Algorithms and Data Structures
Discussion 10 (Week 11)

Keyi Yuan
Teaching Assistant
Nov.18th 2019

Agenda

- Hash:
  - Motivation and Overview
  - Hash Function
  - Collision
    - Direct Addressing: Chaining
    - Open Addressing: Linear Probing, Quadratic Probing
  - Application: Bloom Filter
- Merge Sort:
  - Algorithm
  - Solve Recursive Function: Recursion Tree
    - Divide and Conquer
  - Counting Inversions
  - Solve Recursive Function: Master Theorem

Part 1-1: Hash

Motivation and Overview

Motivation

- We want to find a data structure that can support Insert(), Search() and Delete() operation in least time.
- Recall the data structure we have learned (worst case):
- You can see that all the operations cost more than constant time.

	Insert(key)	**Search(key)**	**Delete(key)**
Array	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$
Heap	$O(\log n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

Motivation

- Use an array with large size?
- When inserting only a few elements?
  - Waste space.

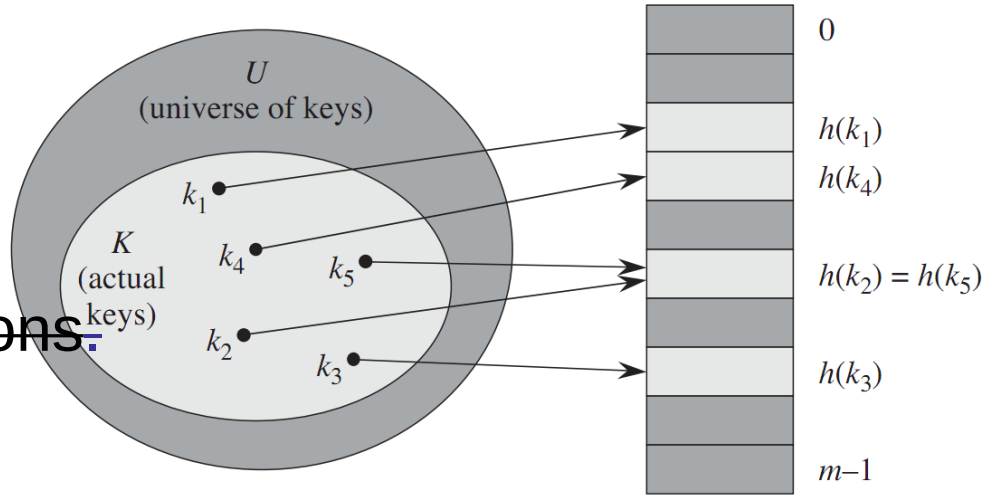
	Insert(key)	Search(key)	Delete(key)
Array	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$
Heap	$O(\log n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

Hash table

- A hash table is a randomized data structure to efficiently implement a dictionary.
- Supports find, insert, and delete operations all in expected $O(1)$ time. **It is only an expectation!**
  - But in the worst case, all operations are $O(n)$.
  - The worst case is provably very unlikely to occur.
- A hash table does not support efficient min / max or predecessor / successor functions.
  - All these take $O(n)$ time on average.

Hash table

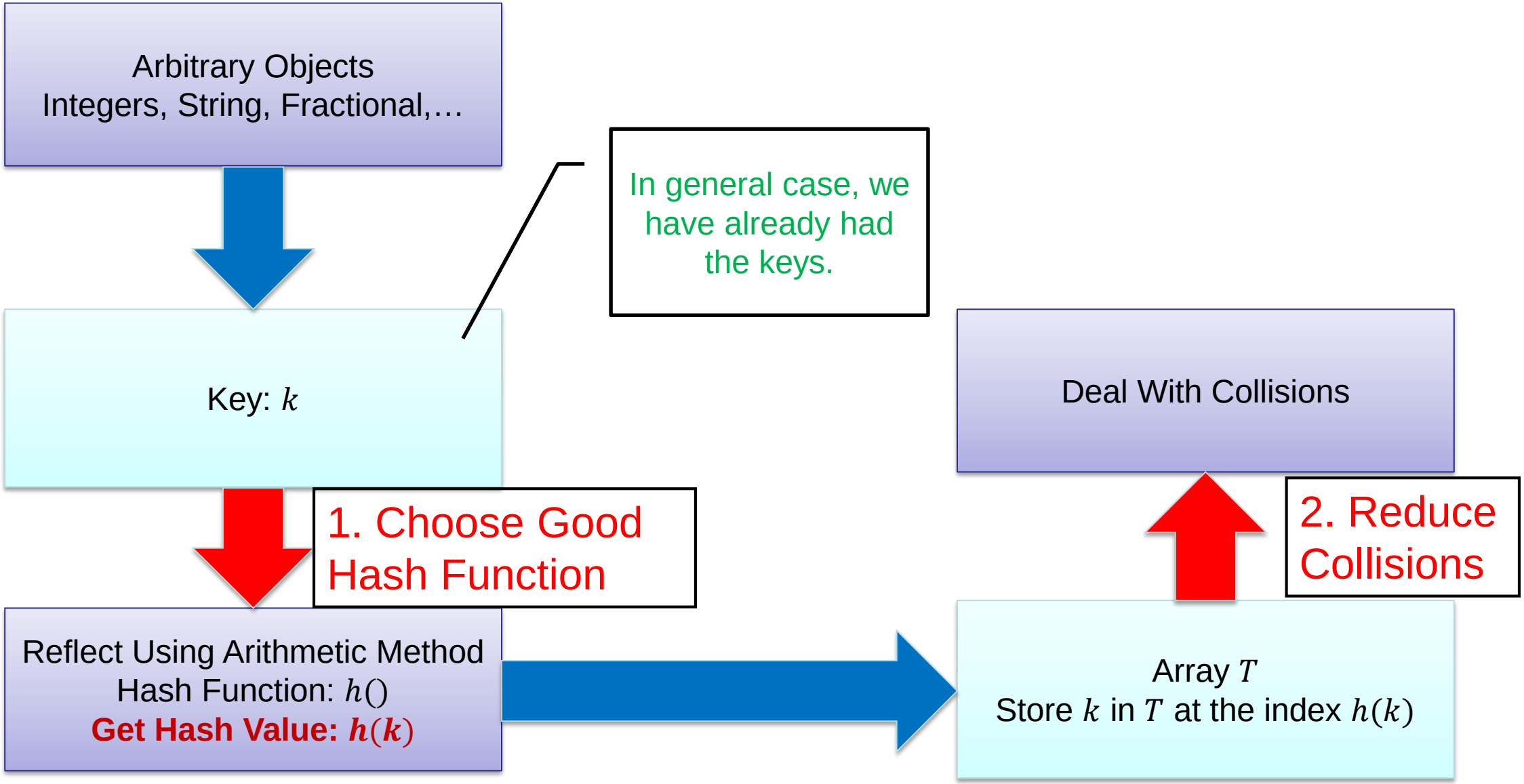
- Reduce the expected time complexity, and uses much less space.
- Idea Instead of storing directly at key's location, convert key to much smaller value, and store at this location.
- A hash table consists of the following:
  - A universe U of keys.
  - An array of T of size m.
  - A hashing function $h: U \to \{0,1,\dots,m-1\}$.
- ~~We'll talk later about how to pick good hash functions.~~
- insert(k, v) Hash key to $h(k)$. Store v in $T[h(k)]$.
  - *In general cases, k is v itself.*
- find(k) Return the value in $T[h(k)]$
- delete(k) Delete the value in $T[h(k)]$
- Assuming $h(k)$ takes $O(1)$ time to compute, all ops still take $O(1)$ time.
- Uses $O(m)$ space.
- If $m \ll |U|$, then hashing uses much less space than direct addressing.
- However, our current scheme doesn't quite work, due to collisions.



Collisions

- We store a key at array position $h(k)$.
- But what if two keys hash to the same location, i.e. $k_1 \neq k_2$, but $h(k_1) = h(k_2)$?
  - This is called a collision.
- Collisions are unavoidable when $|U| > m$.
  - By Pigeonhole Principle, must exist at least two different keys in U that hash to same value.
- Two basic ways to deal with collisions, direct and open addressing.

Overview


Arbitrary Objects
Integers, String, Fractional,…
In general case, we have already had the keys.
Key: k
Deal With Collisions
1. Choose Good Hash Function
2. Reduce Collisions
Reflect Using Arithmetic Method
Hash Function: $h()$
Get Hash Value: $h(k)$
Array T
Store k in T at the index $h(k)$

Part 1-2: Hash

Hash Function

Picking a hash function

- We saw that we want hash functions to hash keys to "random" locations.
  - However, note that each hash function is itself a deterministic function, i.e. h(k) always has the same value.
- It's hard to find such random hash functions, since we don't assume anything about the distribution of input keys.
  - Ex For any hash function, there are always $\geq |U|/m$ keys from the universe hashing to the same location. So if the input is exactly this set, and $|U|/m \geq n$, then all ops take $O(n)$ time.
- In practice, we use a number of heuristic functions.

Heuristic hash functions

- Assume the keys are natural numbers.
  - Convert other data types to numbers.
  - Ex To convert ASCII string to natural number, treat the string as a radix 128 number. E.g. “pt” → (112*128)+116 = 14452.
- Division method $h(k) = k \bmod m$
  - Often choose m a prime number not too close to a power of 2.
  - Why prime?
- Multiplication method $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$, where A is some constant.
  - m is not important.
  - Knuth’s suggestion is $A = \frac{\sqrt{5}-1}{2} \approx 0.618034$ ...

Universal hashing (Optional)

- As we said, regardless of the hash function, an adversary can choose a set of n inputs to make all operations O(n) time.
- Universal hashing overcomes this using randomization.
  - No matter what the n input keys are, every operation takes O(n/m) time in expectation, for a size m hash table.
  - Note O(n/m) time is optimal.
- Instead of using a fixed hash function, universal hashing uses a random hash function, chosen from some set of functions H.
- Say H is a universal hash family if for any keys $x \neq y$

$$\Pr_{h \in H}[h(x) = h(y)] = 1/m$$

- So if we randomly choose a hash function from H and use it to hash any keys x, y, they have 1/m probability of colliding.
- Note the hash functions in H are not random. However, we choose which function to use from H randomly.

Universal hashing (Optional)

- Thm Let H be a universal hash family. Let S be a set of n keys, and let $x \in S$. If $h \in H$ is chosen at random, then the expected number of $y \in S$ s.t. $h(x) = h(y)$ is n/m.
- Proof Say $S = \{x_1, \ldots, x_n\}$.
  - Let X be a random variable equal to the number of $y \in S$ s.t. $h(x) = h(y)$.
  - Let $X_i = 1$ if $h(x_i) = h(x)$ and 0 otherwise.
  - $E[X_i] = \Pr_{h \in H}[h(x_i) = h(x)] \times 1 + \Pr_{h \in H}[h(x_i) \neq h(x)] \times 0 = 1/m$.
    - First equality follows by universal hashing property.
  - $E[X] = E[X_1] + \cdots + E[X_n] = n/m$.

Constructing universal hash family 1 (Optional)

- Choose a prime number p > m.
  - So every key < p.
- Let $h_{ab}(k) = \big((ak + b) \bmod p\big) \bmod m$.
- Let $H_{pm} = \{h_{ab} \mid a \in \{1,2, \dots, p-1\}, b \in \{0, 1, \dots, p-1\}\}$.
- Thm H_{pm} is a universal hash family.
- Proof Let $x, y < p$ be two different keys. For a given h_{ab} let

$$r = (ax + b) \bmod p, s = (ay + b) \bmod p$$

- We have $r \neq s$, because $r - s \equiv a(x - y) \bmod p \neq 0$, since neither a nor $x - y$ divide p.
- Also, each pair (a, b) leads to a different pair (r, s), since

$$a = \big((r - s)(x - y)^{-1} \bmod p\big), b = (r - ax) \bmod p$$

  - Here, $(x - y)^{-1} \bmod p$ is the unique multiplicative inverse of $x - y$ in $\mathbb{Z}_p^*$.

Constructing universal hash family 2 (Optional)

- Since there are $p(p-1)$ pairs (a,b) and $p(p-1)$ pairs (r,s) with $r \neq s$, then a random (a,b) produces a random (r,s).
- The probability x and y collide equals the probability $r \equiv s \bmod m$.
- For fixed r, number of $s \neq r$ s.t. $r \equiv s \bmod m$ is $(p-1)/m$.
- So for each r and random $s \neq r$, probability that $r \equiv s \bmod m$ is $((p-1)/m))/(p-1) = 1/m$.
- So $\Pr_{h_{ab} \in H_{pm}}[h_{ab}(x) = h_{ab}(y)] = 1/m$ and H_{pm} is universal.

Hash Function Properties

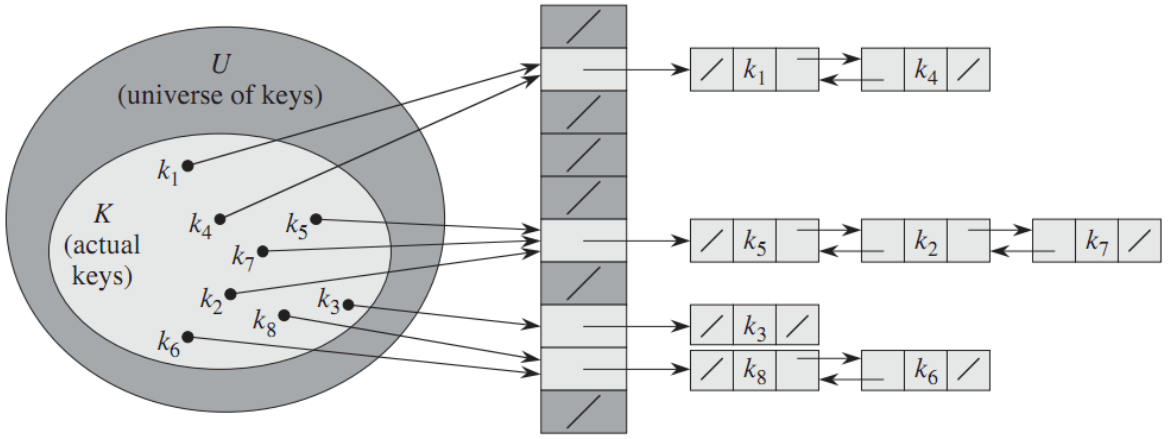
- Should be fast: ideally $\Theta(1)$
- The hash value must be *deterministic*
  - h(k) always has the same value.
- Equal objects hash to equal values
  - $x = y \Rightarrow h(x) = h(y)$
- If two objects are randomly chosen, there should be only a $1/m$ chance that they have the same hash value.

Part 1-3: Hash

Collisions

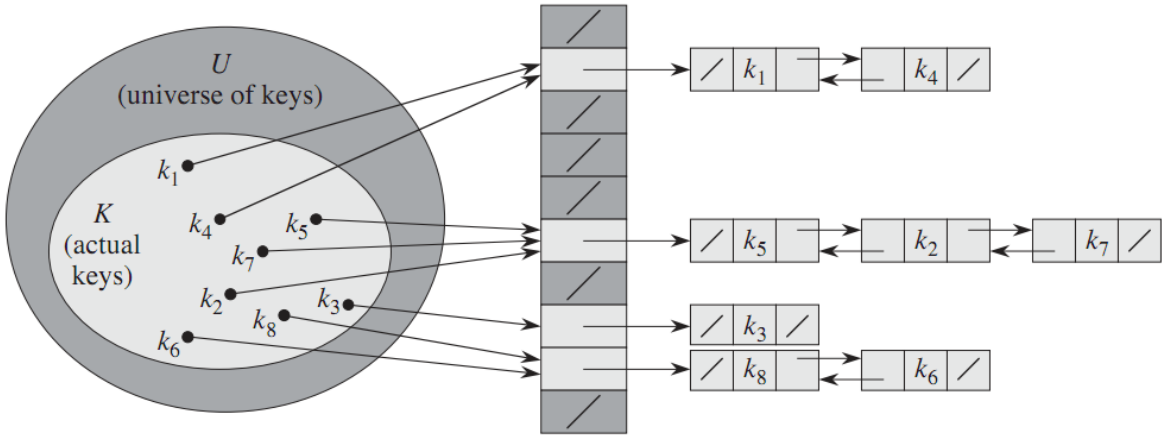
1. Closed addressing: Chaining

- In direct addressing, every entry in hash table points to a linked list.
  - Keys that hash to the same location get added to the linked list.
  - For simplicity, we'll ignore values from now on and only focus on keys.
- insert(k) Add k to the linked list in $T[h(k)]$.
  - **Before insert(), you should find it there exists duplications.**
- find(k) Search the linked list in $T[h(k)]$ for k.
- delete(k) Delete k from the linked list in $T[h(k)]$.
- Suppose the longest list has length $\hat{n}$, and average length list is $\bar{n}$.
  - Each operation takes worst case $O(\hat{n})$ time.
  - An operation on a random key takes $O(\bar{n})$ time.



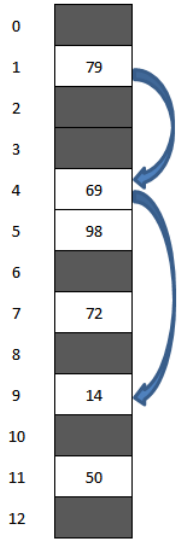
1. Closed addressing: Chaining-Load factor

- The key to making closed addressing hashing fast is to make sure list lengths aren't too long.
- For this, we want the hash function to appear random.
  - Assume that any key is uniformly likely to be hashed to any table location.
- Suppose the hash table contains n items, and has size m.
- Then under the uniform hashing assumption, each table location has on average n/m keys.
  - Call $\alpha = n/m$ the load factor. It is only a measurement.
- So the average(expected) time for each operation is $O(\alpha)$.
- However, even with uniform hashing, in the worst case, all keys can hash to the same location. So the worst case performance is $O(n)$.



2. Open addressing

- In open addressing, each hash table entry stores at most one key.
- When there's a collision, rehash the key to a different location and try to place key there.
  - If there's another collision, rehash again, etc.
- Define a new hash function $h: U \times \{0,1,\dots,m-1\} \rightarrow \{0,1,\dots,m-1\}$ and a probe sequence $h(k,0), h(k,1), \dots, h(k,m-1)$.
  - If $h(k,0), \dots, h(k,i)$ already occupied, try to insert location in $h(k,i+1)$.
- To search for k, also search in locations $h(k,0), h(k,1), \dots, h(k,m-1)$ until
  - $T[h(k,i)] = k$, i.e. we found k in the i'th probe location.
  - Or $T[h(k,i)] = NIL$, meaning k is not in T.
  - Or $T[h(k,m-1)] \neq k$, also meaning k is not in T.



HASH-INSERT(T, k)

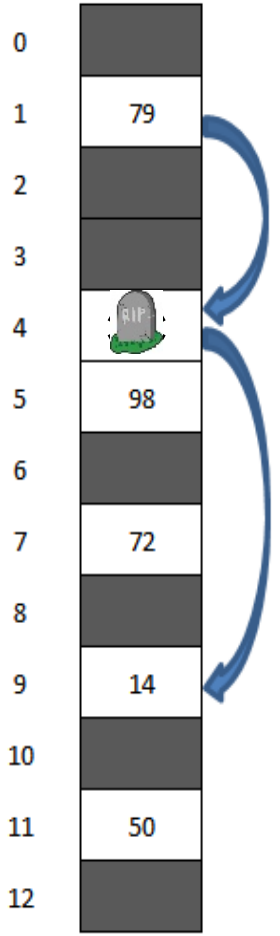
```
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

HASH-SEARCH(T, k)

```
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

2. Open addressing

- Deletion is more complicated. If we just set a location to NIL, then it can cause searches not to find a key still in the table.
  - *In Linear probing, we need to move elements in CS101. (Lazy)*
  - In other case, we mark deleted entries with a “tombstone” more often. (lazy erasing)
- Searches that hit a tombstone keep searching, till they probe the entire table, or hit NIL.
  - However, now search time depends both on current number of table entries, and also number of past deleted ones.
  - Closed addressing is more common when keys need to be deleted.
- Since each table entry stores one key, a table of size m can store at most m keys.
- However, since each table entry stores just the key and no pointers, then for the same amount of storage, an open addressing hash table can store more entries than a closed addressing table.



