Divide and Conquer 2

CS240 Spring 2022

Rui Fan

Counting inversions

- Given a sequence $a_1, \ldots, a_n$, the pair (a_i, a_j) is an inversion if $i < j$ but $a_i > a_j$.
- Ex In 5,2,6,1,4,3, there are 9 inversions.
- Can count number of inversions in $O(n^2)$ time.
- Use divide and conquer to solve in $O(n \log n)$ time.
- Basic idea Divide sequence in half.
  - Count the number of inversions in both halves.
  - Count number of inversions between the halves.

Counting inversions

- Let L and R be the left and right halves of a sequence.
- Observation No matter how we permute L and R, the number of inversions between L and R stays the same.
  - Ex There are 7 inversions betweem 5,2,6 and 1,3,4. There are also 7 inversions between 2,5,6 and 1,3,4.
- Counting inversions between halves is easy if the halves are sorted.
  - Keep a pointer i for L, j for R, initially both 0.
  - If $L_i > R_j$, increment j.
    - Also increment number of inversions by $|L| - i$, because $L_k > R_j$ for all $k \geq i$, because L and R are sorted.
  - Otherwise increment i.
  - Just like merging L and R.
  - Takes $O(n)$ time, where $n = |L| + |R|$.

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_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

6	9	12	11	3	7

Conquer: 2T(n / 2)

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

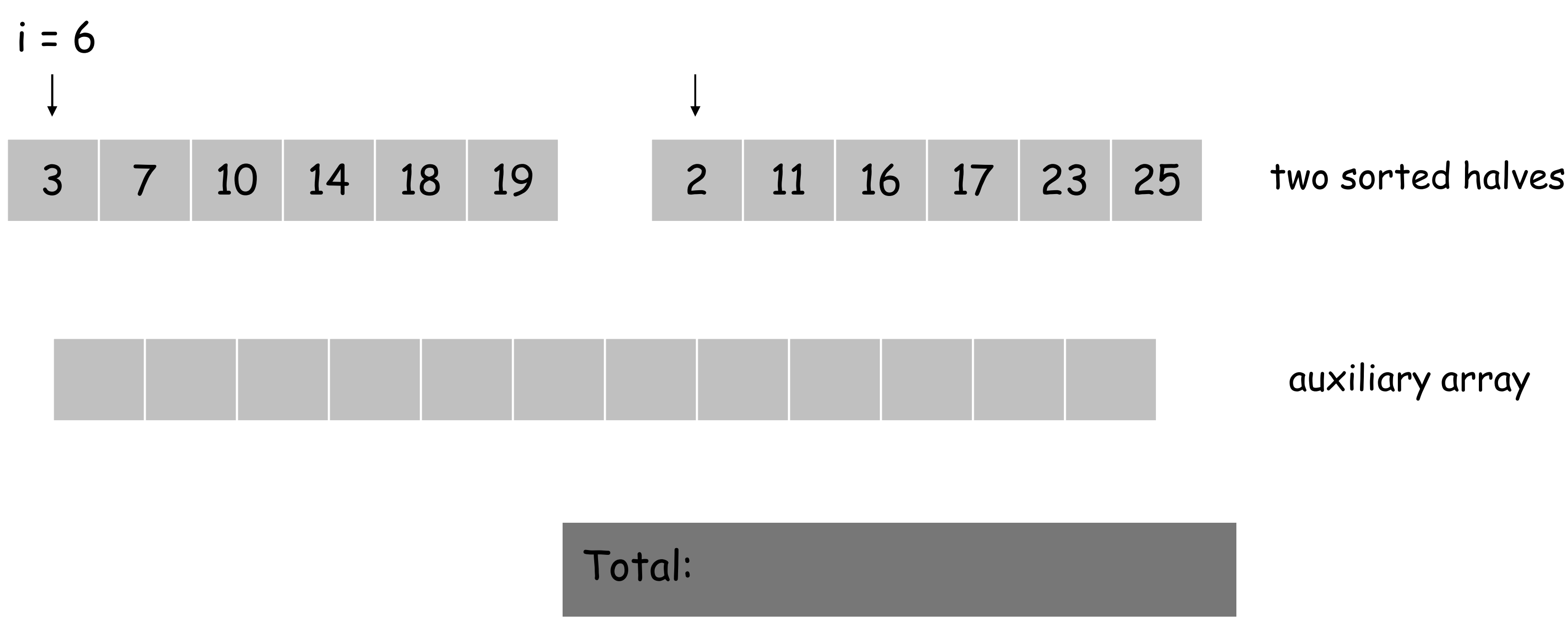
Combine: ???

Total = 5 + 8 + 9 = 22.

Merge and Count

Merge and count step.

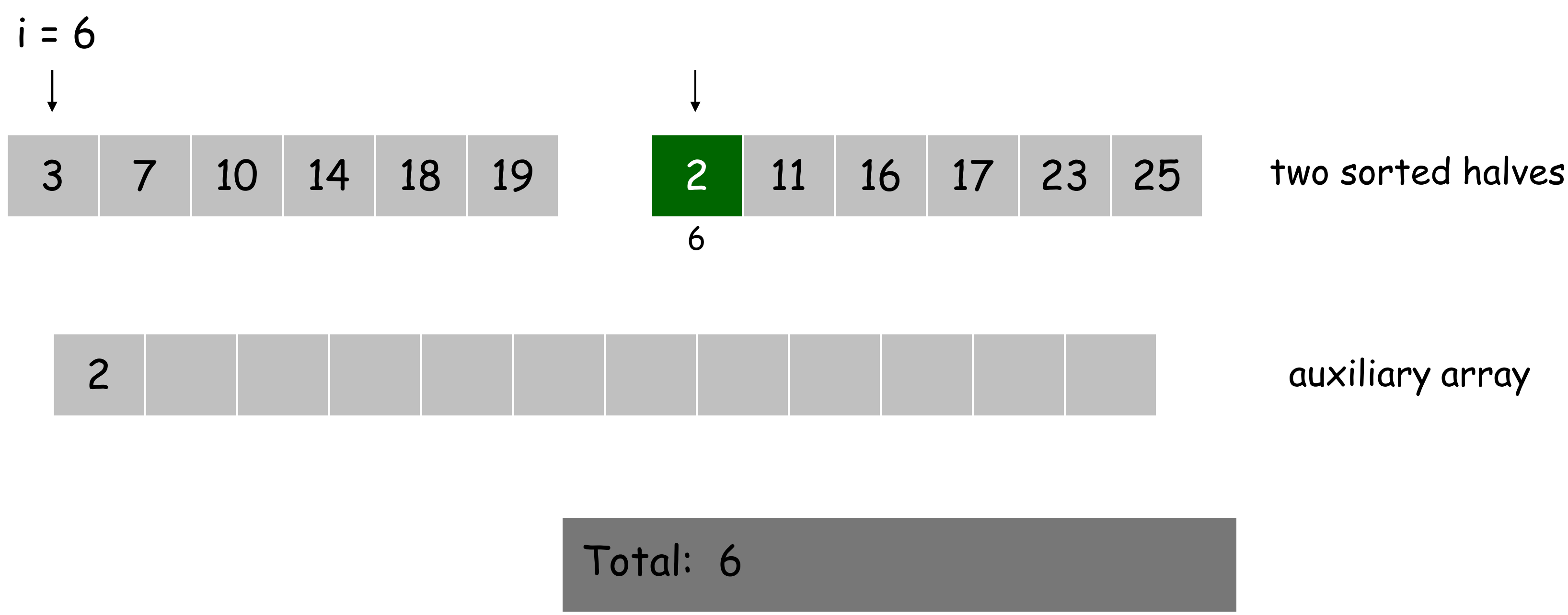
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

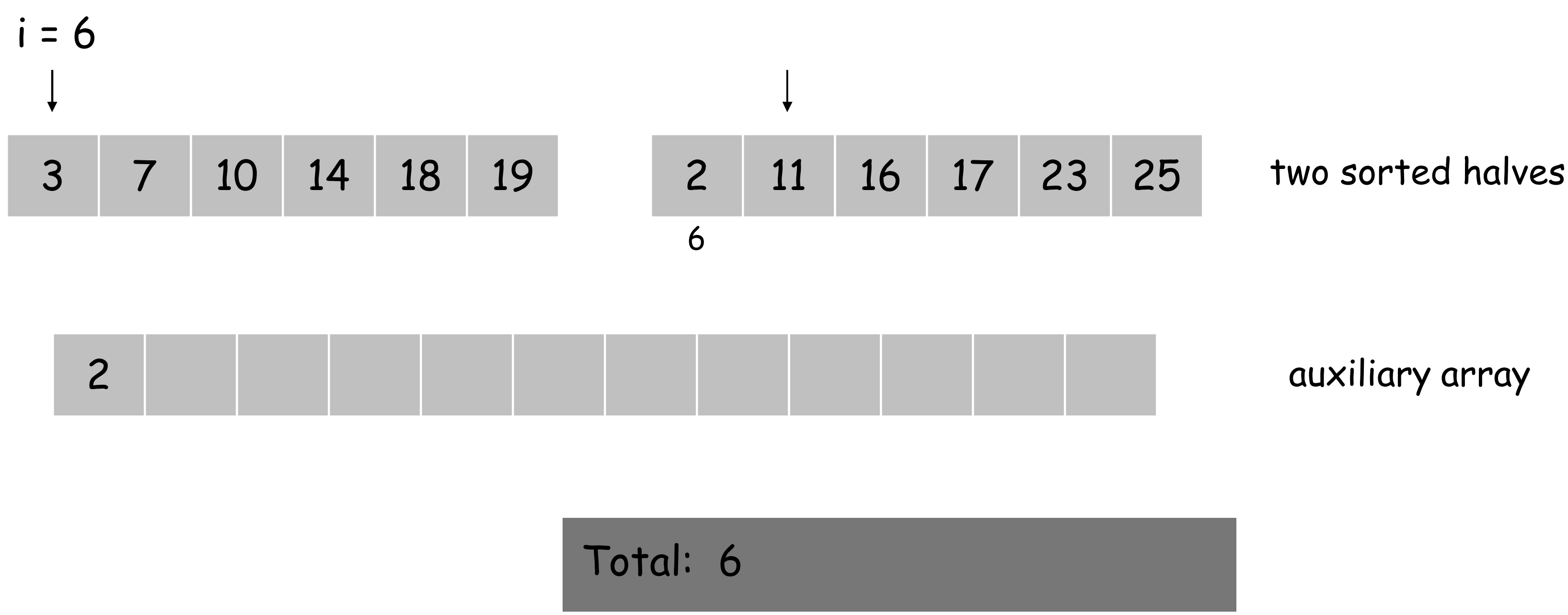
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

