2.807})$

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$

Q. Two 3-by-3 matrices with 21 scalar multiplications?

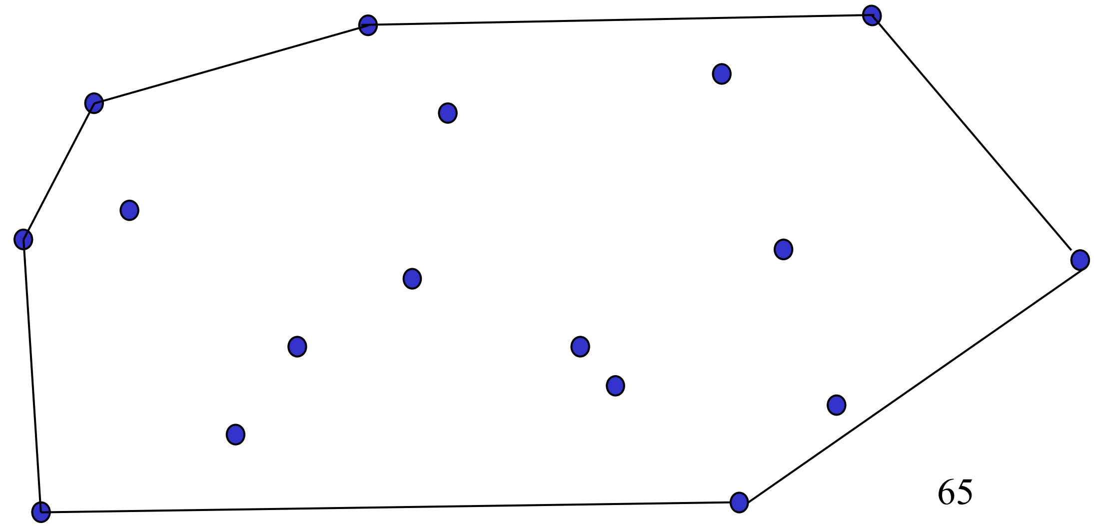
A. Also impossible. $\Theta(n^{\log_3 21}) = O(n^{2.77})$

- Two 20-by-20 matrices with 4,460 scalar mult. $O(n^{2.805})$
- Two 48-by-48 matrices with 47,217 scalar mult. $O(n^{2.7801})$
  A year later. $O(n^{2.7799})$
- December, 1979. $O(n^{2.521813})$
- January, 1980. $O(n^{2.521801})$
- Record holder 1987-2010: $O(n^{2.376})$ [Coppersmith-Winograd, 1987].
- Best Known: $O(n^{2.373})$ [Vassilevska Williams, 2011]
- Conjecture: $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Example 6: Finding the Convex Hull of a set of points (2-dim).

- Given a set A of n points in the plane, the convex hull of A is the smallest convex polygon that contains all the points in A.
- For simplicity, assume no two points have the same x or y coordinate. (otherwise rotate a bit..)
- The output: set of CH vertices in clockwise order.

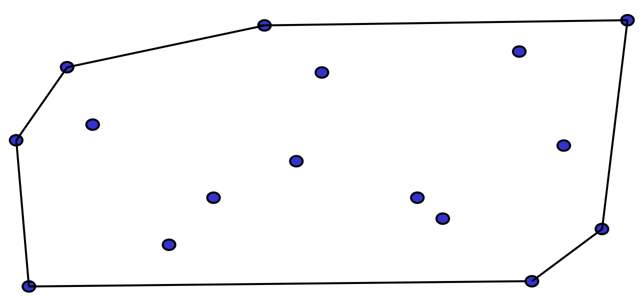
A set S of points is convex if for any two points $x,y \in S$, any point on the line connecting x and y is also in S.



Example 6: Finding the Convex Hull of a set of points.

Intuition:

- Each point is a nail sticking out from a board.
- Take a rubber band and lower it over the nails, so as to completely encompass the set of nails.
- Let the rubber band naturally contract.
- The rubber band gives the edges of the convex hull of the set of points.
- Nails corresponding to a change in slope of the rubberband represent the extreme points of the convex hull.



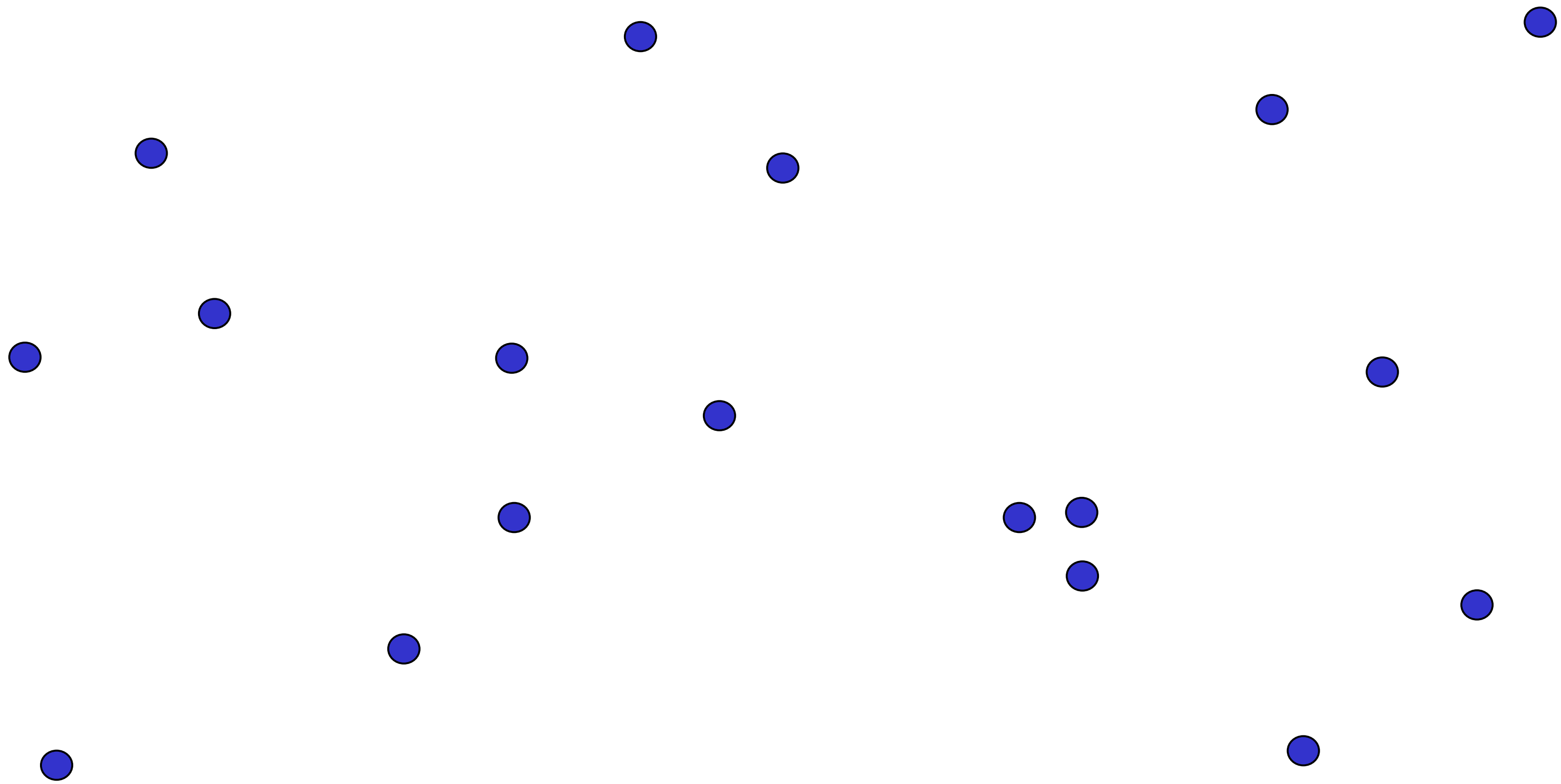
Convex Hull - D&C algorithm.

Let $A = \{p_1, p_2, \ldots, p_n\}$. Denote the convex hull of A by $CH(A)$.

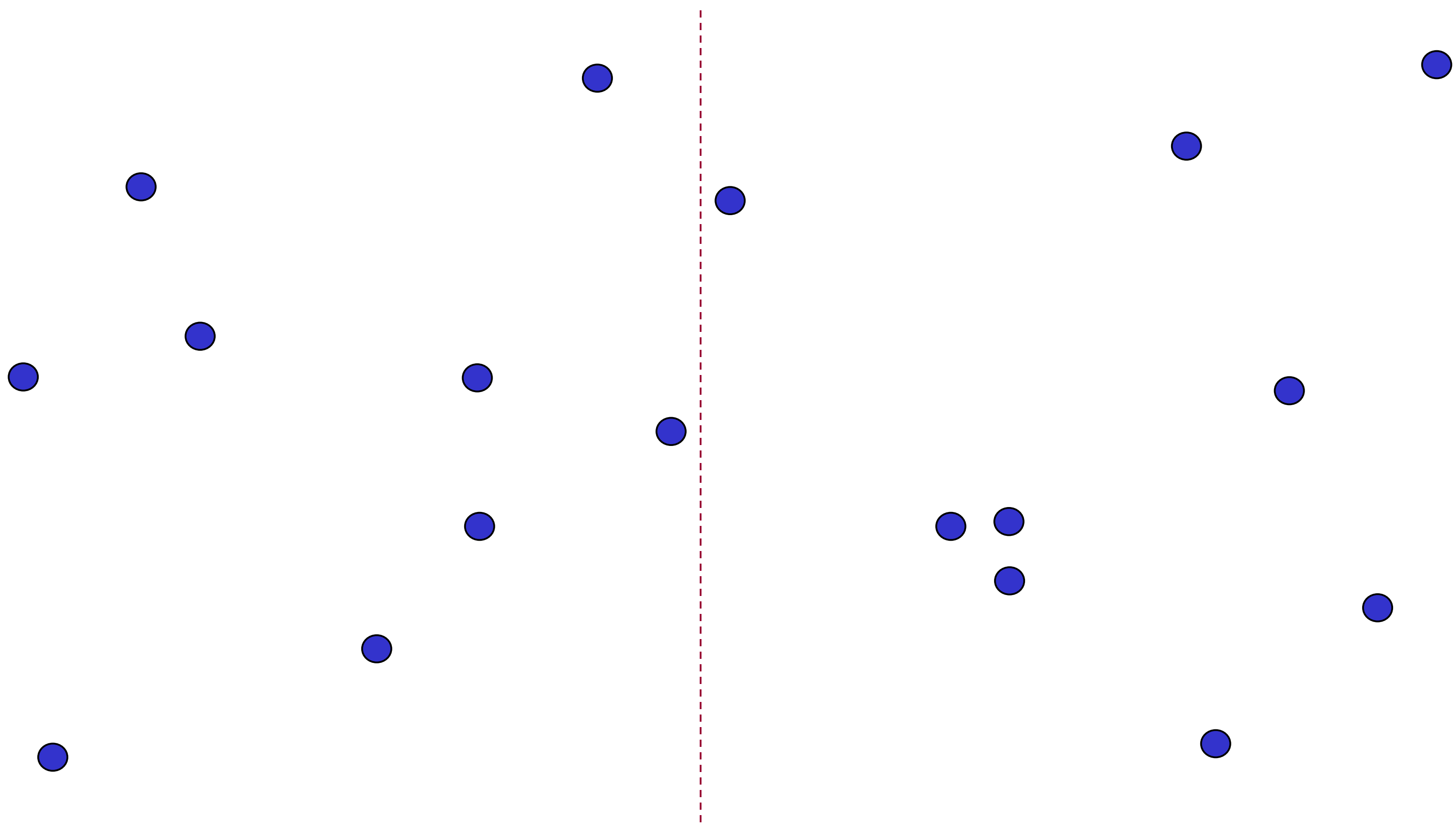
1. Sort the points of A by x-coordinate.
2. If $n \leq 3$, solve the problem directly. Otherwise, apply divide-and-conquer as follows.
3. Divide A into two subsets: $A = L \cup R$.
4. Find $CH(L)$, the convex hull of L.
5. Find $CH(R)$, the convex hull of R.
6. Combine the two convex hulls.

