Algorithms: Lecture 7

Chalmers University of Technology

Today's Lecture

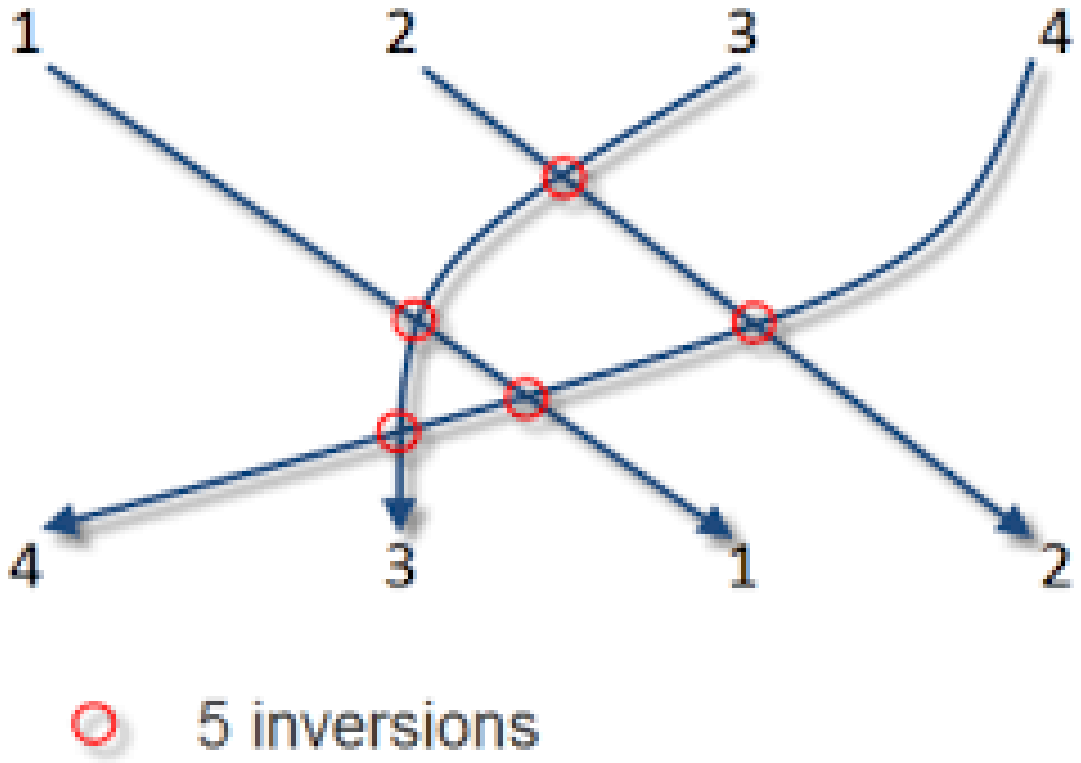
Divide & Conquer

- Counting Inversions
- Closest Pair of Points
- Multiplication of large integers

Intro to the forthcoming problems

- Graphs: Basic Definitions, Applications and Interesting Problems

Counting Inversions



http://www.bogotobogo.com/Algorithms/images/mergesort/inversion_counting_mergesort.png

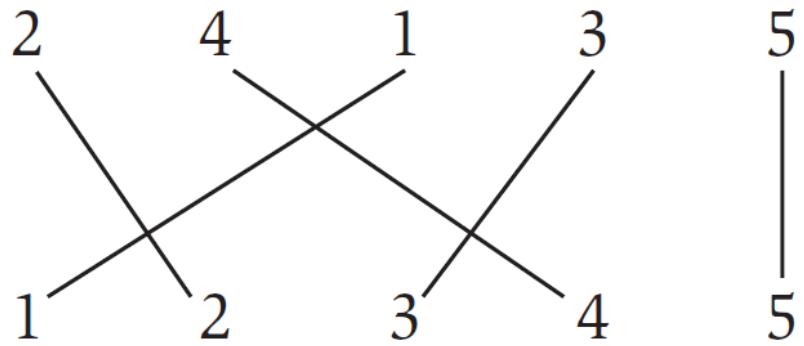
Counting Inversions

Given: a sequence $(a_1, \ldots, a_n)$ of elements where an order relation $<$ is defined.
Goal: Count the inversions in this sequence.

An inversion is a pair of elements where $i < j$ but $a_i > a_j$.

Find inversions in (2,4,1,3,5)

Assume there is a sequence (1, 2, 3, … , n), thus the given sequence is a permutation of it.



What is it:

"degree of unsortedness" of a given sequence.

"Dissimilarity" of two sequences containing the same elements but in different order.

Counting Inversions

RANKING:

Music site tries to match your song preferences with others.

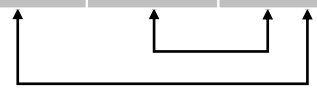
- You rank n songs.
- Music site consults database to find people with similar tastes.

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 $a_i > a_j$.

Songs

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



Inversions

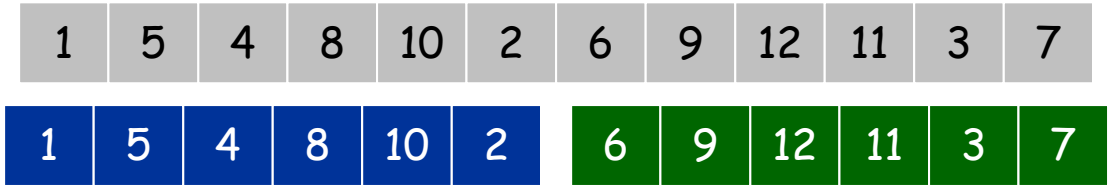
3-2, 4-2

Brute force is Trivial: check all $O(n^2)$ pairs i and j.

Counting Inversions: Divide-and-Conquer

Due to the vague similarity with Sorting, Divide-and-conquer should be applicable

- split the sequence into two equal halves, **A** and **B**.
- recursively count inversions in both **A** and **B** separately.
- count inversions between **A** and **B**, and return sum of the three quantities.



5 blue-blue inversions

8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

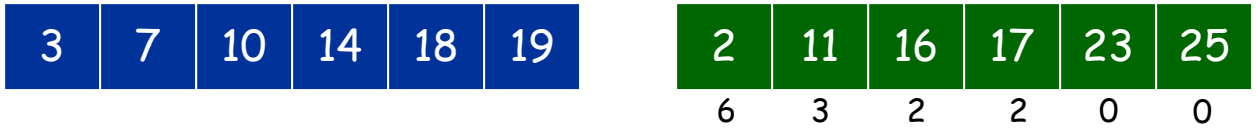
Total = 5 + 8 + 9 = 22.

- $T(n) = 2T(n / 2) + ?$
- Need to replace ? with $O(n)$ to make it better than the Brute Force.
- Sorting the sequence while counting inversions?
  - Can this help? OR Stupid idea as in Median Finding?
- Sorting AND Counting Inversions Simultaneously… Does it sound good?
  - Challenge: merge two sorted sequences A and B, and simultaneously count the inversions between A and B, still everything in $O(n)$ time

Counting Inversions: Merge and Count

Counting **A**-**B** inversions

- Assume both **A** and **B** are sorted.
- Proceed as in the Mergesort
  - While merging whenever the next element copied into the merged sequence is from **B**, this element has inversions with exactly those elements of **A** not visited yet.



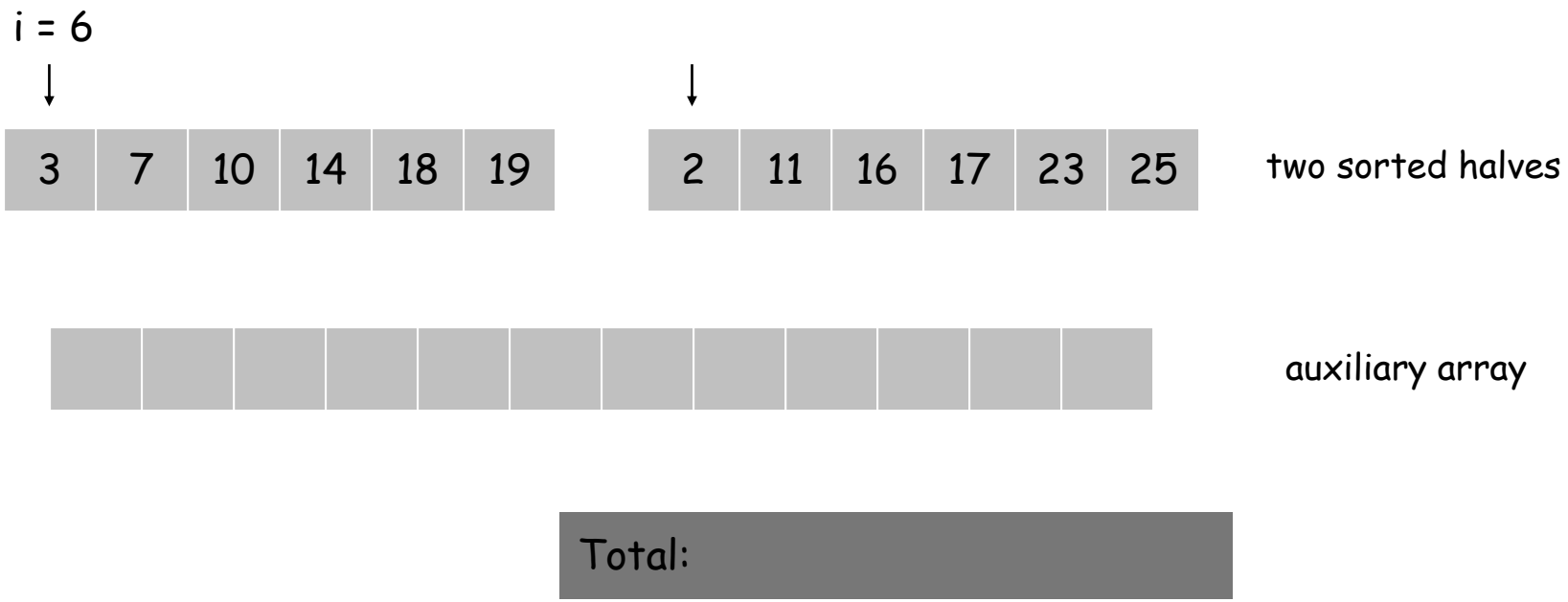
13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

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

Merge and Count

Merge and count step.

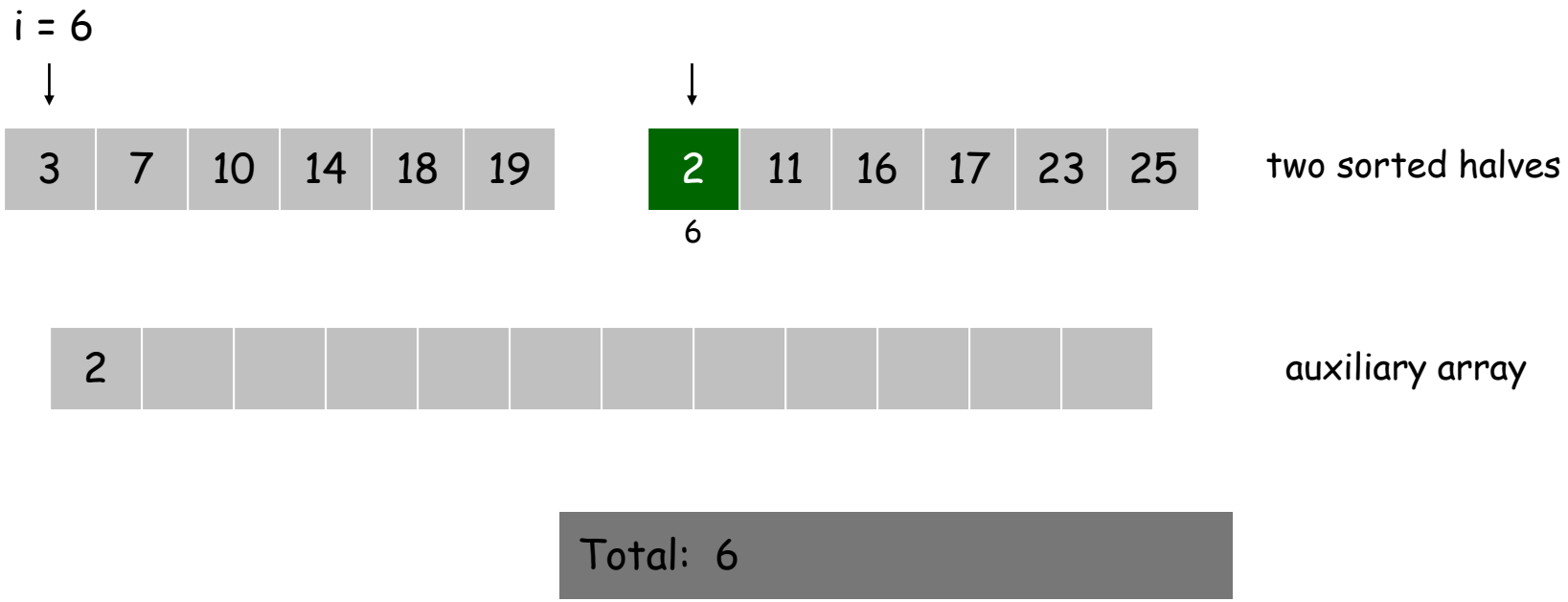
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

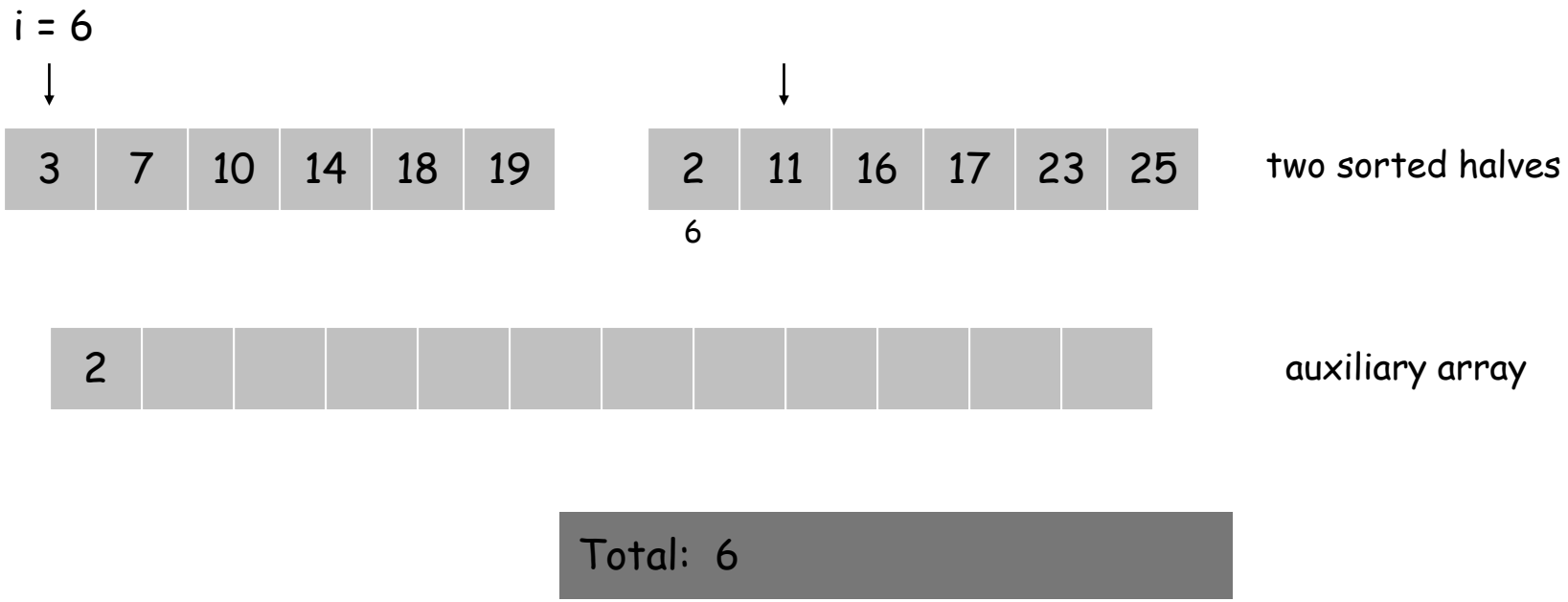
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