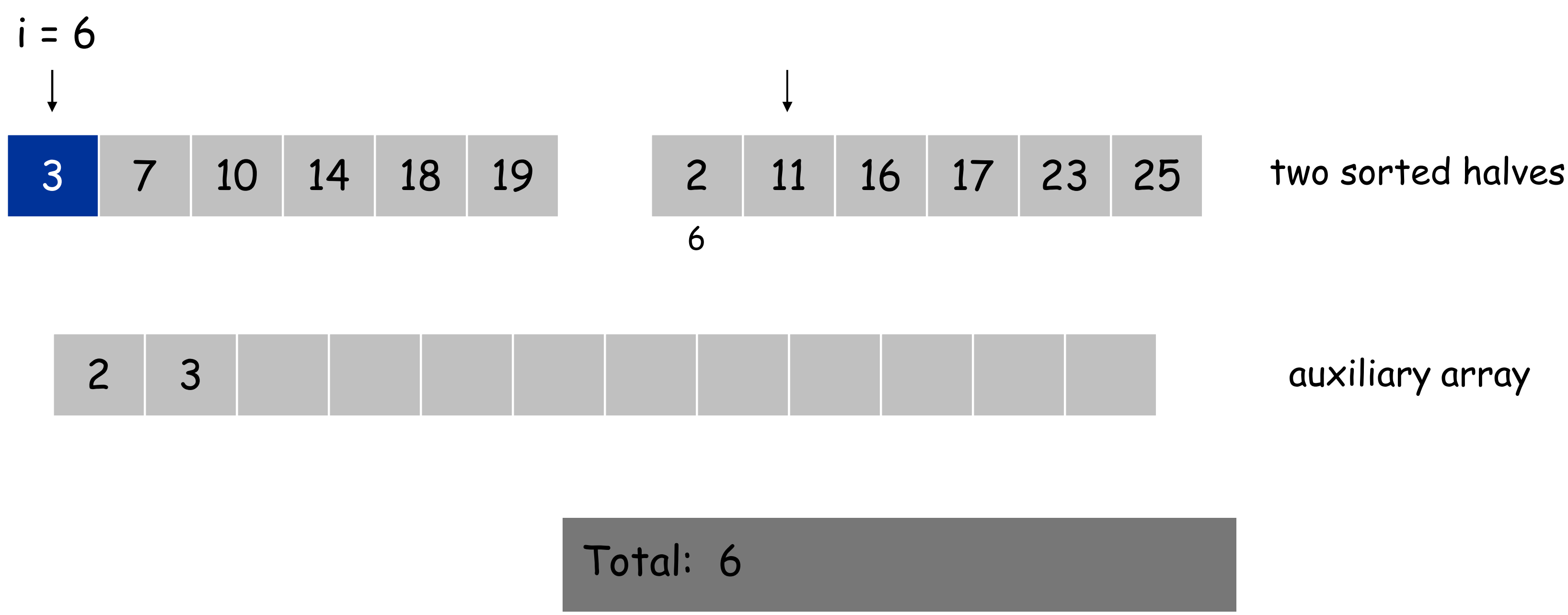
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

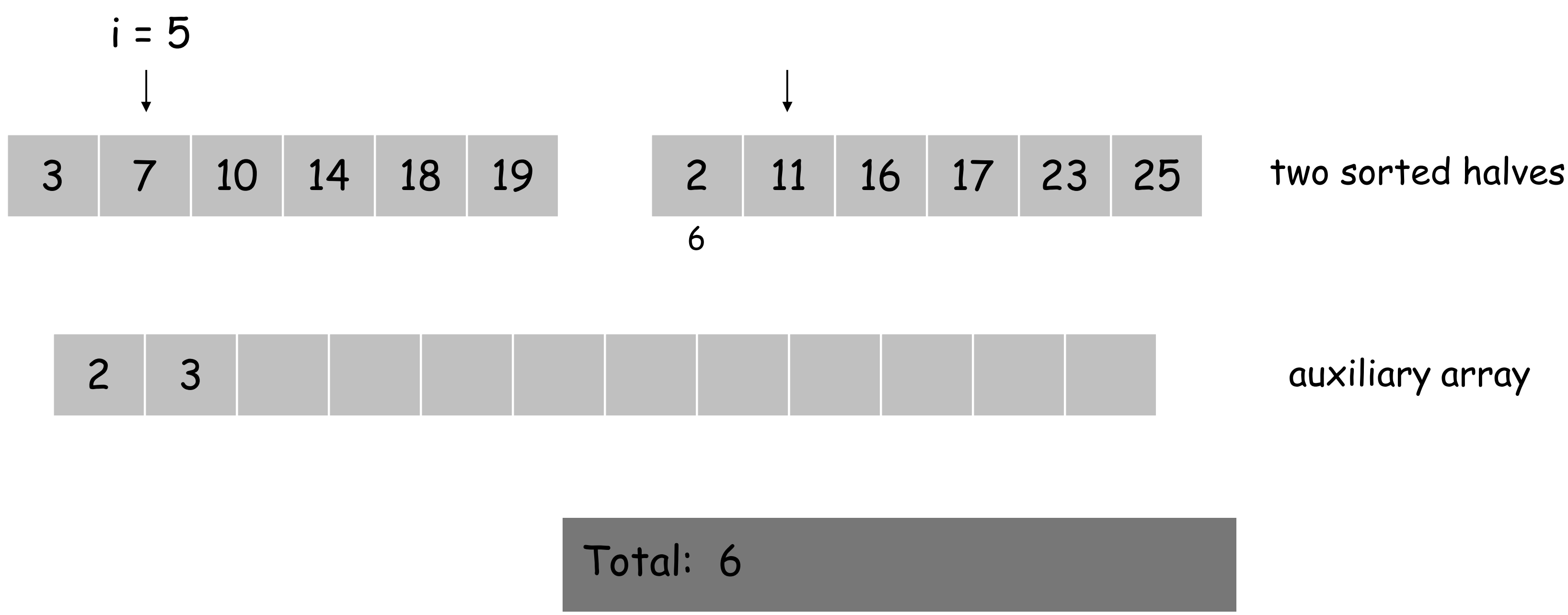
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



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.

i = 5

two sorted halves

3	7	10	14	18	19

2	11	16	17	23	25
6					

auxiliary array

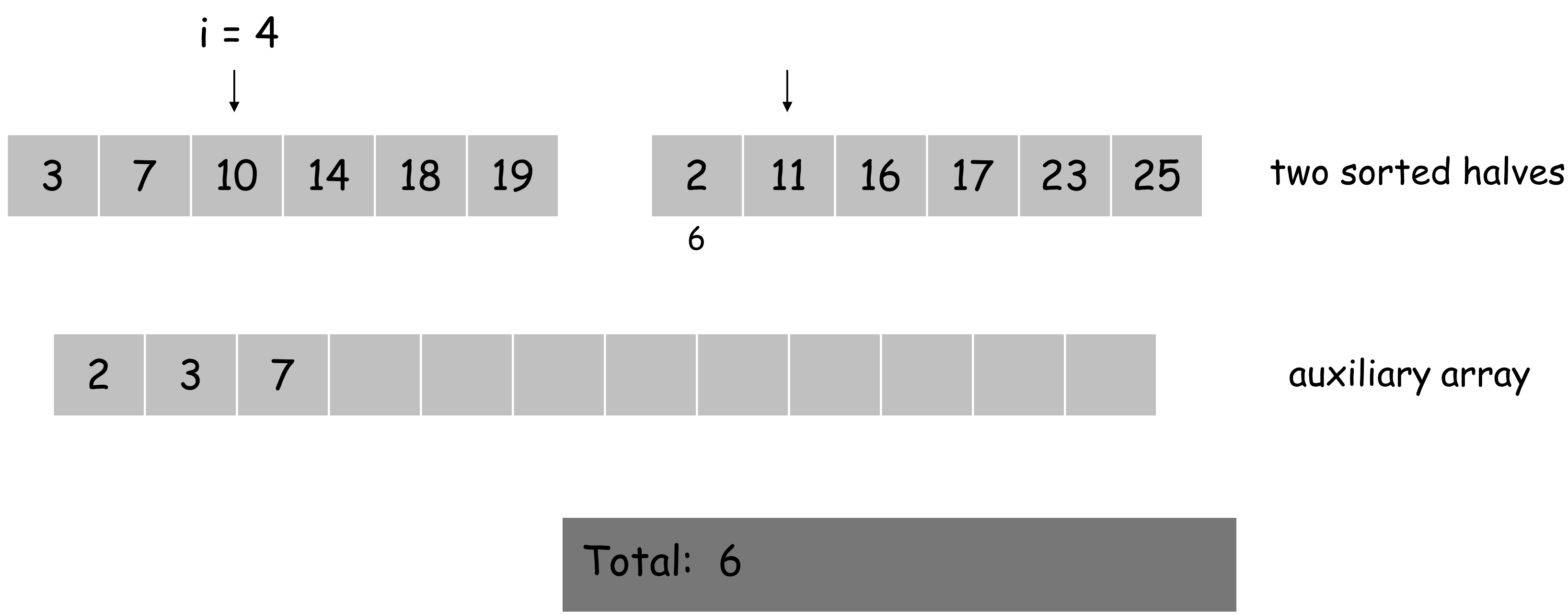
2	3	7									

Total: 6

Merge and Count

Merge and count step.