Probe sequences

- We want two properties from probe sequence
  - $h(k,0), \ldots, h(k,m-1)$ covers all table entries 0,...,m-1.
    - If there's room in the table, an insert will succeed.
  - Given a random key k, $h(k,0), \ldots, h(k,m-1)$ is a random sequence in the m! permutations of {0,...,m-1}.
    - For example, if all keys lead to same probe sequence, then the i'th insert takes O(i) time.
- In practice, it's hard to efficiently create uniformly random probe sequences. So we settle for heuristics.
  - Generally, the more sequences a heuristic can generate, the better it performs.

Heuristic probe sequences

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
  - $h'(k)$ determines entire probe sequence, so there's only m different probe sequences.
  - Poor performance due to primary clustering.
    - An empty slot with i filled slots before it gets filled with probability $(i+1)/m$.
    - So long runs of filled slots, that tend to get even longer.
- Quadratic probing $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where c_1, c_2 are constants.
  - No primary clustering, but secondary clustering, where $h(k_1,0) = h(k_2,0)$ implies both probe sequences are equal.

2-1. Linear probing – Insert & Search

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
- Ex: $h(k) = k \; mod \; 10$, we want to insert $A = [1,8,11,21,22]$

$h(1) = 1$, probe -> 1		1								
$h(8) = 8$, probe -> 8									8	
$h(11) = 1$, probe -> 1,2		1 ~~11~~	11						8	
$h(21) = 1$, probe -> 1,2,3		1 ~~21~~	11 ~~21~~	21					8	
$h(22) = 2$, probe -> 2,3,4		1	11 ~~2~~	21 ~~2~~	2				8	

2-1. Linear probing – Insert & Search

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
- Ex: $h(k) = k \, mod \, 10$, we want to insert $A = [1,8,11,21,22,9,60,99]$

$h(9) = 1$, probe -> 9

	1	11	21	2				8	9

$h(60) = 0$, probe -> 0

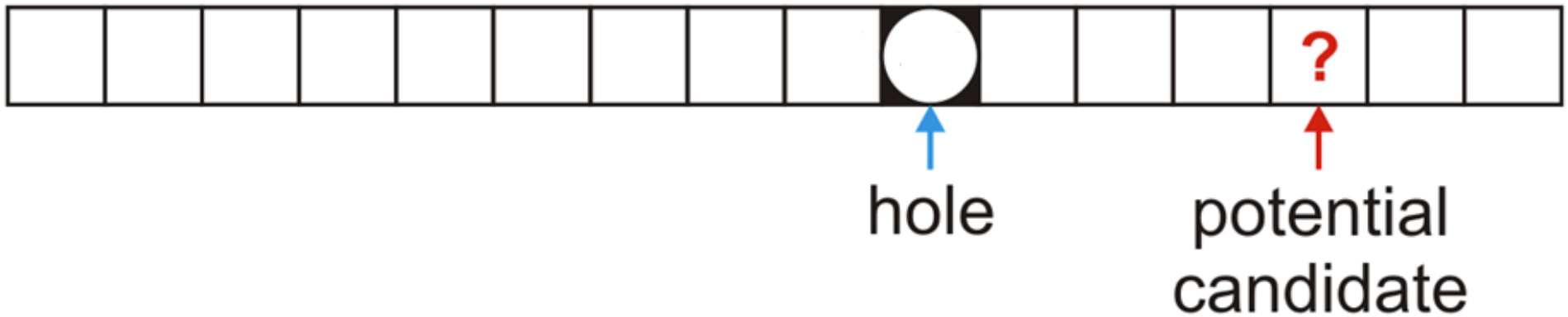
60	1	11	21	2				8	9

$h(99) = 8$, probe -> 9,0,1,2,3,4,5

60 ~~99~~	1 ~~99~~	11 ~~99~~	21 ~~99~~	2 ~~99~~	99			8	9 ~~99~~

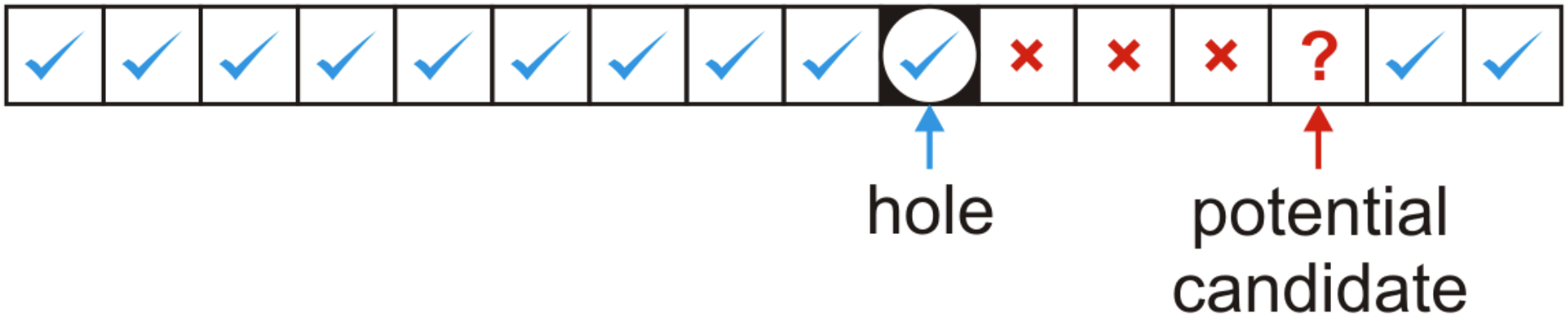
2-2. Linear probing – Delete Alternative 1

- $h(k,0), \ldots, h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
- In general, assume:
  - The currently removed object has created a hole at index hole
  - The object we are checking is located at the position index and has a hash value of `hash`
- Remember: if we are checking the object `?` at location `index`, this means that all entries between `hole` and `index` are both occupied and could not have been copied into the `hole`.
- For each deletion, you should scan the whole hash table until you found that this cell is empty. If you need to move, move it to the `hole`. And the position at potential candidate becomes new hole.



2-2. Linear probing – Delete Alternative 1

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
- The first possibility is that `hole < index`
- In this case, we move the object at `index` only if its hash value is either
  - equal to or less than the `hole` **or**
  - greater than the `index` of the potential candidate.
- For each deletion, you should scan the whole hash table table until you found that this cell is empty. . If you need to move, move it to the `hole`. And the position at potential candidate becomes new `hole`.



2-1. Linear probing – Delete Alternative 1

- The first possibility is that `hole` < `index`
- In this case, we move the object at `index` only if its hash value is either
  - equal to or less than the `hole` **or**
  - greater than the `index` of the potential candidate
- Ex: $h(k) = k \; mod \; 10$, we want to insert delete 1

Initial State, hole = 1

60	~~1~~	11	21	2	99			8	9

$h(11) = 1$, index = 2
(1) $h(11) \leq$ hole ? Yes!
(2) $h(11) >$ index? NO!
Move 11 to the hole=1.
hole = index = 2. Then delete 11 at index = 2.

60	11	~~11~~	21	2	99			8	9

$h(21) = 1$, index = 3
(1) $h(21) \leq$ hole ? Yes!
(2) $h(21) >$ index? NO!
Move 21 to the hole=2.
hole = index = 3. Then delete 21 at index = 3.

60	11	21	~~21~~	2	99			8	~~9~~

2-1. Linear probing – Delete Alternative 1

- The first possibility is that `hole` < `index`
- In this case, we move the object at `index` only if its hash value is either
  - equal to or less than the `hole` **or**
  - greater than the `index` of the potential candidate
- Ex: $h(k) = k \, mod \, 10$, we want to insert delete 1

Initial State, hole = 3

60	11	21		2	99			8	~~9~~

$h(2) = 2$, index = 4
(1) $h(2) \leq$ hole ? Yes!
(2) $h(2) >$ index? NO!
Move 2 to the hole=3.
hole = index = 4. Then delete 2 at index = 4.

60	11	21	2	~~2~~	99			8	9

$h(99) = 1$, index = 5
(1) $h(99) \leq$ hole ? Yes!
(2) $h(99) >$ index? NO!
Move 99 to the hole=4.
hole = index = 5. Then delete 99 at index = 5.

60	11	21	2	99	~~99~~			8	~~9~~

2-1. Linear probing – Delete Alternative 1

- The first possibility is that `hole` < `index`
- In this case, we move the object at `index` only if its hash value is either
  - equal to or less than the `hole` **or**
  - greater than the `index` of the potential candidate
- Ex: $h(k) = k \, mod \, 10$, we want to insert delete 1

Initial State, hole = 5

60	11	21	2	99				8	~~9~~

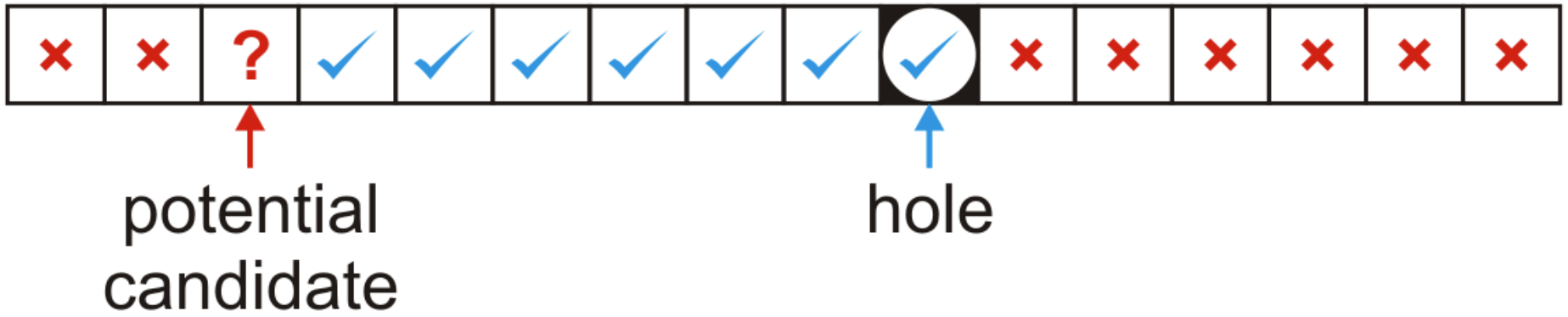
hole = 5 is empty.

Algorithm Ends.

60	11	21	2	99				8	9

2-1. Linear probing – Delete Alternative 2

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
- The second case is we wrapped around the end of the array, that is `hole` > `index`
- In this case, we move the object at `index` only if its hash value is both
  - equal to or less than the `hole` **and**
  - greater than the `index` of the potential candidate.
- For each deletion, you should scan the whole hash table table until you found that this cell is empty. . If you need to move, move it to the `hole`. And the position at potential candidate becomes new `hole`.



2-1. Linear probing – Delete Alternative 2

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Linear probing $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
- Another way of deletion: Lazy Erasing
  - Mark the bin as ERASED
  - Searching: regard it as occupied
  - Insertion: regard it as unoccupied
- What if we want to insert 99 after deleting 9?
  - Search before insertion

60	1	11	21	2	99			8	9

60	1	11	21	2	99			8	X

2-2. Quadratic probing – Insert & Search

- $h(k,0), \ldots, h(k,m-1)$ covers all table entries 0,...,m-1.
- Quadratic probing $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where c_1, c_2 are constants.
- Ex: $h(k) = k \; mod \; 10$, we want to insert $A = [1,8,11,21,22]$
- With probe function: $c_1 i + c_2 i^2 = i/2 + i^2/2$

$h(1) = 1$, probe -> 1		1								
$h(8) = 8$, probe -> 8									8	
$h(11) = 1$, probe -> 1,2		1 ~~11~~	11						8	
$h(21) = 1$, probe -> 1,2,4		1 ~~21~~	11 ~~21~~		21				8	
$h(22) = 2$, probe -> 2,3		1	11 ~~2~~	2	21				8	

LESS!

2-2. Quadratic probing – Delete

- $h(k,0)$, …, $h(k,m-1)$ covers all table entries 0,...,m-1.
- Quadratic probing $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where c_1, c_2 are constants.
- Way of deletion: Lazy Erasing
  - Mark the bin as ERASED
  - Searching: regard it as occupied
  - Insertion: regard it as unoccupied

Heuristic probe sequences (Optional)

- Double hashing $h(k,i) = ((h_1(k) + ih_2(k)) \bmod m.$
- $h_2(k)$ needs to be relatively prime to m to make sure entire table searched.
- One way to ensure this is make m a power of 2, and h_2 always be odd.
- Can also make m prime, m'=m-1, and set

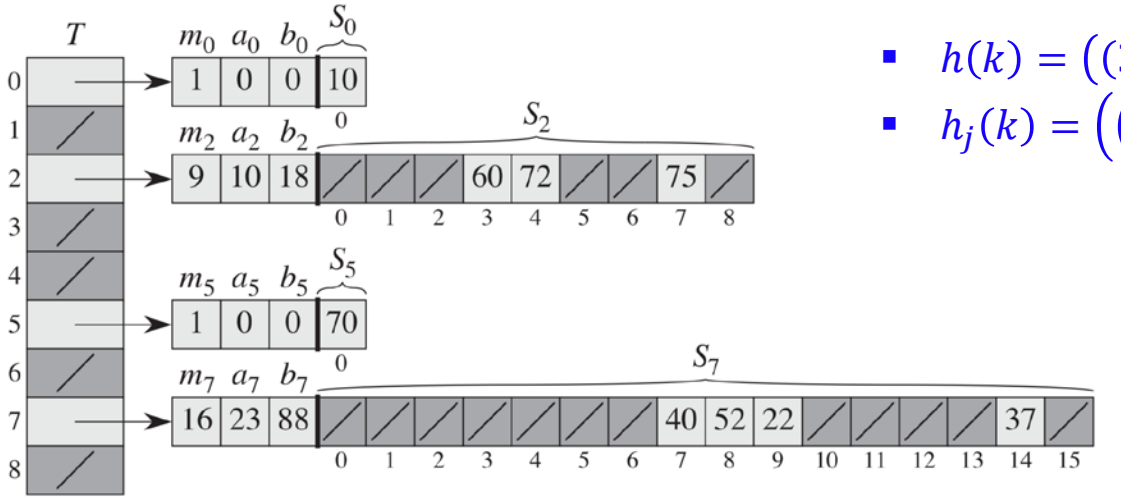
$$h_1(k) = k \bmod m, h_2(k) = 1 + (k \bmod m')$$

- Since each distinct $(h_1(k), h_2(k))$ leads to distinct probe sequence, double hashing can produce $O(m^2)$ probe sequences.
  - Double hashing performs quite well in practice.

Perfect hashing (Optional)

- The hashing methods we've seen can sure O(1) expected performance, but are O(n) in the worst case due to collisions.
- However, if we have a fixed set of keys, perfect hashing can ensure no collisions at all.
  - Perfect hashing maintains a static set, and allows find(k) and delete(k) in O(1) time.
  - It doesn't support insert(k).
- Ex The fixed set of keys may represent the file names on a non-writable DVD.

Perfect hashing (Optional)



- $h(k) = \big((3k + 42) \bmod 101\big) \bmod 9$
- $h_j(k) = \Big(\big(a_j k + b_j\big) \bmod 101\Big) \bmod m_j$

- Perfect hashing uses two levels of universal hashing.
  - Use first layer hash function h to hash key to a location in T.
  - Each location j in T points to a hash table S_j with hash function h_j.
  - If n_j keys hash to location j, the size of S_j is $m_j = n_j^2$.
- We'll ensure there are no collisions in the secondary hash tables $S_1, \ldots, S_m$.
  - So all operations take worst case O(1) time.
- Overall the space use is $O\left(m + \sum_{j=1}^{m} n_j^2\right)$.
  - We'll show this is O(m).
  - So perfect hashing uses same amount of space as normal hashing.

Avoiding collisions (Optional)

- Lemma Suppose we store n keys in a hash table of size m = n^2 using universal hashing. Then with probability $\geq 1/2$ there are no collision.
- Proof There are $\binom{n}{2}$ pairs of keys that can collide.
  - Each collision occurs with probability $1/m = 1/n^2$, by universal hashing.
  - So the expected number of collisions is $\frac{\binom{n}{2}}{n^2} \leq \frac{1}{2}$.
  - By Markov's inequality the Pr[# collisions ≥ 1] $\leq$ E[# collisions] $\leq 1/2$.
- When building each hash table S_j, there's a < 1/2 of having any collisions.
  - If collisions occur, pick another random hash function from the universal family.
  - In expectation, do this twice before finding a hash function causing no collisions.

Space complexity (Optional)

- Lemma Suppose we store n keys in a hash table of size m=n. Then the secondary hash tables use space $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq 2n$, where n_j is the number of keys hashing to location j.
- Proof $E\left[\sum_{j=0}^{m-1} n_j^2\right] = E[\sum_{j=0}^{m-1}(n_j + 2\binom{n_j}{2})] = E[\sum_{j=0}^{m-1} n_j] + 2\,E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right]$
- $\sum_{j=0}^{m-1}\binom{n_j}{2}$ is the total number of pairs of hash keys which collide.
  - By universal hashing, this equals $\binom{n}{2}\frac{1}{m} = \frac{n-1}{2}$.
- $E[\sum_{j=0}^{m-1} n_j] = n$.
- So $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + \frac{2(n-1)}{2} < 2n$.

Part 1-4: Hash

Application: Bloom Filter

Approximate sets

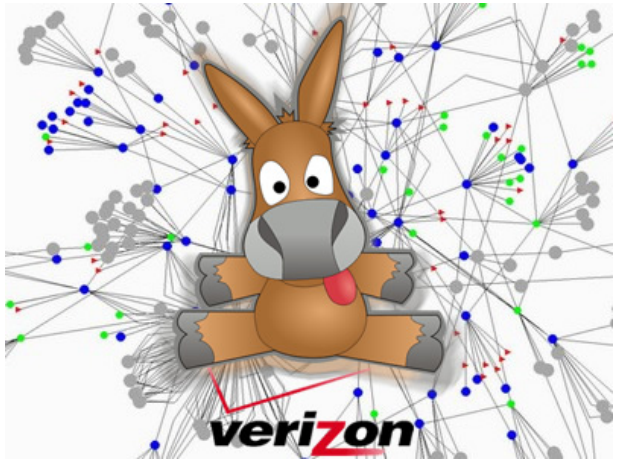
- A Bloom filter is a data structure that can implement a set.
  - It only keeps track of which keys are present, not any values associated to keys.
  - It supports insert and find operations.
  - It doesn't support delete operations.
- Bloom filters use less memory than hash tables or other ways of implementing sets.
- However, Bloom filters are approximate.
  - It can produce false positives: it says an element is present even though it's not.
    - We can bound the probability of false positives.
  - But it doesn't produce false negatives: if it says an element isn't present, then it's not.

Bloom filter applications

- Suppose we have a big database, and querying it to check if an item is present is expensive.
- We store the set of items in the database using a Bloom filter.
  - This tells us whether an item is in database or not.
- If filter an item's not present, it's definitely not in the database.
  - So no need to do an expensive query.
- If filter says an item is present, then either item is present, or there's false positive.
  - When we query the database, there's a small probability we waste time querying for a nonexistent item.
- Overall we save time by checking Bloom filter first before querying database.