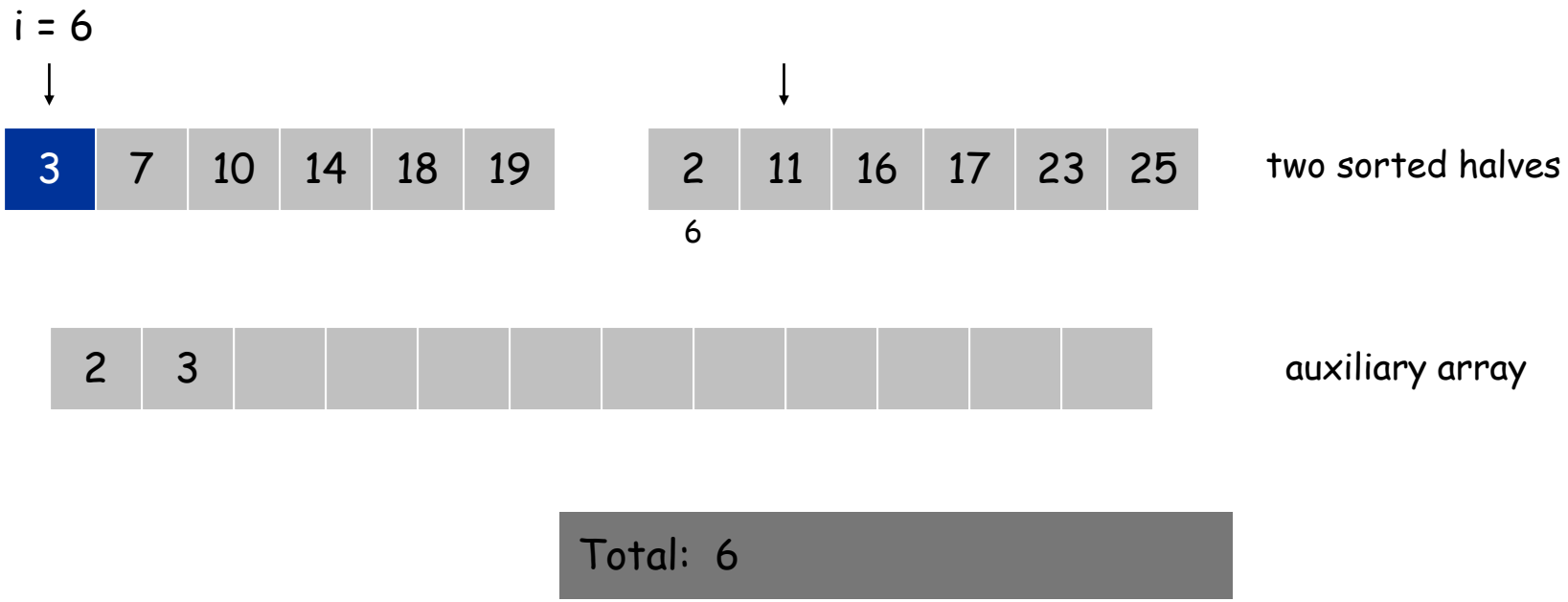
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

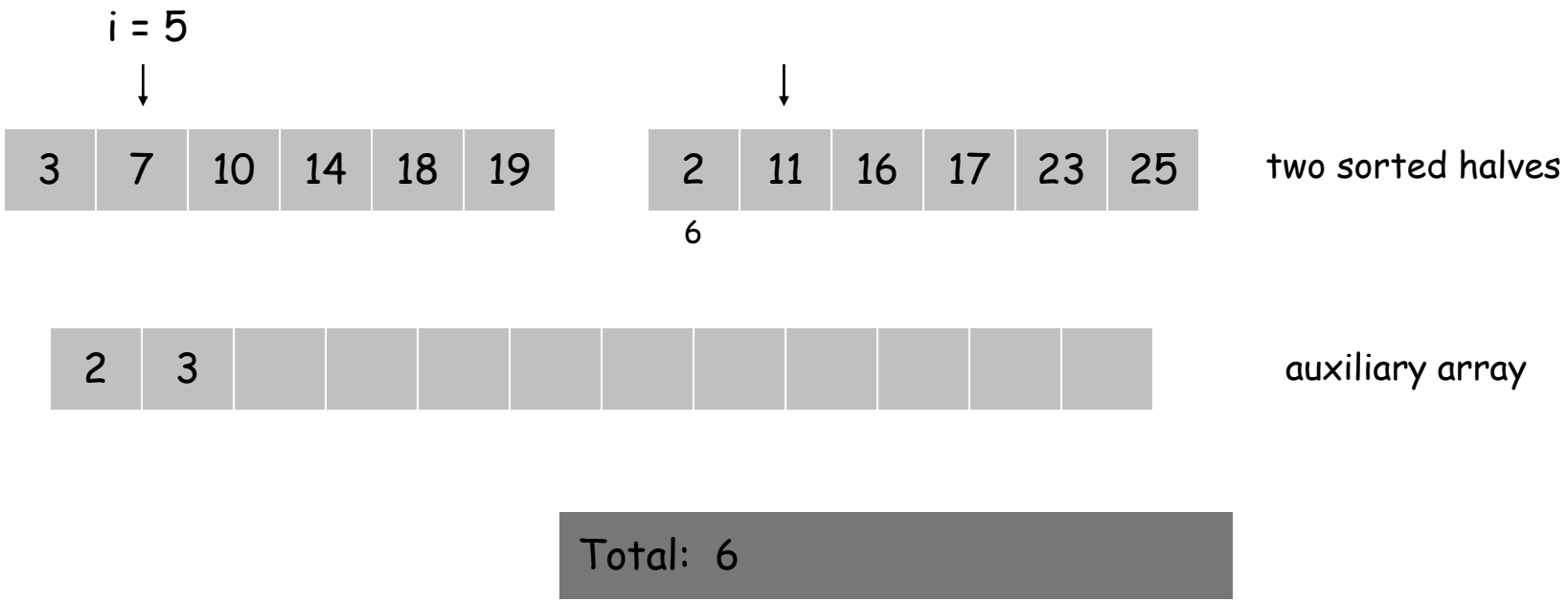
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

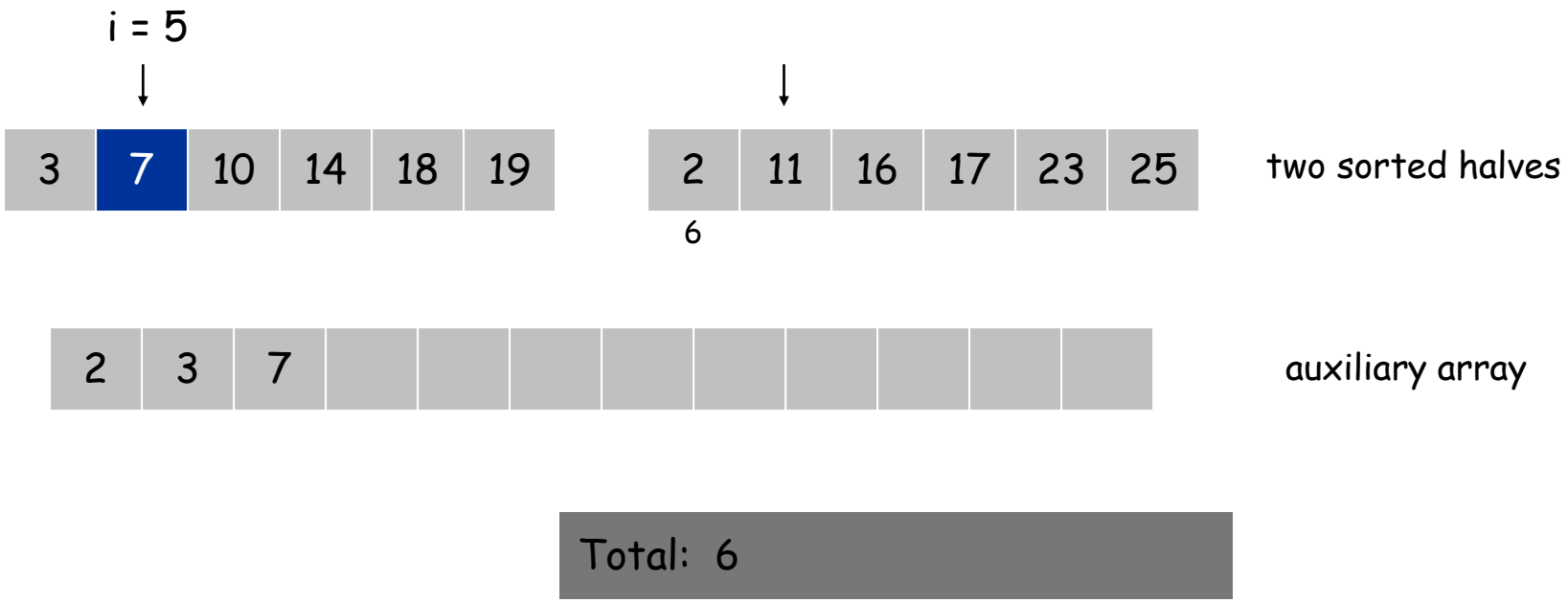
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

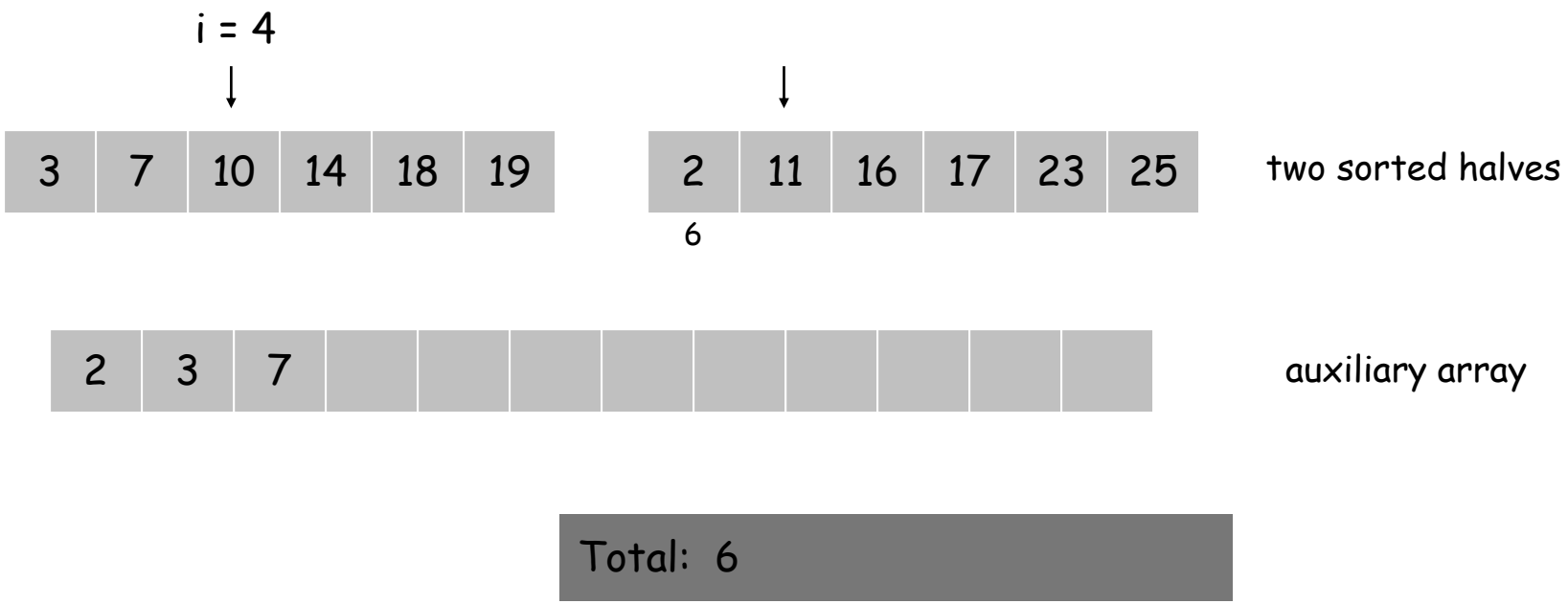
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

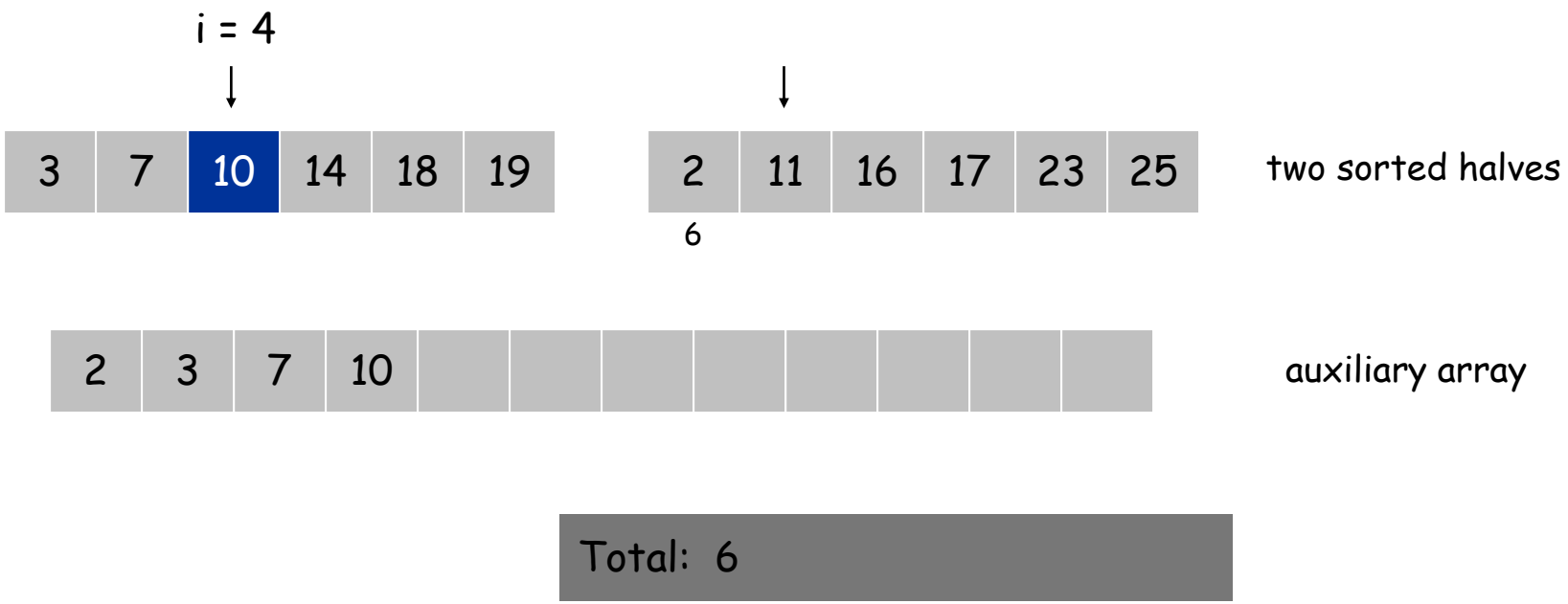
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