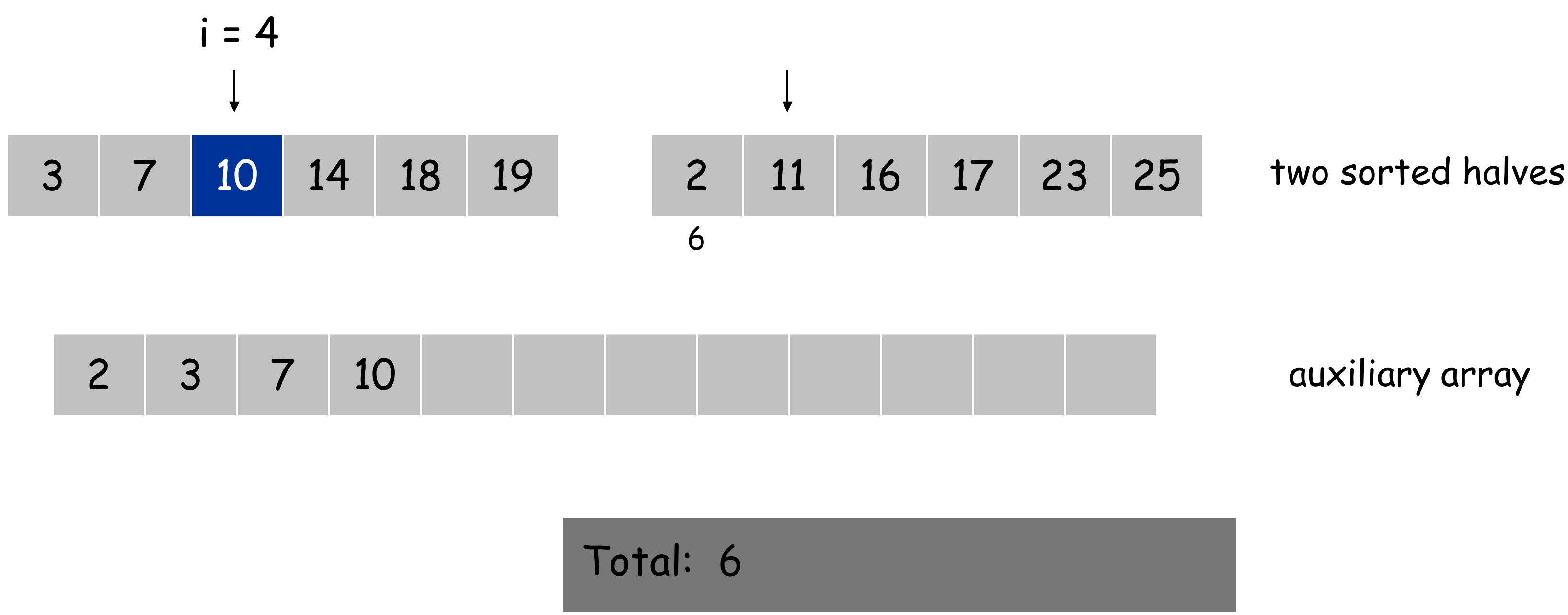
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



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 = 3

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

two sorted halves

2	3	7	10								

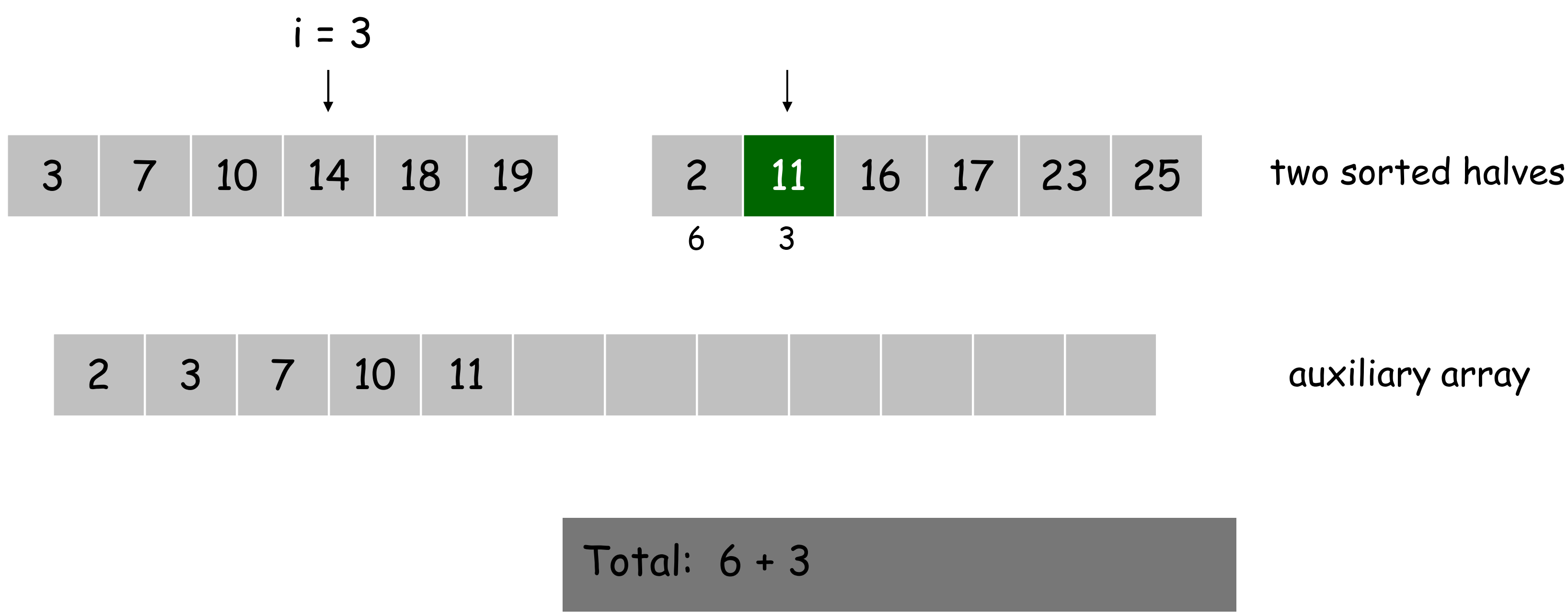
auxiliary array

Total: 6

Merge and Count

Merge and count step.

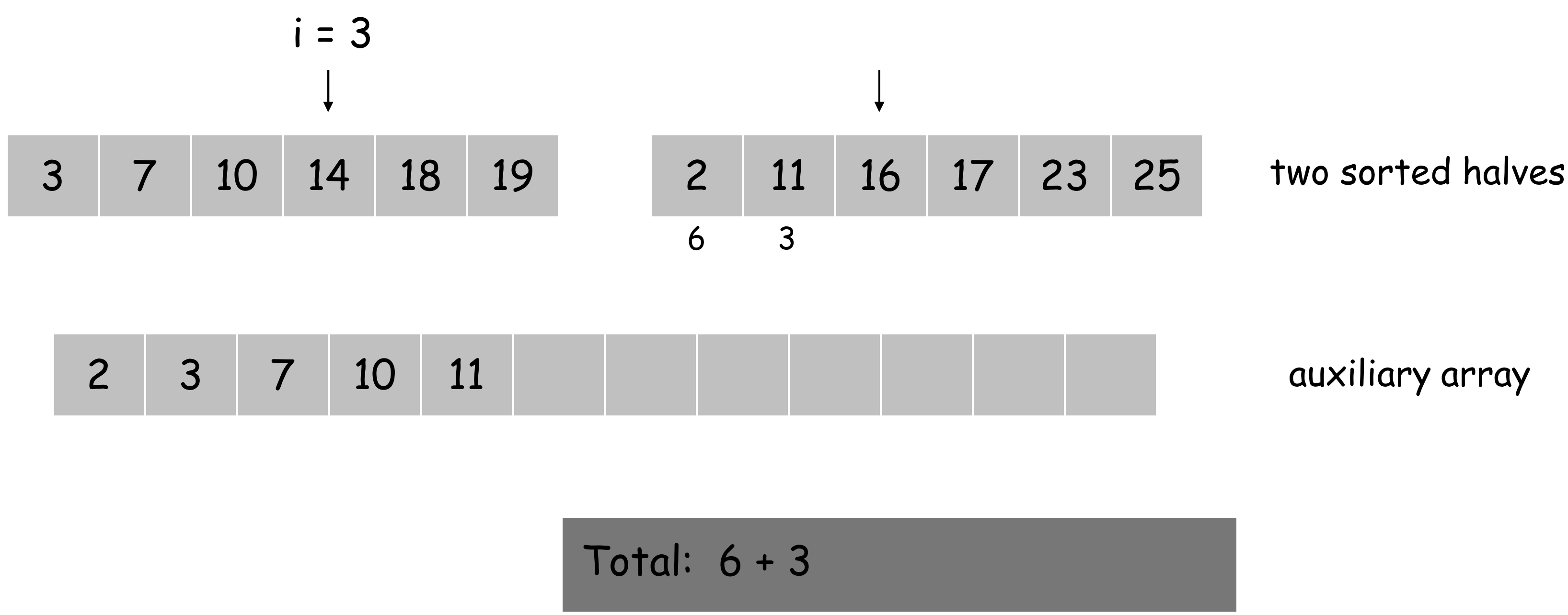
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

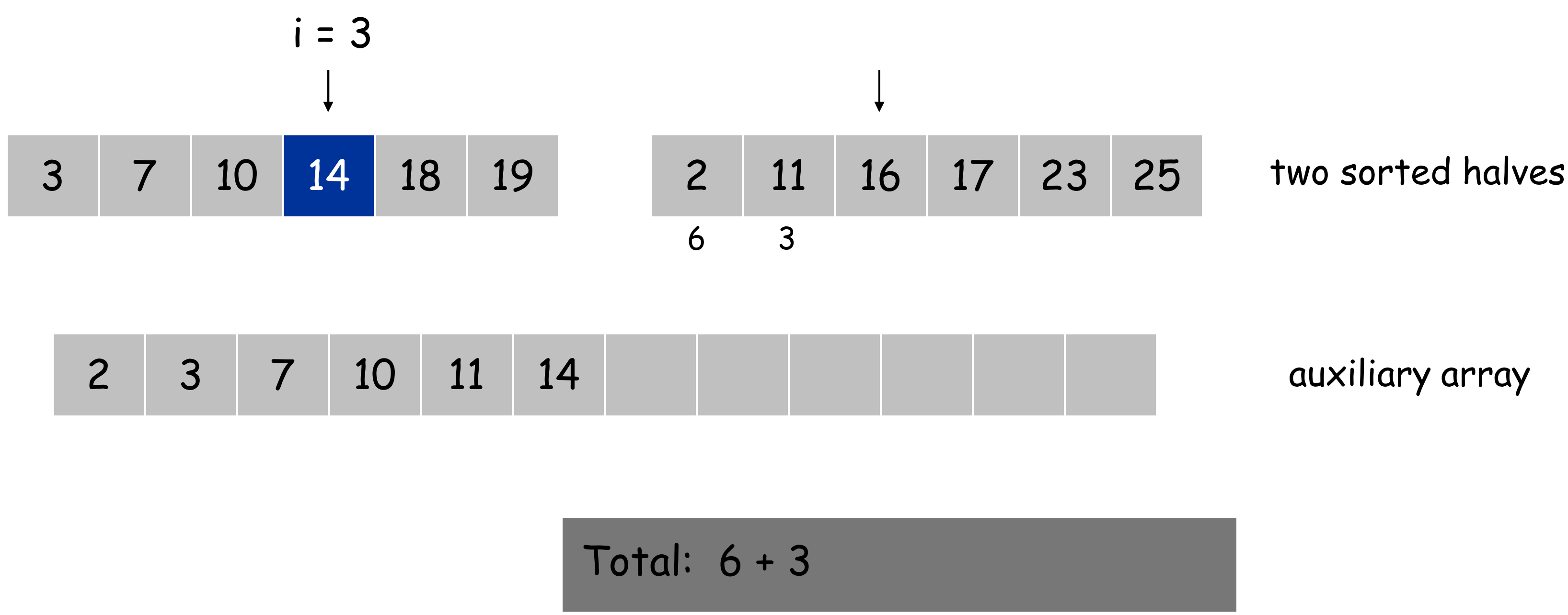
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