Convex Hull - Divide & Conquer

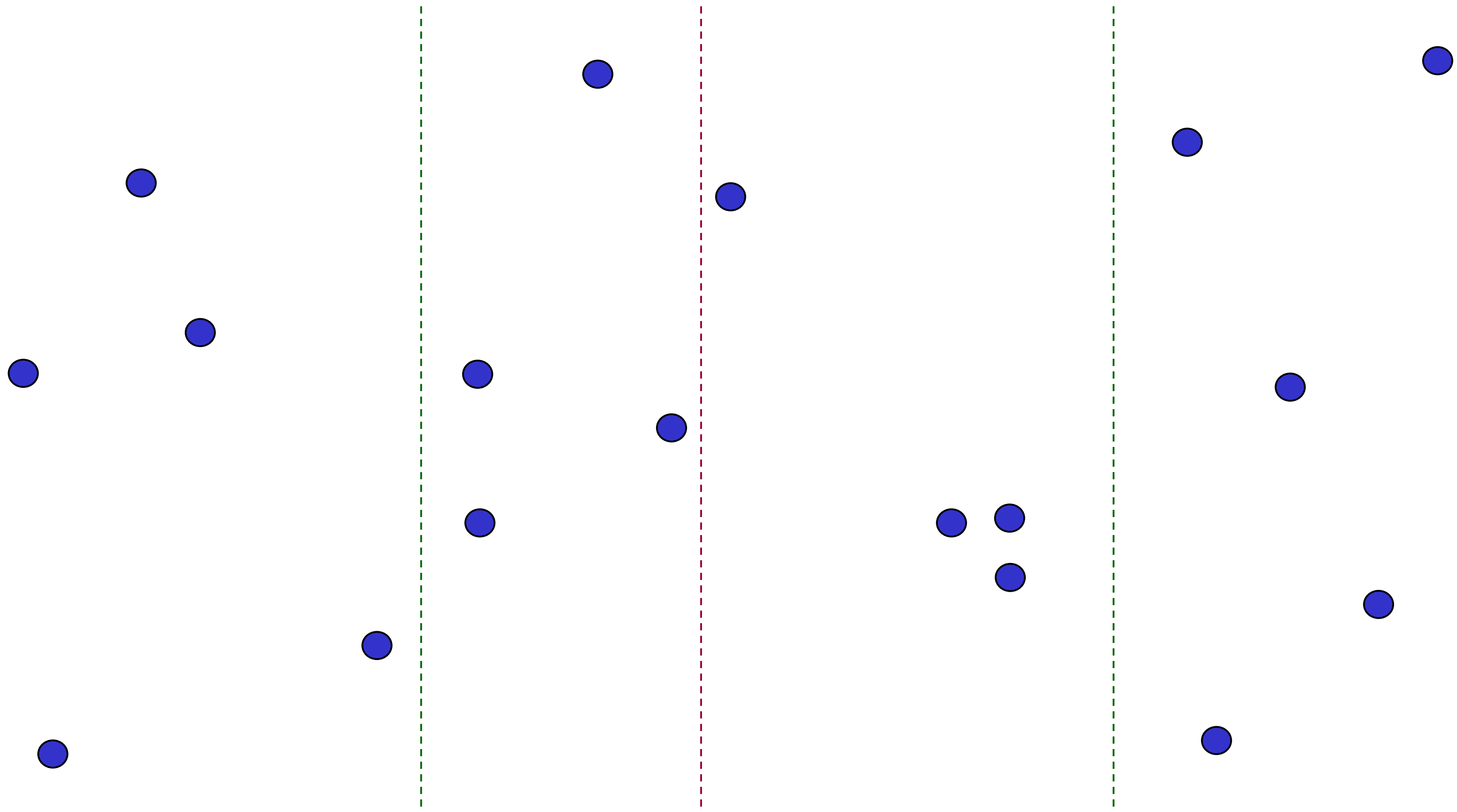
Split set into two, compute convex hull of both, combine.