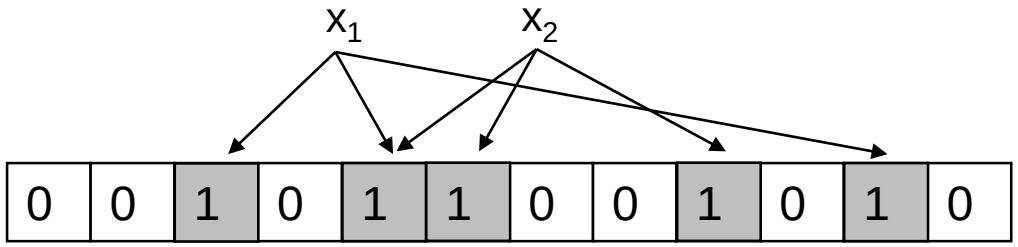
Bloom filter applications

- Consider a P2P network, where each node stores some files.
- If you want to get a file, you need to know which nodes have it.
- Keeping a list of all items stored at each node is too expensive.
- Instead, for every other node, keep a Bloom filter of its files.
- If filter says no for a node, it definitely doesn't have the file.
- If filter says yes, then either node has the file, or there's false positive and we make a useless request.
- Overall we save space, and also won't waste much communication because we rarely make useless requests.



Bloom filters

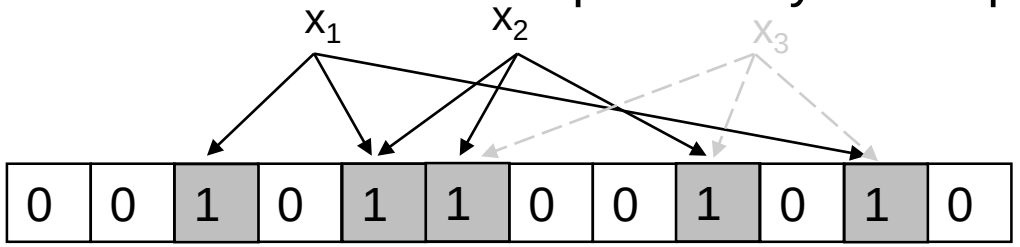
- A Bloom filter consists of
  - An array A of size m, initially all 0's.
  - k independent hash functions $h_1, \ldots, h_k$, each mapping from keys to $\{1, \ldots, m\}$.
- To store key x
  - Set $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$ all to 1.
  - Some locations can get set to 1 multiple times; that's fine.
- To check if key x is in the set
  - Read array locations $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$.
  - If all the values are 1, output "x is in set".
  - Otherwise output "x is not in set".



A Bloom filter with k=3 hash functions storing 2 items.

Correctness

- Let's look at the correctness of the search function.
- If search for x returns no, then at least one of $A[h_1(x)], \ldots, A[h_k(x)]$ equals 0.
  - So x cannot be in the set, because if x had been inserted into the set, then we would have $A[h_1(x)] = \cdots = A[h_k(x)] = 1$.
  - So there are no false negatives.
- If search for x returns yes, then $A[h_1(x)] = \cdots = A[h_k(x)] = 1$.
  - So either x was inserted into the set.
  - Or we inserted some keys that hashed to the same k locations as x.
    - So it looks as if x was inserted, even though it wasn't.
    - This is a false positive. We'll bound the probability this happens.



False positive probability 1

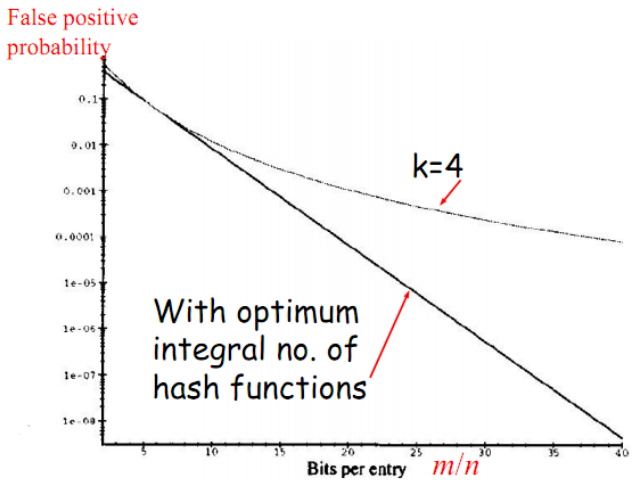
- False positive probability depends on k (number of hash functions), m (size of table) and n (number of keys inserted).
- Assume hash functions hash keys to random locations.
- When inserting one key, we set k random locations to 1.
- Fix any position i. Probability i is set to 1 by a hash function is $1/m$, so probability i stays 0 is $1 - 1/m$.
  - After k hashes, prob. i still 0 is $(1 - 1/m)^k$.
  - To insert n items, we used nk hash functions. So prob. i still 0 after all these is $p = (1 - 1/m)^{nk}$.
- We now use an approximation $\left(1 - \frac{1}{m}\right)^{nk} \approx e^{-\frac{nk}{m}}$.

False positive probability 2

- So probability any position i is 1 after n keys inserted is $1 - p \approx 1 - e^{-\frac{nk}{m}}$.
- Now, assume there are (1-p)m positions that are 1 in the array.
  - This isn't quite correct. The actual number of 1's in the array is a random variable, whose expectation is (1-p)m.
  - However, we can make the argument rigorous by showing that the actual number of 1's is $(1-p)m + \sqrt{m \log m}$ with high probability.
- We only get a false positive if when we check k random locations, they're all 1.
  - Probability is $f = (1-p)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k$.

False positive probability 3

- Notice the false prob. $\left(1 - e^{-\frac{nk}{m}}\right)^k$ is a function of k, the number of hash functions we use.
- We find k to minimize the false positive prob. by differentiating f wrt k and solving.
- The optimum k is $\frac{m \ln(2)}{n}$, which leads to $f = \left(\frac{1}{2}\right)^k \approx 0.6185^{\frac{m}{n}}$.
  - Notice that m/n is the average number of bits per item. So error rate decreases exponentially in space usage.



Improvements

- Right now Bloom filters can't handle deletes.
  - Say keys k_1, k_2 hash to two overlapping sets of locations. If you delete k_1 by setting some of its locations to 0, you could also delete k_2.
- Deletes can be done by storing a count of how many keys hashed to that location, and inc / dec the counts when inserting or deleting.
  - But this uses more memory.
  - Also, what if the counts overflow?
- Neat trick Given Bloom filters for sets S_1, S_2, we can create Bloom filter for $S_1 \cap S_2$ and $S_1 \cup S_2$ just by bitwise ANDing or ORing S_1 and S_2's filters.

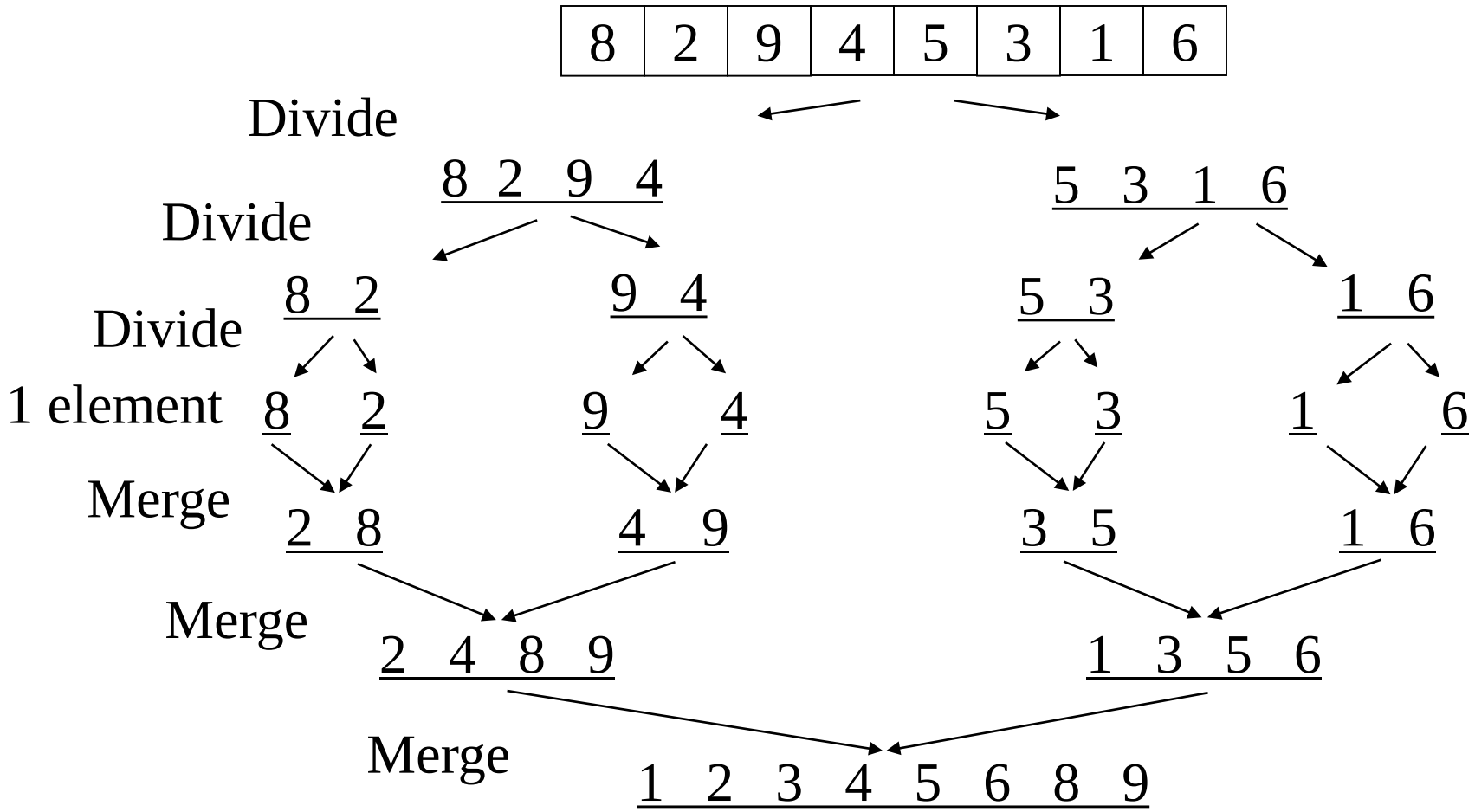
Part 2-1: Merge Sort

Algorithm

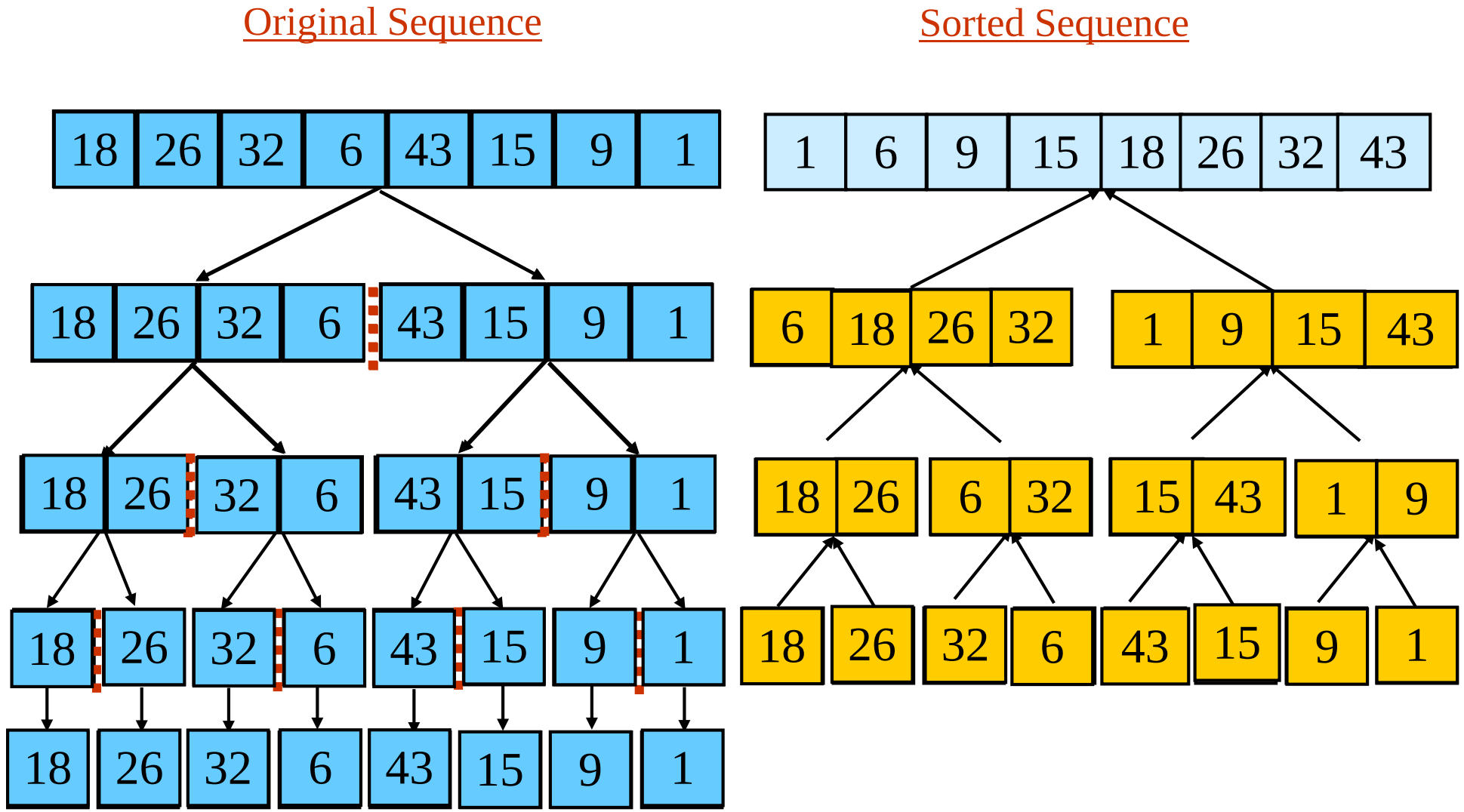
Divide And Conquer

- Merging two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the left side then the right side and then merging the left and right back together.