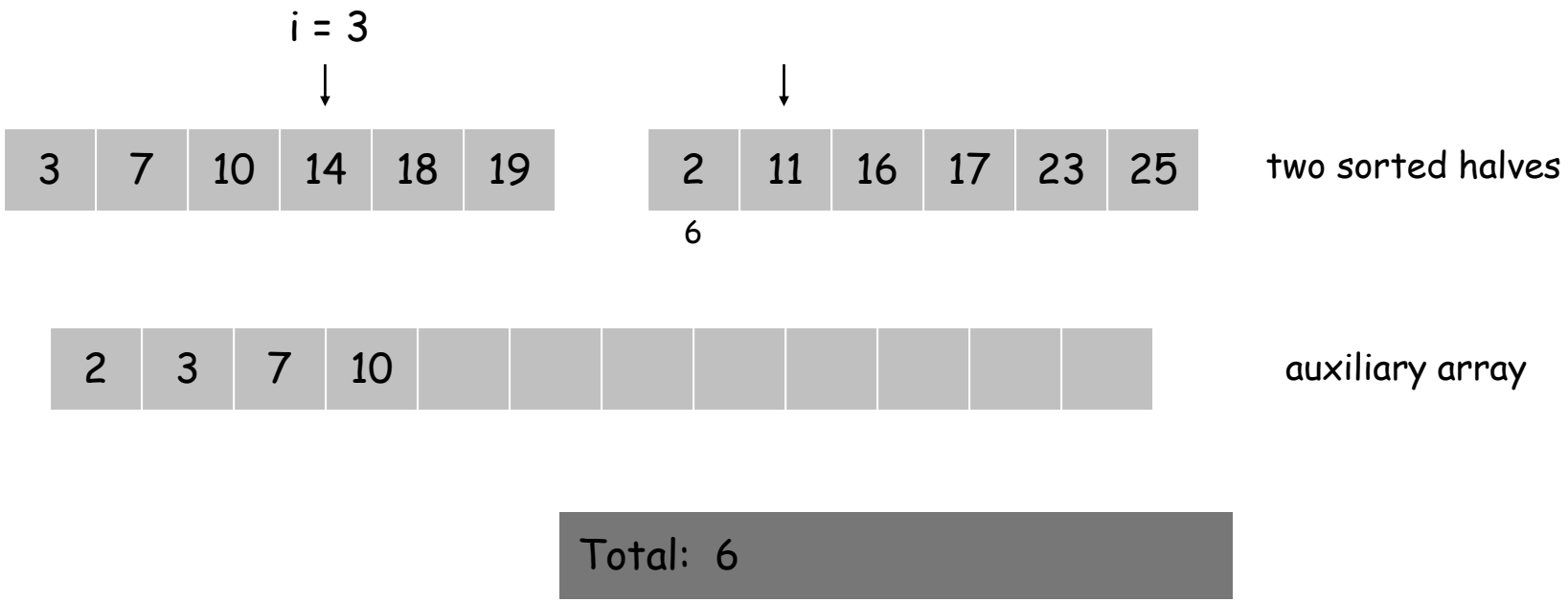
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

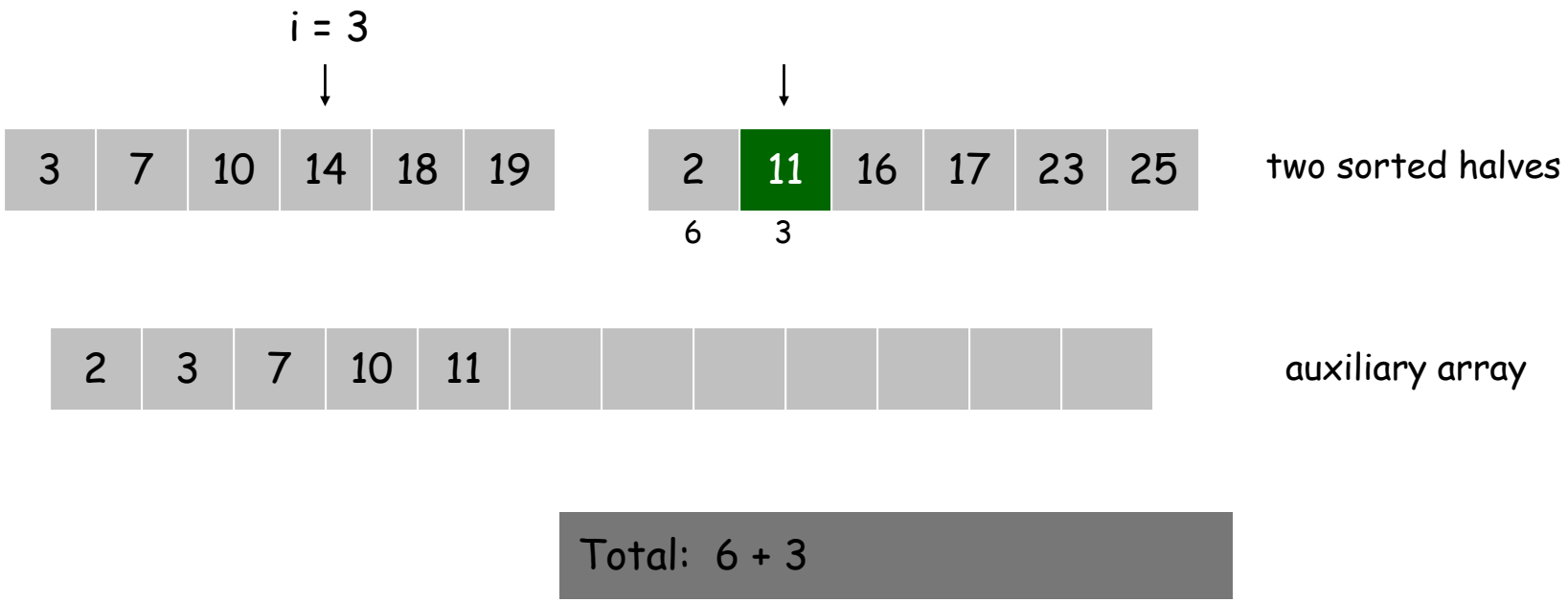
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

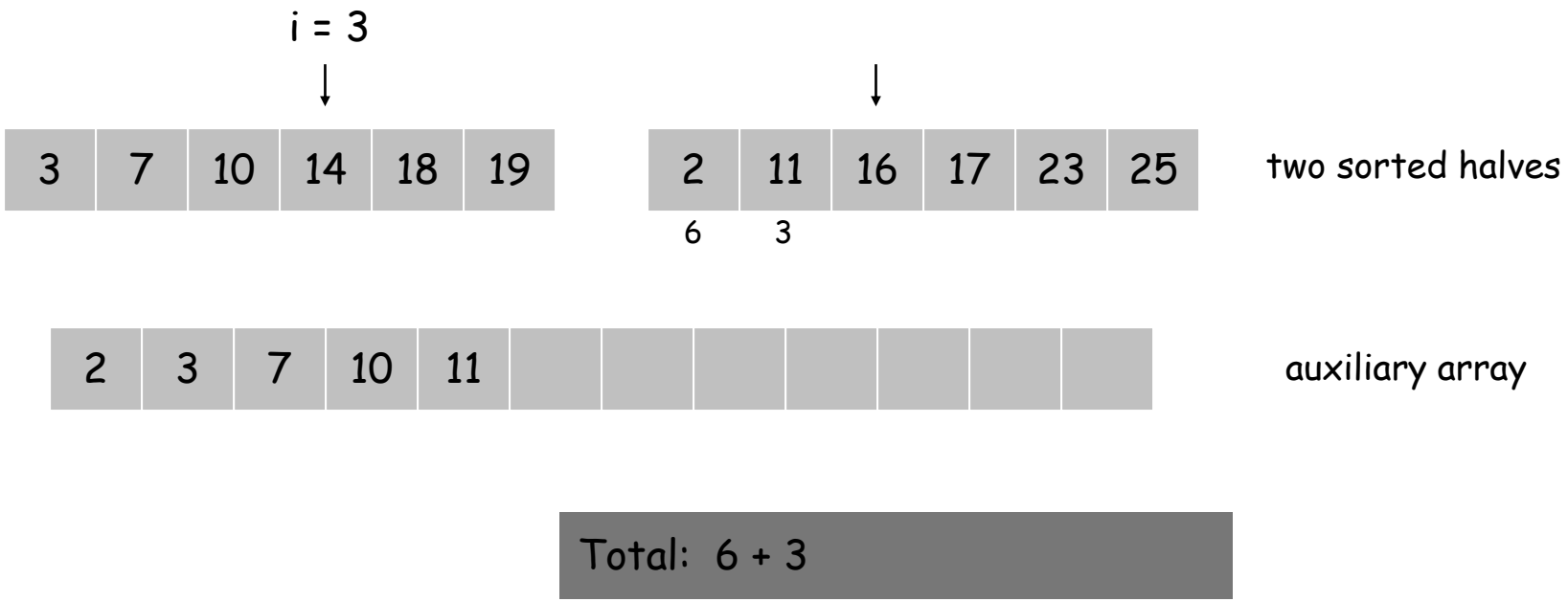
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

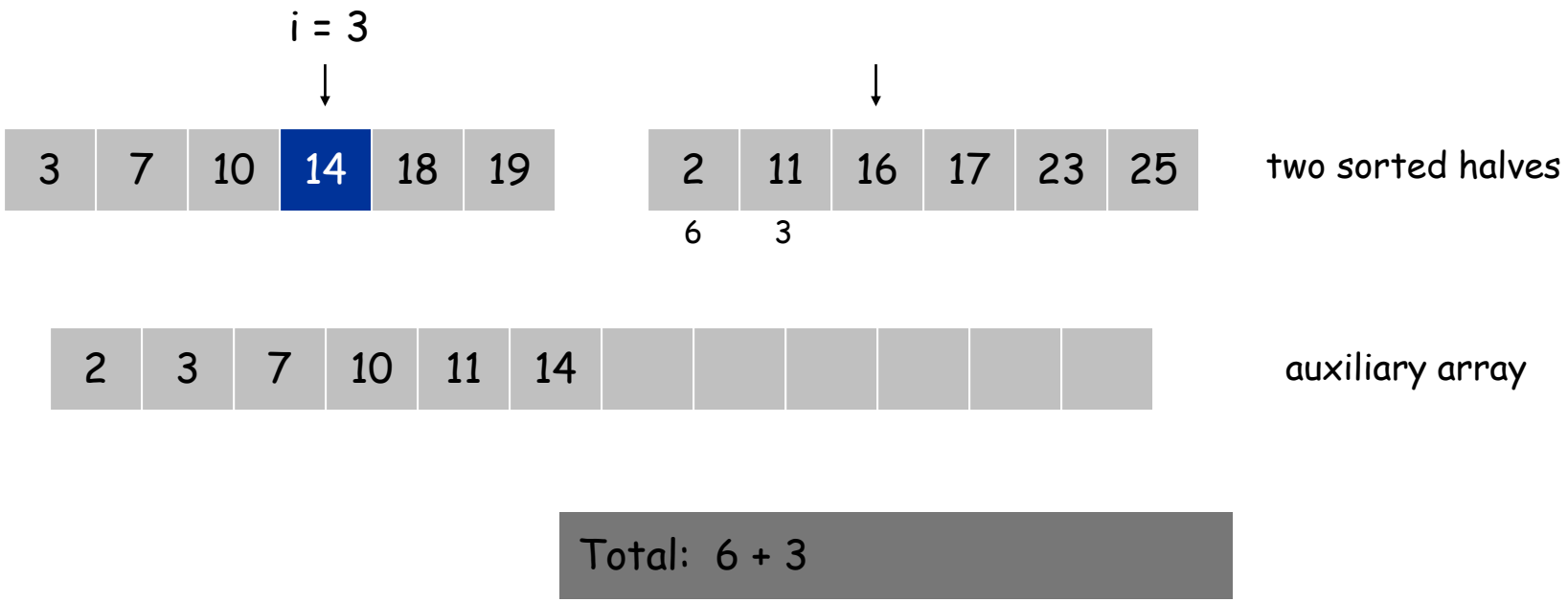
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

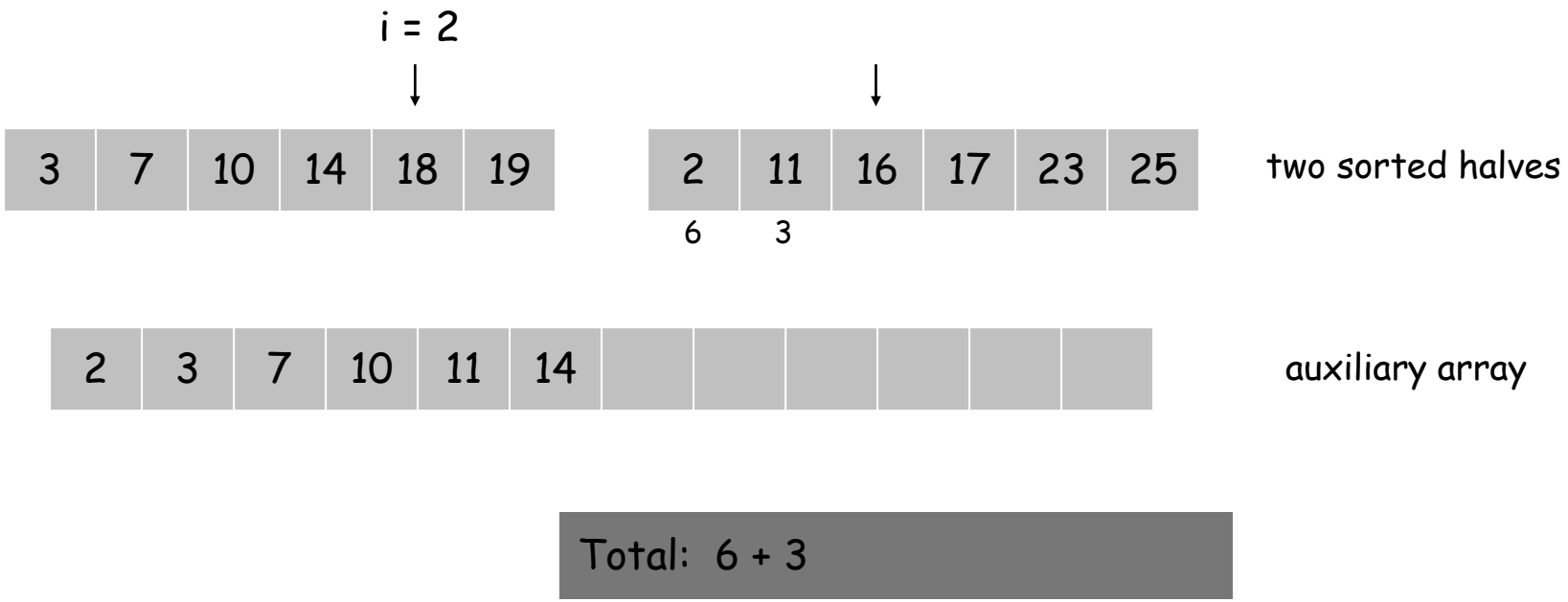
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