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 = 2

two sorted halves

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

auxiliary array

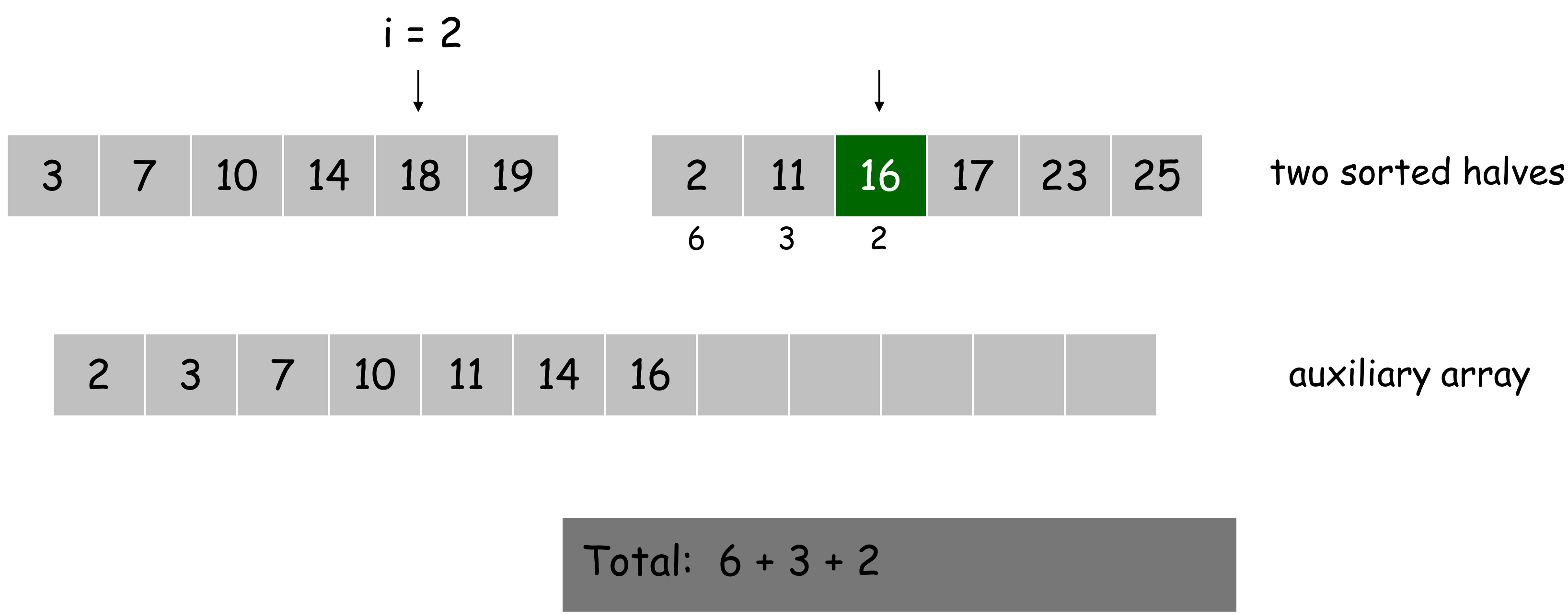
2	3	7	10	11	14						

Total: 6 + 3

Merge and Count

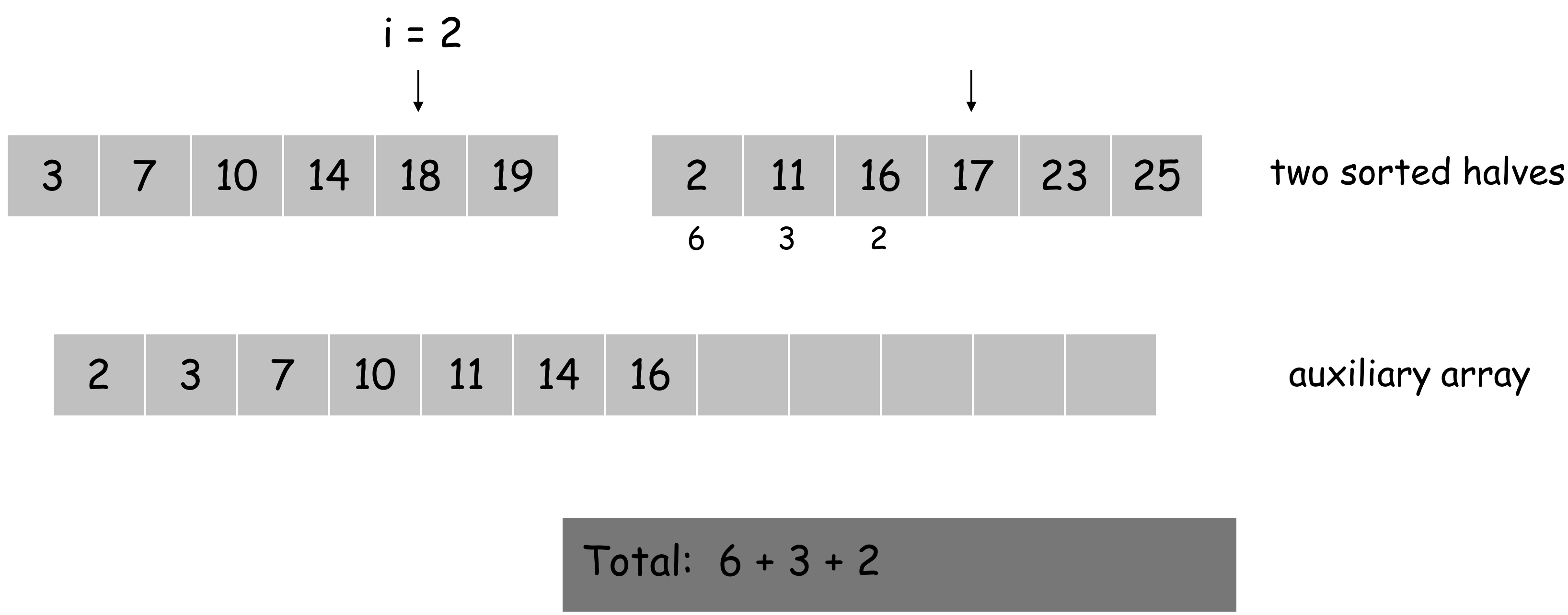
Merge and count step.

- 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

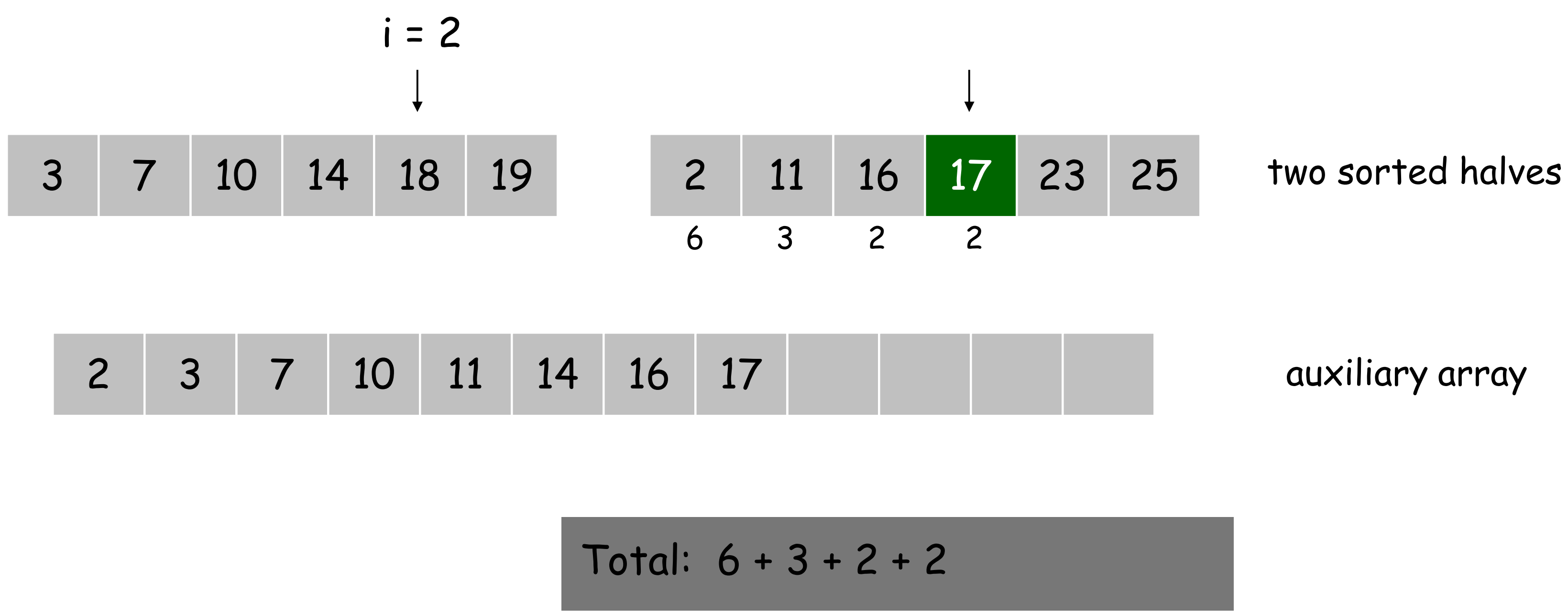
Merge and count step.
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