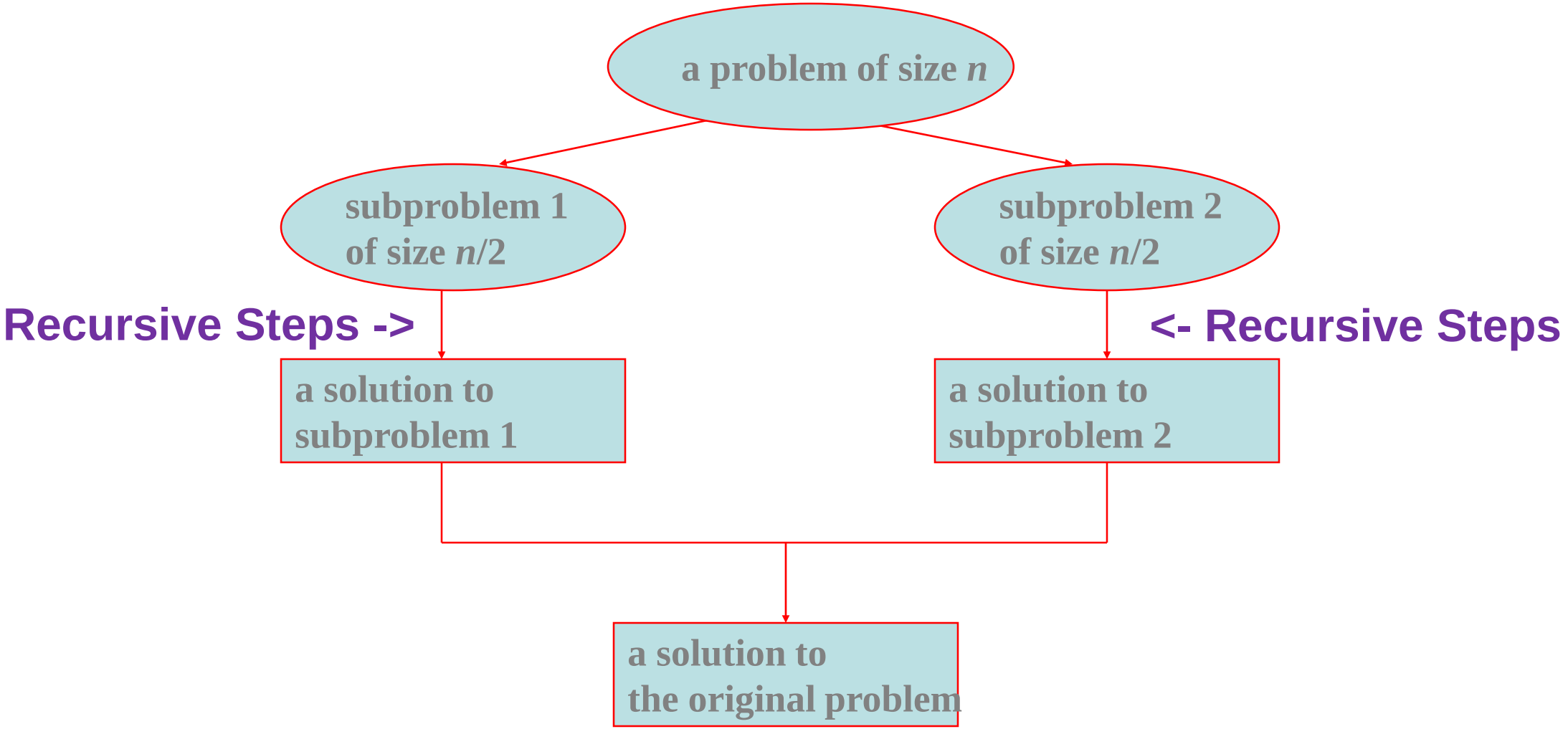
Merge Sort Overview



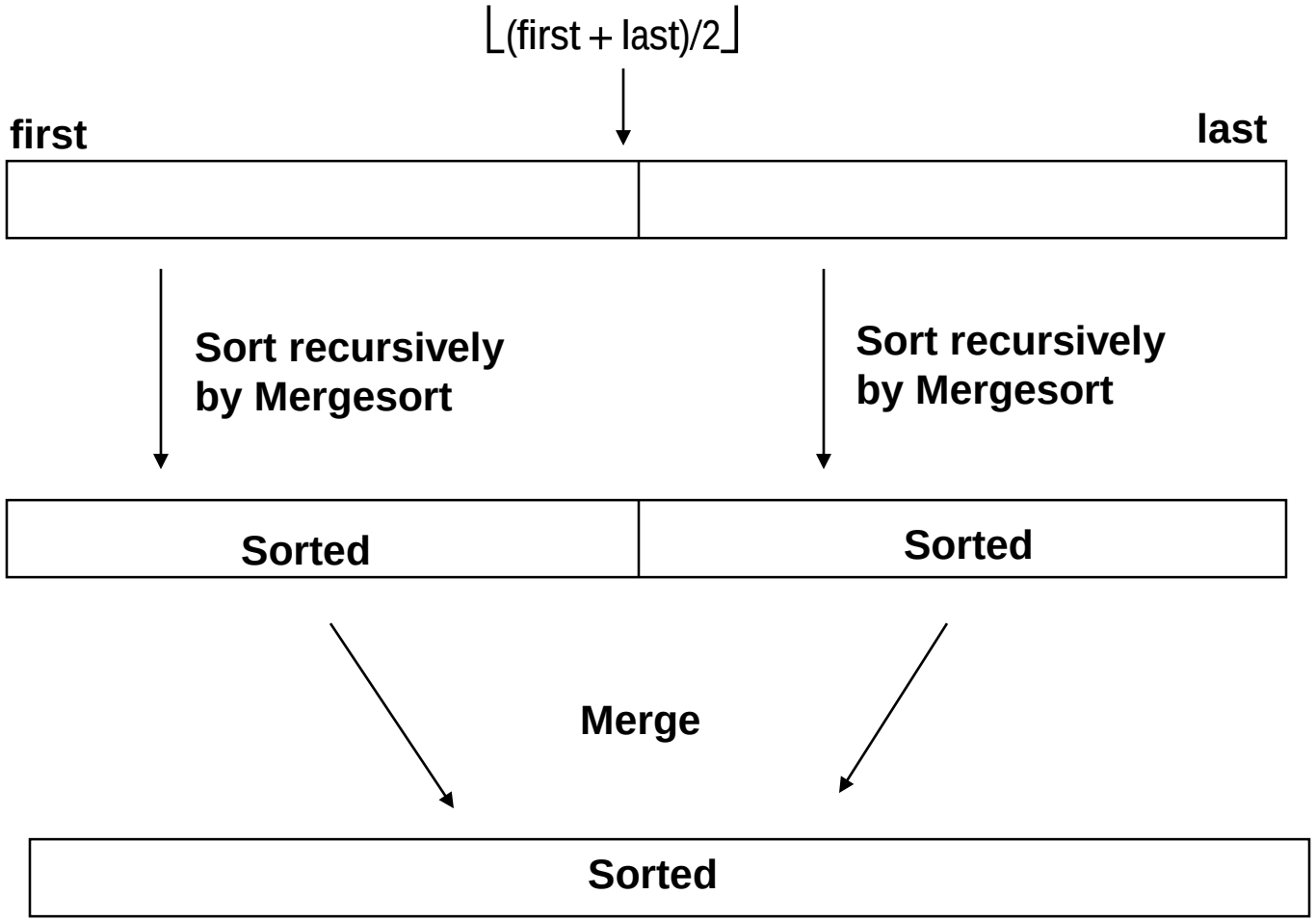
Merge Sort – Example



Divide and Conquer



Using Divide and Conquer: Merge Sort strategy

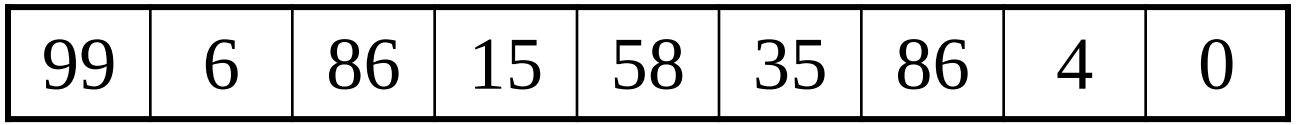


Merge Sort Algorithm

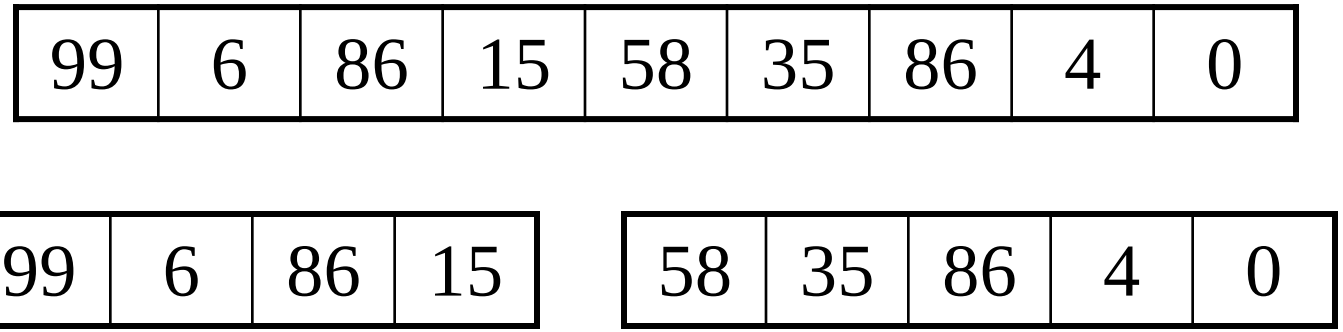
- Divide: Partition n elements array into two sub lists with $n/2$ elements each. Partition until the length of sub lists is 1.
- Conquer: Sort sub list1 and sub list2 by merging.
- Combine: Merge sub list1 and sub list2 at last.

```
void mergesort(n) {
  if (n==1)
    return;
  else {
    L=mergesort(n/2);
    R=mergesort(n/2);
  }
  // takes O(n) time
  merge(R,L);
}
```