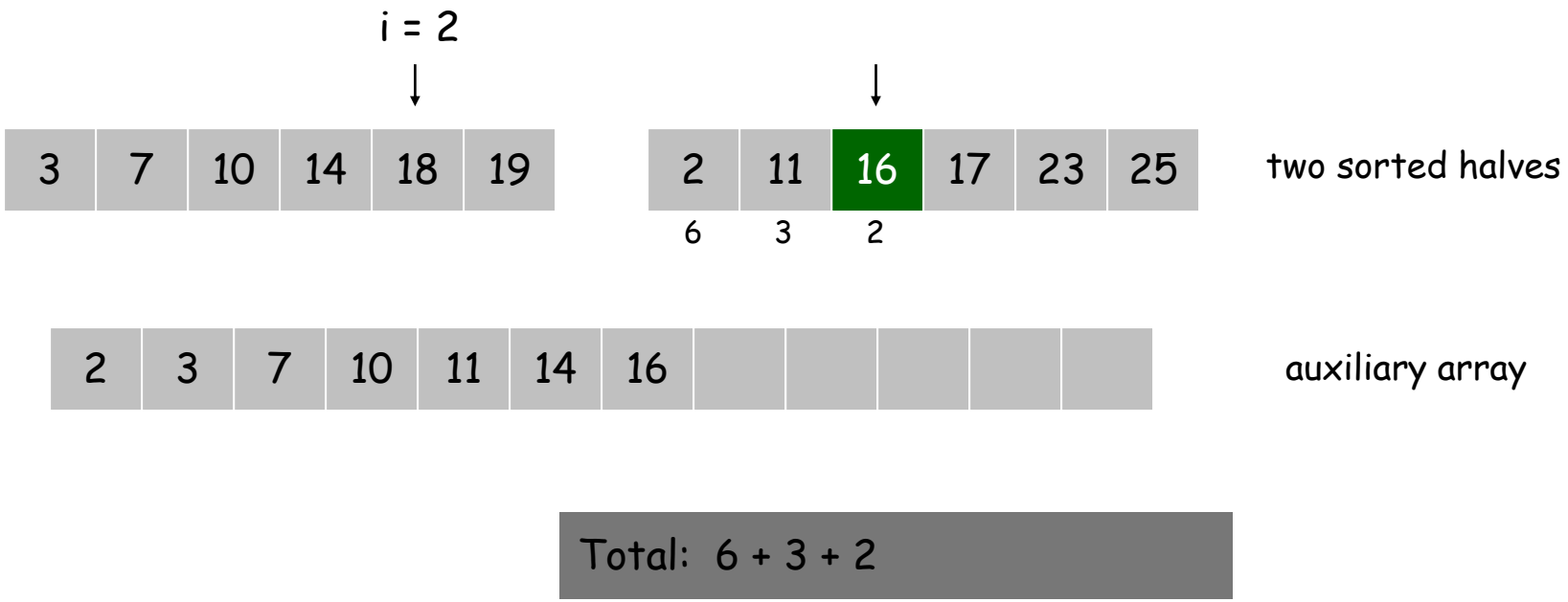
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

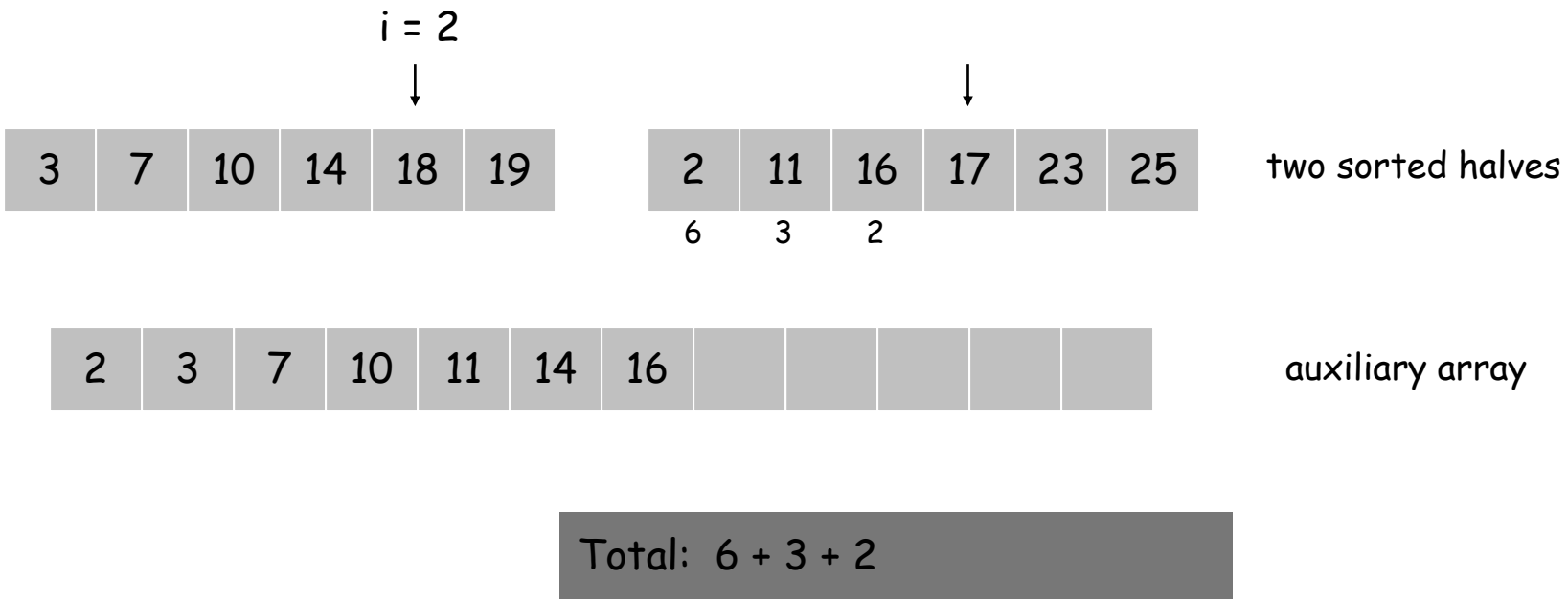
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

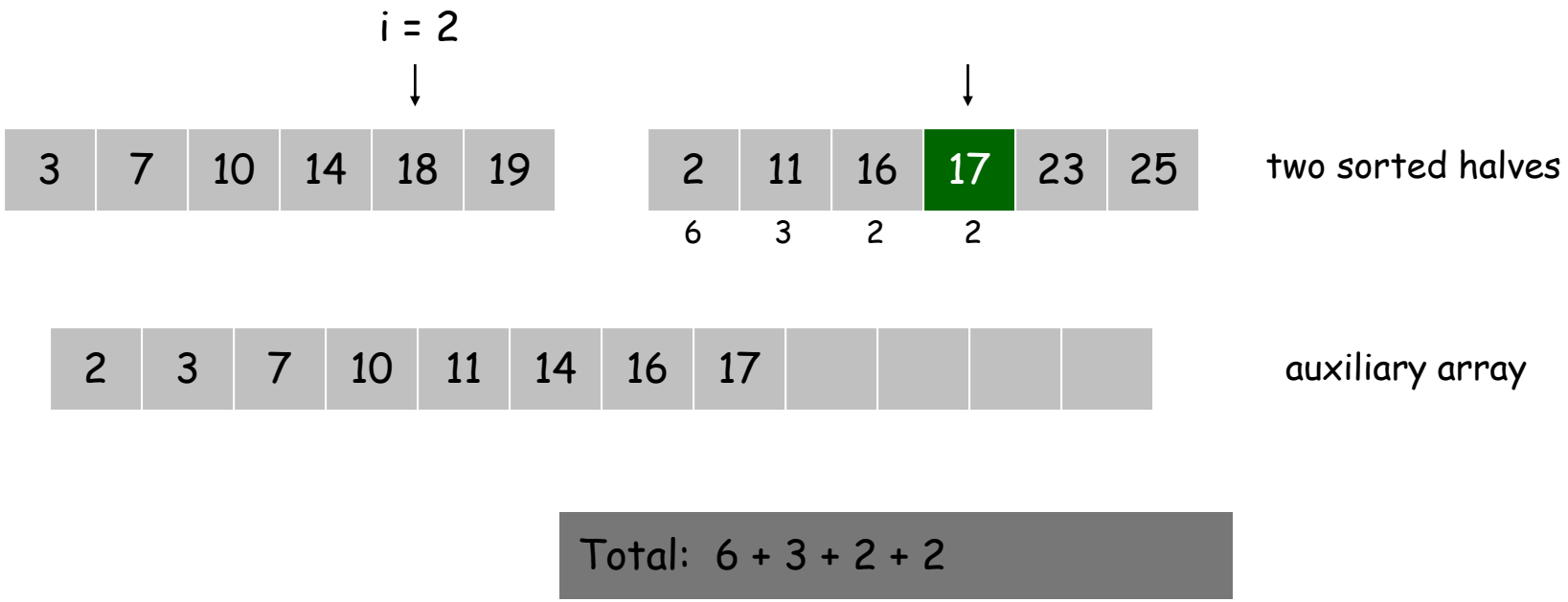
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

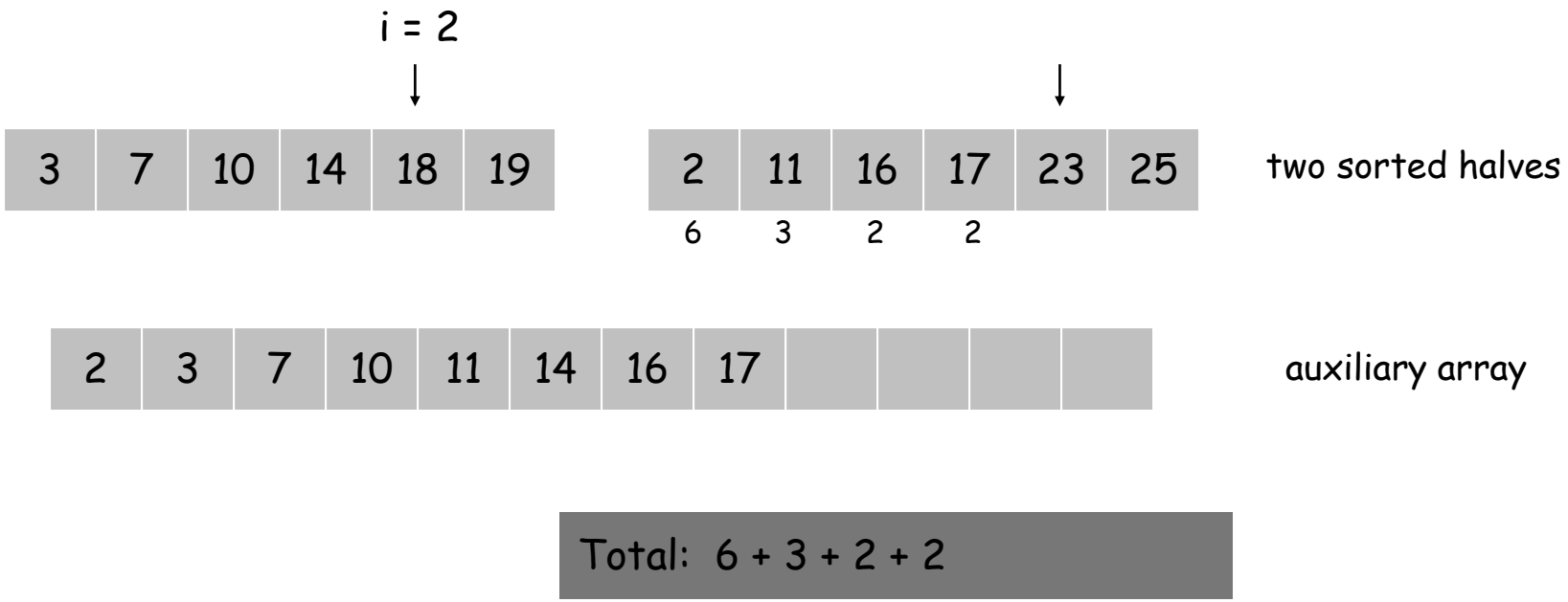
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

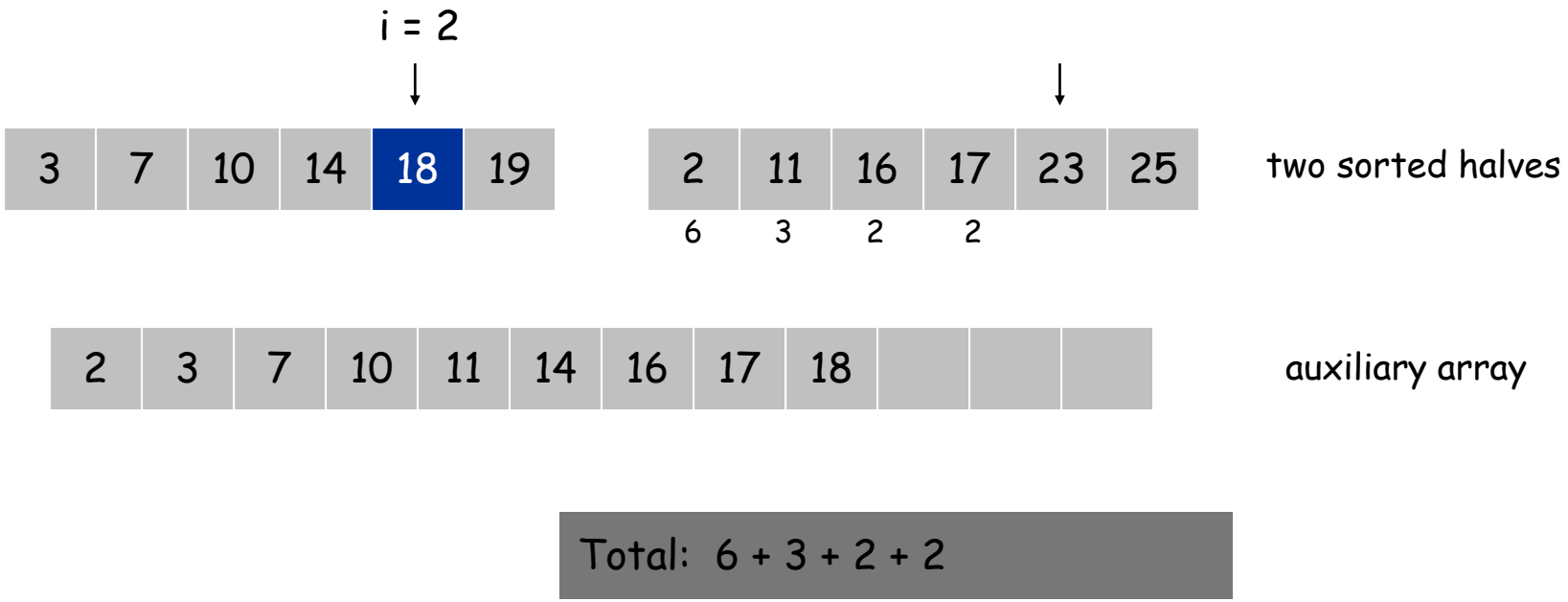
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

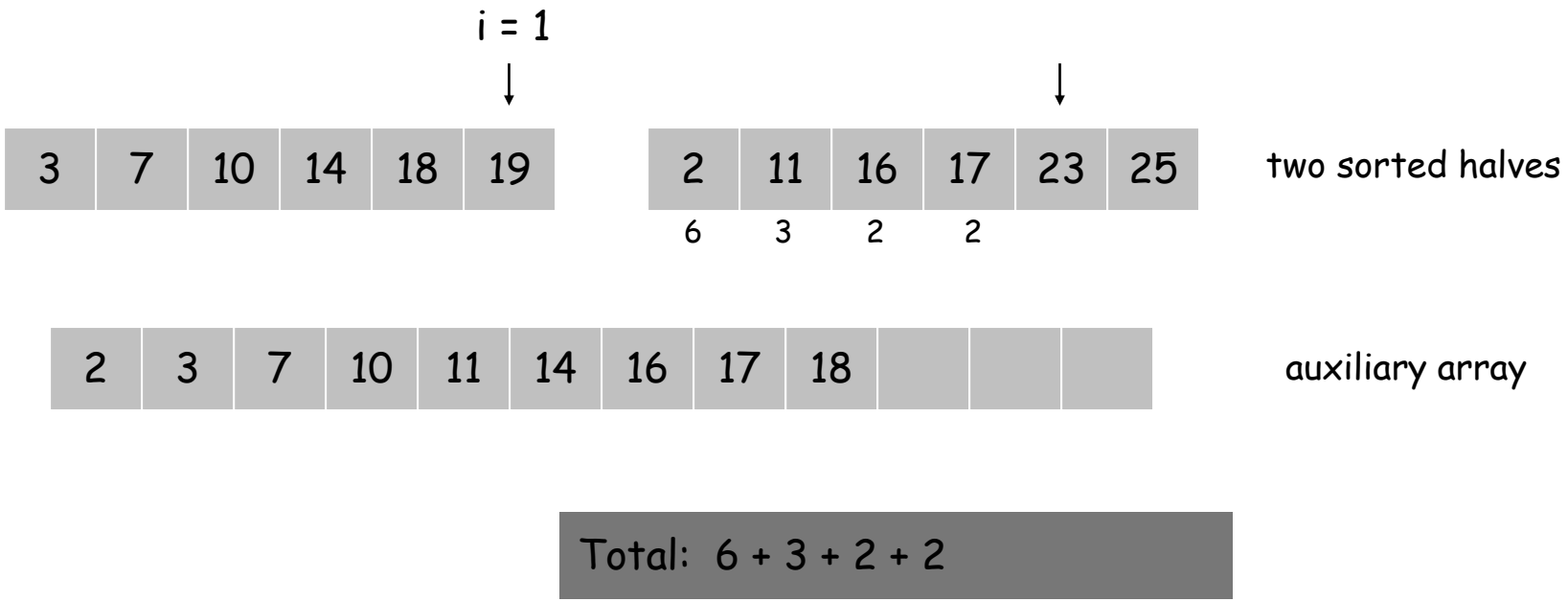
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