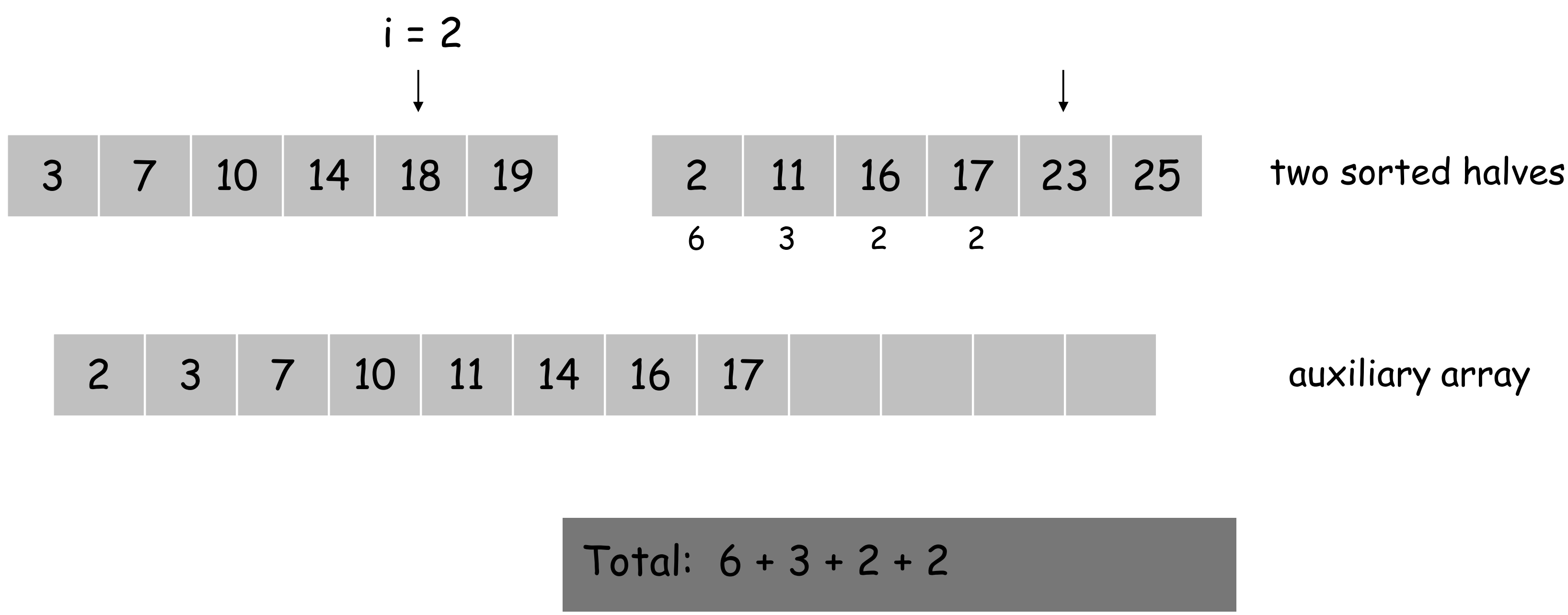
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

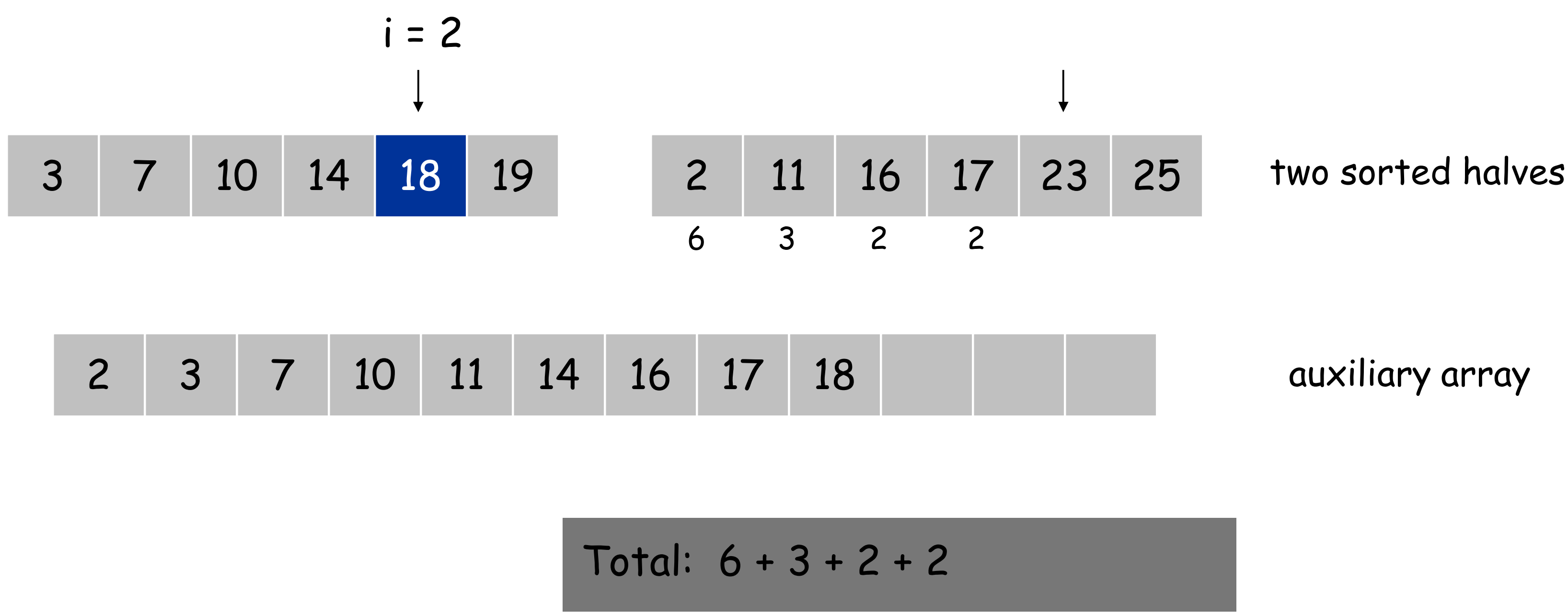
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



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.

i = 1

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

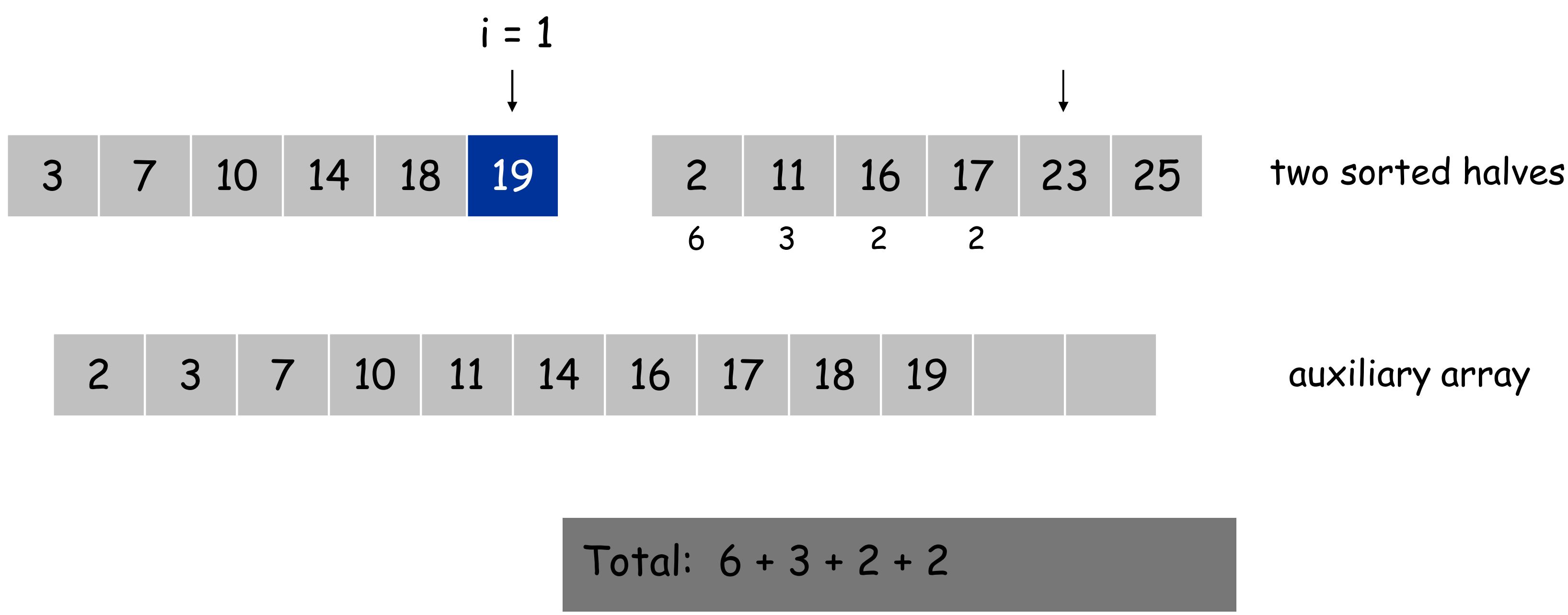
2	3	7	10	11	14	16	17	18				auxiliary array

Total: 6 + 3 + 2 + 2

Merge and Count

Merge and count step.

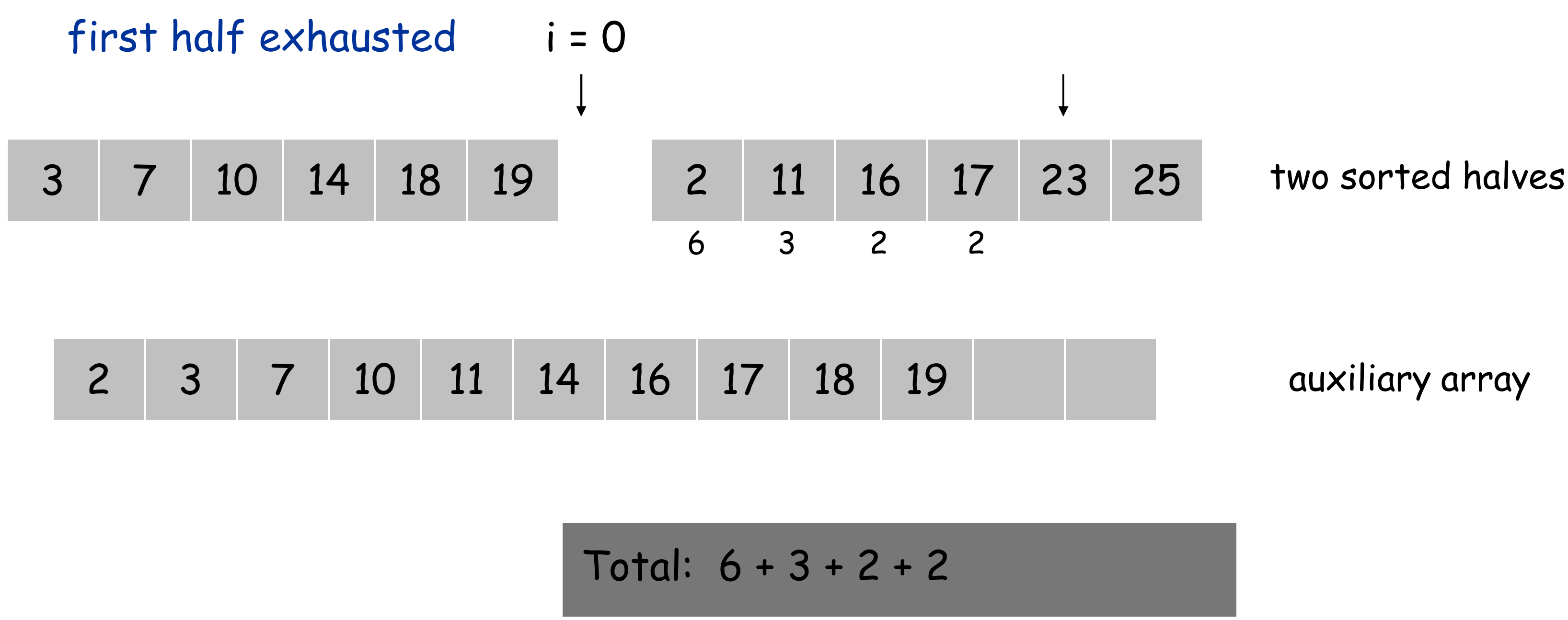
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