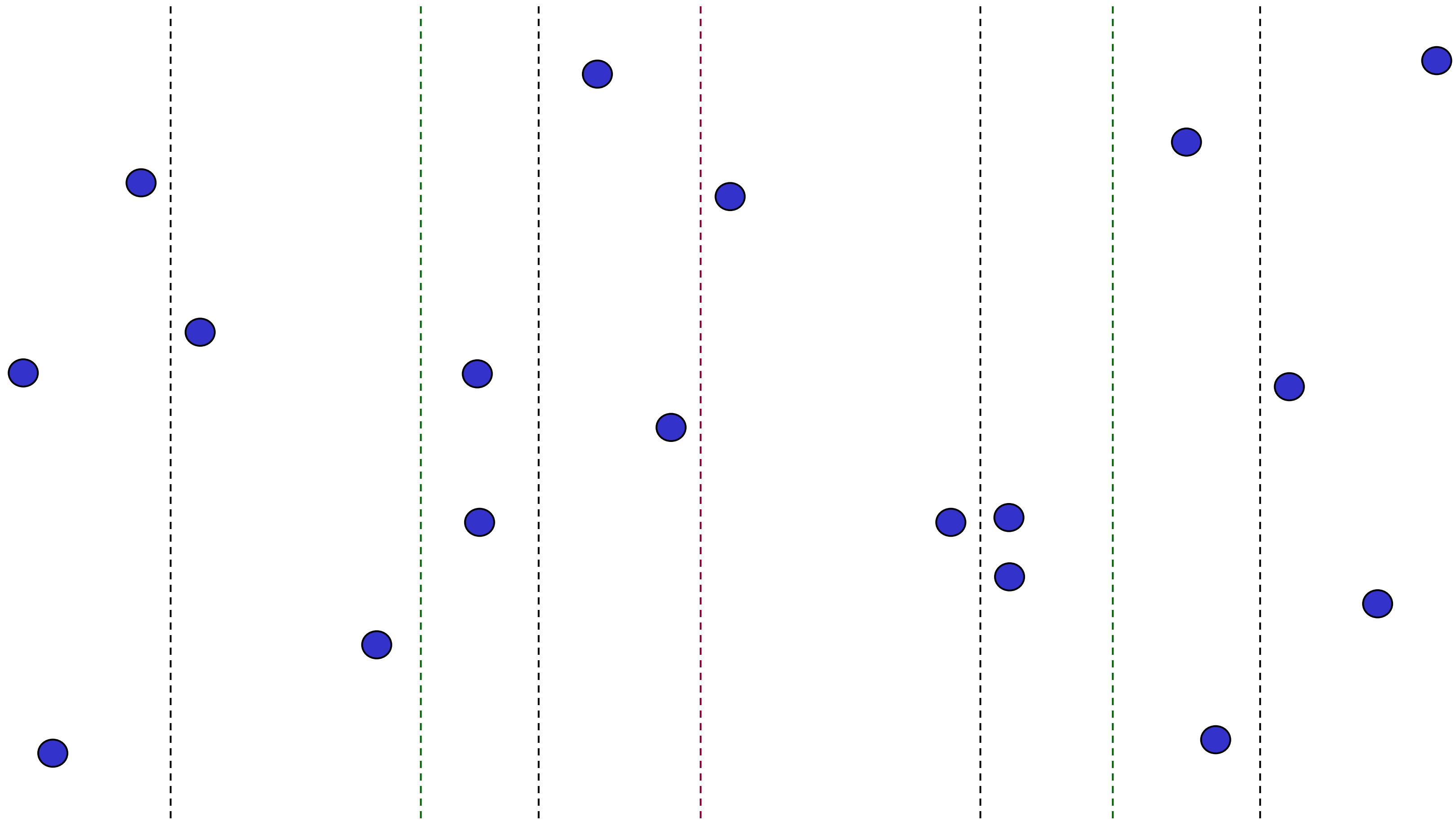
Convex Hull - Divide & Conquer



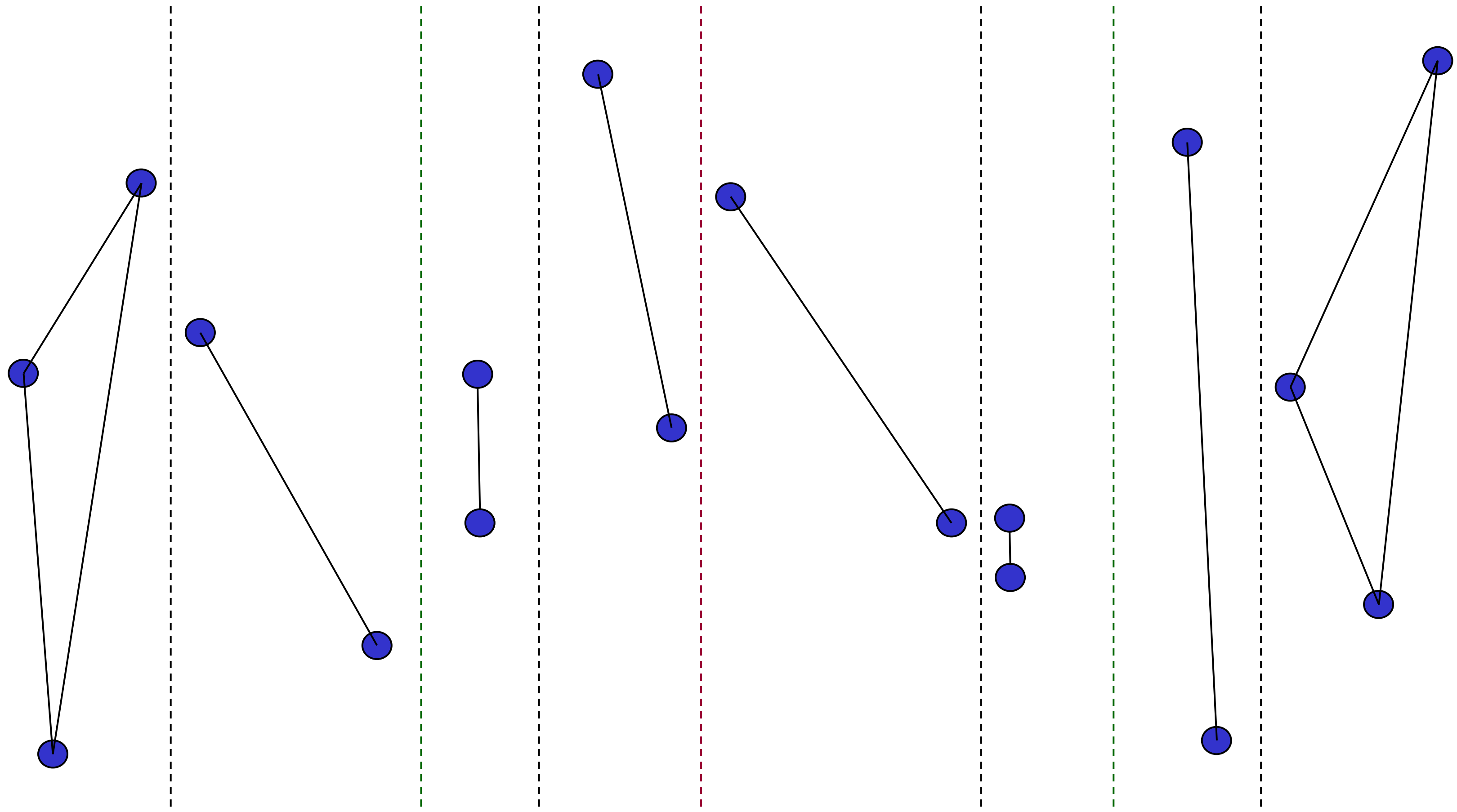
Convex Hull - Divide & Conquer



Convex Hull - Divide & Conquer

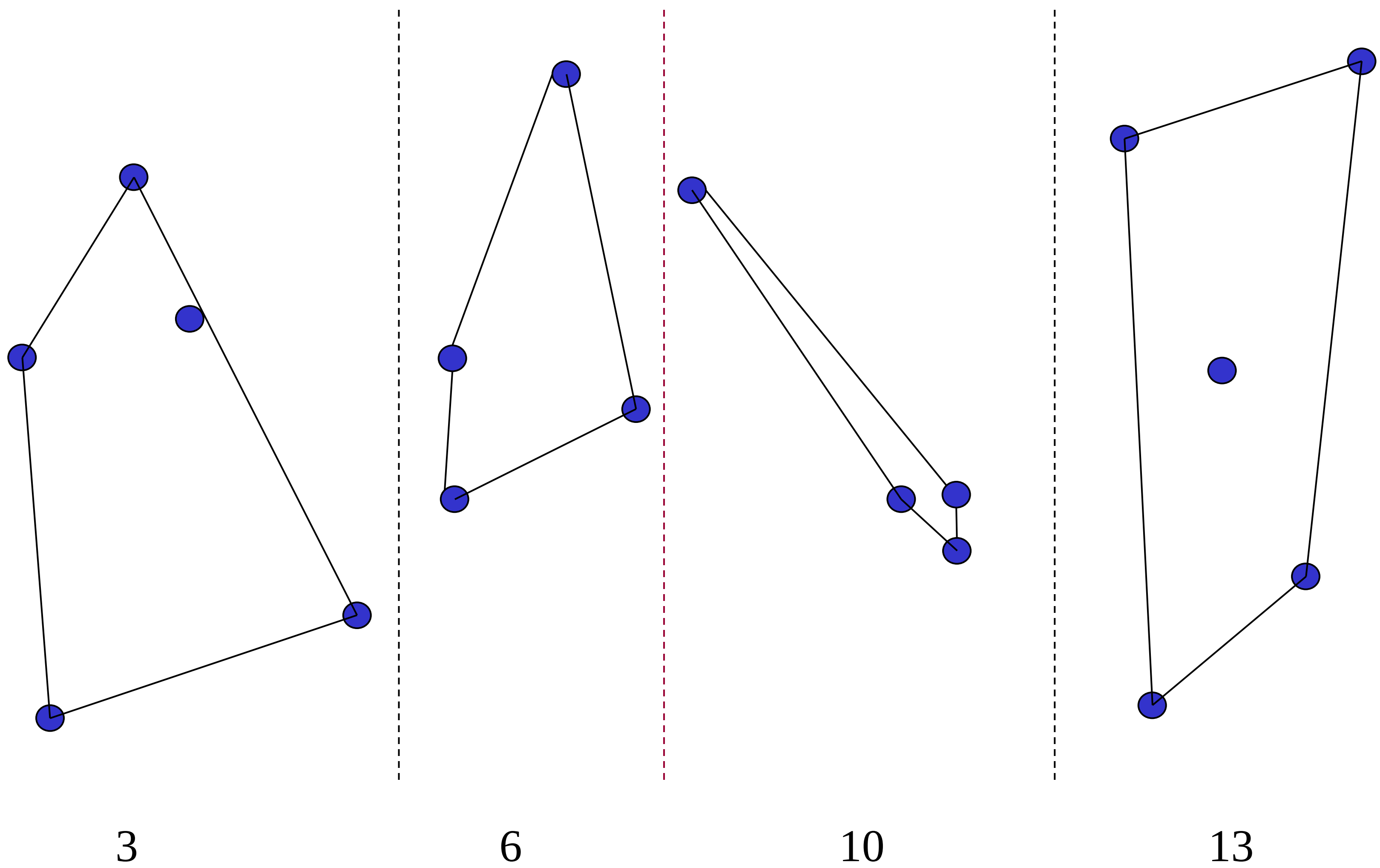


Convex Hull - Divide & Conquer

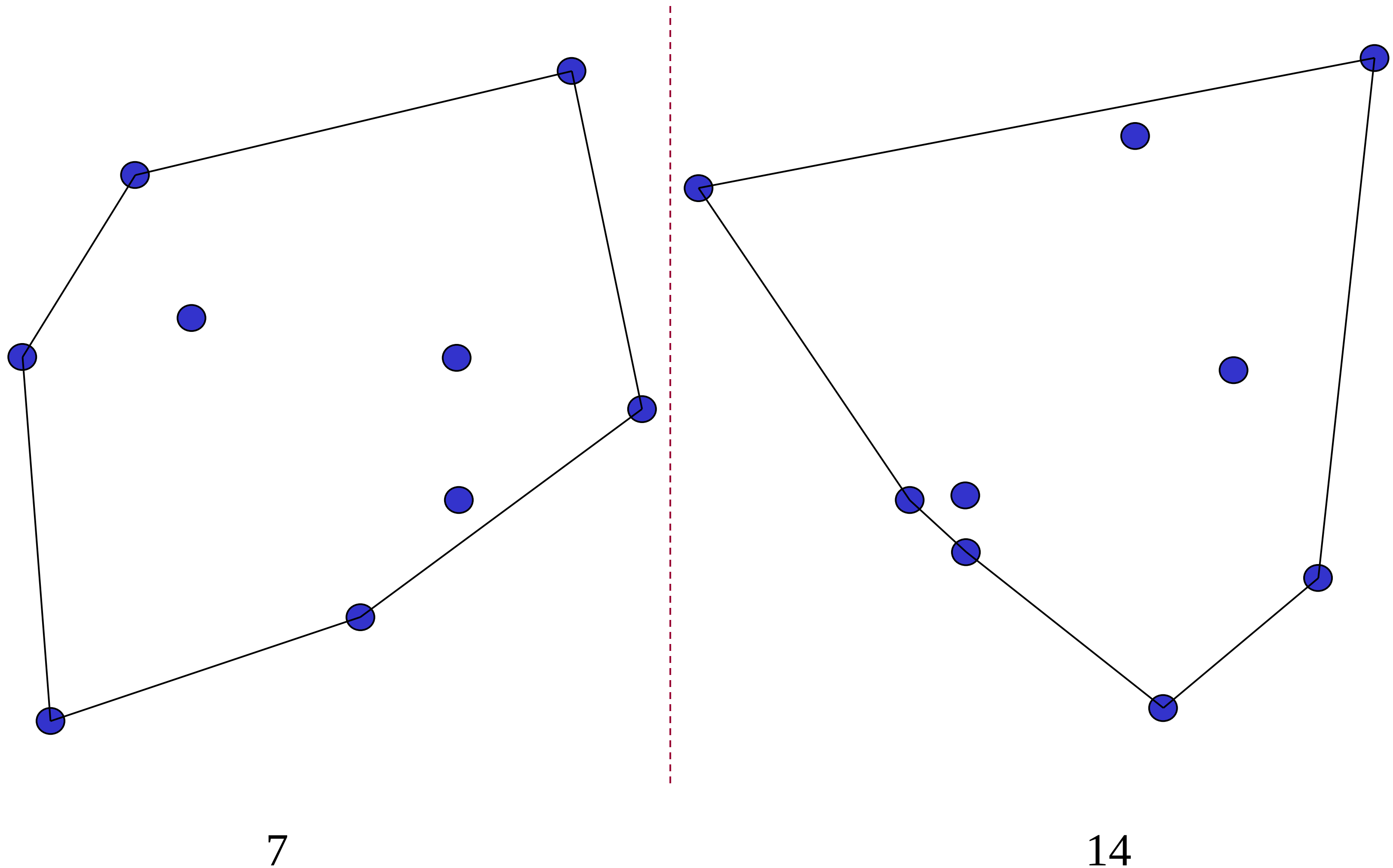


Solution order: 1 2 4 5 8 9 11 12

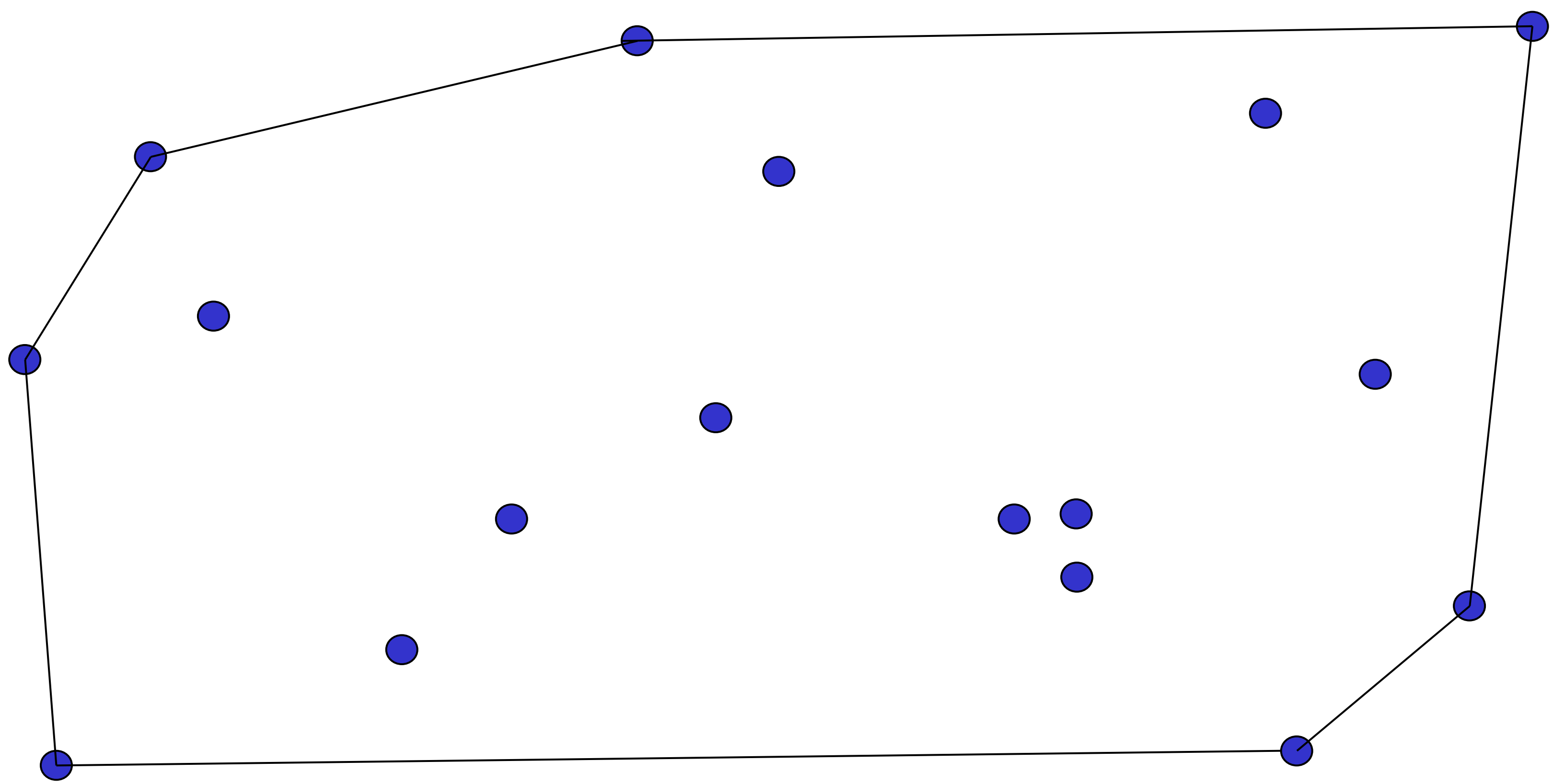
Convex Hull - Divide & Conquer



Convex Hull - Divide & Conquer



Convex Hull - Divide & Conquer



15

Combine CH(B) and CH(C) to get CH(A)

1. We need to find the "upper bridge" and the "lower bridge" that connect the two convex hulls.
2. The lower bridge is the edge vw, where v ∈CH(L) and w∈CH(R), such that all other vertices in CH(L) and in CH(R) are above vw.
3. Suffices to check if both neighbors of v in CH(L) and both neighbors of w in CH(R) are all above vw.

Combine CH(B) and CH(C) to get CH(A)

4. Find the lower bridge as follows:

   (a) v = the rightmost point in CH(B);

   w = the leftmost point in CH(C).

   (b) Loop

   if counterclockwise neighbor(w) lies below the line vw

   then w = counterclockwise neighbor(w)

   else if clockwise neighbor(v) lies below the line vw

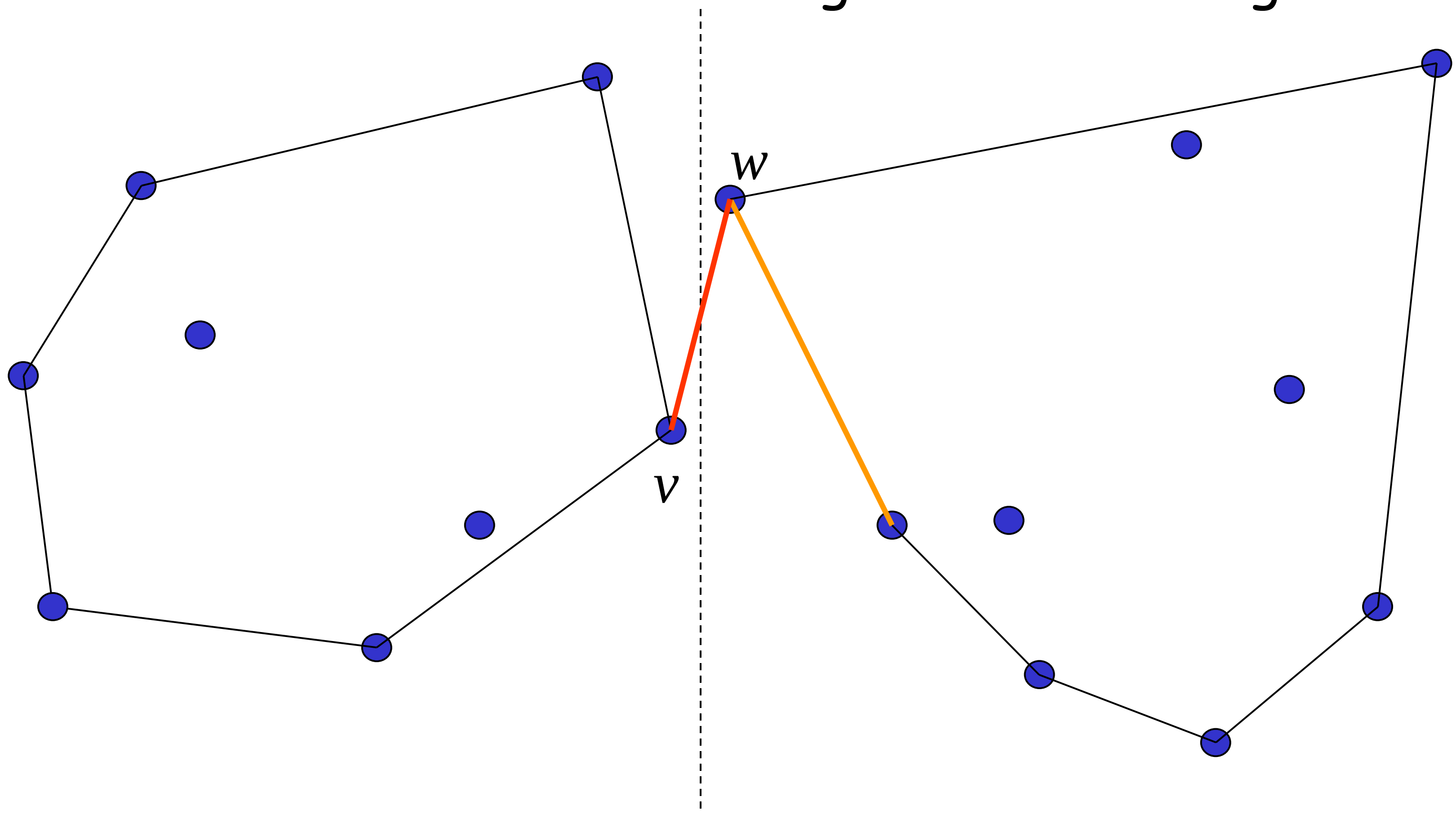
   then v = clockwise neighbor(v)

   (c) vw is the upper bridge.