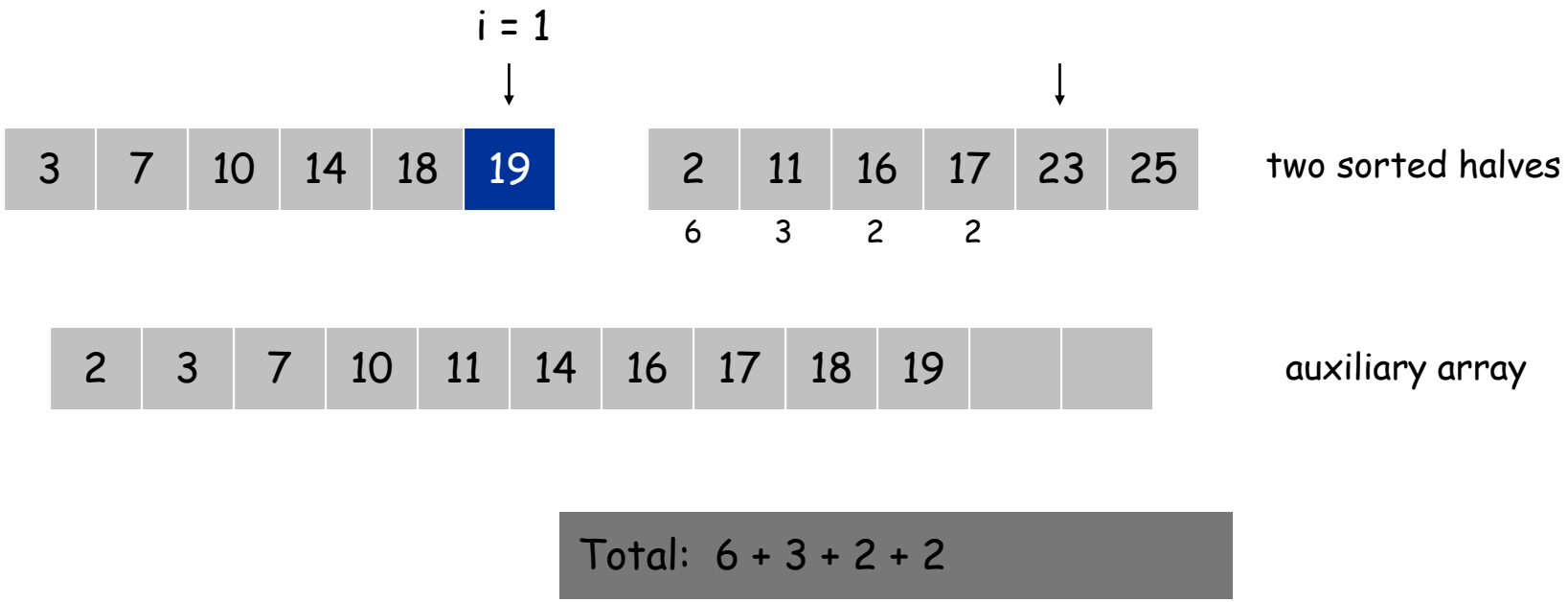
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

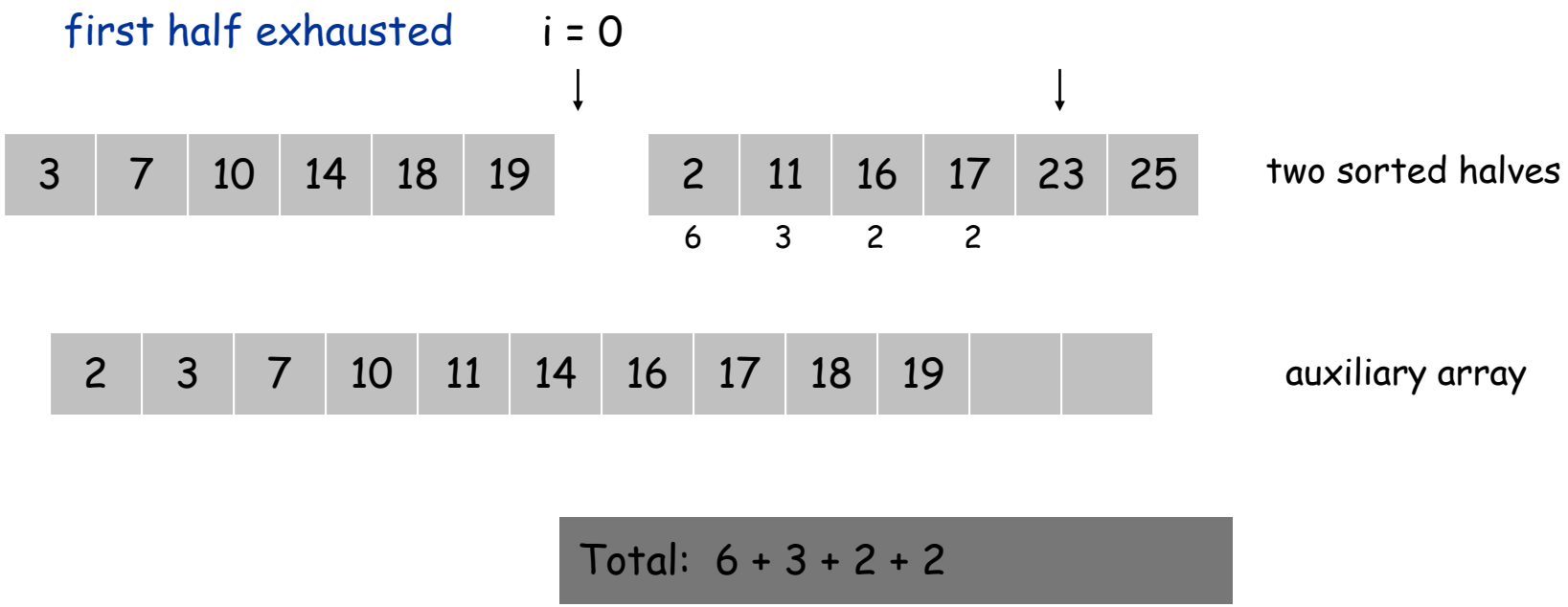
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

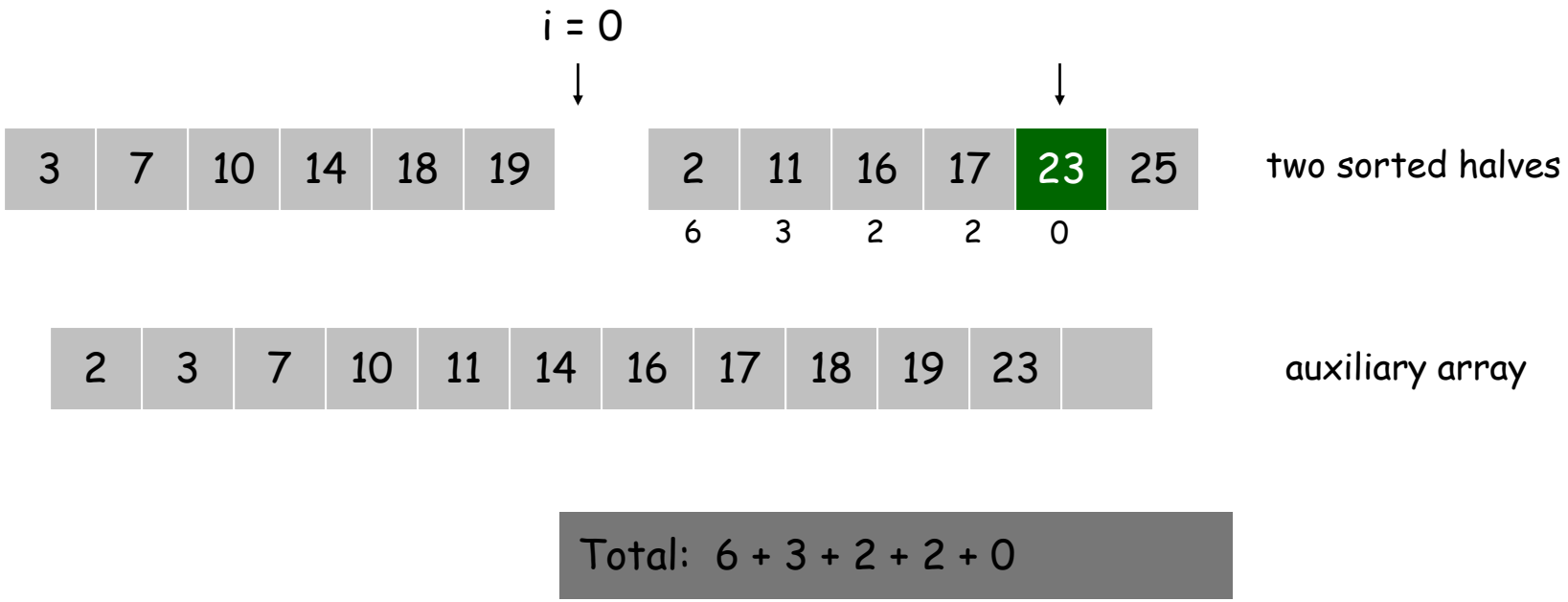
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

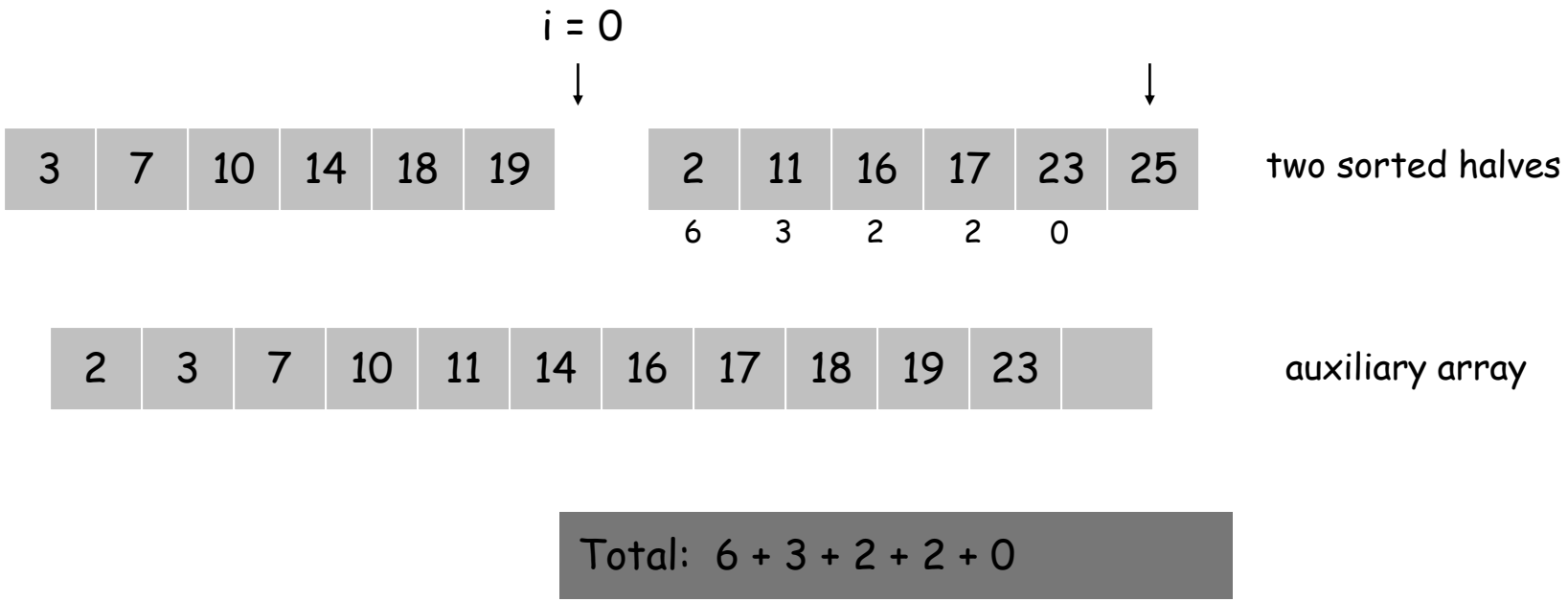
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

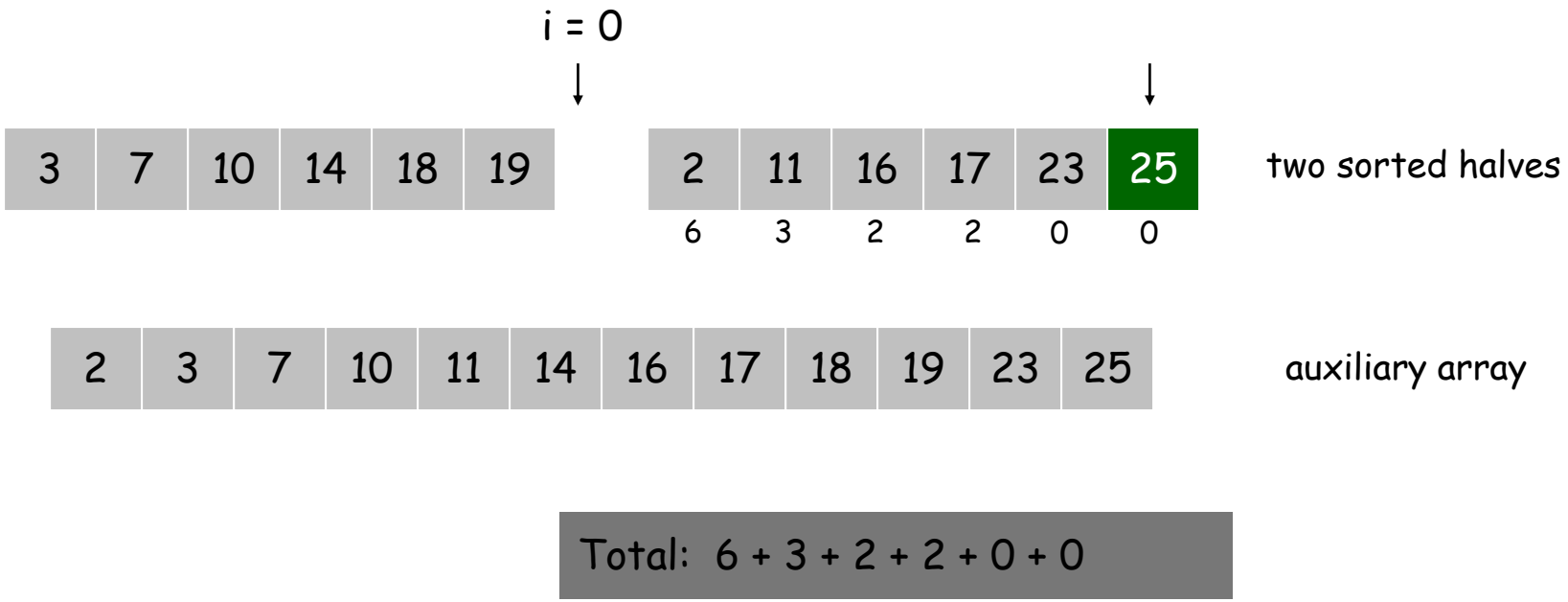
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