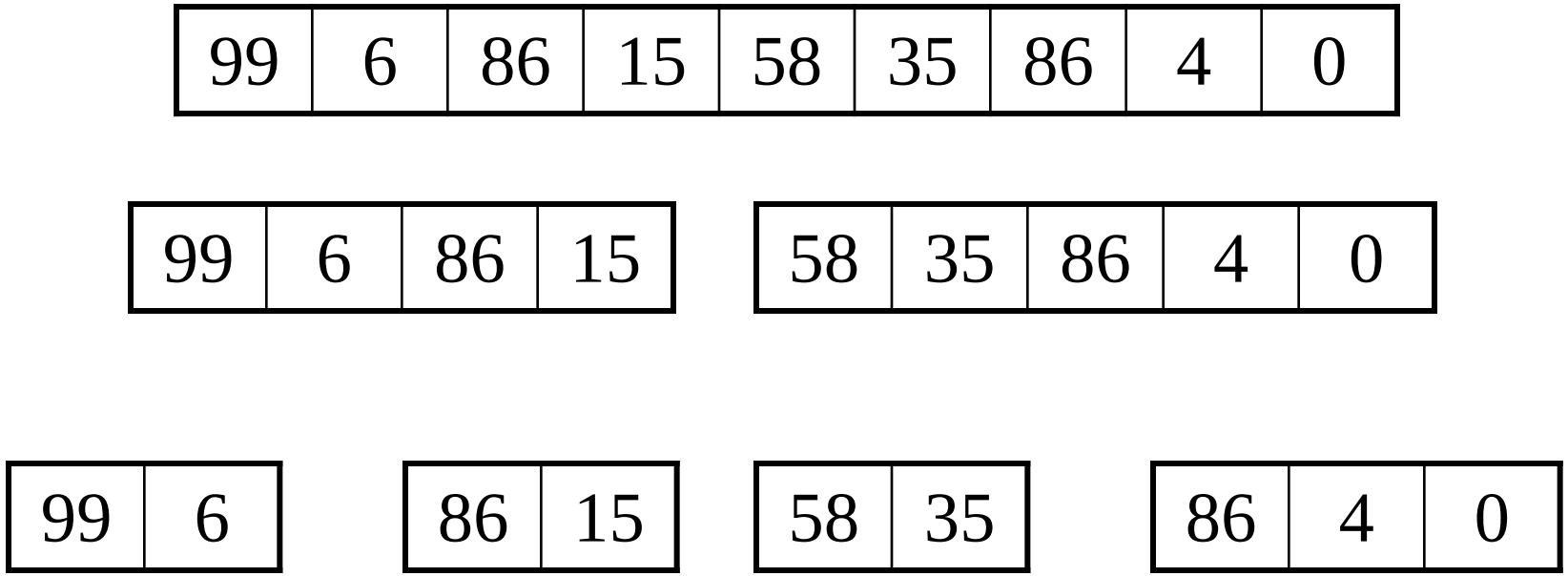
Merge Sort Example - Divide



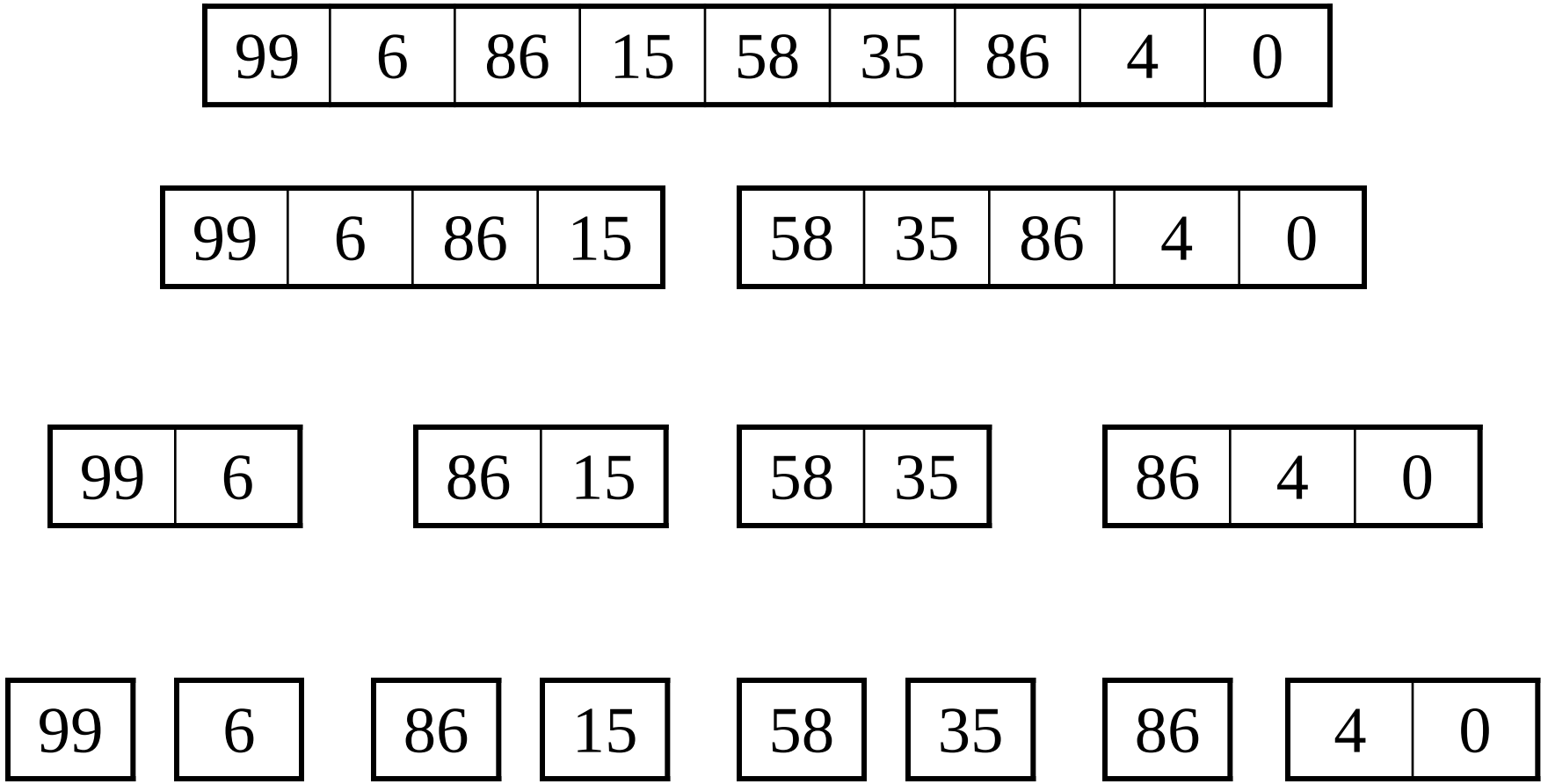
Merge Sort Example - Divide



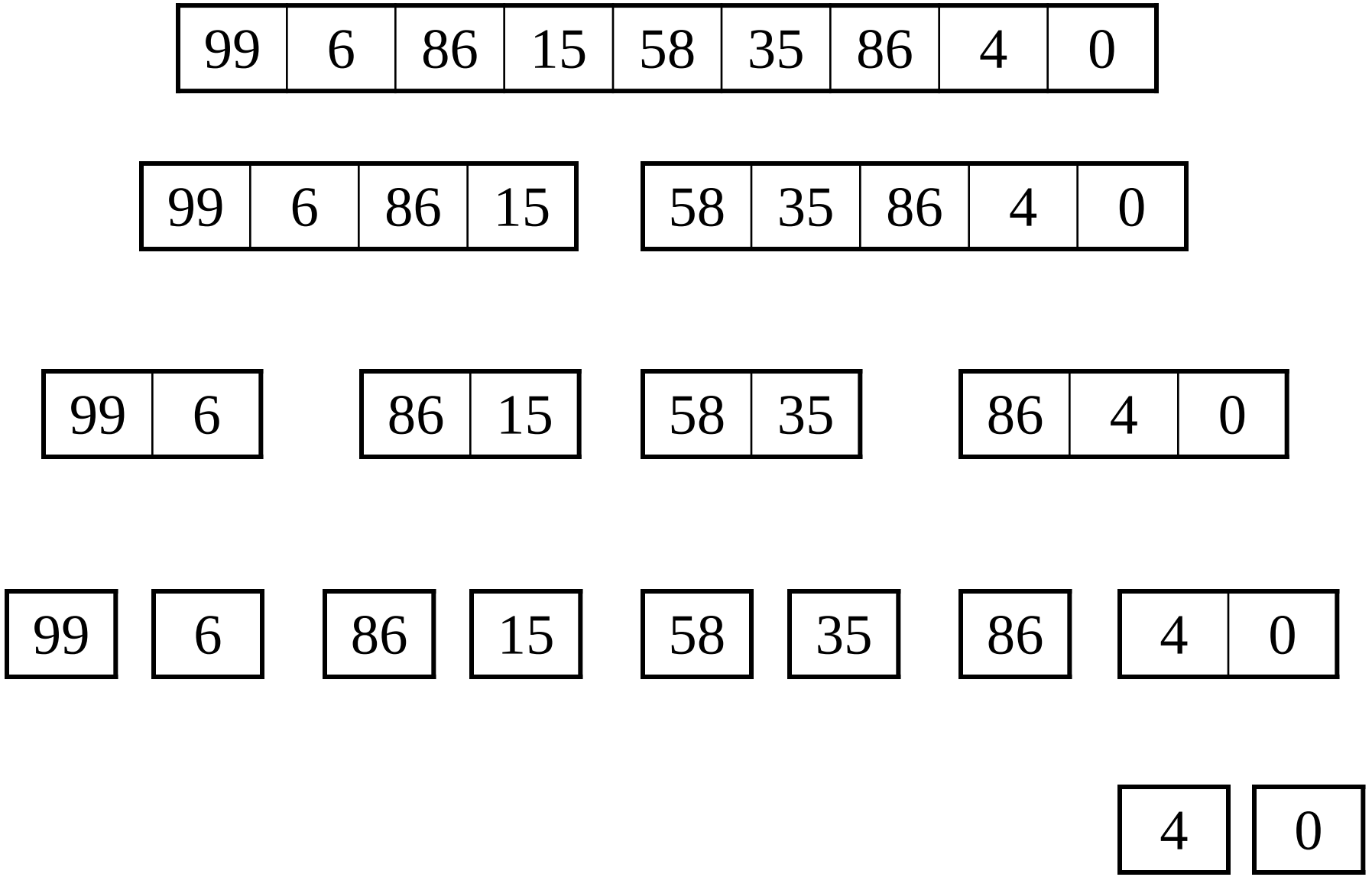
Merge Sort Example - Divide



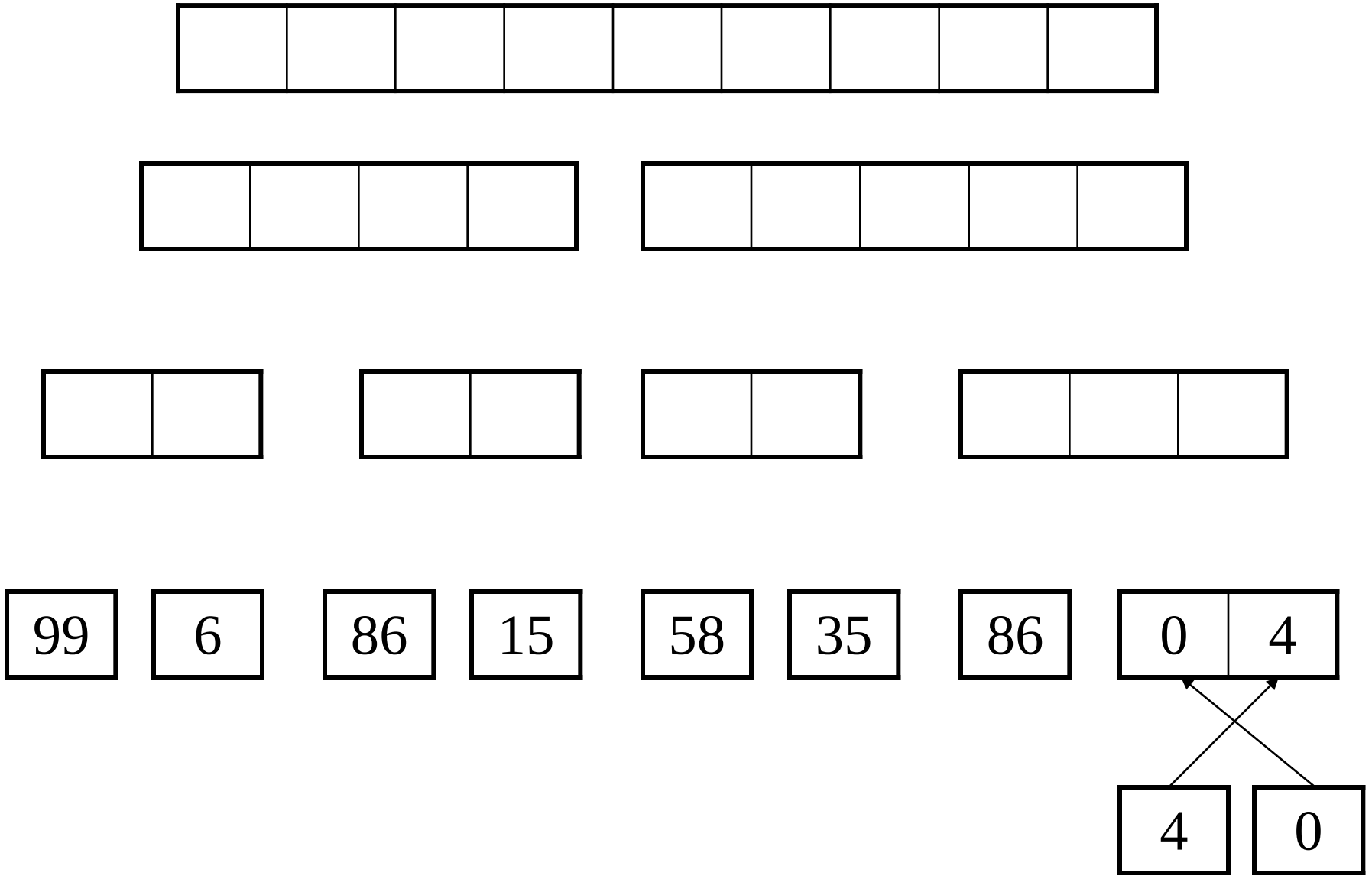
Merge Sort Example - Divide



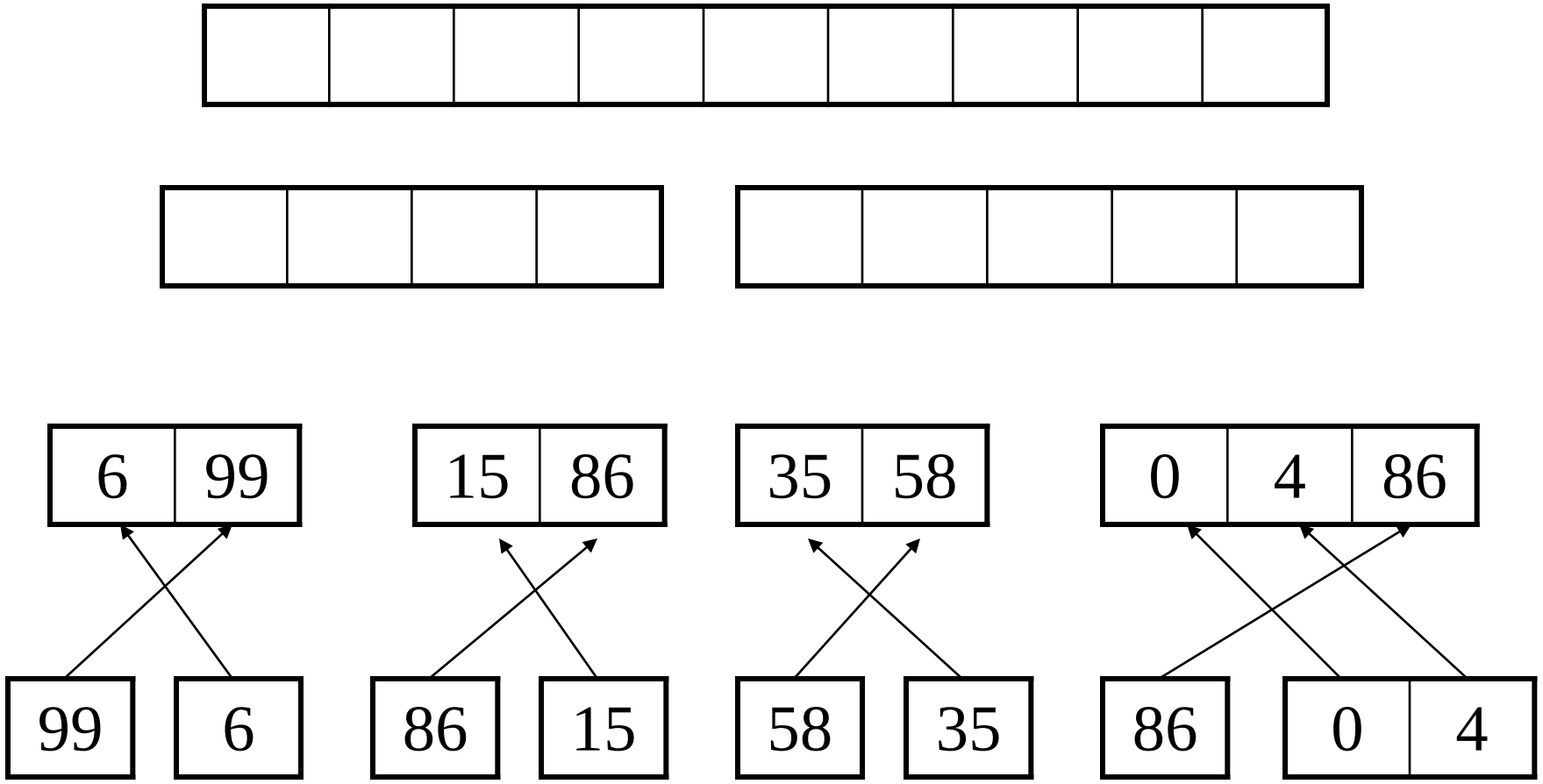
Merge Sort Example - Divide



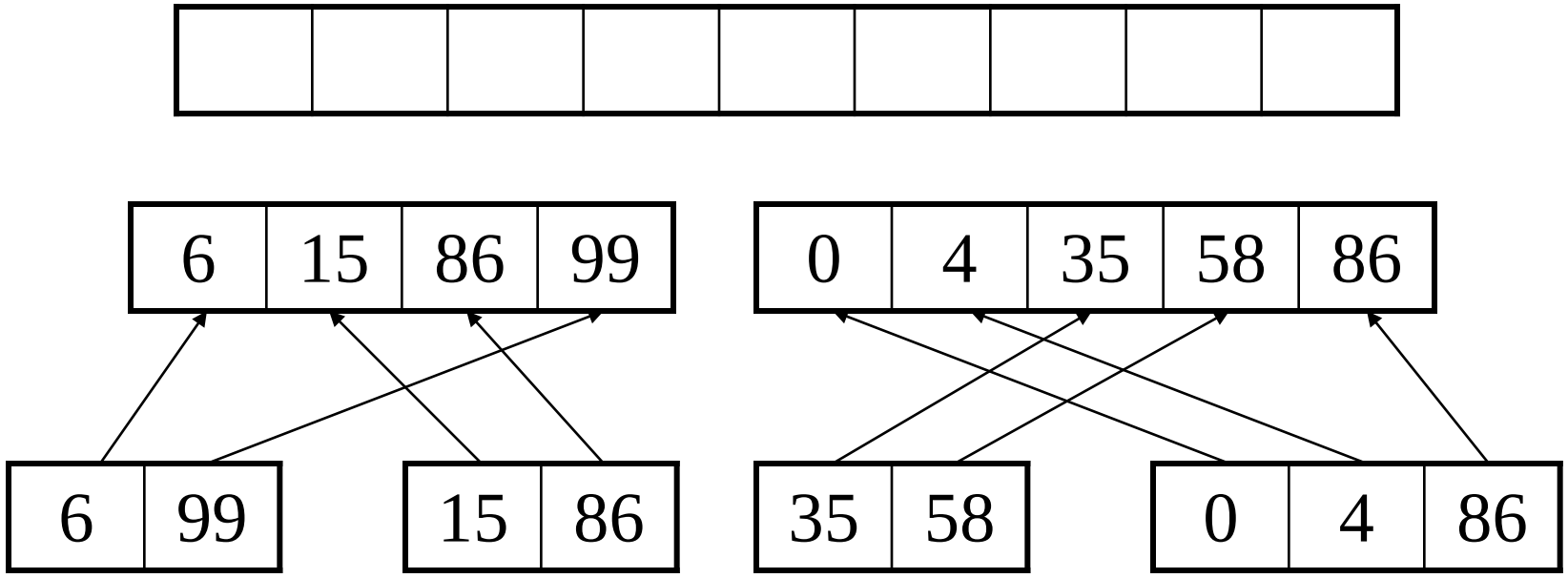
Merge Sort Example - Merge



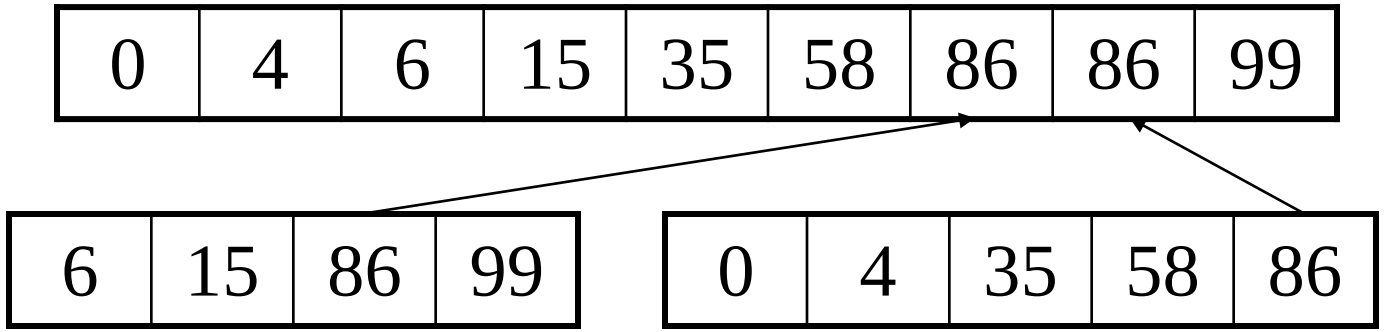
Merge Sort Example - Merge



Merge Sort Example - Merge



Merge Sort Example - Merge



Merge Sort Example - Merge

0	4	6	15	35	58	86	86	99

Time Complexity Analysis

- Worst case: $O(n \log n)$
- Average case: $O(n \log n)$
- Best case: $O(n \log n)$
- Additional Space: $O(n)$ -> Not in place. Why?
- How to prove time complexity?

Time Complexity Analysis

- Compared to additional space, Why we use Merge Sort more often instead of Heap Sort?
- Although time complexity is the same, generally merge sort will be significantly faster on a typical system with a 4 or greater way cache, since merge sort will perform sequential reads from two runs and sequential writes to a single merged run, while heap needs to maintain its heap property.
- It is related to **computer architecture.**

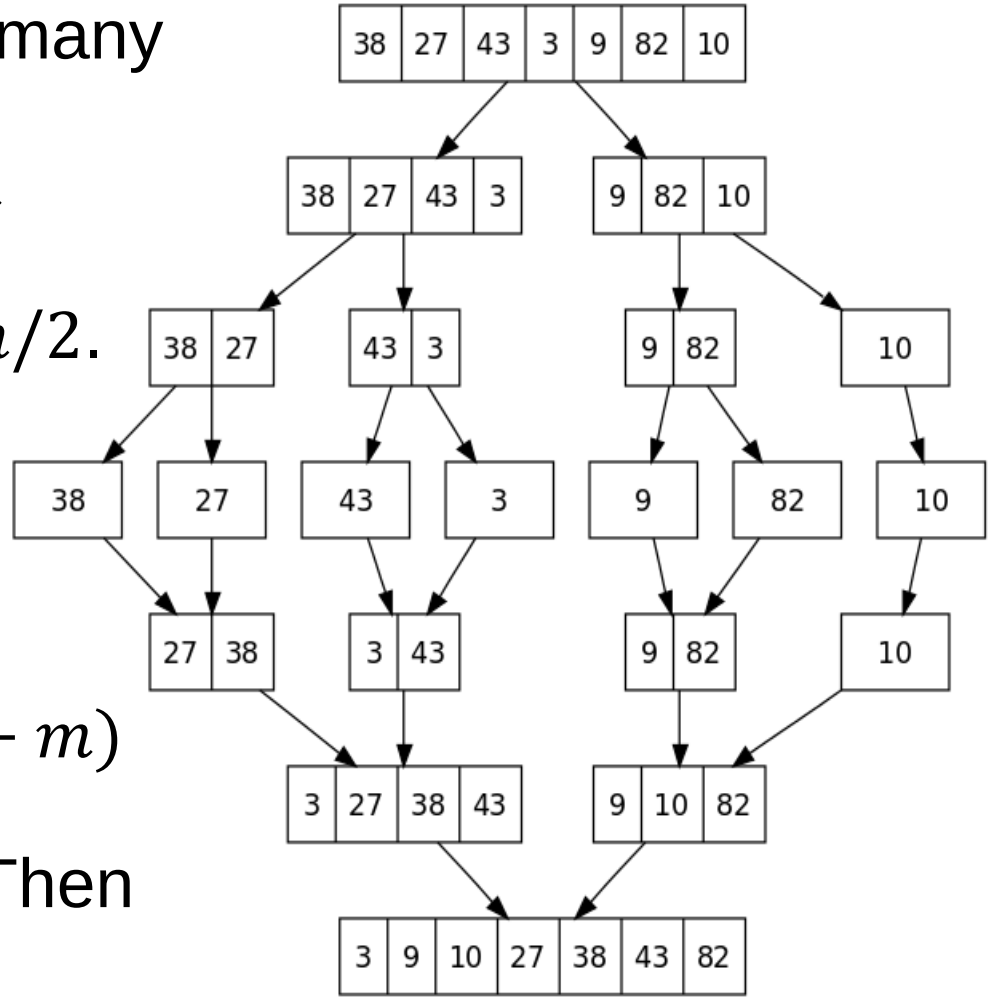
	Best	**Average**	**Worst**	**Additional Space**
Insert Sort	$\Omega(n)$ $O(n+d)$ *inversion*	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$\Omega(n)$ Optimization	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Heap Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

Part 2-2: Merge Sort

Solve Recursive Function: Recursion Tree

Solve Recursive Function: Recursion Tree

- Used for recursive algorithms that split into many branches.
- Ex Mergesort algorithm to sort an array of n numbers.
- Divide the array into two subarrays of size $n/2$.
- Recursively sort each subarray.
  - If array size = 1, just return the array.
- Merge two sorted subarrays into one sorted array.
  - Merging lists of size n and m takes $O(n+m)$ time.
- Let $S(n)$ be time complexity of mergesort. Then $S(n) = 2S\left(\frac{n}{2}\right) + O(n), S(1) = 1.$

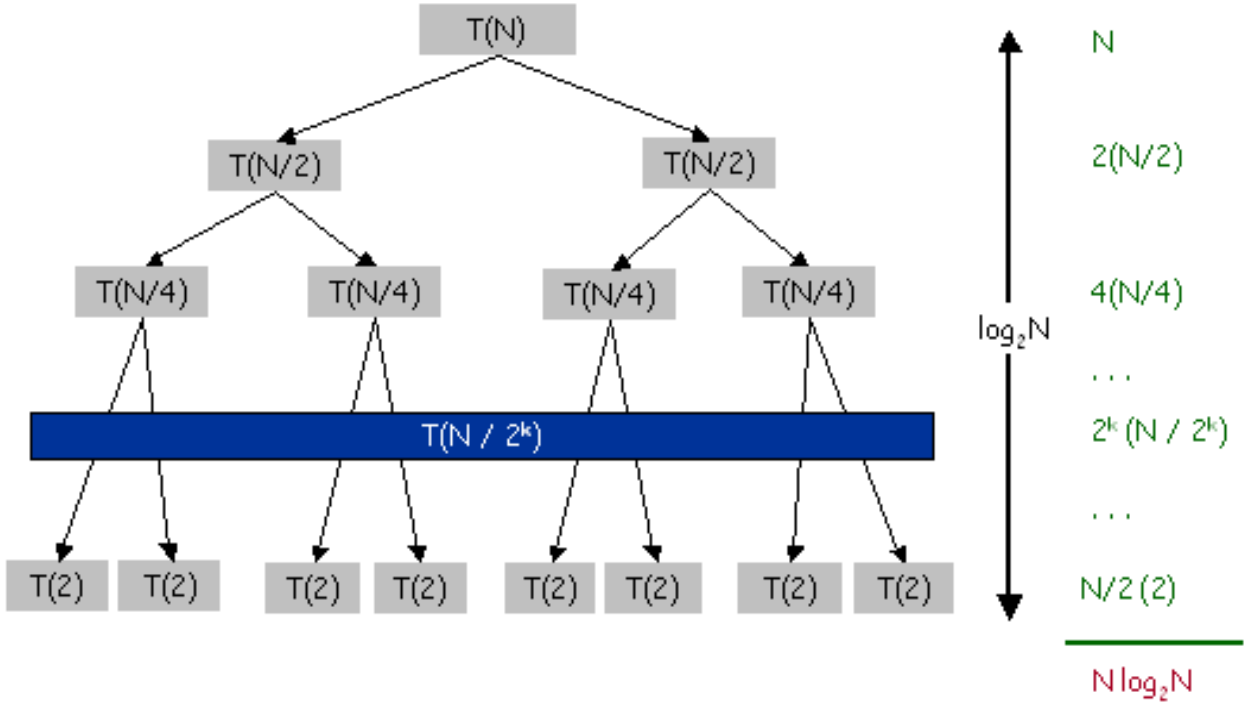


Solve Recursive Function: Recursion Tree

- Visualize the recursive calls that occur during mergesort(n).
- There are $\log_2 n$ levels in the recursion tree.
- At level i, there are 2^i recursive calls mergesort($n/2^i$).
- Each call does $n/2^i$ work in merge function.
  - So total work at level i is $2^i * \left(\frac{n}{2^i}\right) = n$.
- So total work overall is $S(n) = n \log_2 n$.

```
void mergesort(n) {
  if (n==1)
    return;
  else {
    L=mergesort(n/2);
    R=mergesort(n/2);
  }
  // takes O(n) time
  merge(R,L);
}
```

- (1) Calculate how many levels.
- (2) Calculate how many recursive calls each level has.
- (3) Calculate how many work does each recursive call does.
  - Then calculate the total work on each level.
- (4) Calculate how many total work you need to do.



Part 2-3: Merge Sort

Counting Inversions

Counting Inversions

- In an array A, the elements at indices i and j (where $i < j$) form an inversion if $A[i] > A[j]$.
- Now you need to count the pairs of inversions in an array.
- Methods 1: Scan the array for each elements, which costs $O(n^2)$.
- Methods 2: Divide and Conquer, using merge sort.

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

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.

Divide: $O(1)$

Conquer: $2T(\frac{n}{2})$

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

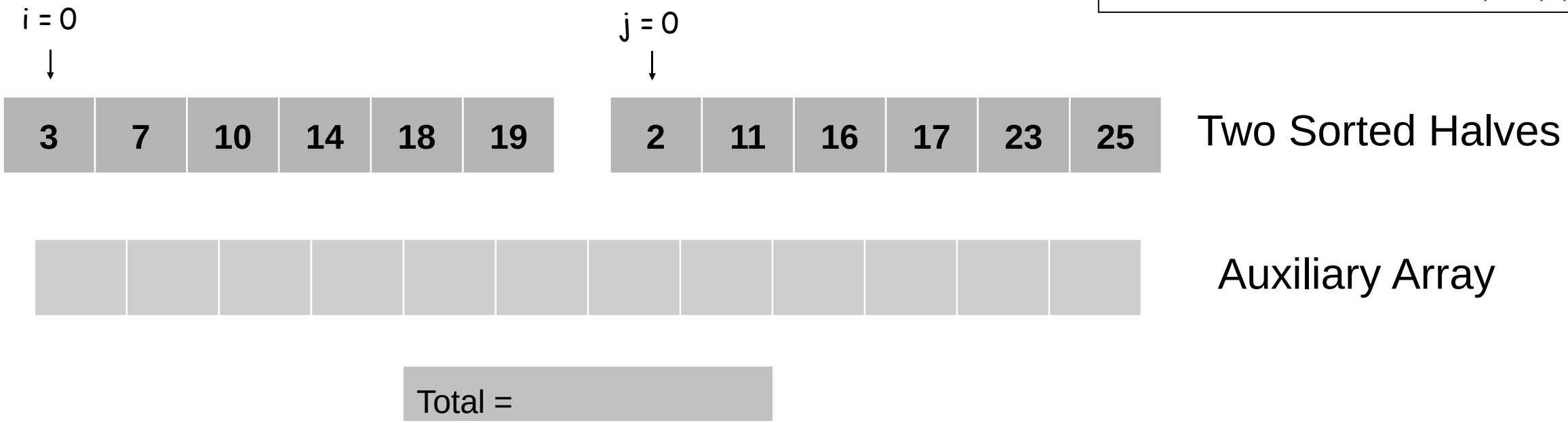
Combine: ?

Total = 5 + 8 + 9 = 22.

Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

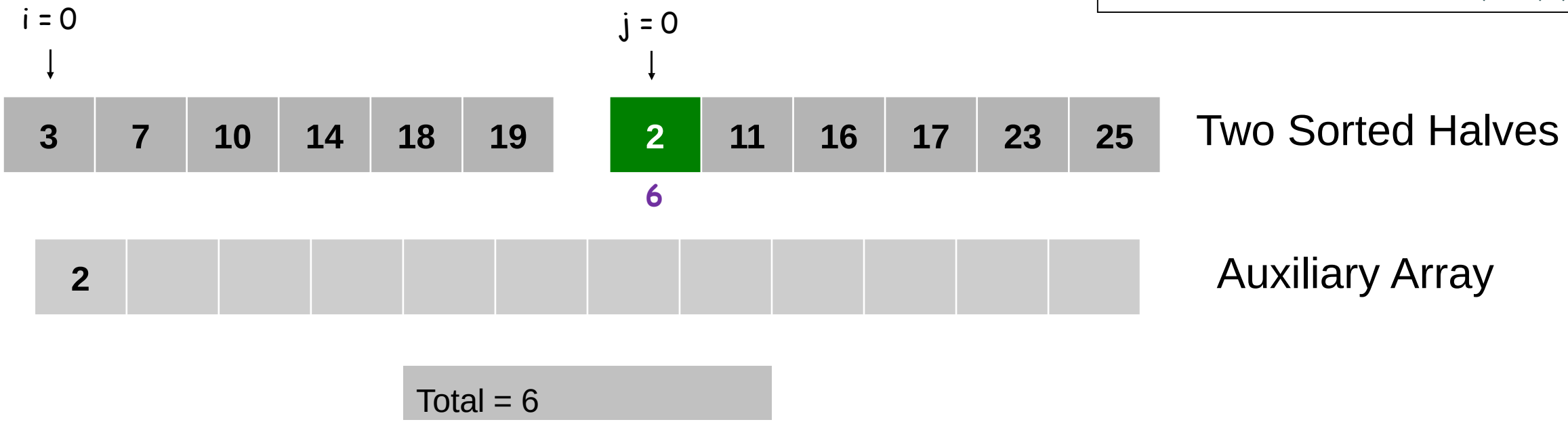
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

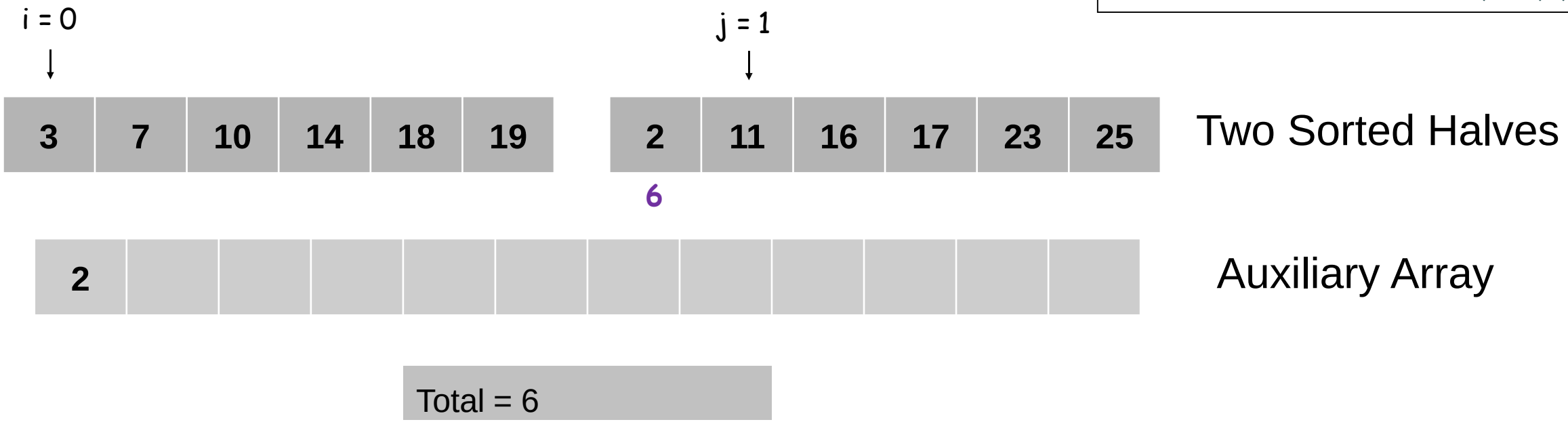
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

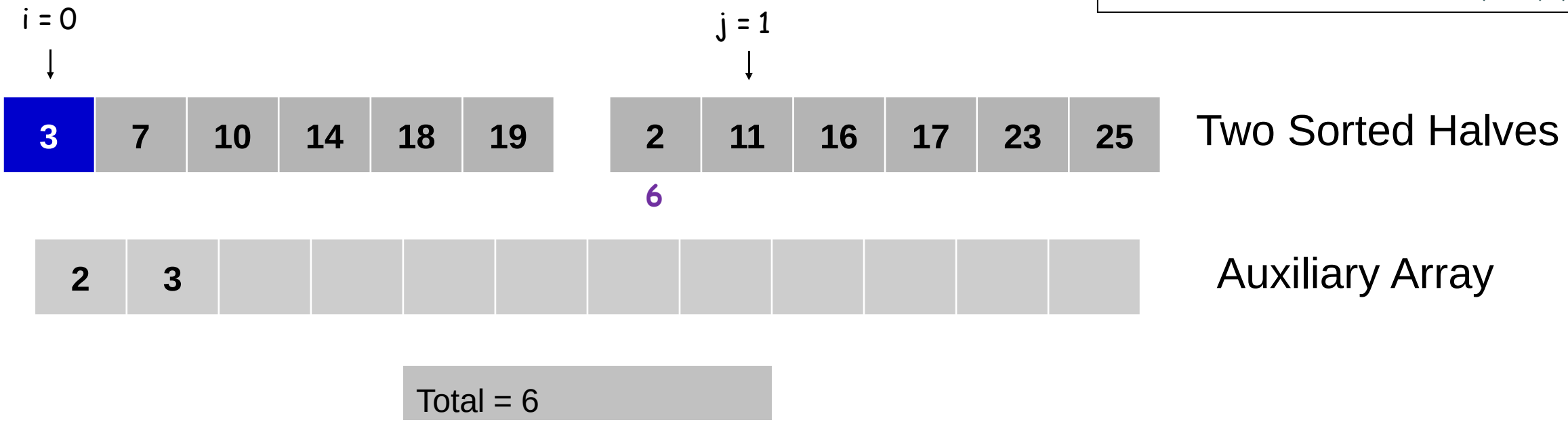
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

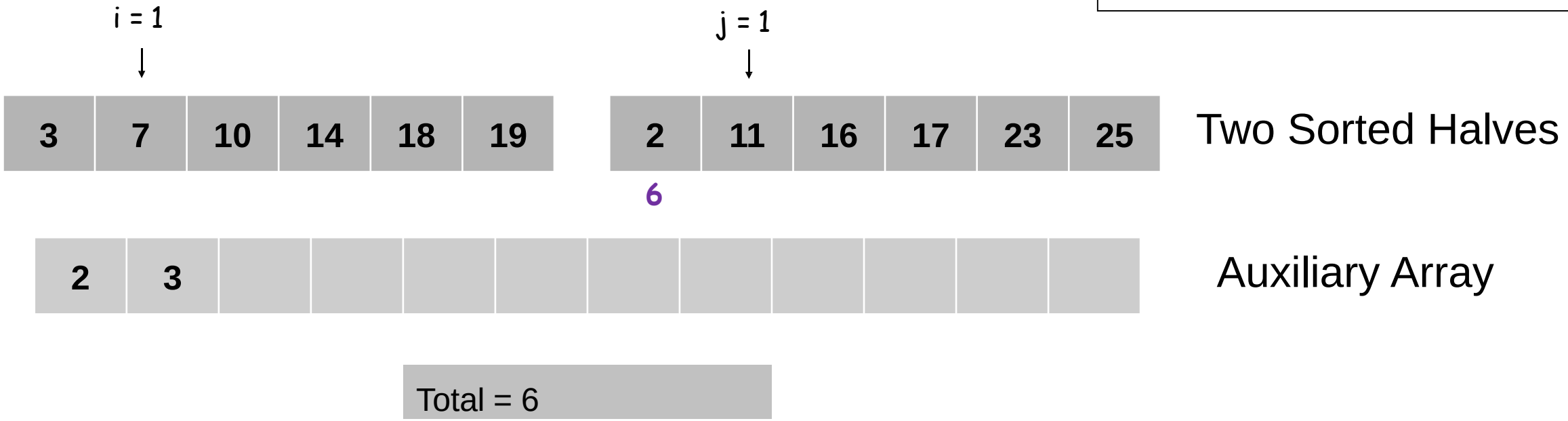
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

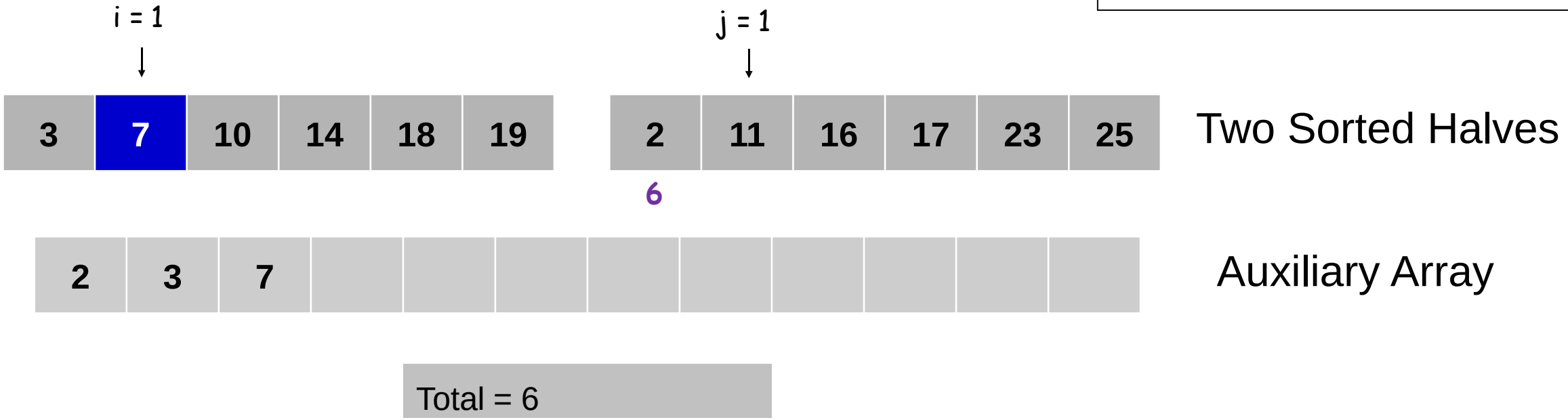
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

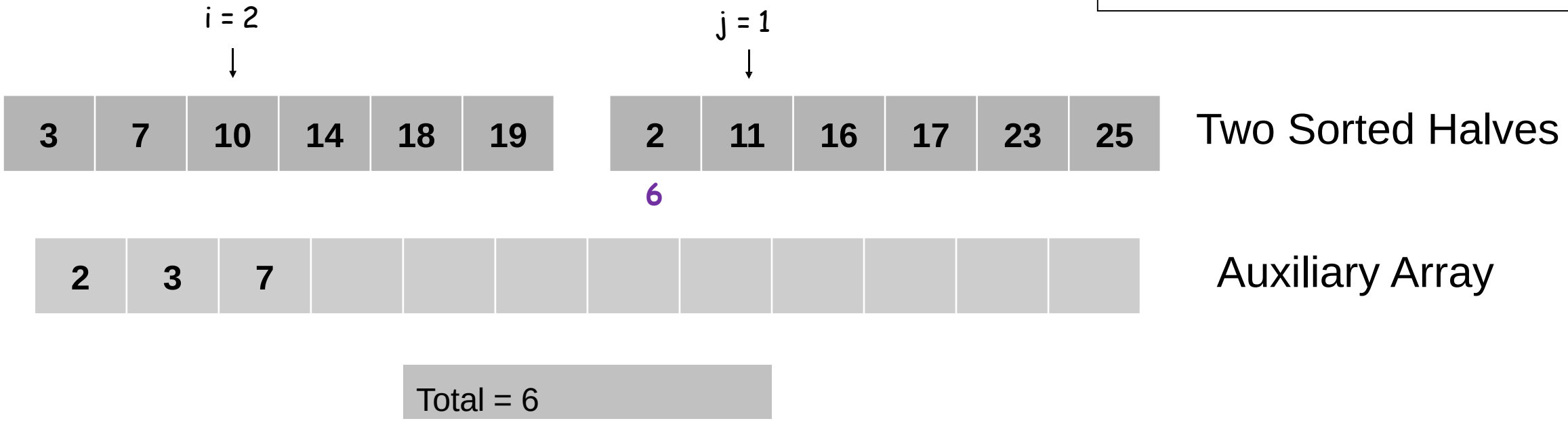
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

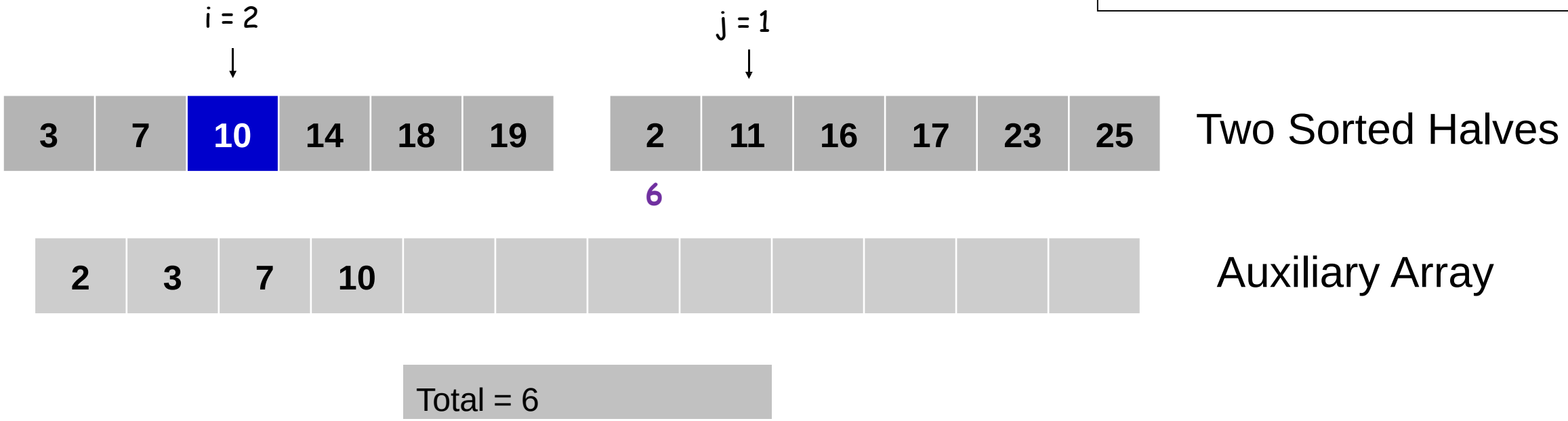
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

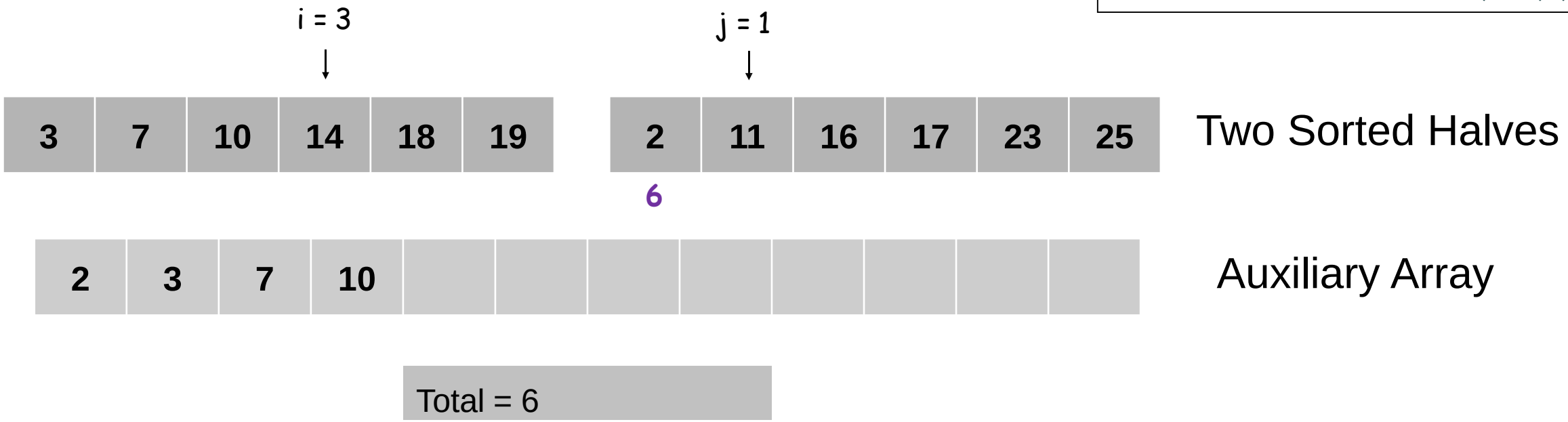
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

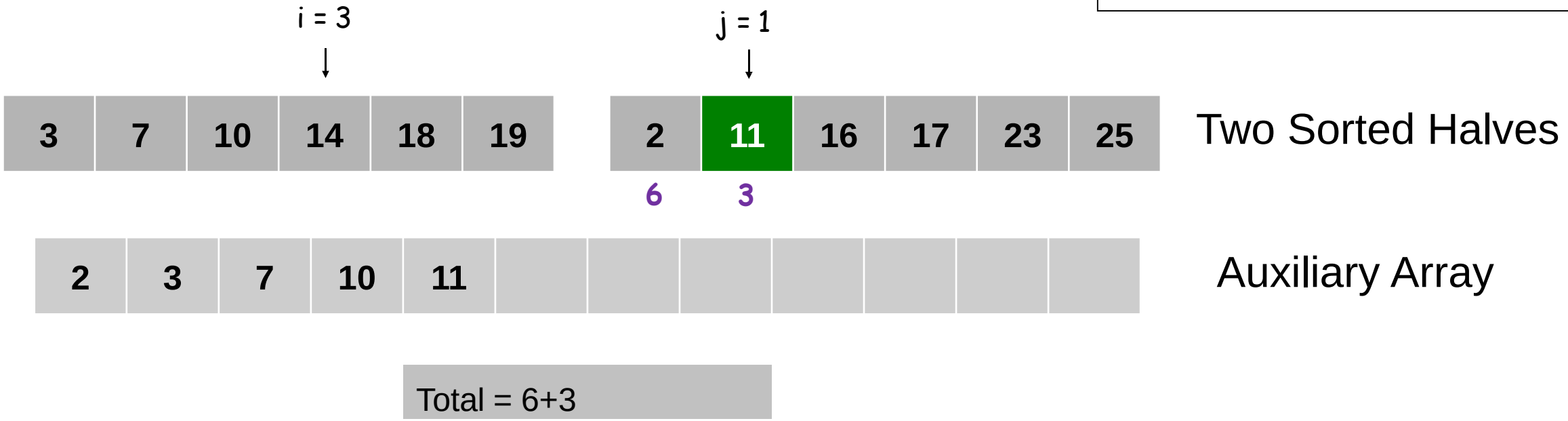
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

