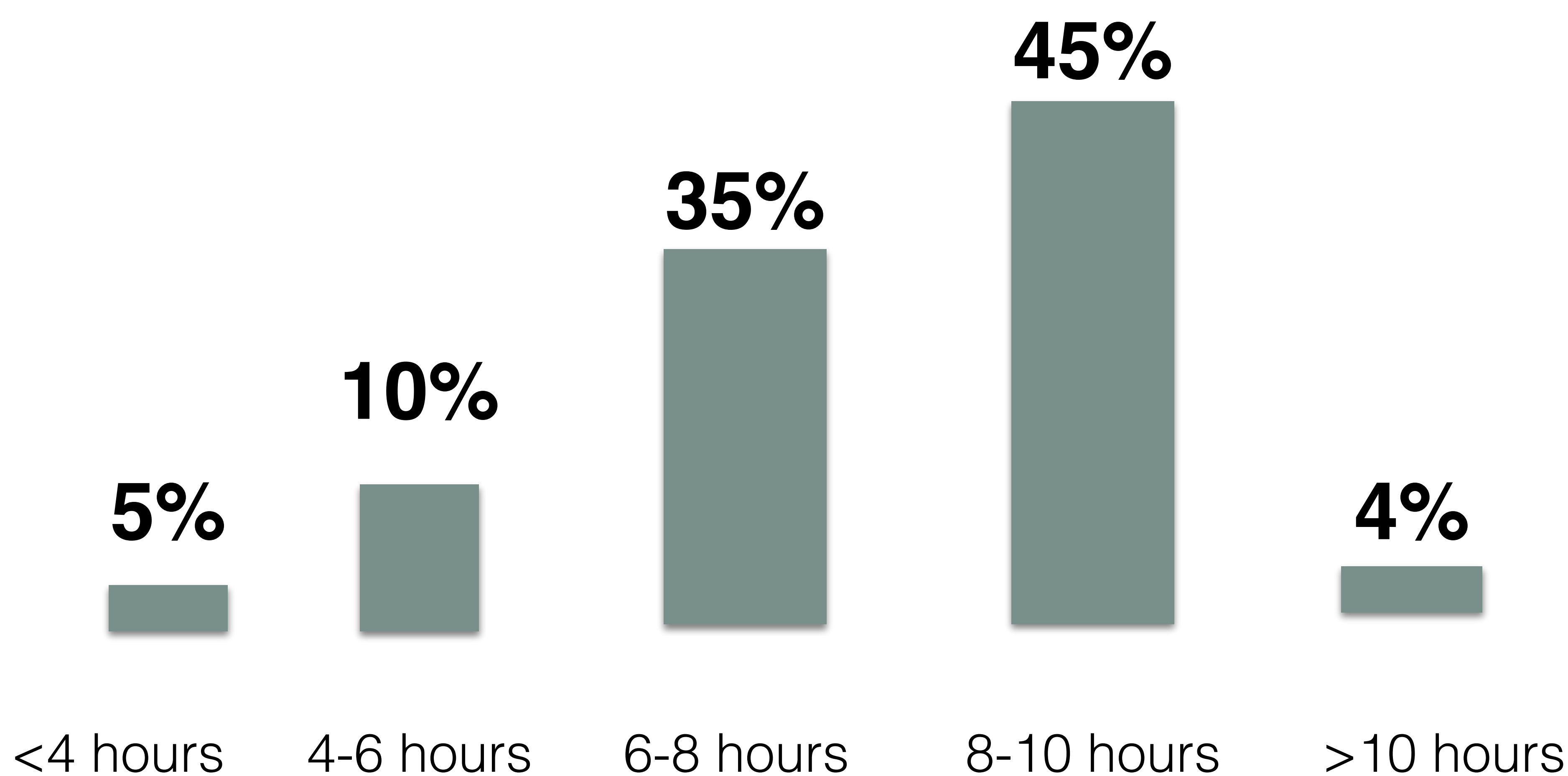
Divide and Conquer: Sorting and Recurrences

Check-in and Announcements

- Tomorrow’s office hours shifted to 10 am-noon
- **Midterm on April 2nd** (no class), 24-hours take home
- Problem set workload feedback (distribution)



Divide & Conquer: The Pattern

- **Divide** the problem into several independent smaller instances of exactly the same problem
- **Delegate** each smaller instance to the **Recursion Fairy** (technically known as induction hypothesis)
- **Combine** the solutions for the smaller instances
    - Assume the recursion fairy correctly solves the smaller instances, how can you combine them?