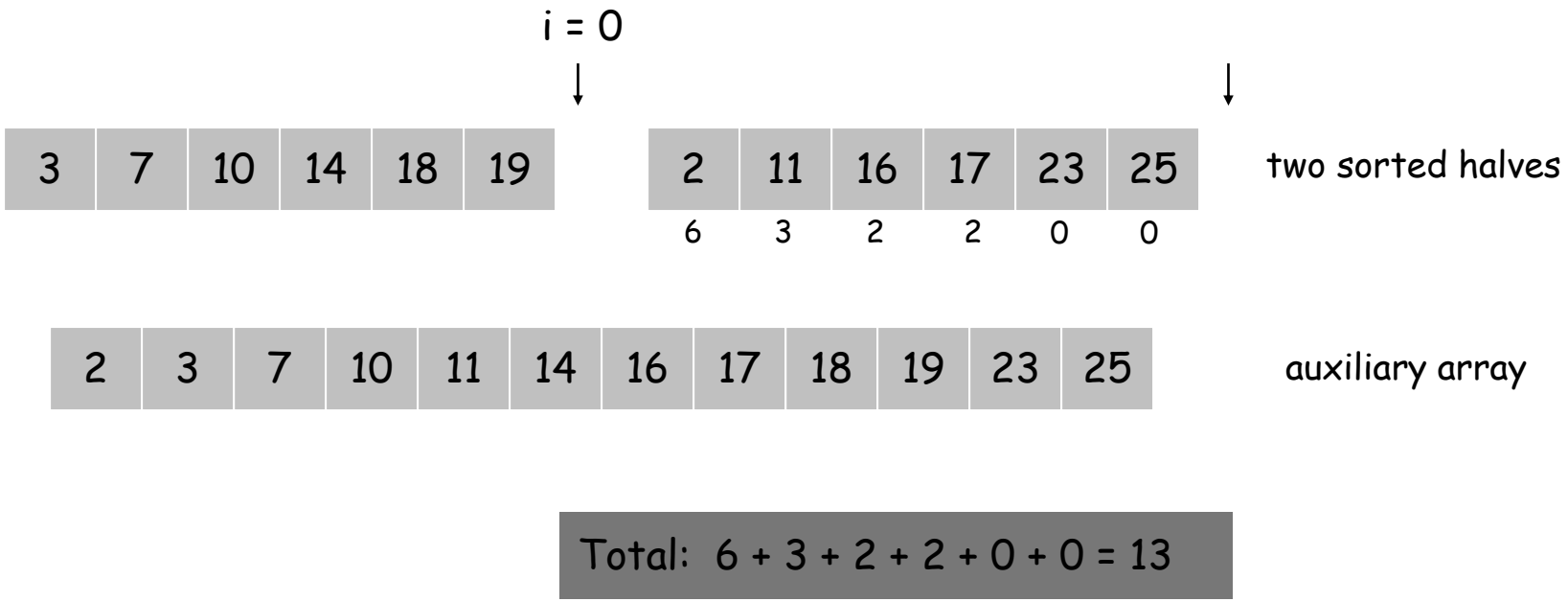
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



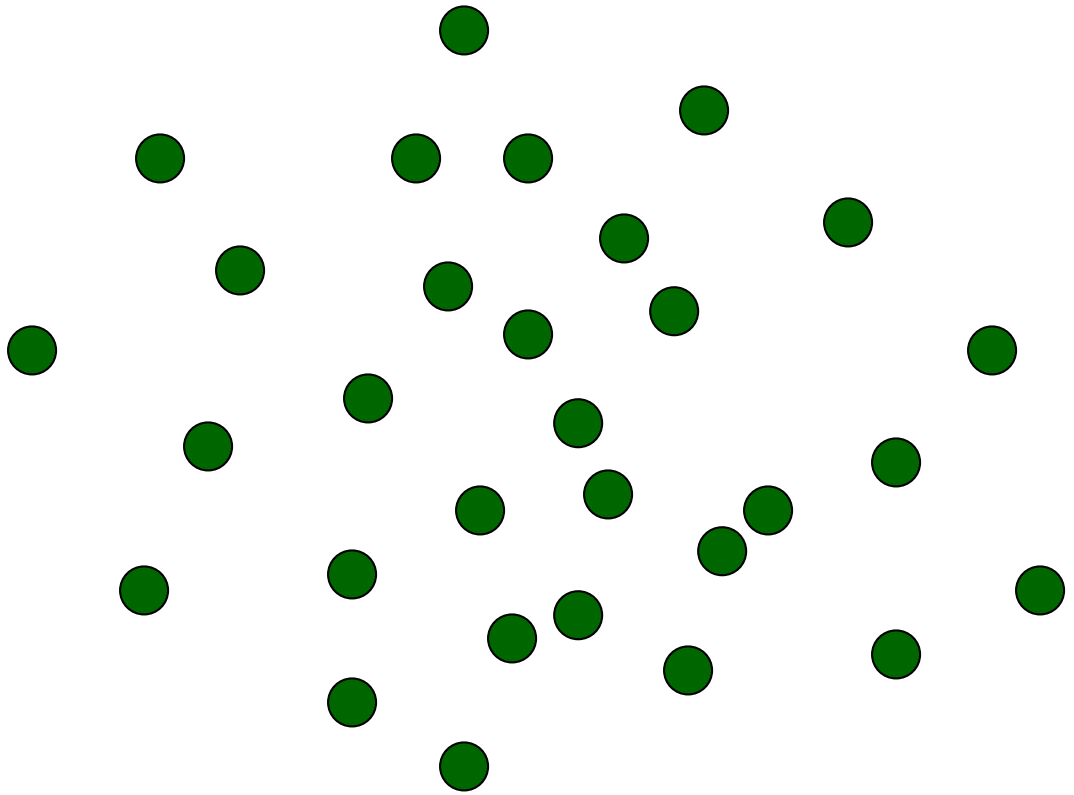
Merge and Count

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.



Closest Pair of Points



Closest Pair of Points

Given: Set of n points given as Cartesian coordinates in the plane

Goal: Find a pair of points with minimum distance.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.

Brute force: Check all pairs of points p and q with $O(n^2)$ comparisons.

Our aim: Using Divide and Conquer, $T(n) = 2T(n / 2) + O(n)$ to get to $O(n \log n)$

Closest Pair of Points

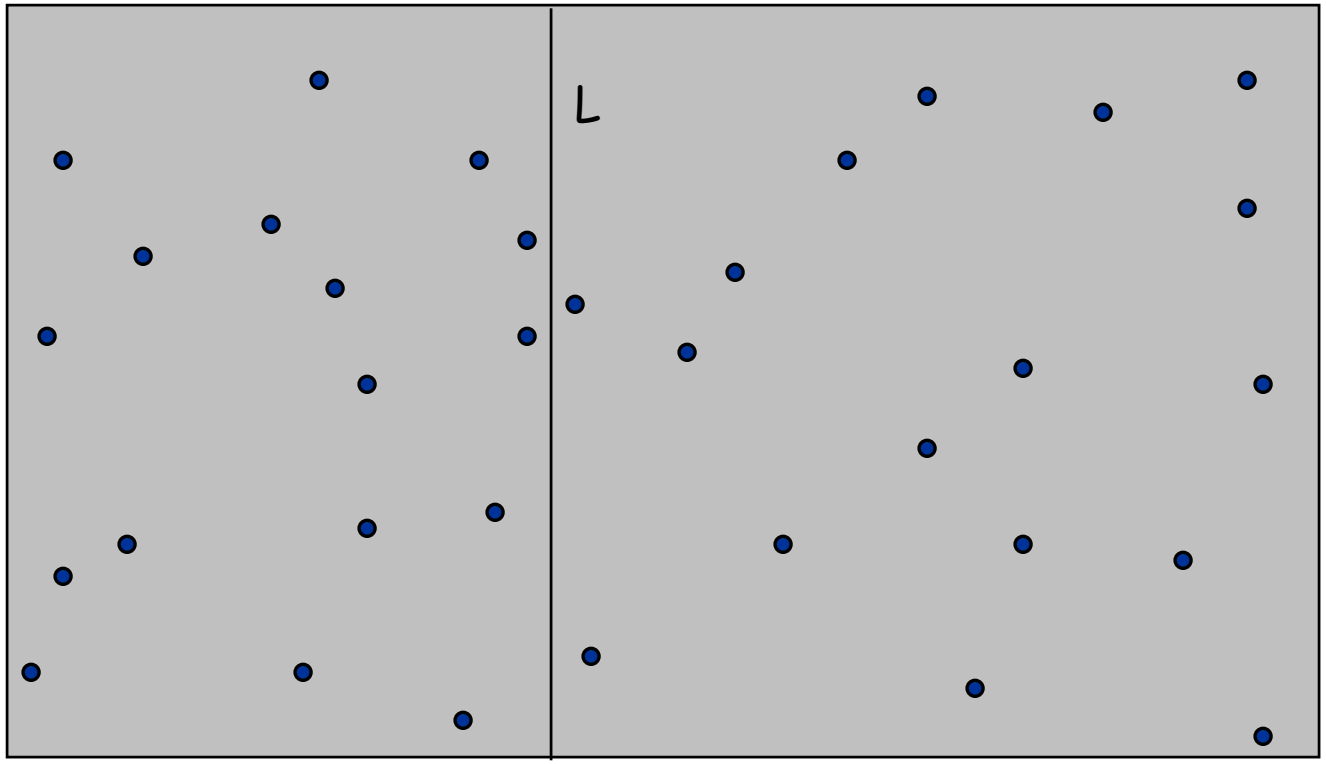
How it works:

Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.

Points sorted by their x-coordinate ⟵ Why sorting? find median directly in O(n)

Take the median z of all x-values and put all points with coordinate $x < z$ and $x > z$, respectively in two sets, S_1 and S_2

Compute the closest pairs in both S_1 and S_2 recursively.



Closest Pair of Points

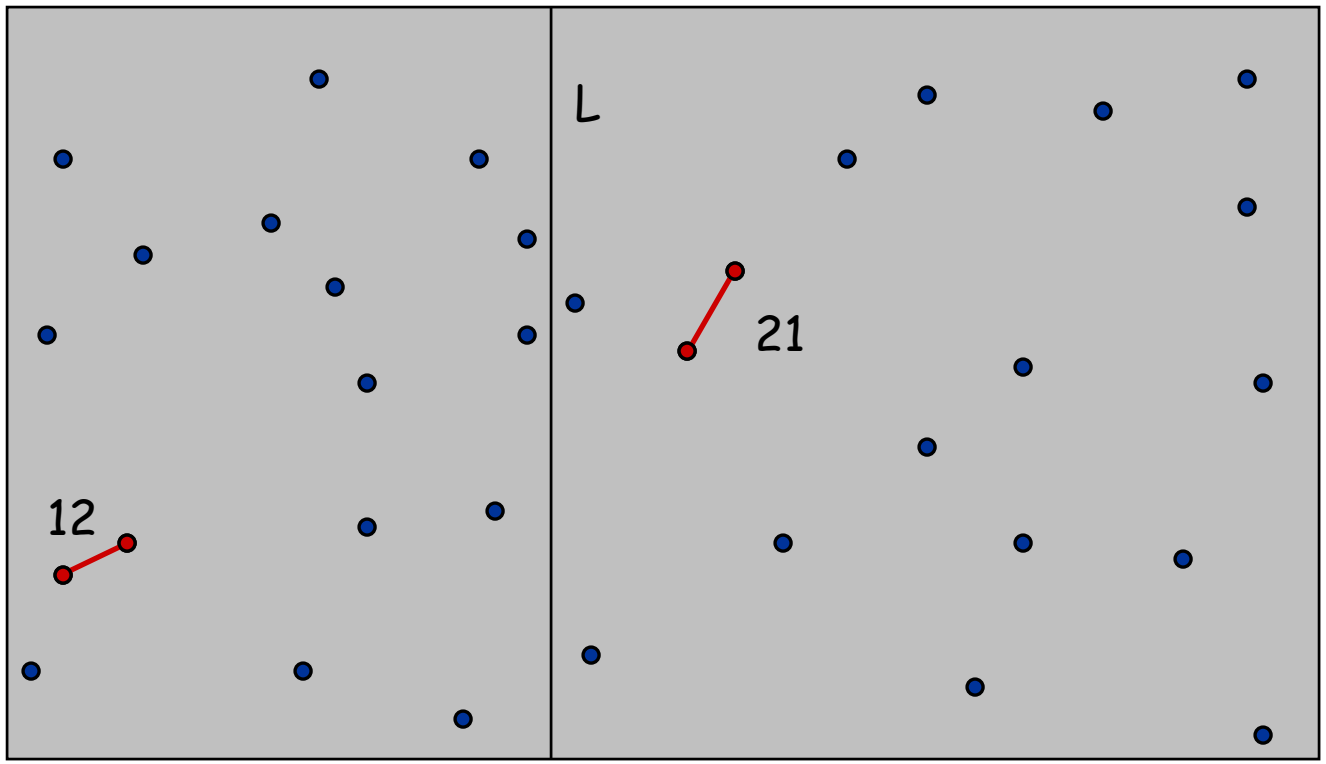
How it works:

Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.

Points sorted by their x-coordinate ⟵ Why sorting? find median directly in O(n)

Take the median z of all x-values and put all points with coordinate $x < z$ and $x > z$, respectively in two sets, S_1 and S_2

Compute the closest pairs in both S_1 and S_2 recursively.



Closest Pair of Points

How it works:

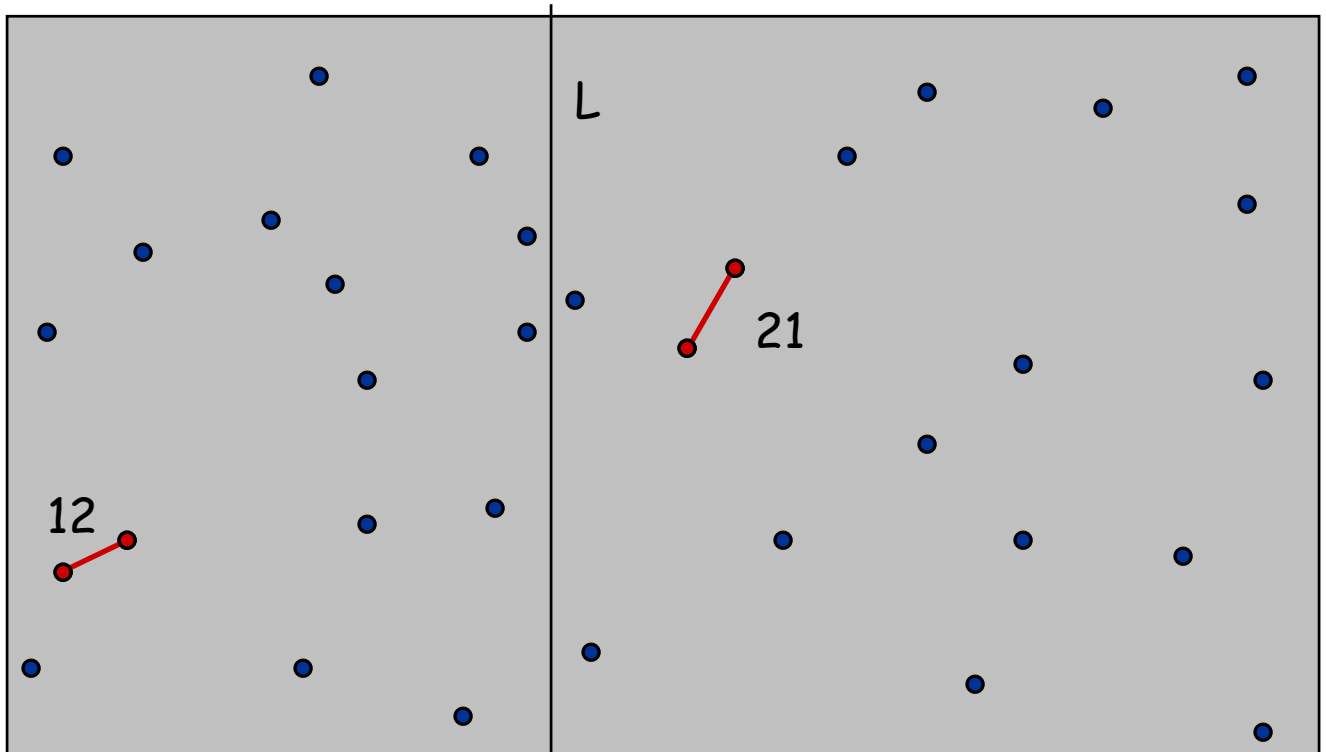
Draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.