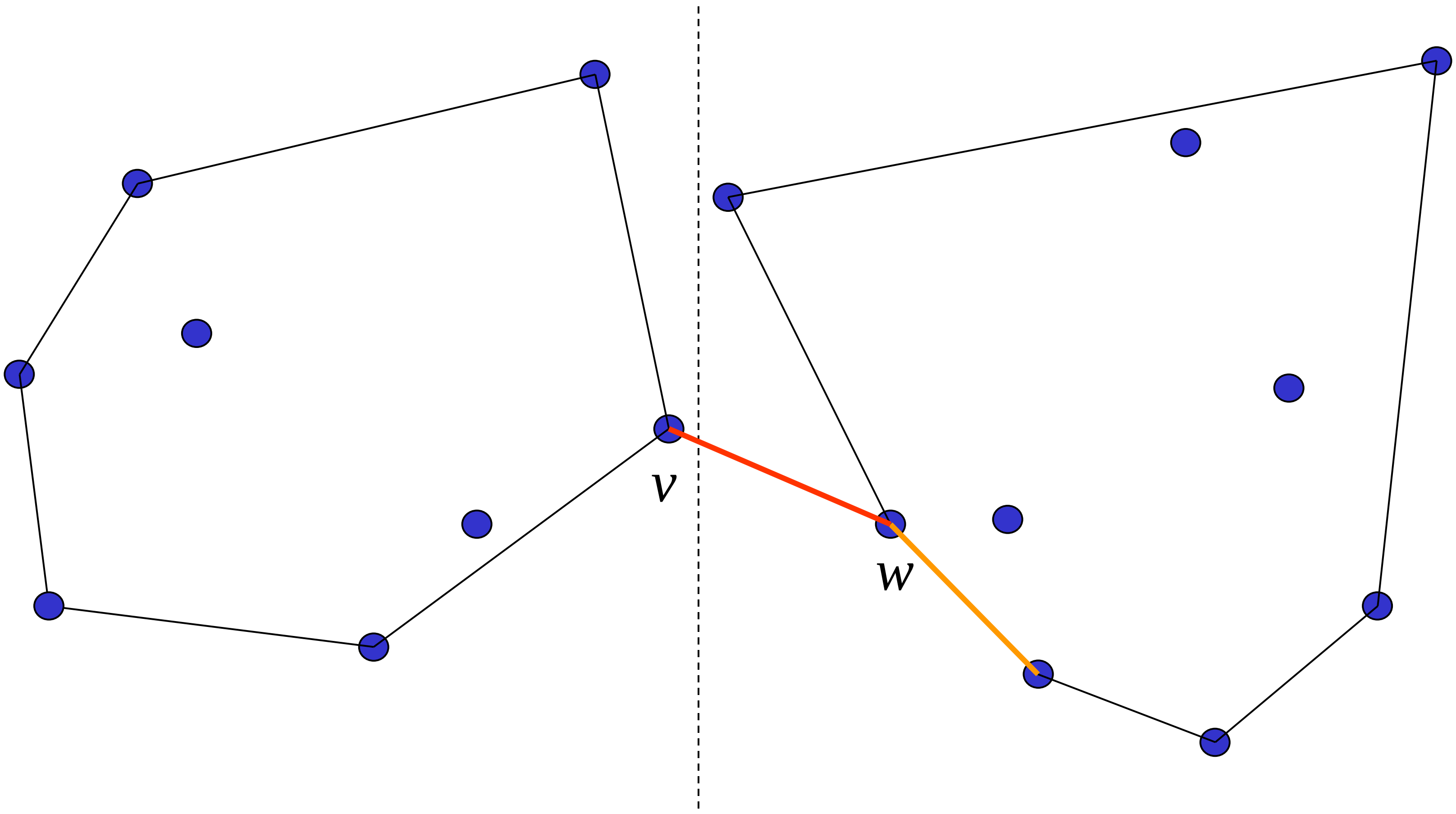
5. Find the upper bridge similarly.

Convex Hull - Divide & Conquer

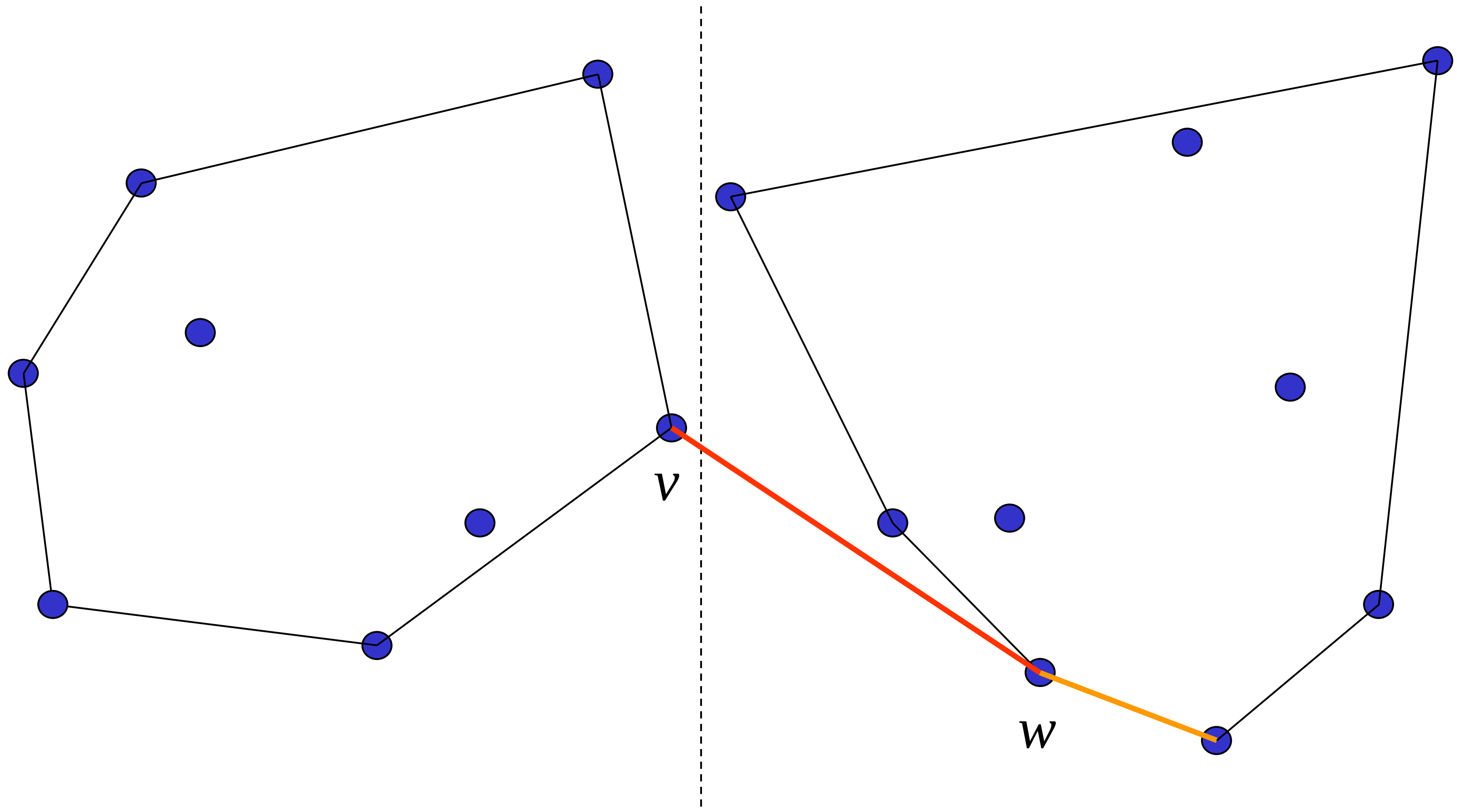
Combine two convex hulls: Finding the lower bridge.