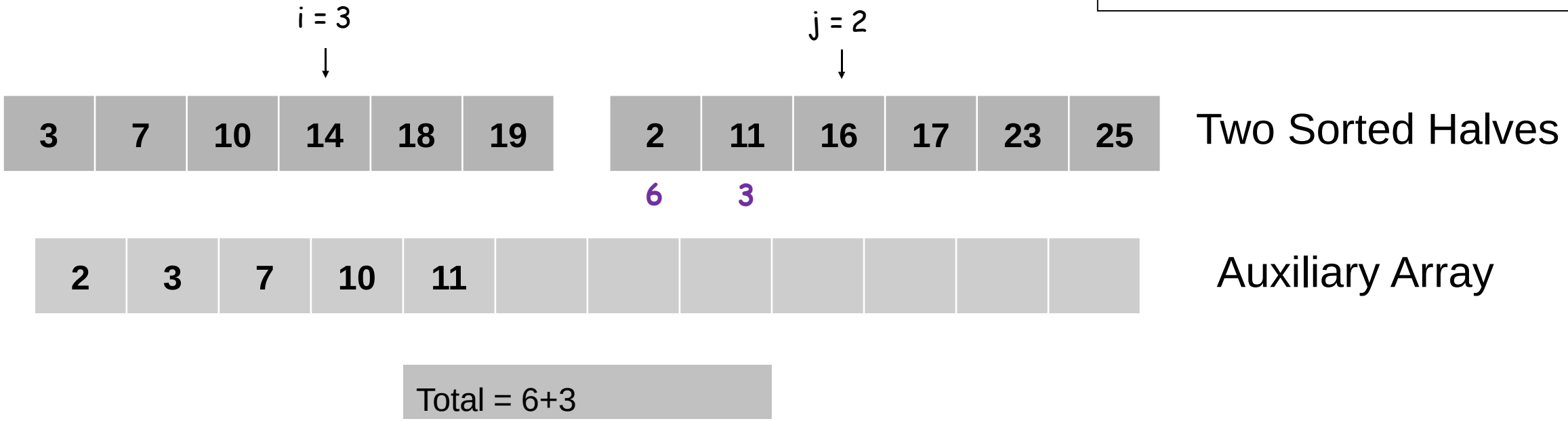
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

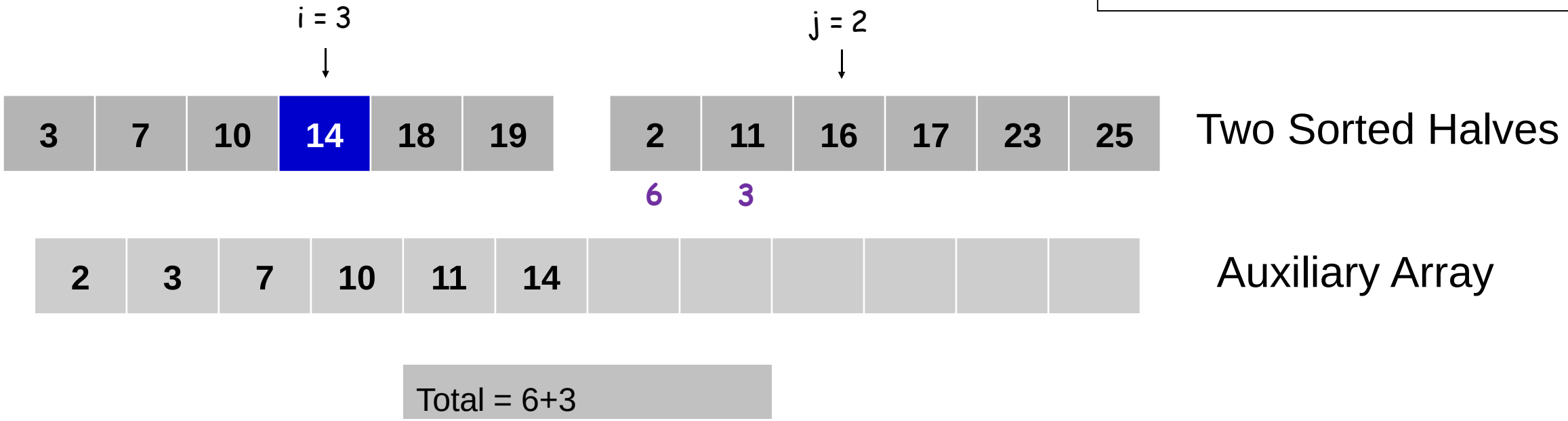
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

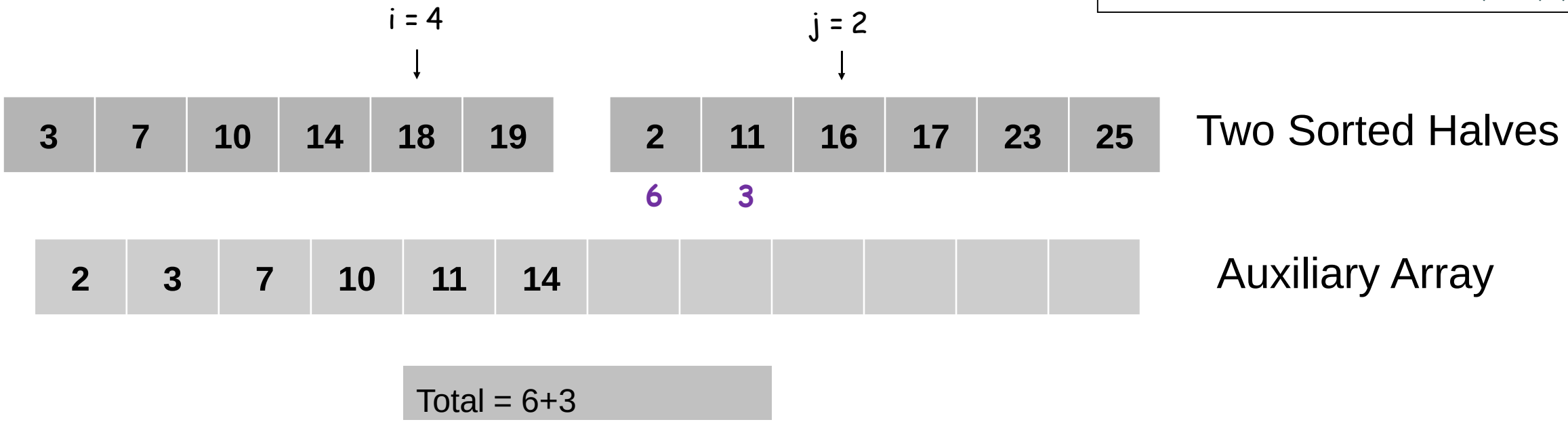
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

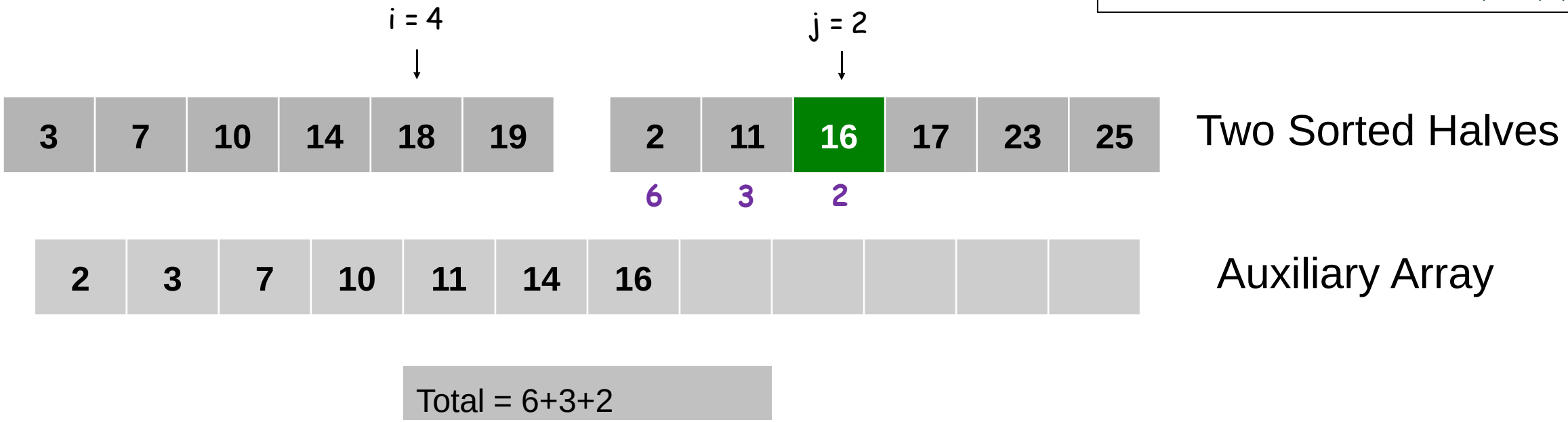
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

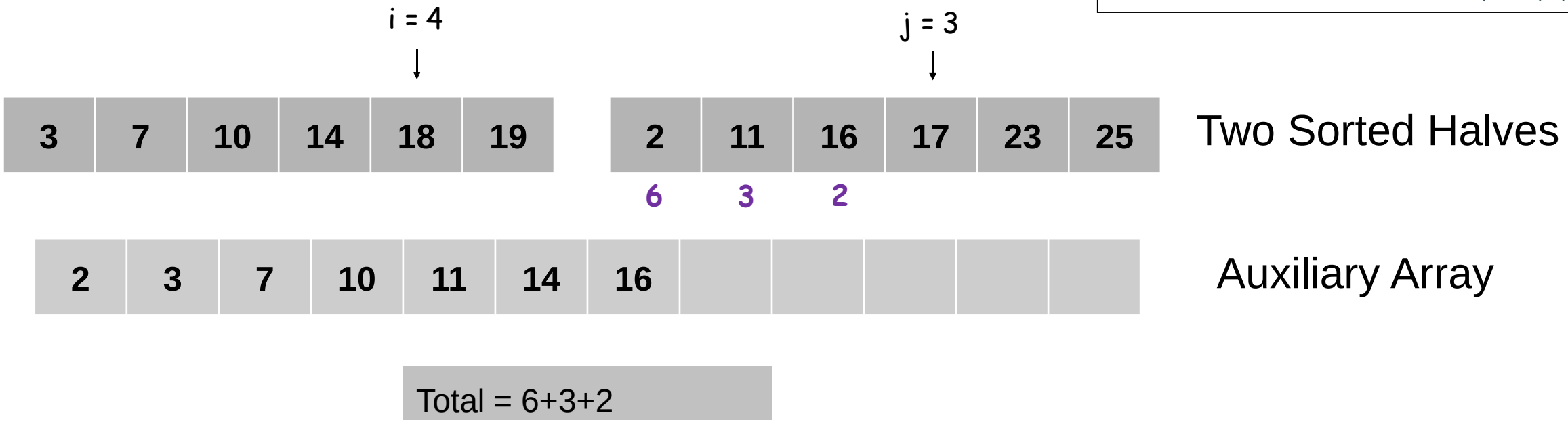
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

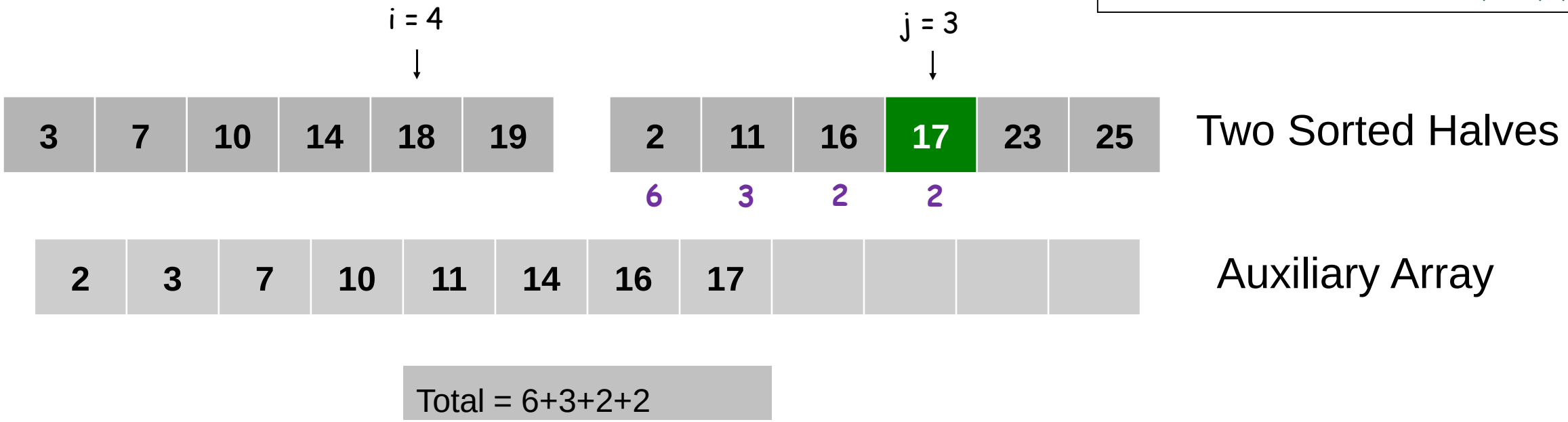
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

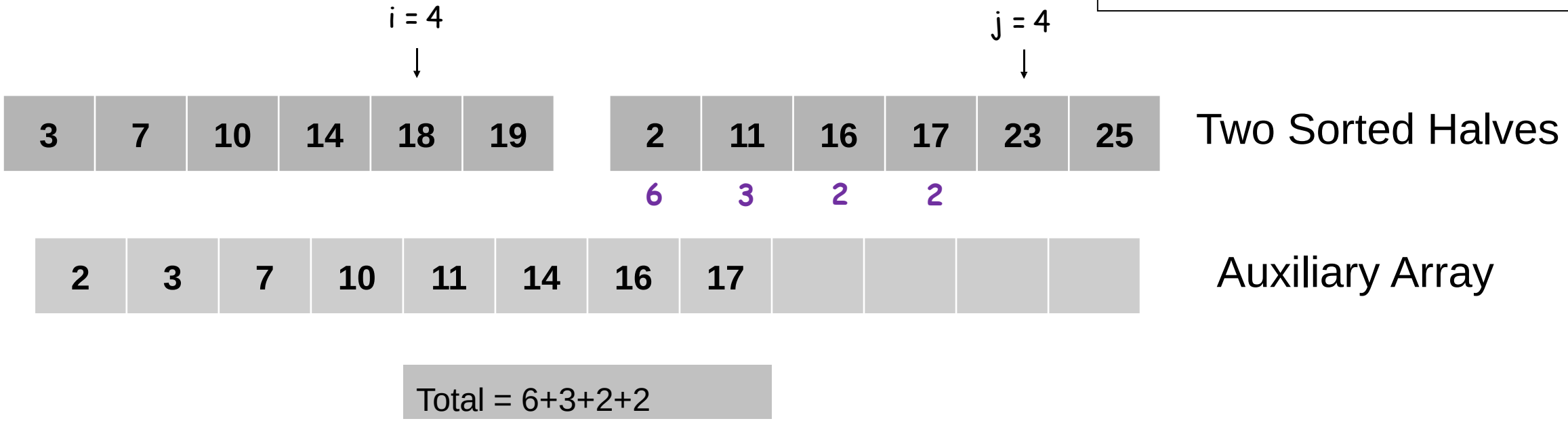
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

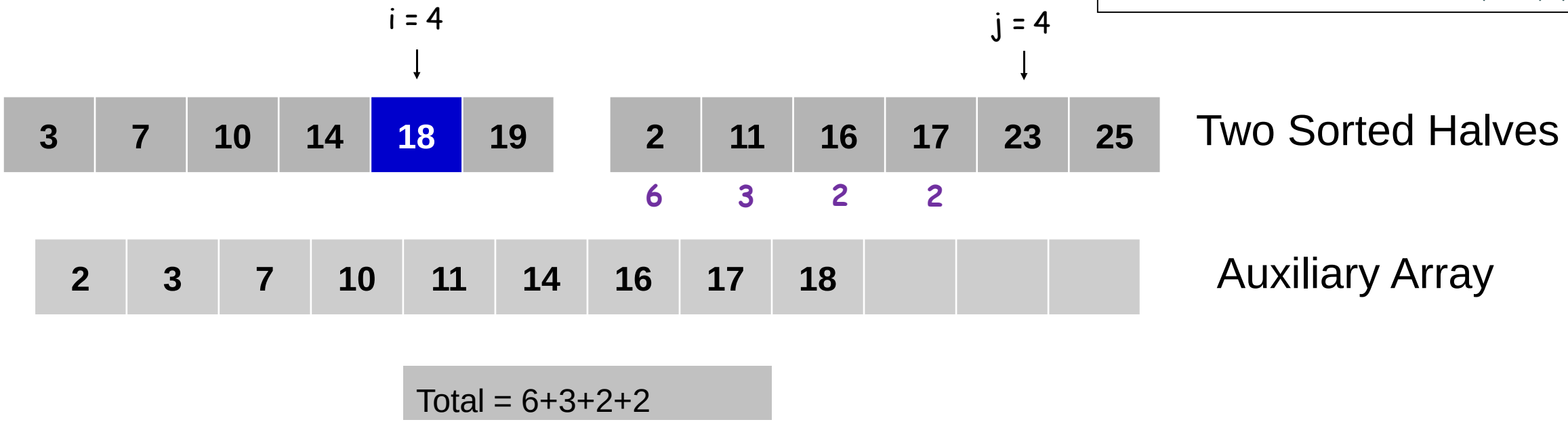
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

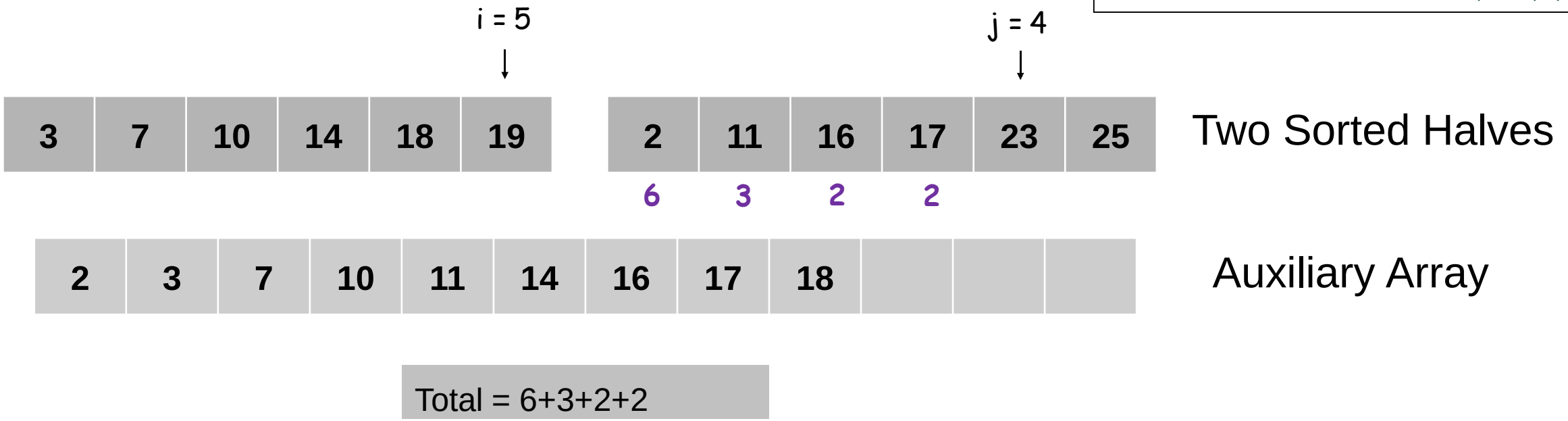
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