Points sorted by their x-coordinate ⟵ Why sorting? find median directly in O(n)

Take the median z of all x-values and put all points with coordinate $x < z$ and $x > z$, respectively in two sets, S_1 and S_2

Compute the closest pairs in both S_1 and S_2 recursively.

Compute closest pairs where one point is in S_1 while the other in S_2 ???

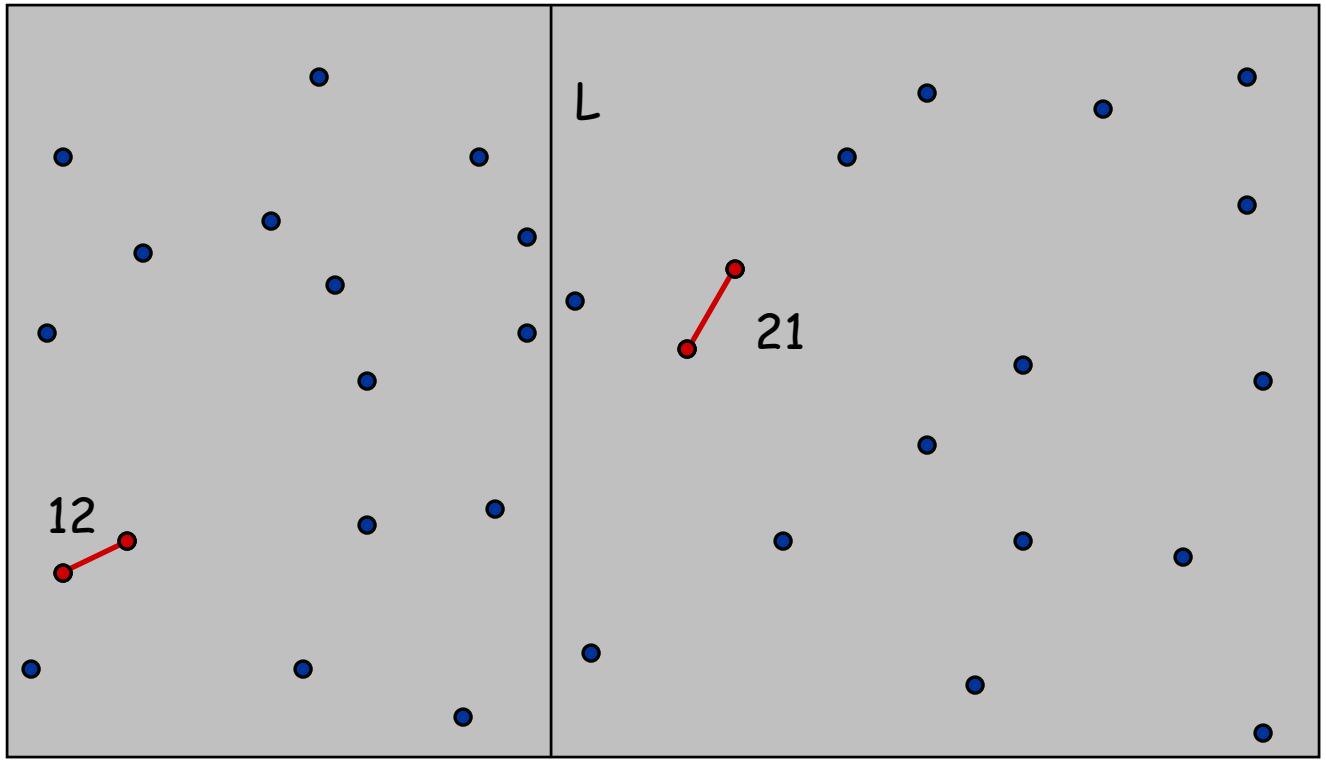


Closest Pair of Points

Let d_1 be the distance between the closest pair in S_1

Similarly d_2 for the set S_2

$d := \min(d_1, d_2)$



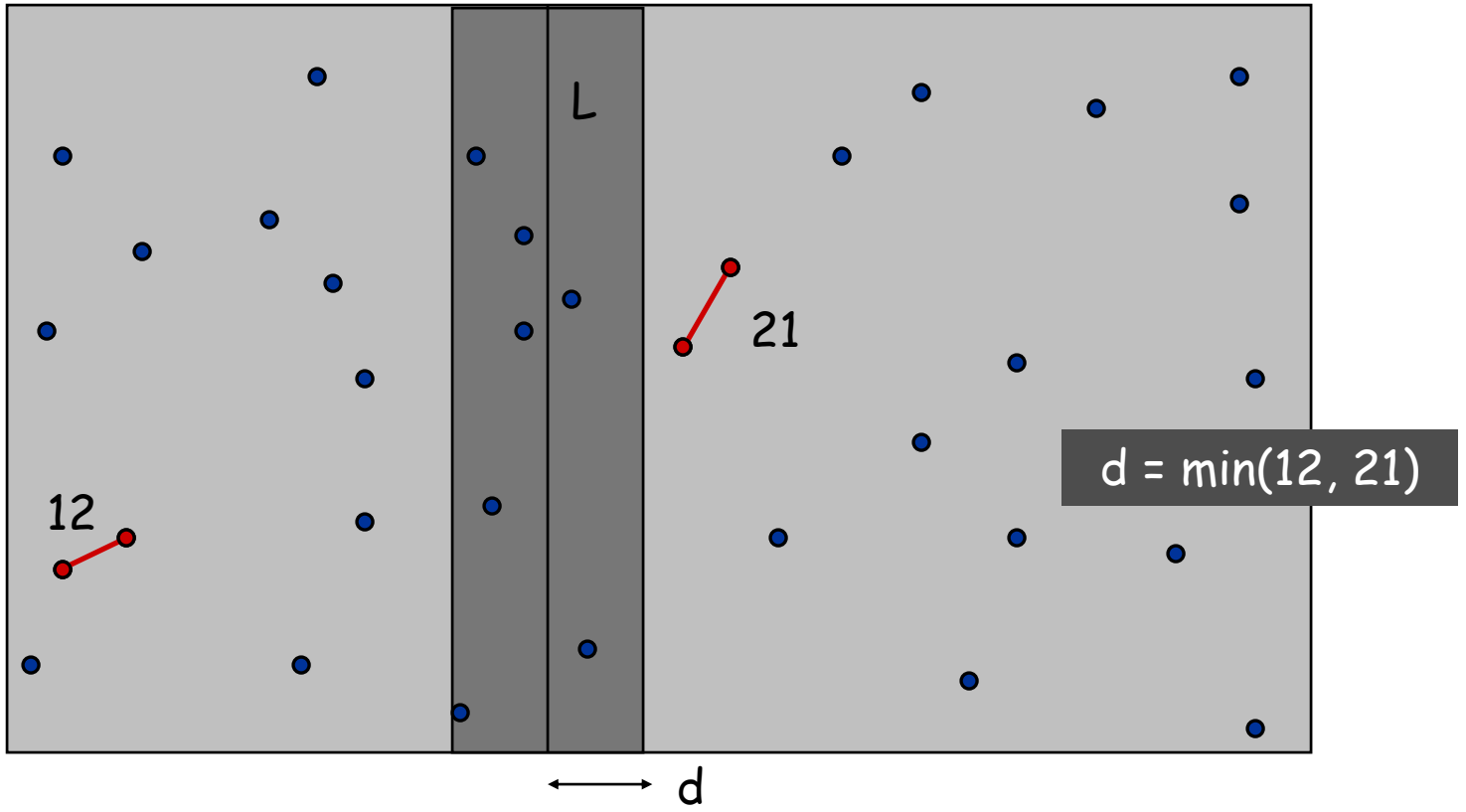
Closest Pair of Points

Find closest pair with one point in S_1 while the other in S_2

Because we already have closest pair with distance d

The candidates for such pairs of points are in a stripe of breadth d on both sides of the separating line L

Consider the pairs with one point in each set, assuming that distance < d.



Closest Pair of Points

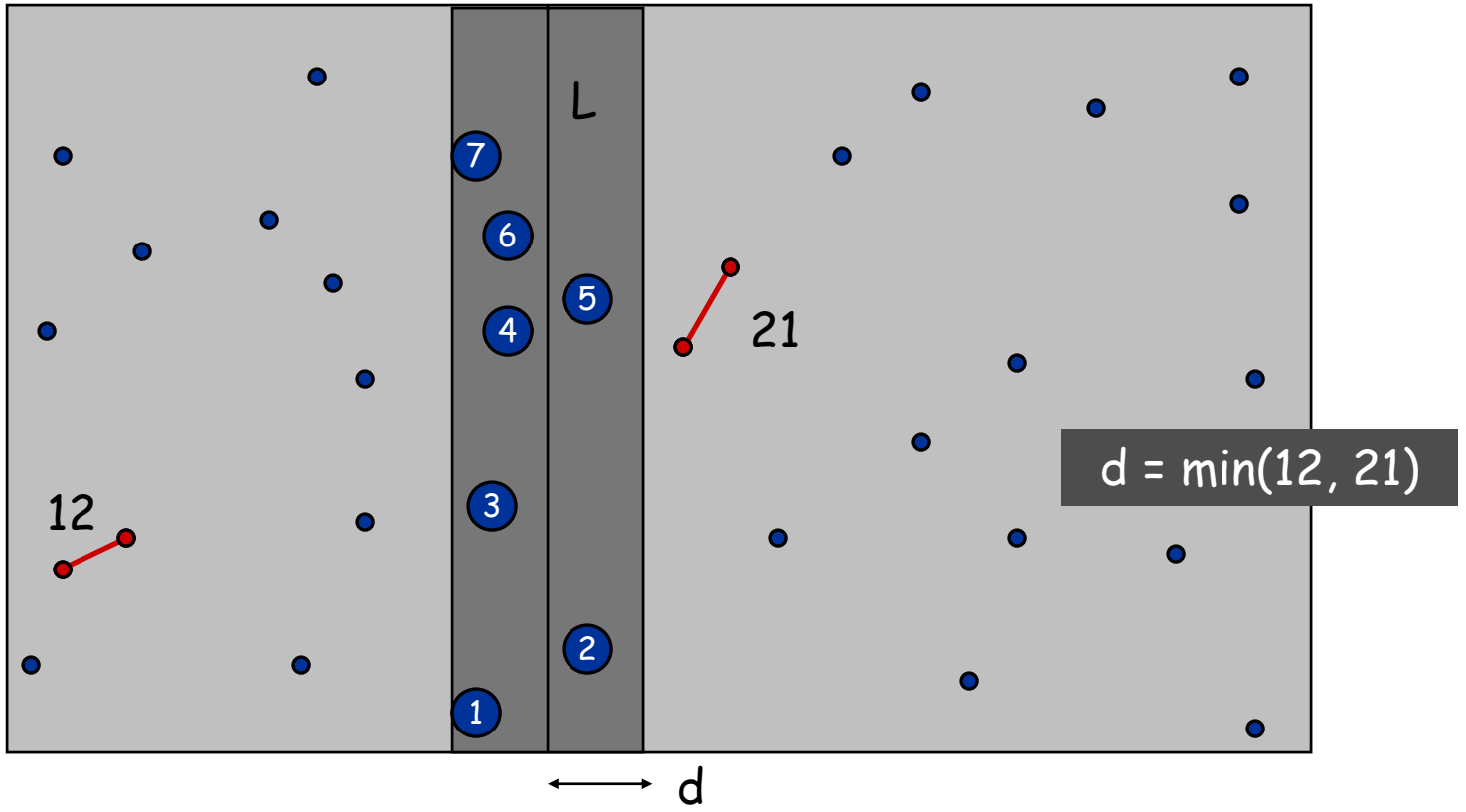
Find closest pair with one point in S_1 while the other in S_2

Sort points in 2d-strip by their y coordinate.

Only check distances of those within 11 positions in sorted list! (d is 12)

Managed everything in O(n) for the conquer step.

Finally, standard recurrence T(n) = 2T(n / 2) + O(n) with solution O(n log n)



Multiplication of large integers

```
    209506709303468099431859684686779409766717133476767930
X   592017509177763470967767934292909701230895667999301092 1
-----------------------------------------------------------------

    971548028394508438309485670104364384579021796570295676 7
+   124243109823409905732907509717989843092877957927759797 7
-----------------------------------------------------------------
```

Multiplication of large integers

a , b are both n-digit integers

If we use the brute-force approach to compute c = a * b, what is the time efficiency?

Remember the "School Algorithm"

addition of n integers, each with O(n) digits requires O (n^2)

Already discussed in Lecture 1.

$$
\begin{array}{r}
5678 \cdot 4321 \\
\hline
22712000 \\
01703400 \\
00113560 \\
00005678 \\
\hline
24534638
\end{array}
$$

Divide and Conquer: Can we do better?

Split the decimal representations of the factors a and b into two halves,

and to multiply, use the distributive law.

3 × 6

= 3 × (2 + 4)

= 3×2 + 3×4

Multiplication of large integers

$a = a_1a_0$ and $b = b_1b_0$

$c = a * b$

$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

For instance: a = 123456, b = 117933:

Then $c = a * b = (123*10^3+456)*(117*10^3+933)$

$=(123 * 117)10^6 + (123 * 933 + 456 * 117)10^3 + (456 * 933)$

What we achieve:

multiplication of n-digit numbers is reduced to:

several multiplications of n/2-digit numbers and some additions

In fact: we need 4 multiplications involving a_1,a_0,b_1, and b_0

Multiplications with 10^n and $10^{n/2}$ are trivial: Append the required number of 0s.