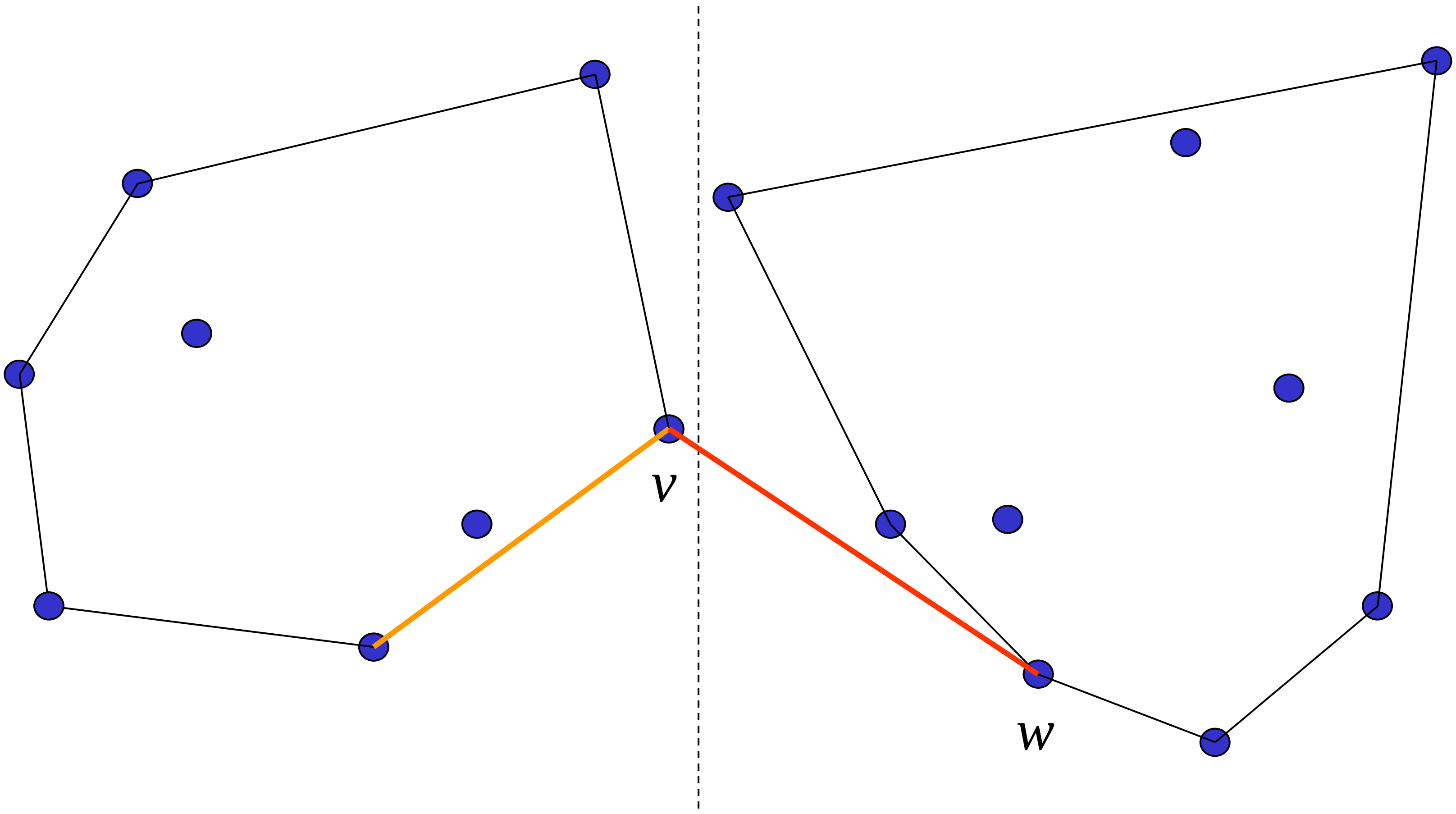
Convex Hull - Divide & Conquer



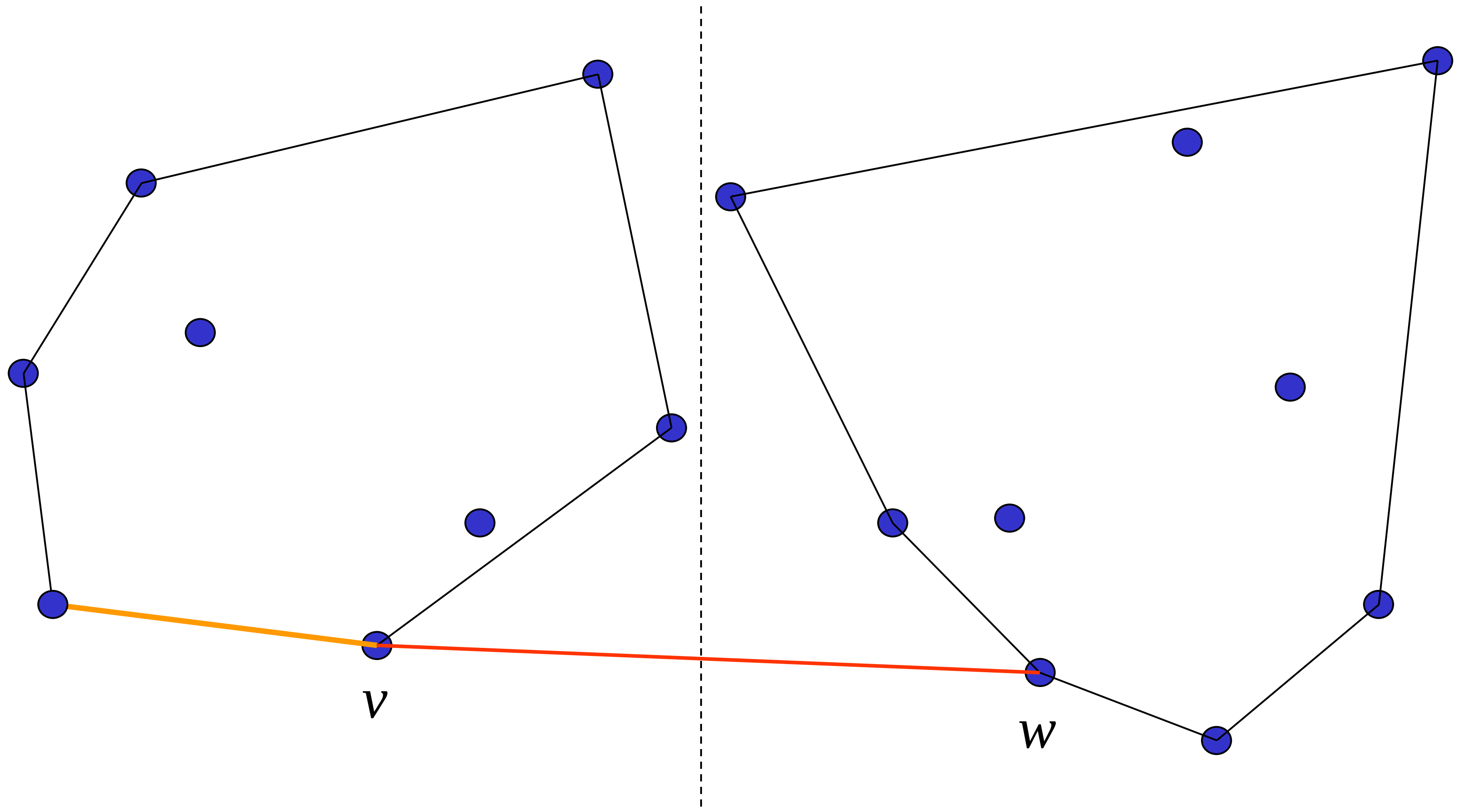
Convex Hull - Divide & Conquer



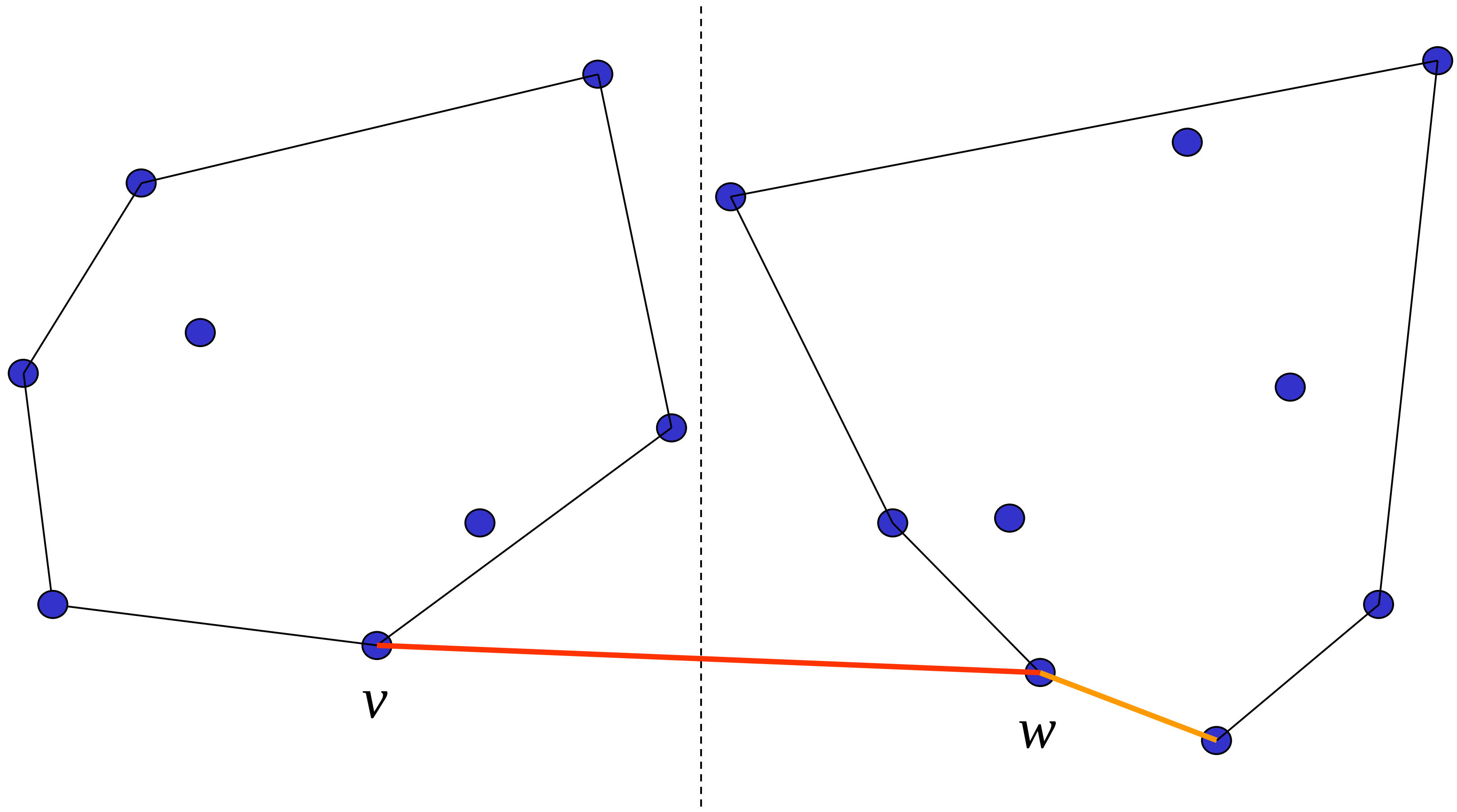
Convex Hull - Divide & Conquer



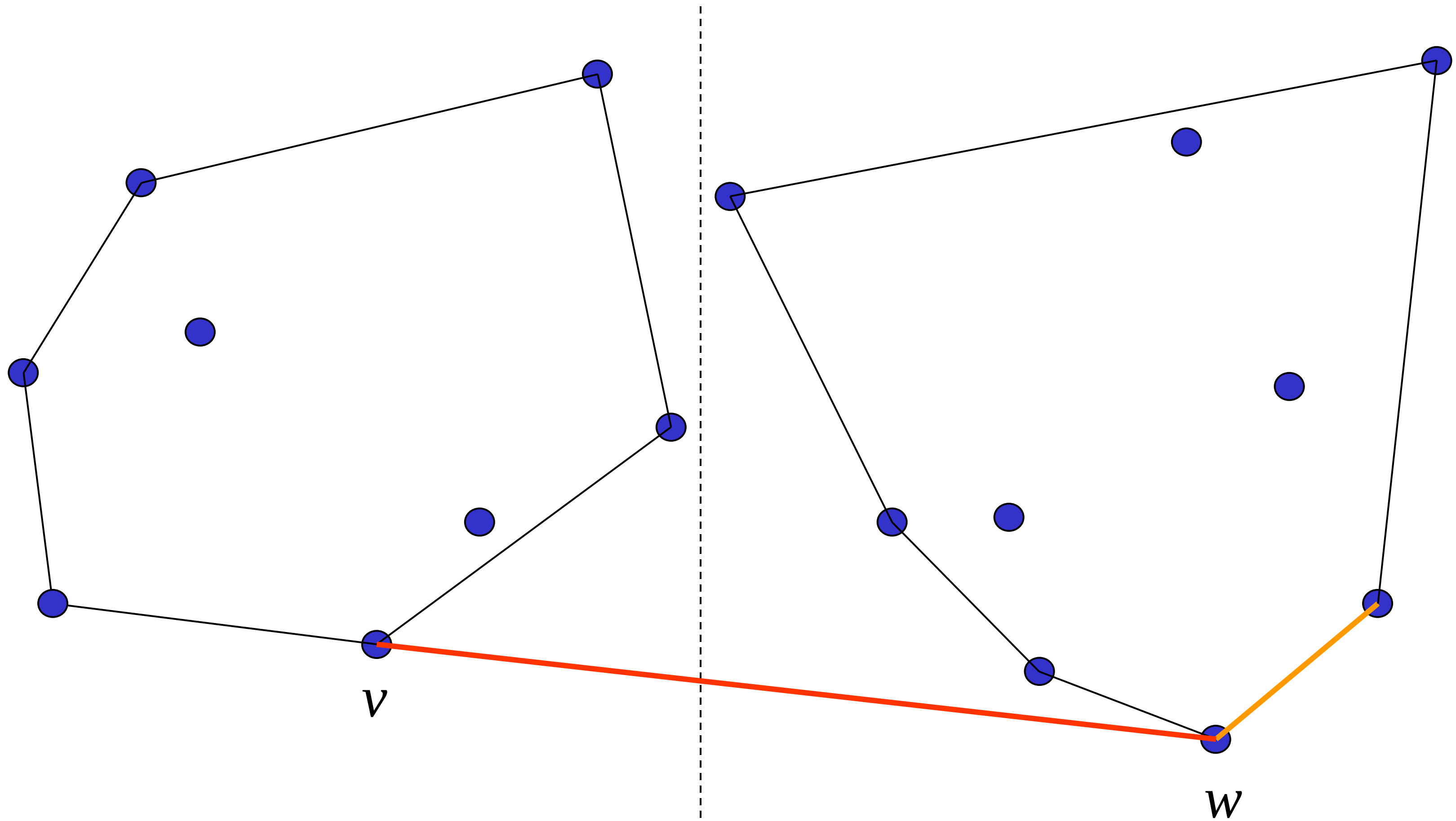
Convex Hull - Divide & Conquer



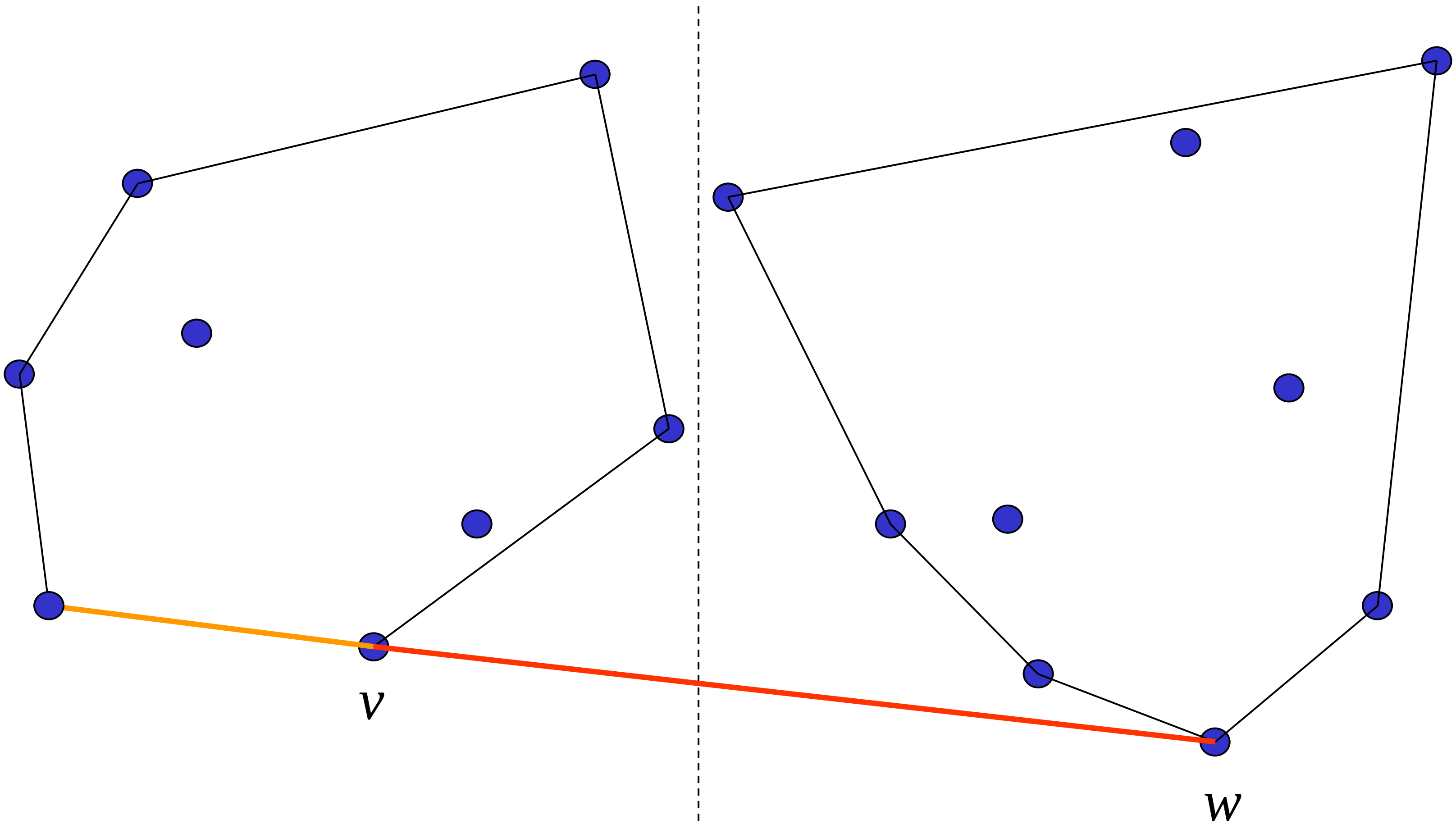
Convex Hull - Divide & Conquer



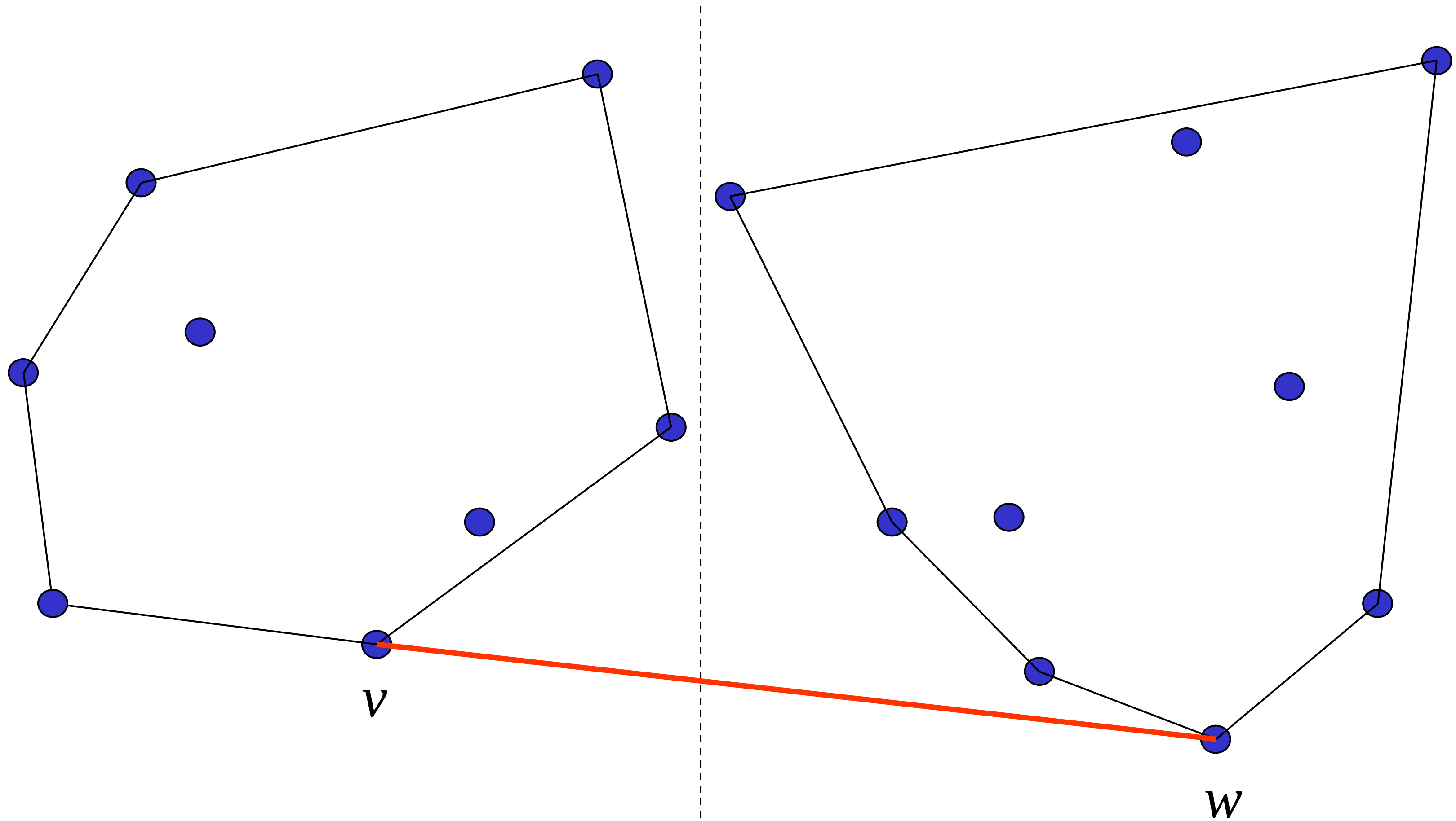
Convex Hull - Divide & Conquer



Convex Hull - Divide & Conquer



Convex Hull - Divide & Conquer



Convex Hull - D&C algorithm.

Analysis:

1. Preprocessing: O(n log n)
2. Recursion: Each of the Divide and Combine steps takes O(n): When calculating the bridges, each point is considered at most once, O(1) for each point.

Therefore:

$$T(n) = \begin{cases} O(1) & n \leq 3 \\ 2T(n/2) + cn & n > 3 \end{cases}$$

Implying T(n)=O(n log n) (like mergesort)

Can we do better? Maybe not by D&C?

Convex Hull - lower bound.

Theorem: Any algorithm for calculating convex hull takes $\Omega(n \log n)$ time.

Proof: Given *n* positive numbers, $x_1 x_2 \ldots, x_n$, correspond to each number x_i the point (x_i , x_i^2), and find a convex hull of the n points.

These points all lie on the parabola $y = x^2$. The convex hull of this set consists of a list of the points sorted by x-coordinate.