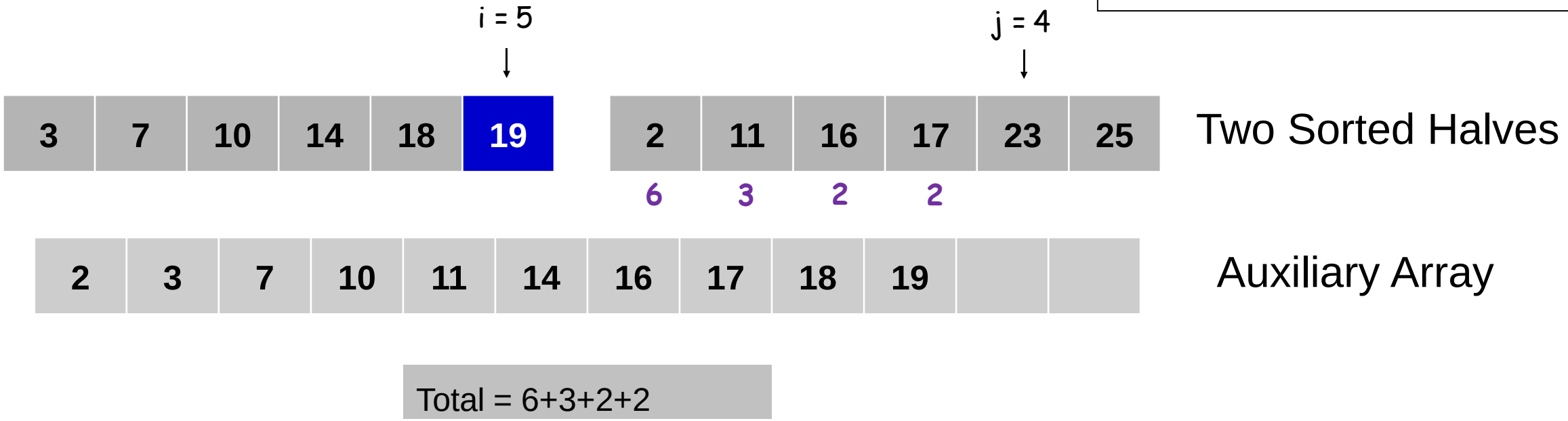
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

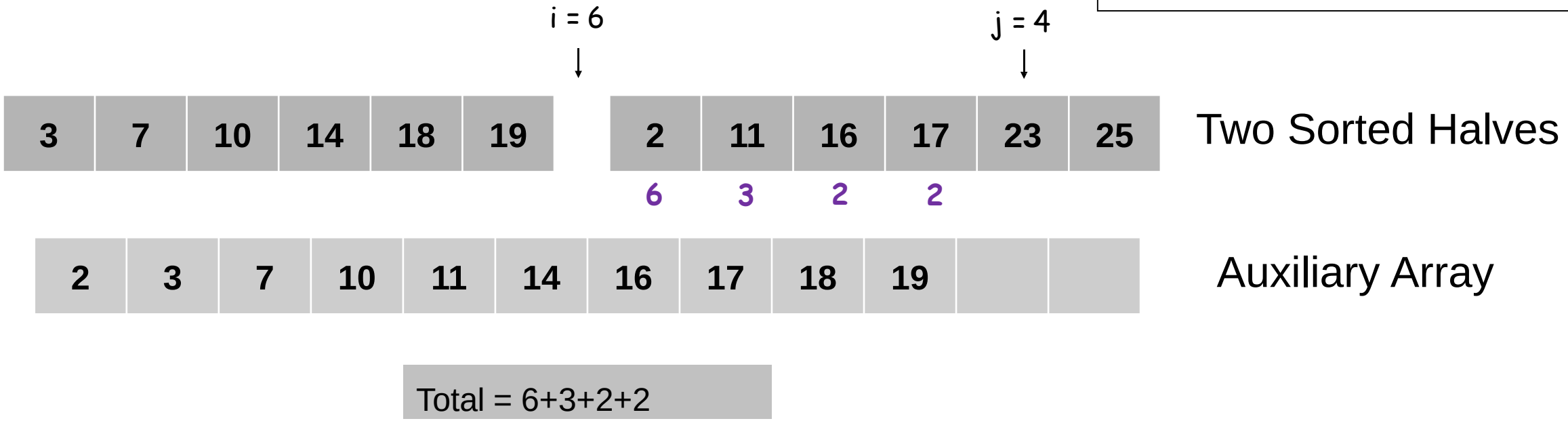
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

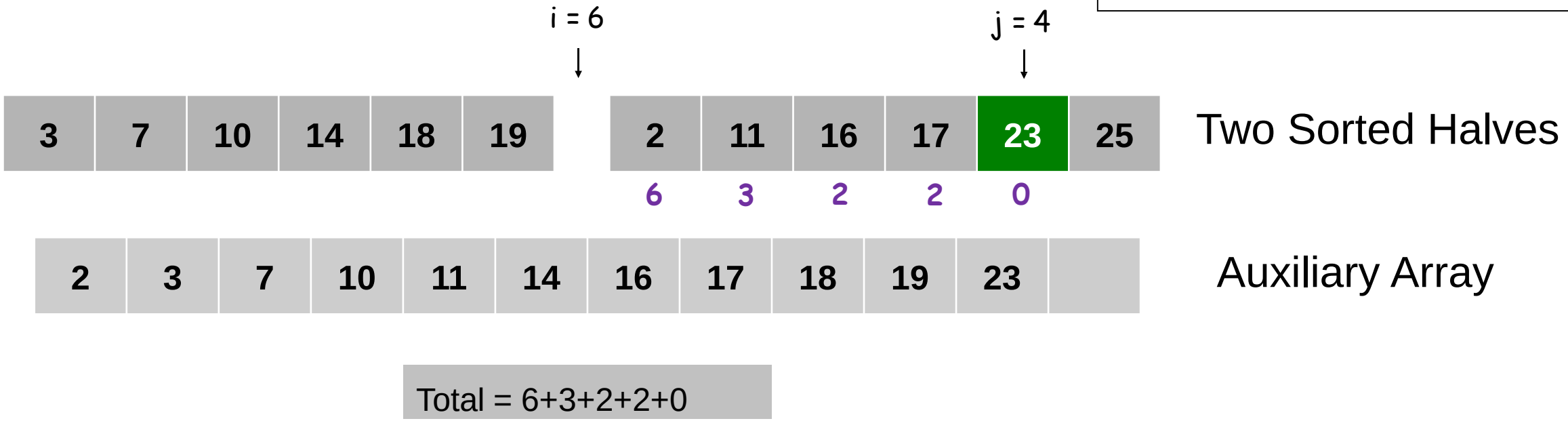
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

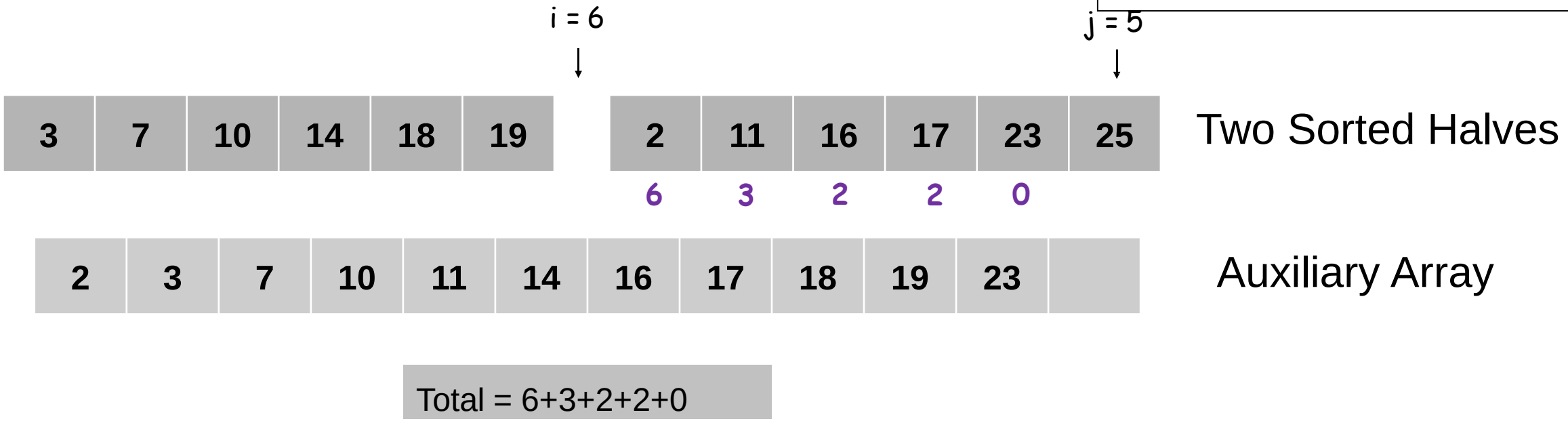
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

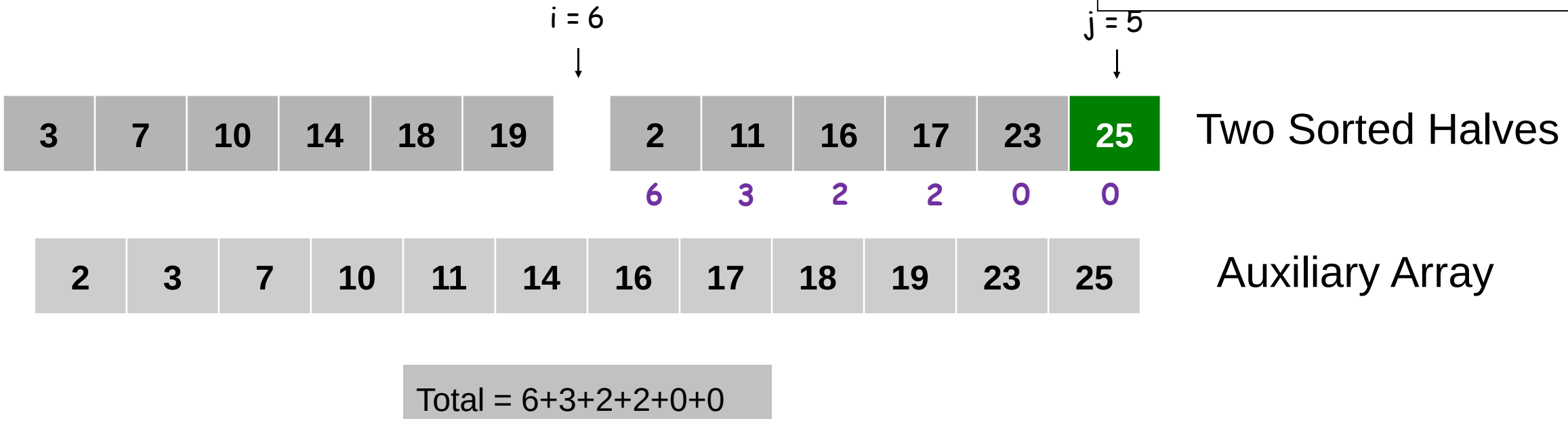
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

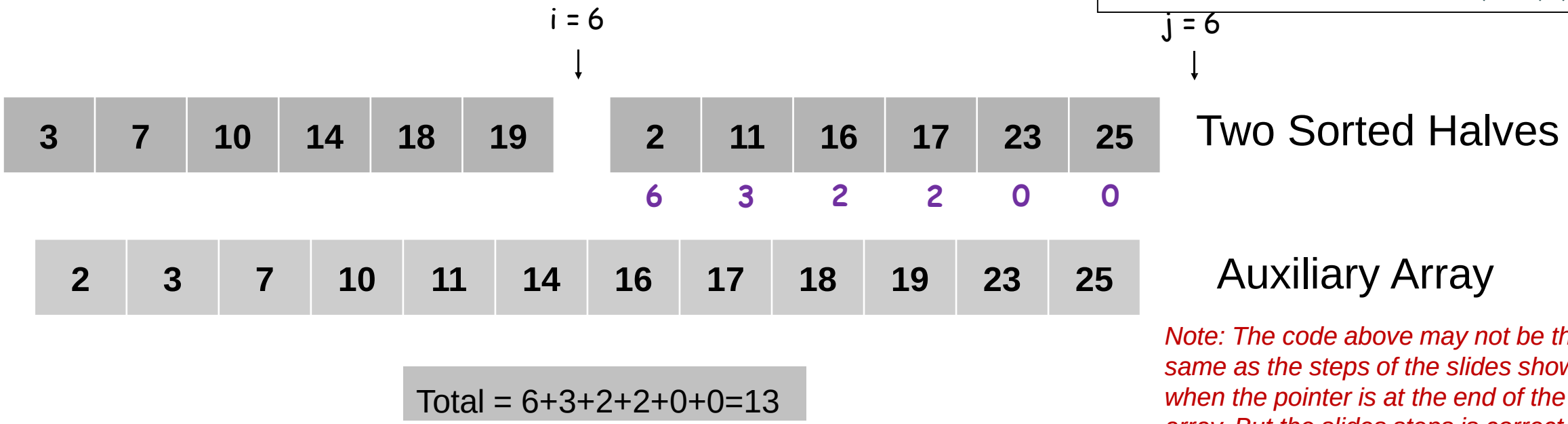
```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



Counting Inversions: Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves.
- Combine two sorted halves a and b into sorted whole one array c.
- Suppose in sorted halves, we have no inversions.
  - Otherwise, you should add them.

```
inversions = 0 + a_inver + b_inver
i=j=0
   while i < len(a) and j < len(b):
       if a[i] <= b[j]:
           c.append(a[i])
           i += 1
       else:
           c.append(b[j])
           j += 1
           inversions += (len(a)-i)
```



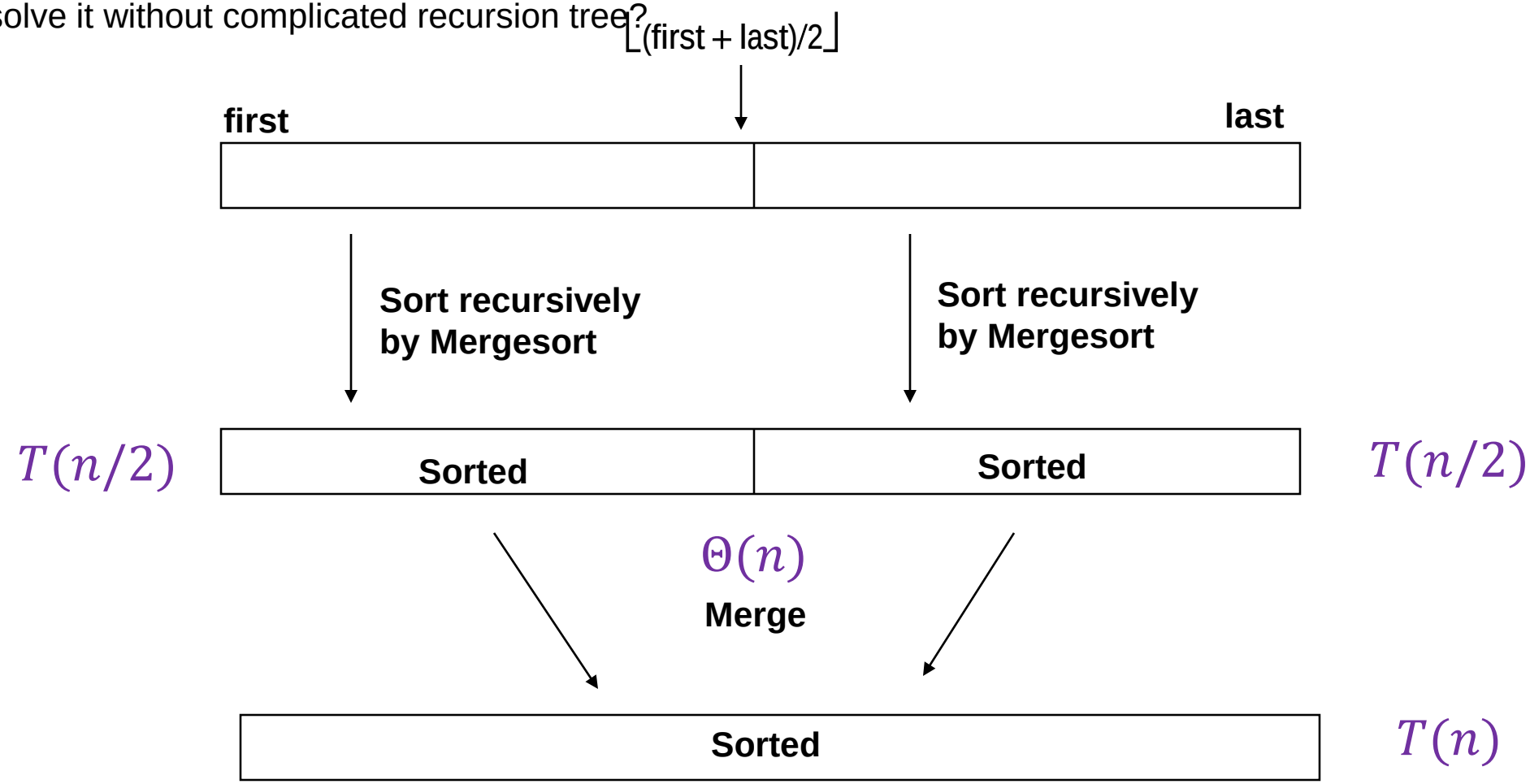
Note: The code above may not be the same as the steps of the slides shows when the pointer is at the end of the array. But the slides steps is correct.

Part 2-4: Merge Sort

Solve Recursive Function: Master Theorem

Merge Sort: Recursive Function

- Can we write a recursive function for merge sort?
- Recall the procedure.
- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- Can we solve it without complicated recursion tree?



Solve Recursive Function: Master Theorem

- "Plug and play" method for solving a common type of recurrence.
  - Based on comparing the nonrecursive complexity $f(n)$ with $n^{\log_b a}$.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Solve Recursive Function: Master Theorem

- Ex $T(n) = 9T\left(\frac{n}{3}\right) + n$
  - $a = 9, b = 3, f(n) = n, \log_b a = 2.$
  - Check $f(n) = O(n^{2-\epsilon})$, so use case 1 of Master theorem.
  - So $T(n) = \Theta(n^2)$.
- Ex $T(n) = T\left(\frac{2n}{3}\right) + 1.$
  - $a = 1, b = \frac{3}{2}, f(n) = 1, \log_b a = 0.$
  - $f(n) = \Theta(n^0)$, so use case 2 of theorem.
  - So $T(n) = n^0 \log n = \Theta(\log n)$.
- Ex $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$.
  - $a = 3, b = 4, f(n) = n \log n, \log_b a \approx 0.793.$
  - $f(n) = \Omega(n^{0.793+\epsilon})$, so use case 3 of theorem.
  - So $T(n) = \Theta(n \log n)$.

Solve Recursive Function: Master Theorem

- Note in cases 1 and 3, $f(n)$ needs to be smaller (resp. larger) than $n^{\log_b a}$ by a polynomial factor n^{ϵ}.
  - If this doesn't hold, we can't use the theorem.
- Ex $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$.
  - $a = 2, b = 2, f(n) = n\log n\,, \log_b a = 1$.
  - However, case 2 of the Master theorem doesn't apply, since $f(n) \neq \Theta(n)$.
  - Case 3 also doesn't apply, since $n\log n \neq \Theta(n^{1+\epsilon})$ for any $\epsilon > 0$.
  - So we can't use the Master theorem to solve this recurrence.
- For a proof of the Master theorem, see Section 4.5 in Cormen et al.
  - Proof basically formalizes the recursion tree method.

Extend Reading: Akra-Bazzi (Optional)

- Ex $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$
  - We can not solve it by Master Theorm.
  - The Akra–Bazzi method applies to recurrence formulas of the form

$$T(x) = g(x) + \sum_{i=1}^{k} a_i T(b_i x + h_i(x)) \qquad \text{for } x \geq x_0$$

  - We first find the unique real number p such that $\sum_{i=1}^{k} a_i b_i^p = 1$.
  - Then the solution is:

$$T(n) = \Theta\left(x^p \left(1 + \int_1^x \frac{f(u)}{u^{p+1}}\, du\right)\right)$$

  - More about the theorm: https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method