Multiplication of large integers

$a = a_1a_0$ and $b = b_1b_0$

$c = a * b$

$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

Solve by recursive application of the above:

At every recursion step: problem reduces to 4 recursive calls AND addition

Clearly: $T(n) = 4T(n / 2) + O(n)$,

$a = 4$, $b = 2$, $k = 1$. Master Theorem implies: $T(n) = O(n^{\log_2 4}) = O(n^2)$

Still same like Brute Force… we need to avoid 4 recursive calls at each level

Multiplication of large integers

$a = a_1a_0$ and $b = b_1b_0$

$c = a * b$

$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

Can we achieve?

multiplication of n-digit numbers by three multiplications of n/2-digit numbers + O(n)

<presenting the idea using the rectangle example>

Multiplication of large integers

$a = a_1a_0$ and $b = b_1b_0$

$c = a * b$

We are aiming at:

$c = c_2 10^n + c_1 10^{n/2} + c_0$, where computing each of c_2, c_1 and c_0 requires ONE multiplication

This is achievable as follows:

$c = (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

$= c_2 10^n + c_1 10^{n/2} + c_0$,

where

$c_2 = a_1 * b_1$ is the product of their first halves

$c_0 = a_0 * b_0$ is the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Multiplication of large integers

Finally
$c = c_2 10^n + c_1 10^{n/2} + c_0$,
where
$c_2 = a_1 * b_1$
$c_0 = a_0 * b_0$
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

Clearly: $T(n) = 3T(n / 2) + O(n)$,
$a = 3$, $b = 2$, $k = 1$. Master Theorem implies: $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$ far better than $O(n^2)$

Caution: Factors $(a_1 + a_0)$ and $(b_1 + b_0)$ may have $(n/2) + 1$ digits
Split off the first digit and have recursive calls of $(n/2)$
Can only cause $O(n)$ extra work, won't affect the time bound

Acceleration takes effect only for rather large n (more than some 100 digits)
Recursive calls being the overhead

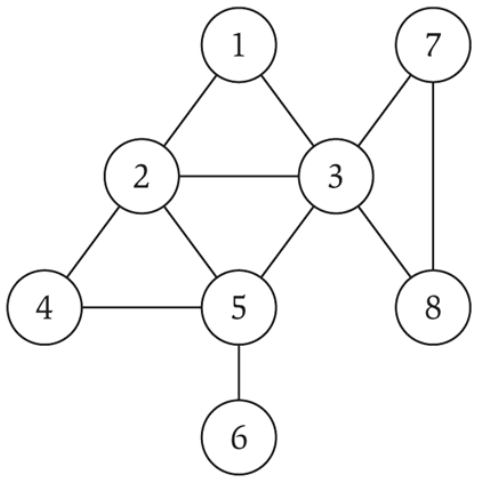
Graphs

Basic Definitions, Applications and Interesting Problems

Graphs

Graph. $G = (V, E)$

- V = nodes.
- E = edges between pairs of nodes.
- Captures binary relationships between objects (nodes).
  - Symmetric relationships: G is undirected, all edges (u, v) are unordered: (u, v) and (v, u) are identical
  - Asymmetric relationships : G is directed, All edges (u, v) are ordered: (u, v) and (v, u) are different
- Graph size parameters: $n = |V|$, $m = |E|$.
- A node and an edge are incident if the edge contains this node.
- Two vertices(nodes) joined by an edge are called adjacent.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

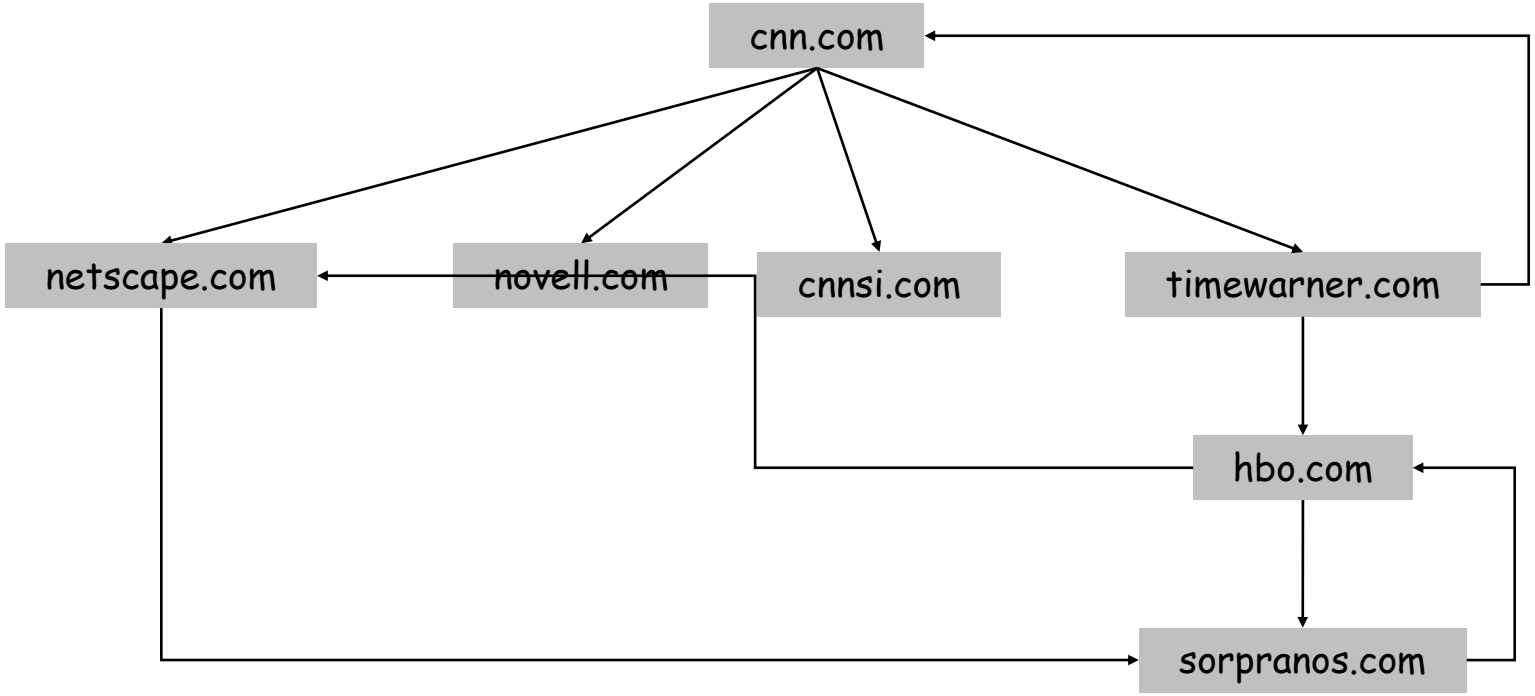
Some Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

World Wide Web

Web graph.

- Node: web page.
- Edge: hyperlink from one page to another.



Social network

Social network graph.

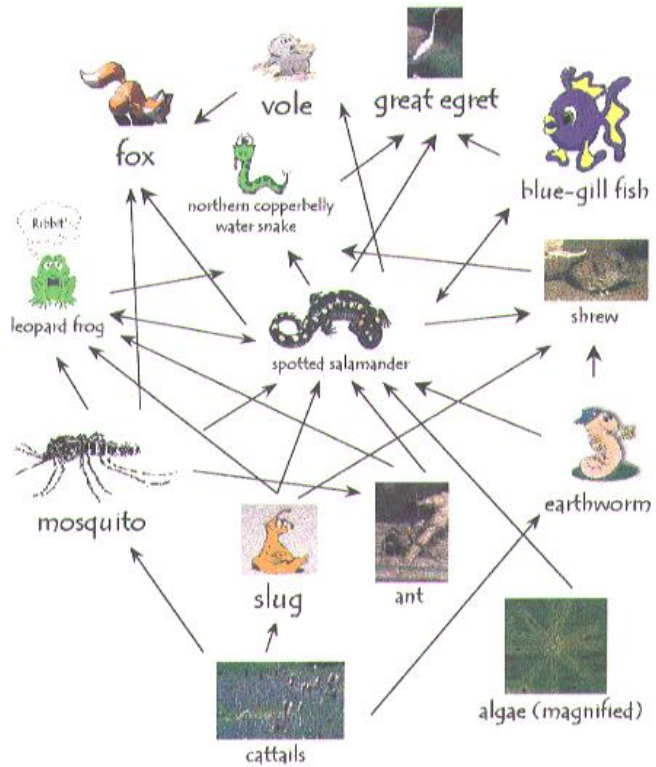
- Node: people.
- Edge: relationship between two people.

Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator. (victim to killer)

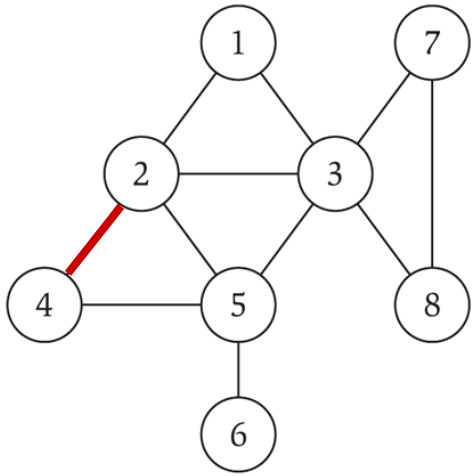


Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with A_{uv} = 1 if (u, v) is an edge, 0 otherwise

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $O(1)$ time.
- Identifying all edges takes $O(n^2)$ time.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

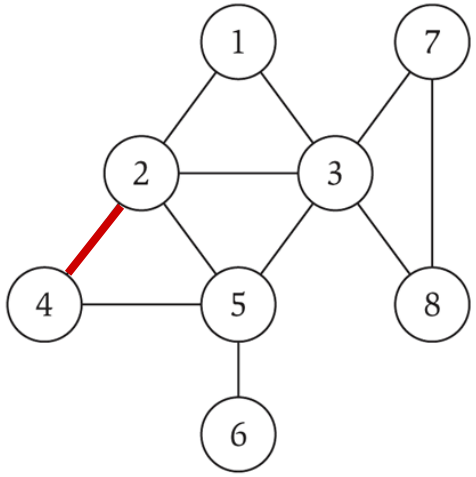
Graph Representation: Adjacency List

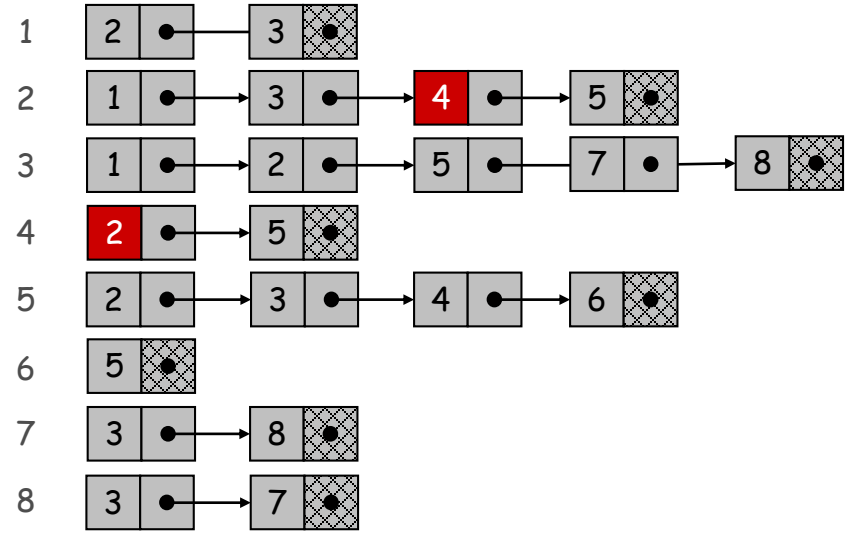
Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(deg(u)) time.

degree = number of incident edges

- Identifying all edges takes O(m + n) time.
  - In most graph algorithms, adjacency lists are preferable, as they do not waste space for non-edges.
- Nodes in Directed graphs:
  - **in-degree:** the number of incoming edges
  - **out-degree:** the number of outgoing edges.





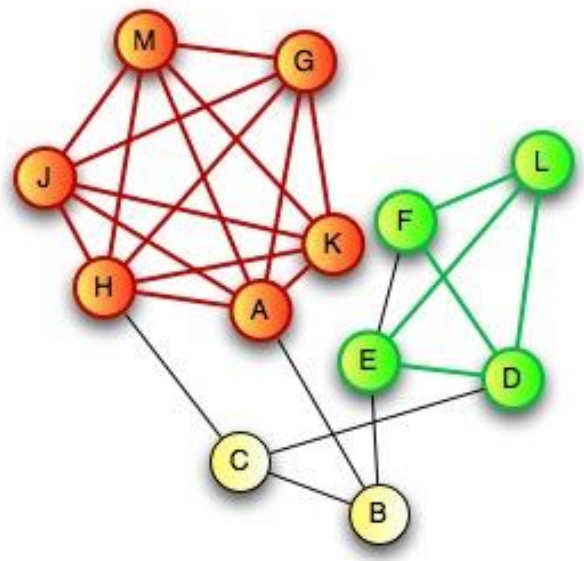
Graph Problems: Clique

Given: An Undirected Graph G.

Goal: Find a clique of maximum size.

a subset K of nodes such that there is an edge between any two nodes in K

Motivation: The graph models an interaction network (persons in a social network, proteins in a living cell, etc.), where an edge means some close relation between two "nodes".
We may wish to identify big groups of pairwise interacting "nodes".



http://scienceblogs.com/goodmath/wp-content/blogs.dir/476/files/2012/04/i-24b2db50d71be775f98de9a464113aca-maximal-cliques.jpg

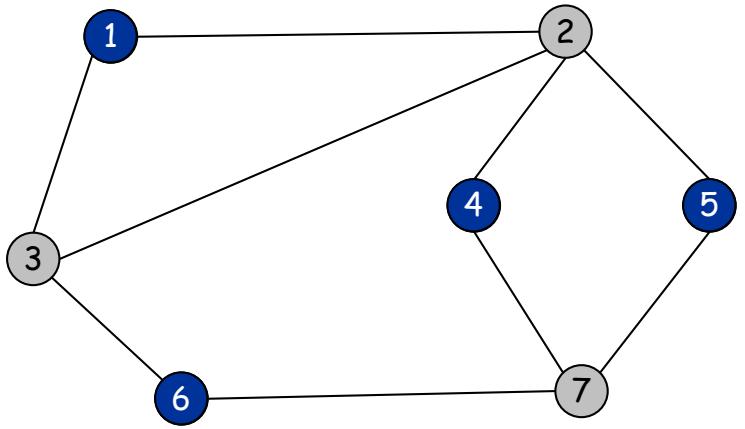
Graph Problems: Independent Set

Given: An Undirected Graph G.

Goal: Find and independent set of maximum size.

subset of nodes such that no two joined by an edge

Motivation: The graph models conflicts between items, and we wish to select as many as possible items conflict-free.



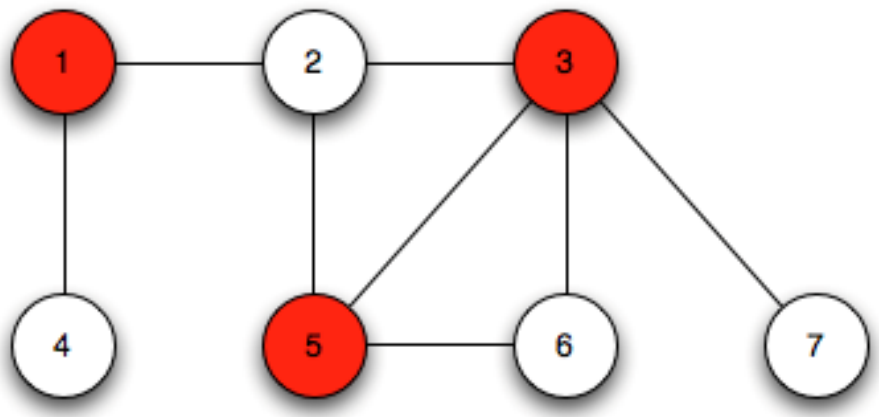
Graph Problems: Vertex Cover

Given: An Undirected Graph G.

Goal: Find a vertex cover of minimum size.

a subset of nodes such that every edge of G has at least one of its two nodes in it

Motivation: How can we place a minimum number of guards in a museum building so that they can watch all corridors?



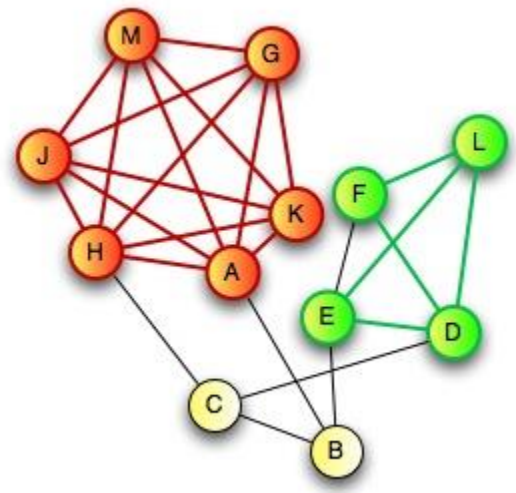
http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/images/lecture29/ApproxVert4.png

Graph Problems: Difficulty

How difficult is to solve these Graph problems (some hints):

Maximal Clique: Find a clique of maximum size.

a subset K of nodes such that there is an edge between any two nodes in K



k = 2 is trivial (every pair of adjacent nodes)

But, we need to consider all possible k.

Furthermore, for every k, we need to check all k-subsets of nodes.

In other words, we need to check all possible 2^n subsets of the node set

n could be very large…

This 2^n is not special here, same applies to e.g., Interval Scheduling…

The difference is… Our Algorithm Design Approaches seem to fail here…

Next Week: Going to be hard…

for whom…?

Of course for these problems.

However, to fight against a hard opponent, we need to work hard!