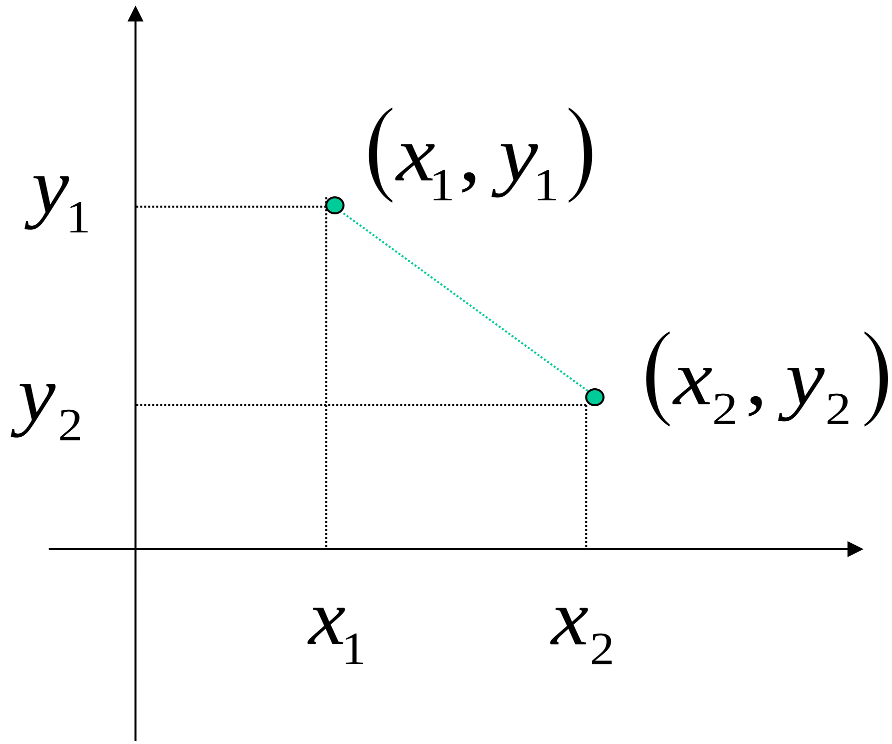
Therefore, if we could find a convex hull in time $T(n)$ then we could sort in time $T(n)+O(n)$.

It is known that sorting takes $\Omega(n \log n)$, therefore, this lower bound applies also to finding the convex hull.

Example 7: Closest Pair Problems

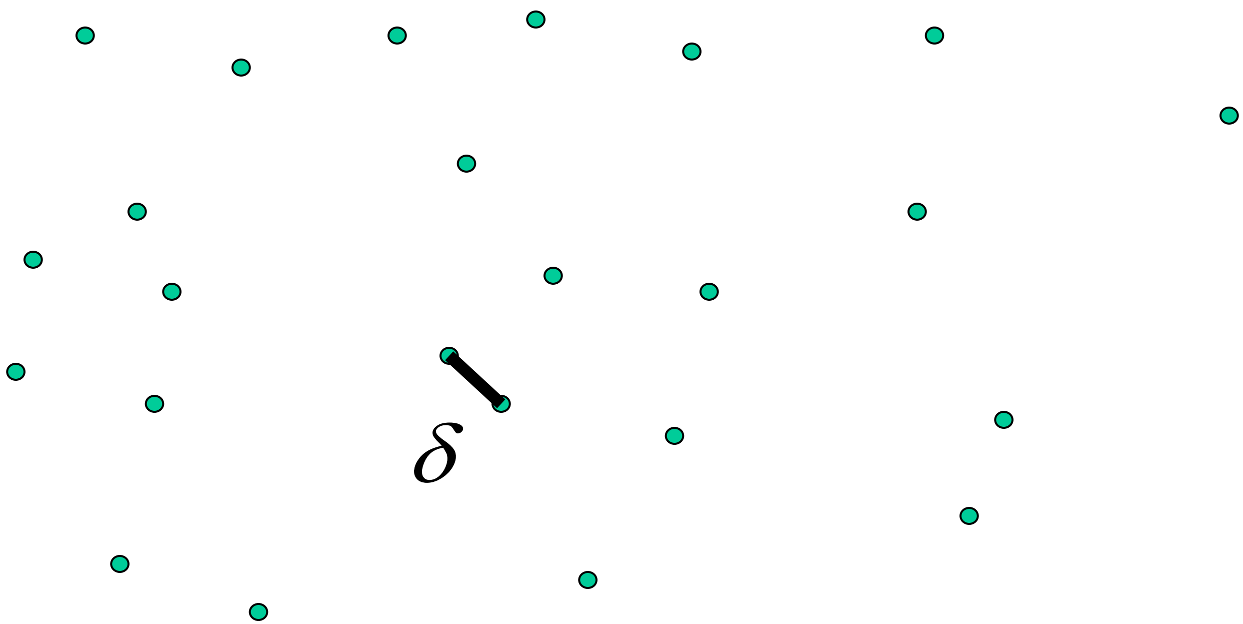
- Input:
  - A set of points $P = \{p_1, \ldots, p_n\}$ in two dimensions
- Output:
  - The pair of points p_i, p_j with minimal Euclidean distance between them.

Euclidean Distances



$$\|(x_1, y_1) - (x_2, y_2)\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Closest Pair Problem



Closest Pair Problem

- $O(n^2)$ time algorithm is easy
- Assumptions:
  - No two points have the same x-coordinates
  - No two points have the same y-coordinates

  (otherwise rotate a bit)
- How do we solve this problem in one-dimension (this is very easy)?
  - Sort the numbers and scan from left to right looking for the minimum gap
- Let's apply divide-and-conquer to the 1-dim problem:

D&C for 1-dim closest pair

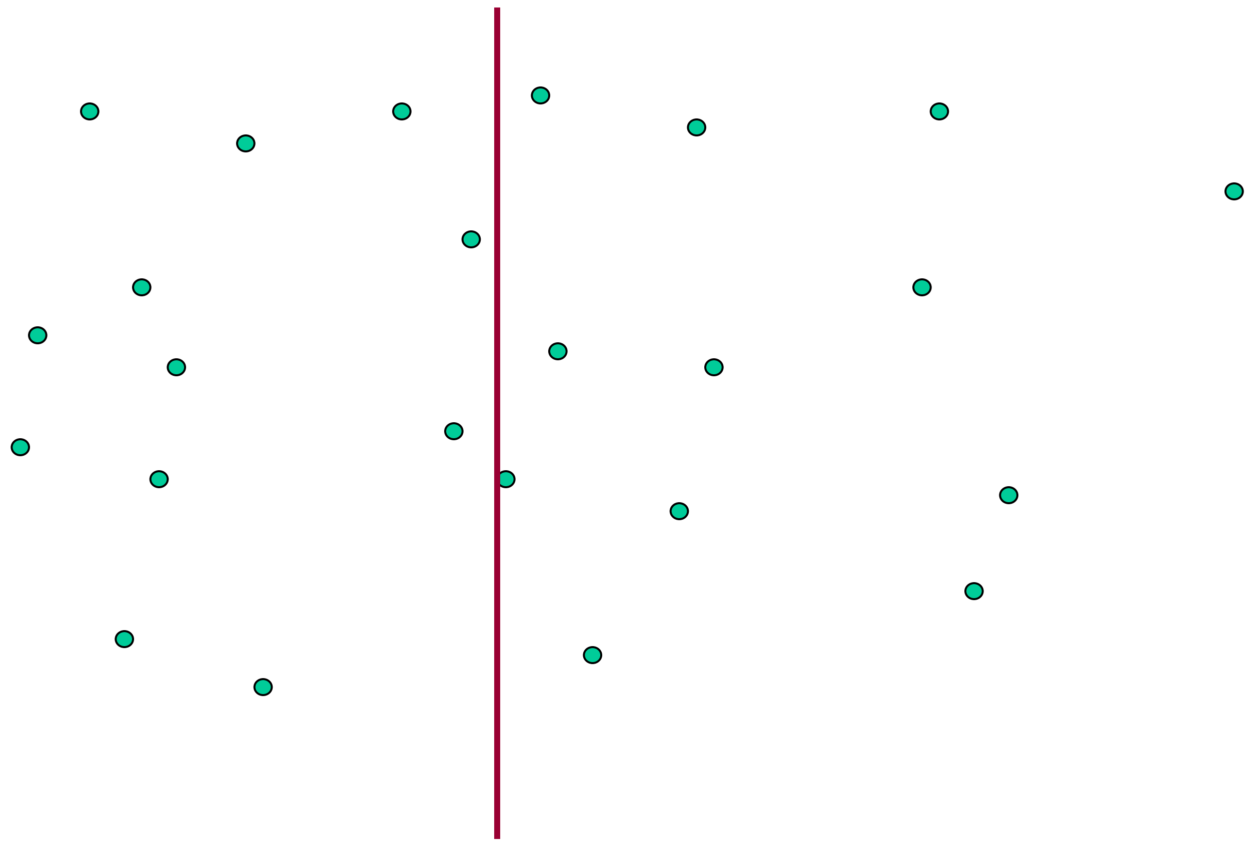
- Divide
  - $t = n/2$
- Conquer
  - δ_1 = Closest-Pair(A,1,t)
  - δ_2 = Closest-Pair(A,t+1,n)
- Combine
  - Return min(δ_1, δ_2, A[t+1]-A[t])