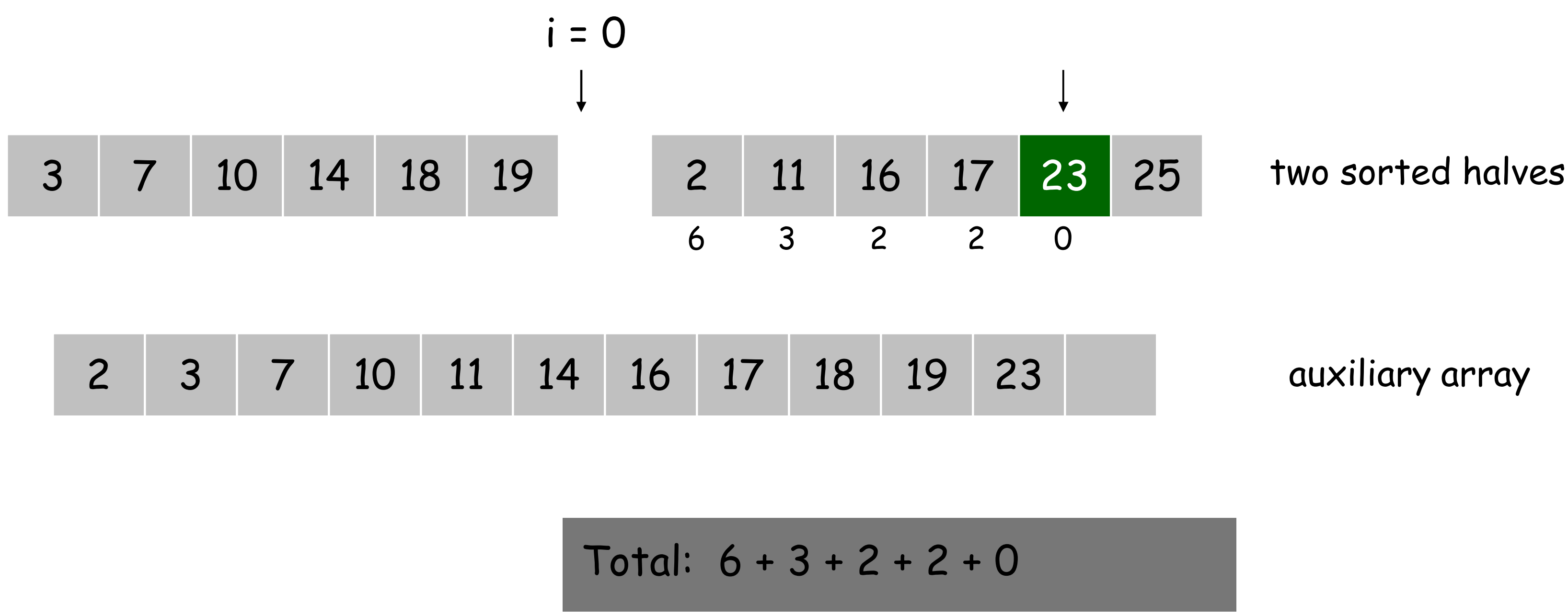
Merge and count step.

- 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

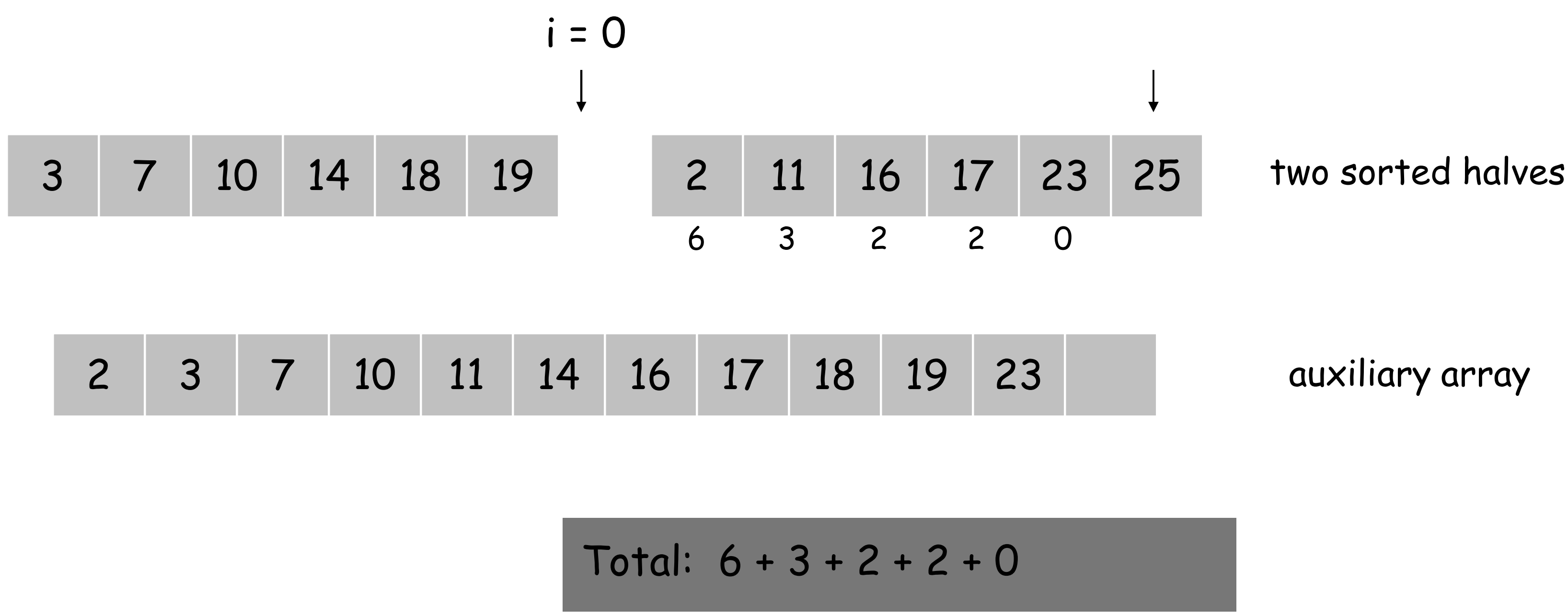
Merge and count step.
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



Merge and Count

Merge and count step.

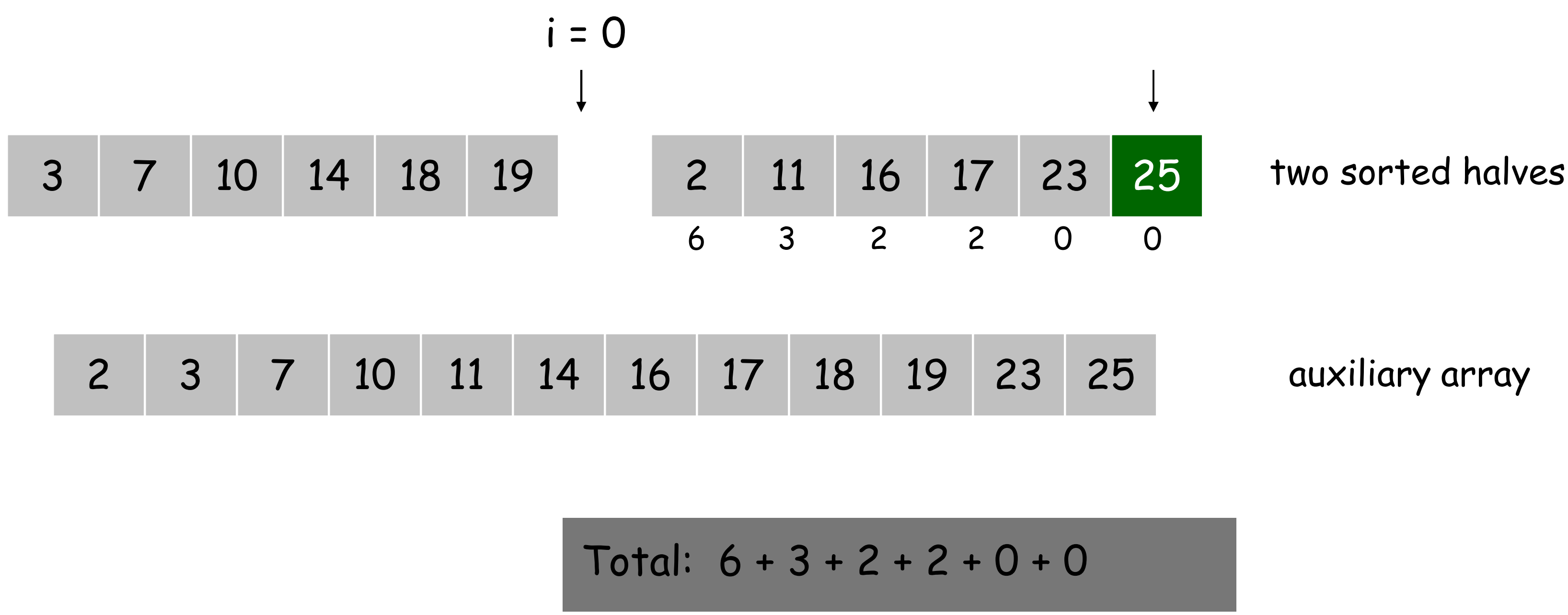
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves into sorted whole.



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.

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

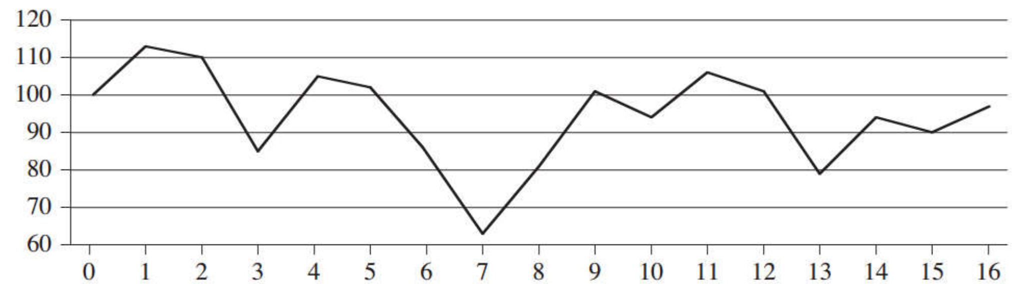
Total: 6 + 3 + 2 + 2 + 0 + 0 = 13

Counting inversions

- Let A be a sequence with left / right halves L / R.
- Function C counts the number of inversions in A, and returns A in sorted order.
  - Compute x = C(L), y = C(R).
    - After this, L and R are sorted.
  - Merge L and R, while counting number of inversions z.
  - Return x+y+z, and the merged sequence.
- Let T be the time complexity for C.
  - $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$.
  - Thus, $T(n) = O(n \log n)$.