Review: Merge Sort

MERGE-SORT(L)

IF (list L has one element)

  RETURN L.

Divide the list into two halves A and B.

$A \leftarrow$ MERGE-SORT(A). $\longleftarrow T(n / 2)$

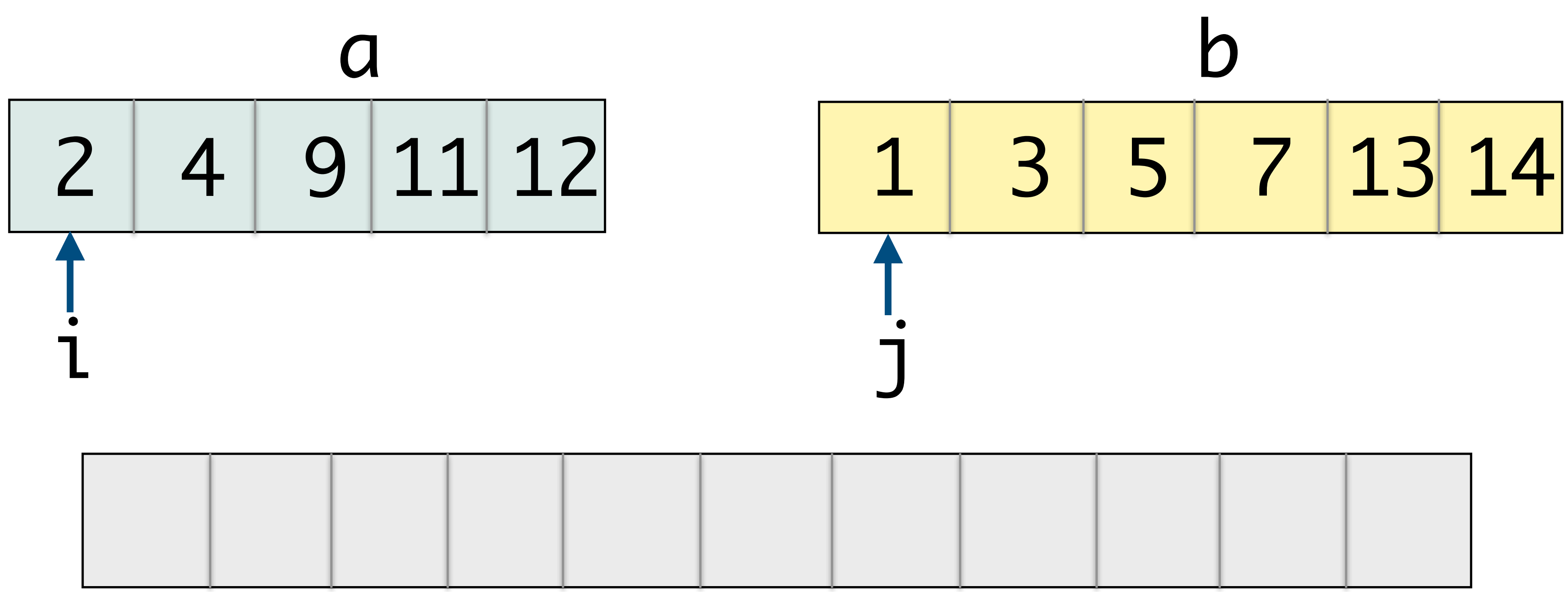
$B \leftarrow$ MERGE-SORT(B). $\longleftarrow T(n / 2)$

$L \leftarrow$ MERGE(A, B). $\longleftarrow \Theta(n)$

RETURN L.