Time: $T(n)=2T(n/2)+c$ ➔ $T(n)=\Theta(n)$

Divide and Conquer: 2-dim

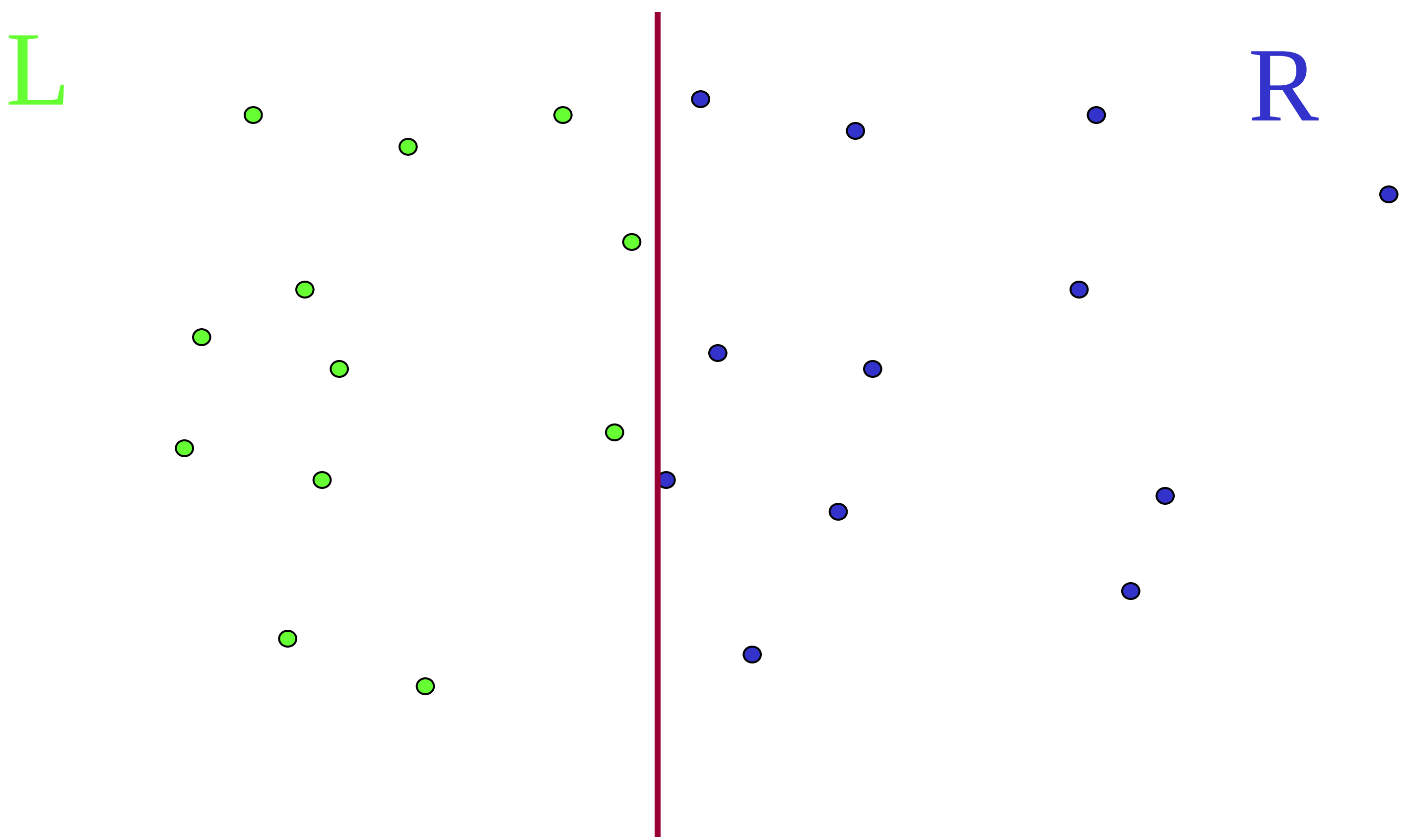
- We will do better than $O(n^2)$.
- Intuitively, there is no need to really compare each pair.
- Divide and conquer can avoid it.