Maximum subarray

- Motivation Make money on stocks by buying and selling on days with largest price difference.

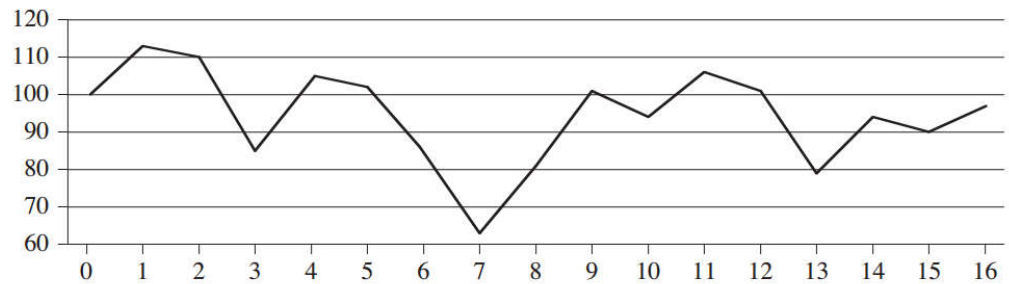


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Source: *Introduction to Algorithms* Cormen et al

- Ex Buy on day 7, sell on day 11, make \$106 - \$63 = \$43.
- If there are n days, can compute price difference of all $O(n^2)$ pairs of days and take the max.
- Is there a faster way?

Maximum subarray



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Let P be the array of stock prices.
- Goal Find $i < j$ such that $P[j]$ – $P[i]$ is maximum.
- We first compute the price change on consecutive days.
  - Ex On day 4, the price change is \$105-\$85=\$20.
  - Call the array of changes A.
  - So $A[i] = P[i] - P[i-1]$, for $i = 1, \ldots, n$.

Maximum subarray

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

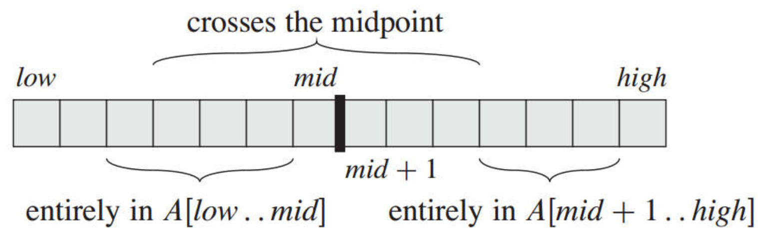
maximum subarray

- Observation Finding $i < j$ with max $P[j]-P[i]$ is the same as finding $i < j$ with max $\sum_{k=i+1}^{j} A[k]$.
  - Ex $P[11] - P[7] = 43 = A[8] + A[9] + A[10] + A[11]$.
- Thus, we want to find a subarray of A with the maximum sum.
  - I.e. want to find a continuous set of elements of A with the largest sum.
  - Ex For A above, it's the 8th to 11th elements.

Maximum subarray

- Goal Given array A, find $i < j$ with max $\sum_{k=i+1}^{j} A[k]$.
- Seems no easier than initial problem... Still $O(n^2)$ pairs i, j to consider.
  - In fact, computing $\sum_{k=i+1}^{j} A[k]$ takes O(n) time, so finding max subarray seems to take $O(n^3)$ time!
  - Actually, can find $\sum_{k=i+1}^{j} A[k]$ for all pairs i,j in $O(n^2)$ time. How?
- But with divide and conquer, can find max subarray in $O(n \log n)$ time.

A divide and conquer algorithm



Observation

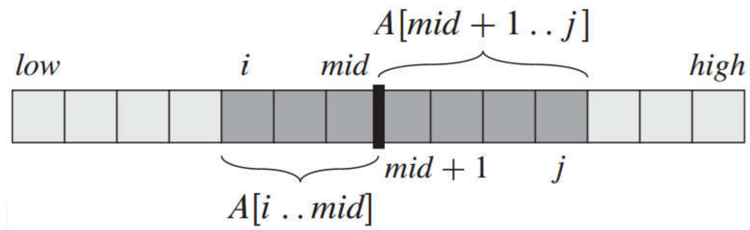
- Divide A down the middle. Then a max subarray of A either
  - Lies entirely in the left half.
  - Lies entirely in the right half.
  - Crosses the midpoint.

Algorithm

- Break A into left and right halves.
- Compute the max subarrays in each half.
- Compute the max subarray crossing the midpoint.
- Return max of these three subarrays.

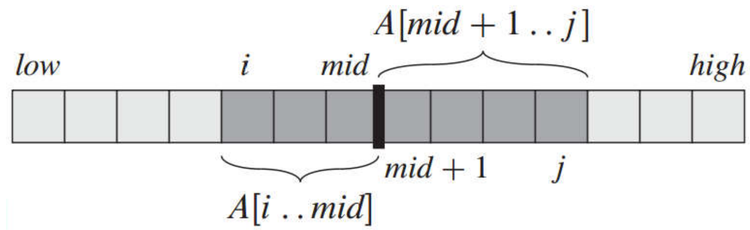
- Analysis $S(n) = 2S(n/2) + T(n) + O(1)$.
  - Finding max subarray in each half takes $S(n/2)$ time.
  - $T(n)$ = time to find max subarray crossing midpoint.
- We will show $T(n) = O(n)$.
- So $S(n) = O(n \log n)$.

Max crossing subarray



- Goal Find max subarray crossing the midpoint.
- Solution Find the max leftwards subarray from the midpoint.
  - I.e. find a subarray containing the midpoint and lying to the left, that has the max sum.
  - Also find the max rightwards subarray from the midpoint.
  - Combine them and return this.
- Ex A = [3,2,-8,1,6,7,-4,2,8,2,-4,1,-2, 3,1].
  - Max leftwards subarray from 2 is [1,6,7,-4,2].
  - Max rightwards subarray from 2 is [2,8,2].
  - Max crossing subarray is [1,6,7,-4,2,8,2].

Max crossing subarray



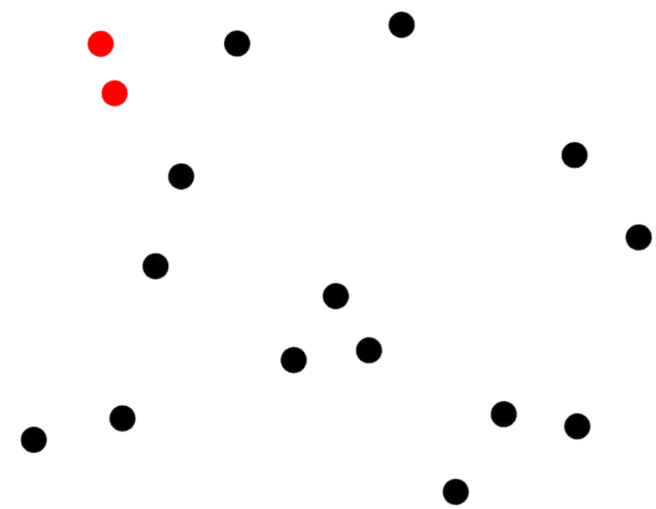
- Algorithm To find max leftwards subarray, sum array elements leftwards starting from midpoint.
  - Whenever sum exceeds current max, remember the index as the current max.
  - Similar for rightwards subarray.
- Analysis Scan through once to left and right. $O(n)$ time.