Divide and Conquer for the Closest Pair Problem



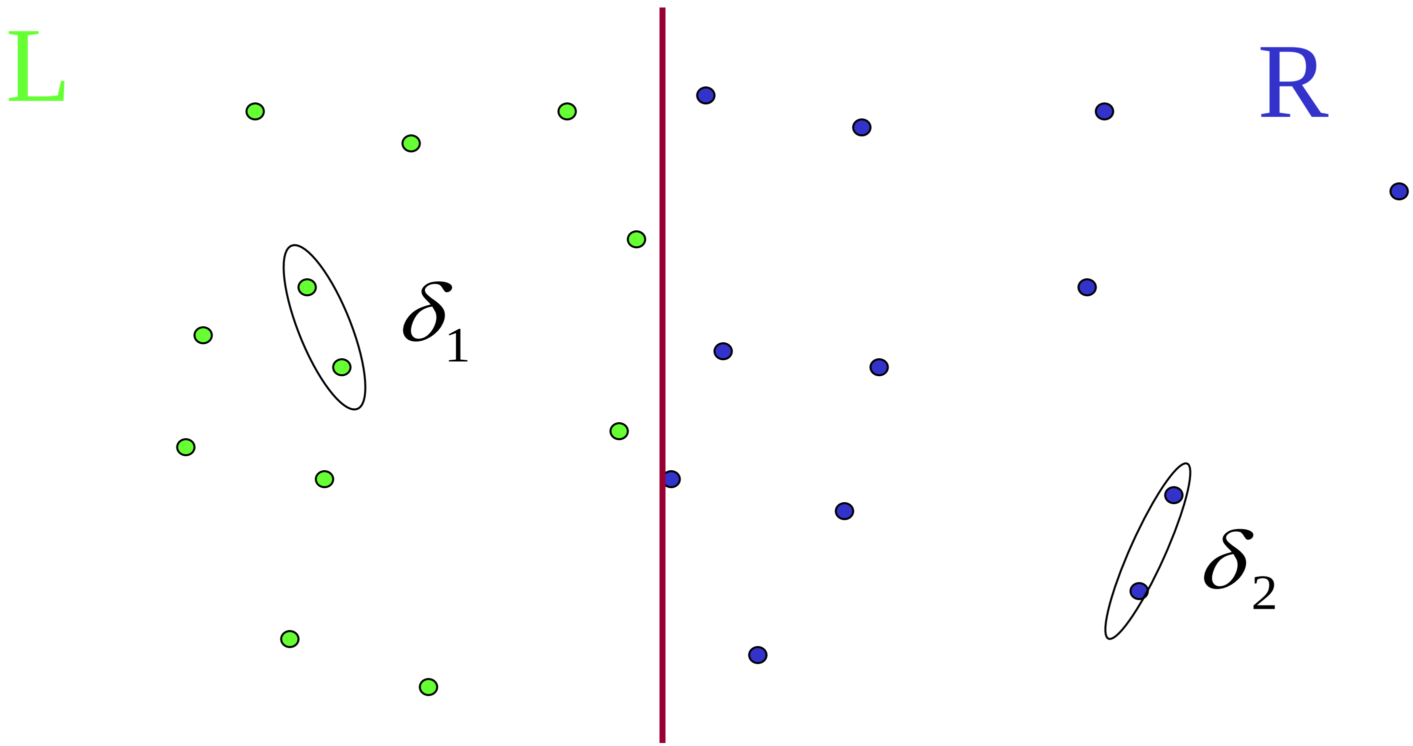
Divide by x-median

Divide



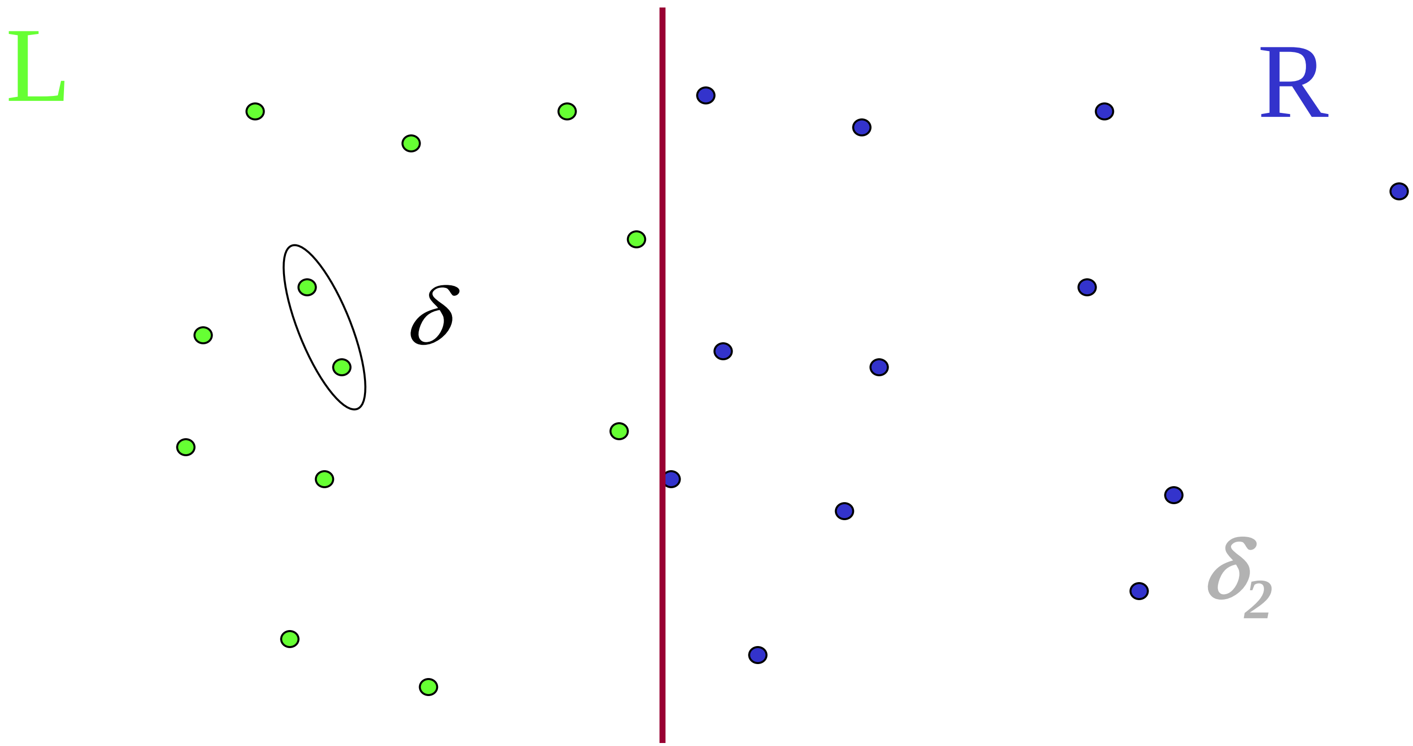
Divide by x-median

Conquer



Conquer: Recursively solve L and R

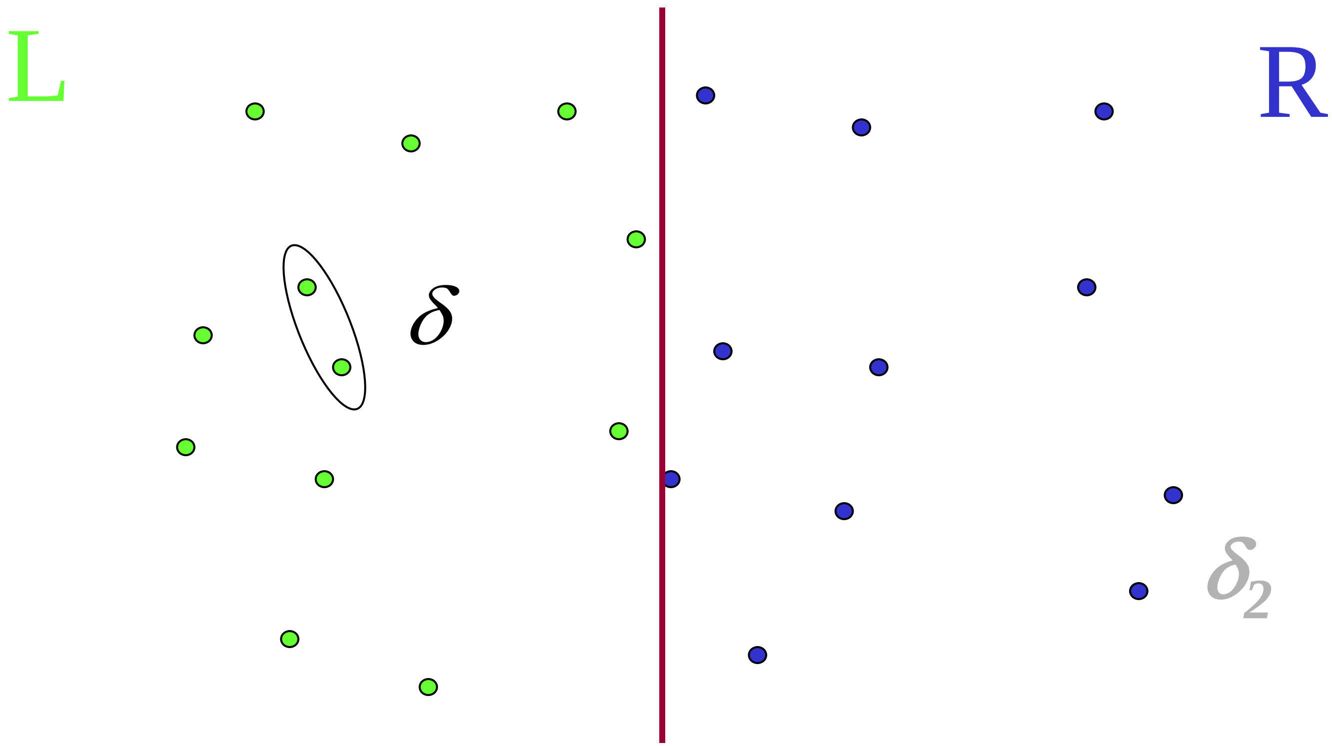
Combine I



Take the smaller one of δ_1, δ_2: $\delta = \min(\delta_1, \delta_2)$

Combine II

but maybe there is a point in L and a point in R whose distance is smaller than δ?



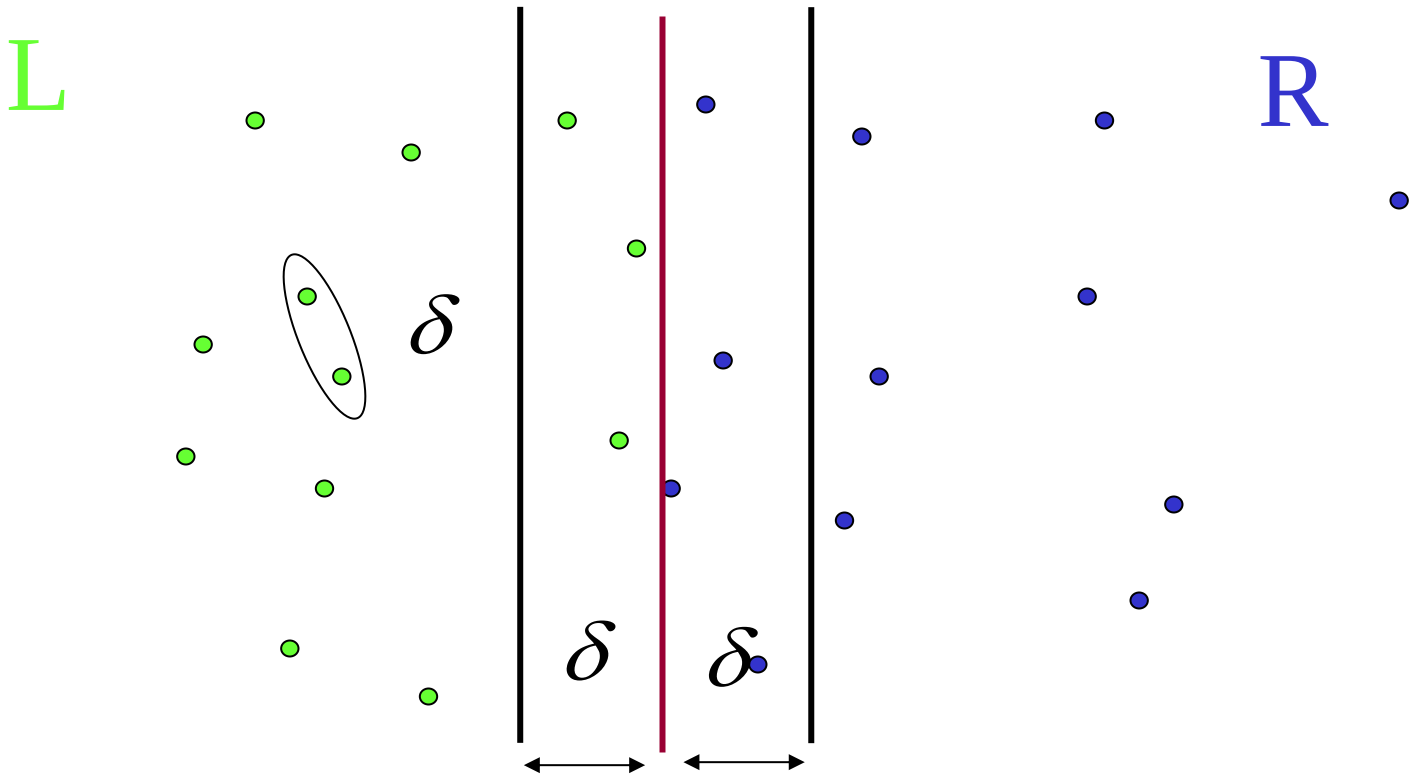
Take the smaller one of δ_1, δ_2: $\delta = \min(\delta_1, \delta_2)$

Combine II

- If the answer is "no" then we are done.
- If the answer is "yes" then the closest such pair forms the closest pair for the entire set
- How do we determine this?

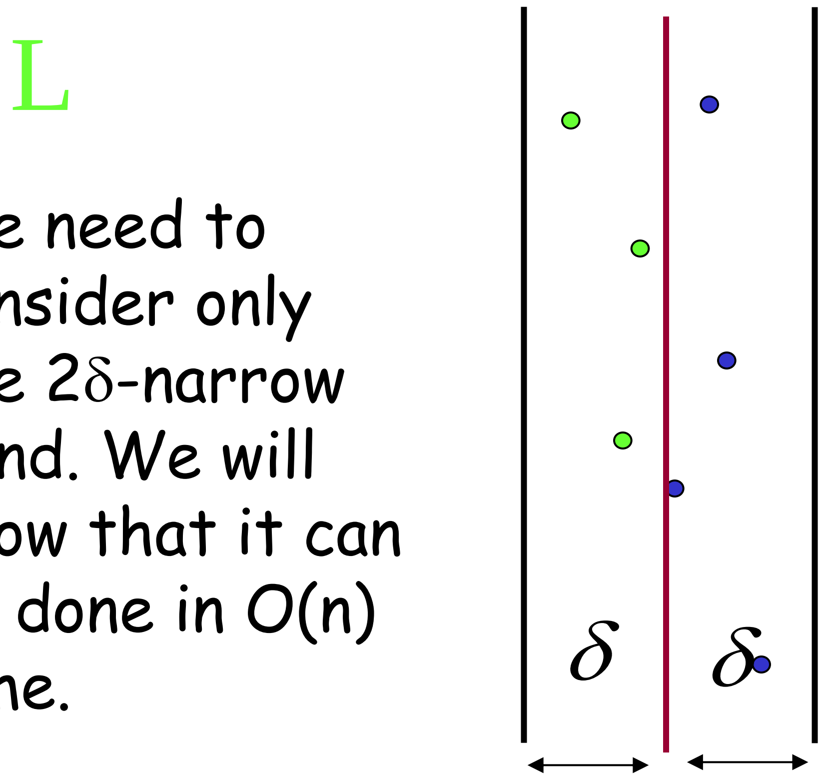
Combine II

Is there a point in L and a point in R whose distance is smaller than δ?



Combine II

Is there a point in L and a point in R whose distance is smaller than δ ?



We need to consider only the 2δ-narrow band. We will show that it can be done in O(n) time.

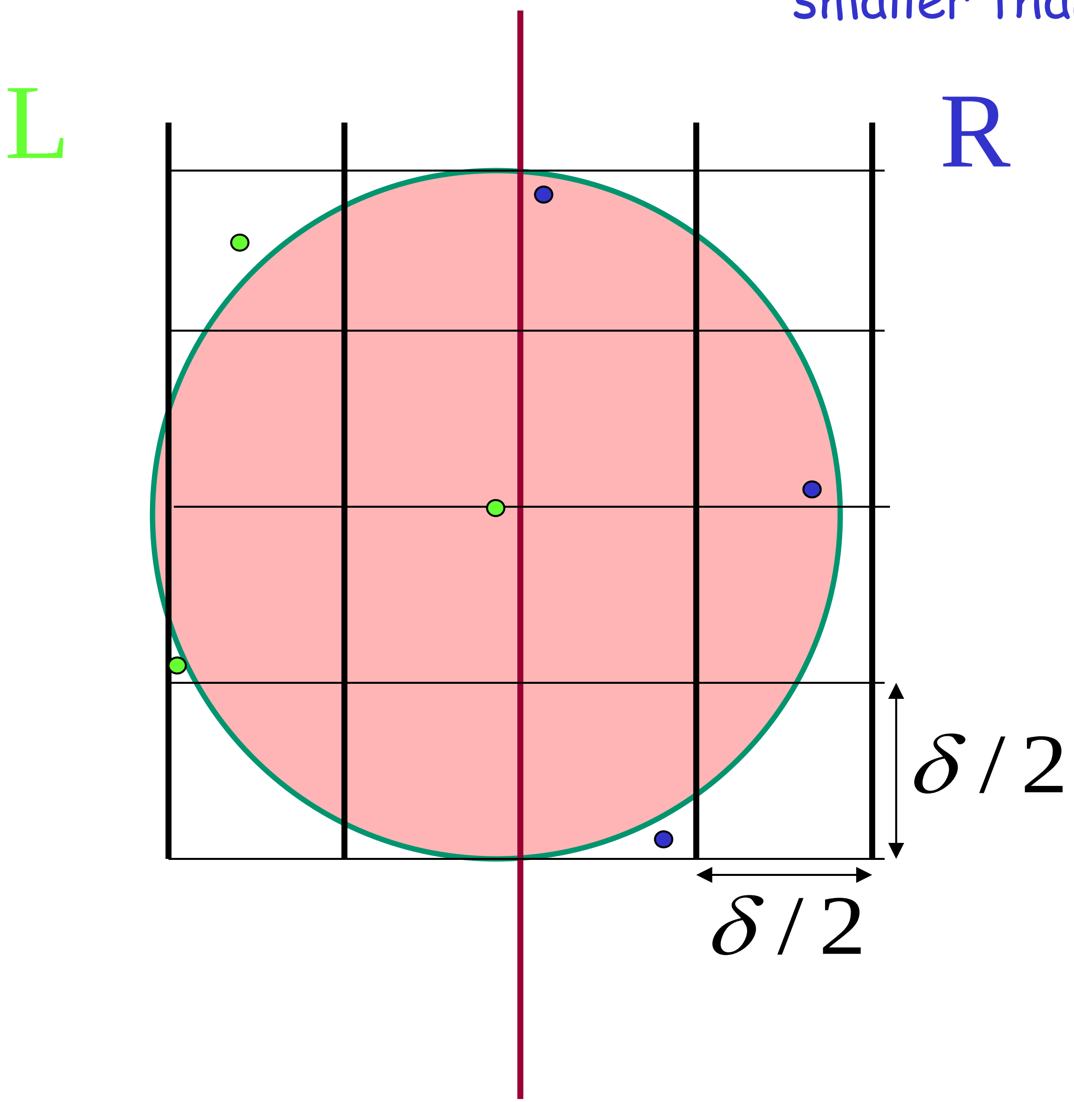
Denote this set by S. Assume S_y is a sorted list of S by y-coordinate.

Combine II

- There exists a point in L and a point in R whose distance is less than δ if and only if there exist two points in S whose distance is less than δ (why?).
- If S is the whole thing, did we gain anything?
- Amazing claim: If s and t in S have the property that $||s-t|| < \delta$, then s and t are within 8 positions of each other in the sorted list S_y.

Combine II

Is there a pair of points, one in L and one in R, whose distance is smaller than δ?



There is at most one point in each box.

Top half of circle intersects 8 boxes.

In fact, can prove less than 8.

D&C Algorithms for Closest-Pair

- Preprocessing:
  - Construct P_x and P_y as sorted-list by x- and y-coordinates
- Divide
  - Construct L, L_x , L_y and R, R_x , R_y
- Conquer
  - Let δ_1= Closest-Pair(L, L_x , L_y)
  - Let δ_2= Closest-Pair(R, R_x , R_y)
- Combine
  - Let $\delta = \min(\delta_1, \delta_2)$
  - Construct S and S_y
  - For each point in S_y, check each of the next 8 points in S_y.
  - If the distance is less than δ, then update δ to be the new distance

Closest-Pair - Time Analysis

- Preprocessing: $O(n \log n)$ time
- Divide: $O(n)$
- Conquer: $2T(n/2)$
- Combine: $O(n)$

$T(n)=2T(n/2)+O(n) \rightarrow O(n \log n)$ time