FIND-MAX-CROSSING-SUBARRAY(A, *low*, *mid*, *high*)

```
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

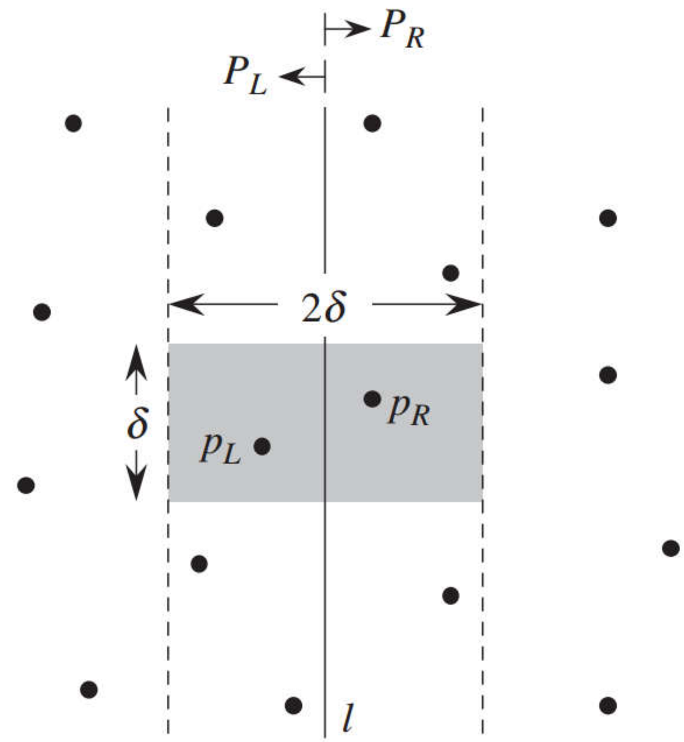
Closest point pair

- Given a set of n points in the plane, find the pair that's closest.
- Naive algorithm computes distances between all $O(n^2)$ pairs of points and chooses min.
- Use divide and conquer to improve complexity to $O(n \log n)$.



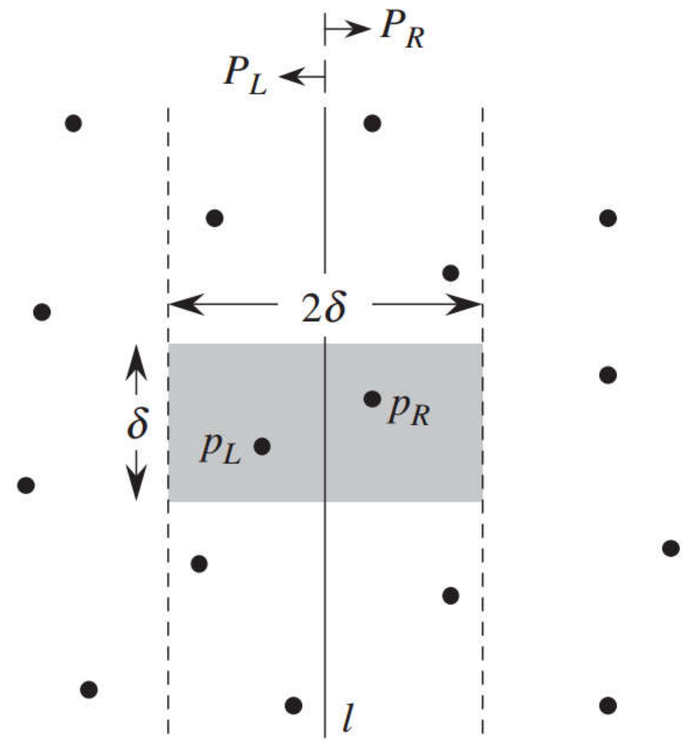
Closest point pair

- Split the points evenly using a vertical line, i.e. half the points lie on the left and half on the right.
- Observation The closest pair of points either
  - Both lie in the left half
  - Both lie in the right half, or
  - Straddles the line, i.e. one point on each side.
- This suggests the following algorithm.



Closest point pair

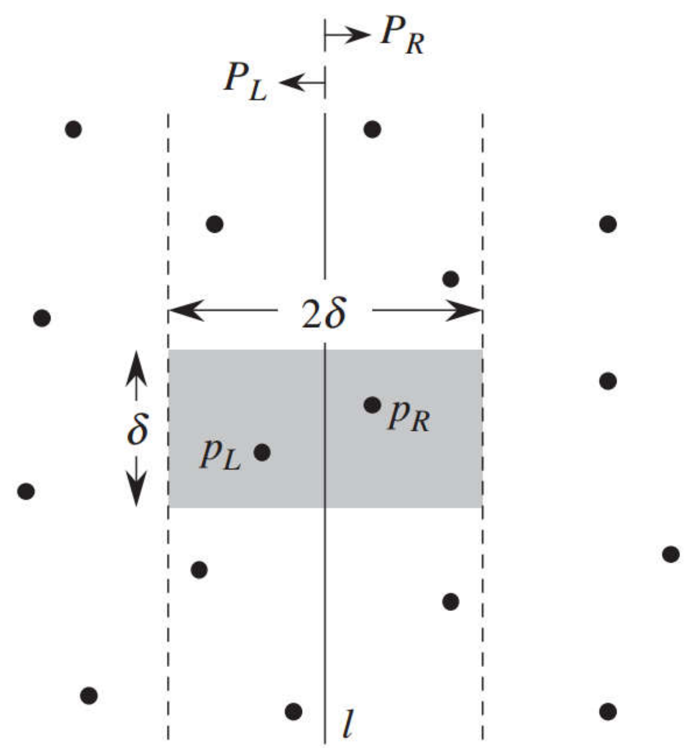
- Divide points evenly using vertical line.
- Recursively find closest point pair in left half and right half.
  - Let the min distance between any point pair in either half be δ.
- Look for closest pair of points straddling line with distance < δ.
  - Don't need to consider straddling pairs with distance $\geq \delta$, since we already found such pairs on the left or right.
- If pair exists, return their distance.
- Else return δ.



Algorithm analysis

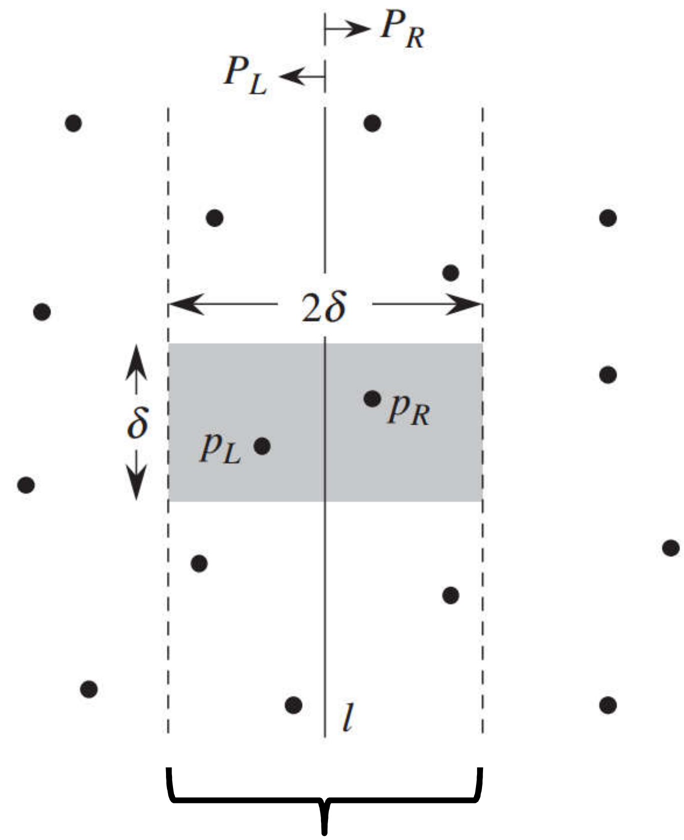
- Let $S(n)$ be time to find closest point pair of n points.
- $S(n) = 2S(n/2) + O(n)$
  - Can divide the points in $O(n)$ time.
    - Details on slide 17.
  - $2S(n/2)$ time to recursively find closest point pair in both halfs.
  - Can find closest straddling pair in $O(n)$ time.
    - Details next slide.
- $S(2) = O(1)$.
  - If only two points, they're the closest pair.
- So $S(n) = O(n \log n)$.

- Divide the points evenly.
- Recursively find closest pair on left and right.
- Find closest straddling pair.
- Return the min of the three.



Closest straddling point pair

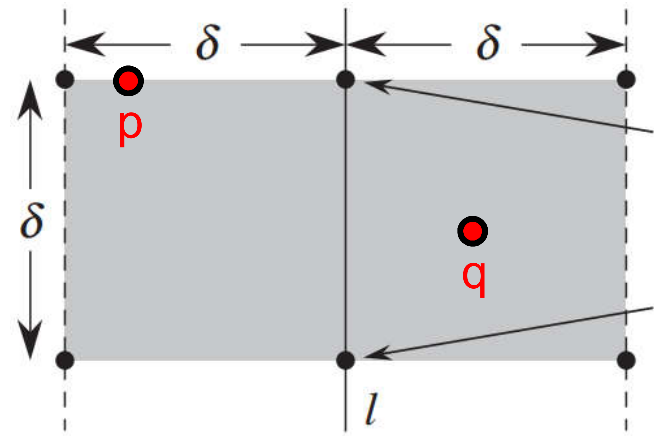
- Goal Find closest straddling pair, assuming their distance is $< \delta$.
- Only need to consider points within a band of width 2δ centered on dividing line.
  - Pairs outside band can't be closer than δ.
- Let B be set of points in band.
  - To form B, iterate through all points in any order, pick ones within distance δ from line.
  - Takes $O(n)$ time.
- Assume points in B sorted by y coordinate, i.e. from top to bottom.
  - By iterating in the right order when forming B, can get this property "for free", without actually sorting B.
  - Details later.
- Now, use following lemma to find closest straddling pairs.



2δ band around divding line

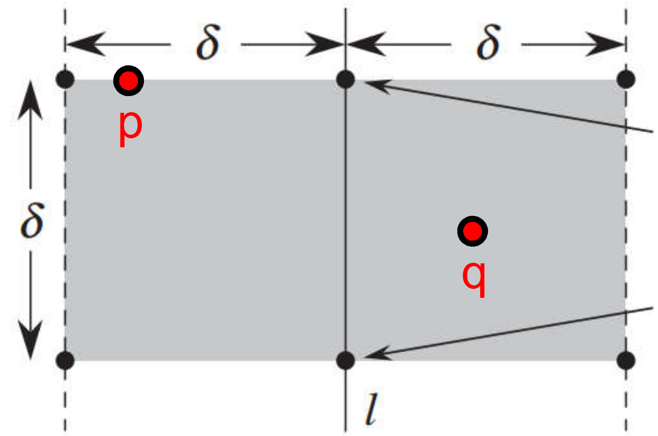
Sparsity lemma

- Lemma Let $p, q \in B$. Suppose q is below p, and has distance $< \delta$ from p. Then
  1. q lies in a $\delta \times 2\delta$ rectangle centered on the dividing line, and with p on the top edge.
  2. The rectangle contains at most 6 points from B (including p and q).
  3. If we list the points in order B from top to bottom, the points in the rectangle immediately follow p in the ordering.



Sparsity lemma proof

1. Any point below the rectangle is $> \delta$ distance from p.
2. Any two points in rectangle on same side of the line are distance $\geq \delta$ apart.
   - Because δ is the min distance between any pair of points on either side.
   - So, at most 6 points in B fit in the rectangle.
   - Ex The 6 points can fit in the corners and the middle, as shown.
3. Points in the rectangle precede any points below it in y ordering.



Closest straddling point pair

- Algorithm Sweep through points in B from top to bottom.
  - For each point p, check next 5 points in R below it.
  - Let δ_p be distance to nearest one.
  - After sweeping through all points in B, return the minimum δ_p value or δ, whichever is smaller.
- Correctness By sparsity lemma, only next 5 points in B below p can be distance $< \delta$ from p.
  - Since we return the closest pair among these 5 points, we find overall closest straddling pair.
  - If no straddling pairs have distance $< \delta$, we return δ.
- Analysis Algorithm takes O(n) time.
  - B contains O(n) points.
  - For each point in B, check its distance to 5 other points.

Dividing points evenly

- At the beginning of the algorithm, sort all points horizontally and store in an array H.
  - Takes $O(n \log n)$ time.
- Assume at some level of recursion, input array is sorted horizontally.
- Then points to the left / right of dividing line are points in the first / second half of array.
  - Outputting either half takes $O(n)$ time.
- These points are sorted horizontally, for the next level of recursion.
  - So at every level of recursion, can get points in sorted order in $O(n)$ time.
- Add this $O(n \log n)$ preprocessing time to algorithm's running time.
  - Algorithm still $O(n \log n)$.

Sorting R points by y coordinate

- At the beginning of the algorithm, also sort all the points vertically. Store them in a separate array V.
  - Takes $O(n \log n)$ time.
- Points in H and V have pointers to each other.
  - I.e. given p in H, its pointer gives p's index in V. Similarly given p in V, we can get p's index in H.
- When picking out points left (or right) of dividing line using H, mark them in V by following the pointers.
- Next, iterate through V (in vertical order) and pick out marked points.
  - These points are again sorted vertically.
  - Takes $O(n)$ time.
- Add this $O(n \log n)$ preprocessing time to algorithm's running time. Algorithm still $O(n \log n)$.