Merge Step: $\Theta(n)$

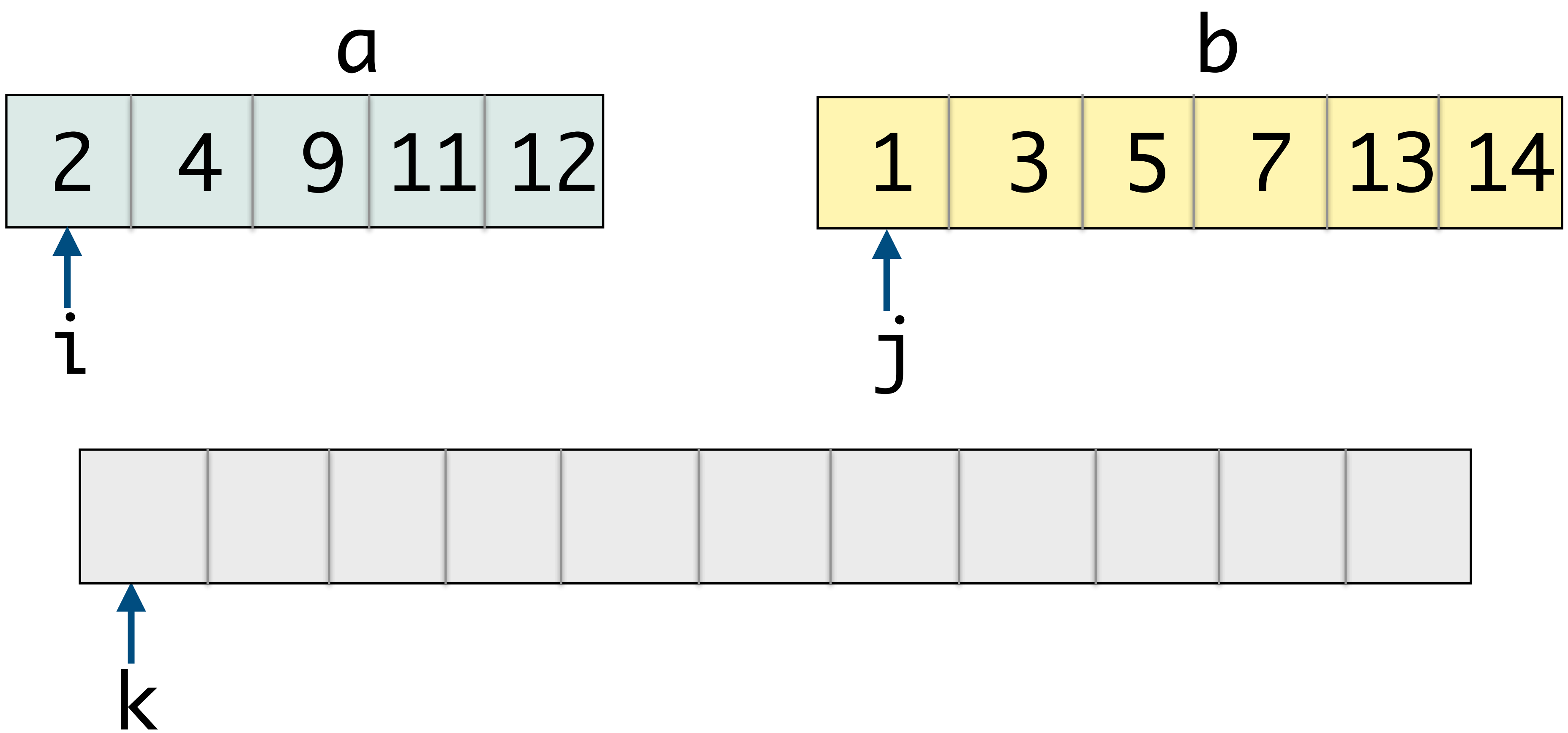
- Scan subarrays from left to right
- Compare element by element; create new merged array



Merge Step: $\Theta(n)$

Is a[i] <= b[j] ?

- Yes, a[i] appended to c
- No, b[j] appended to c

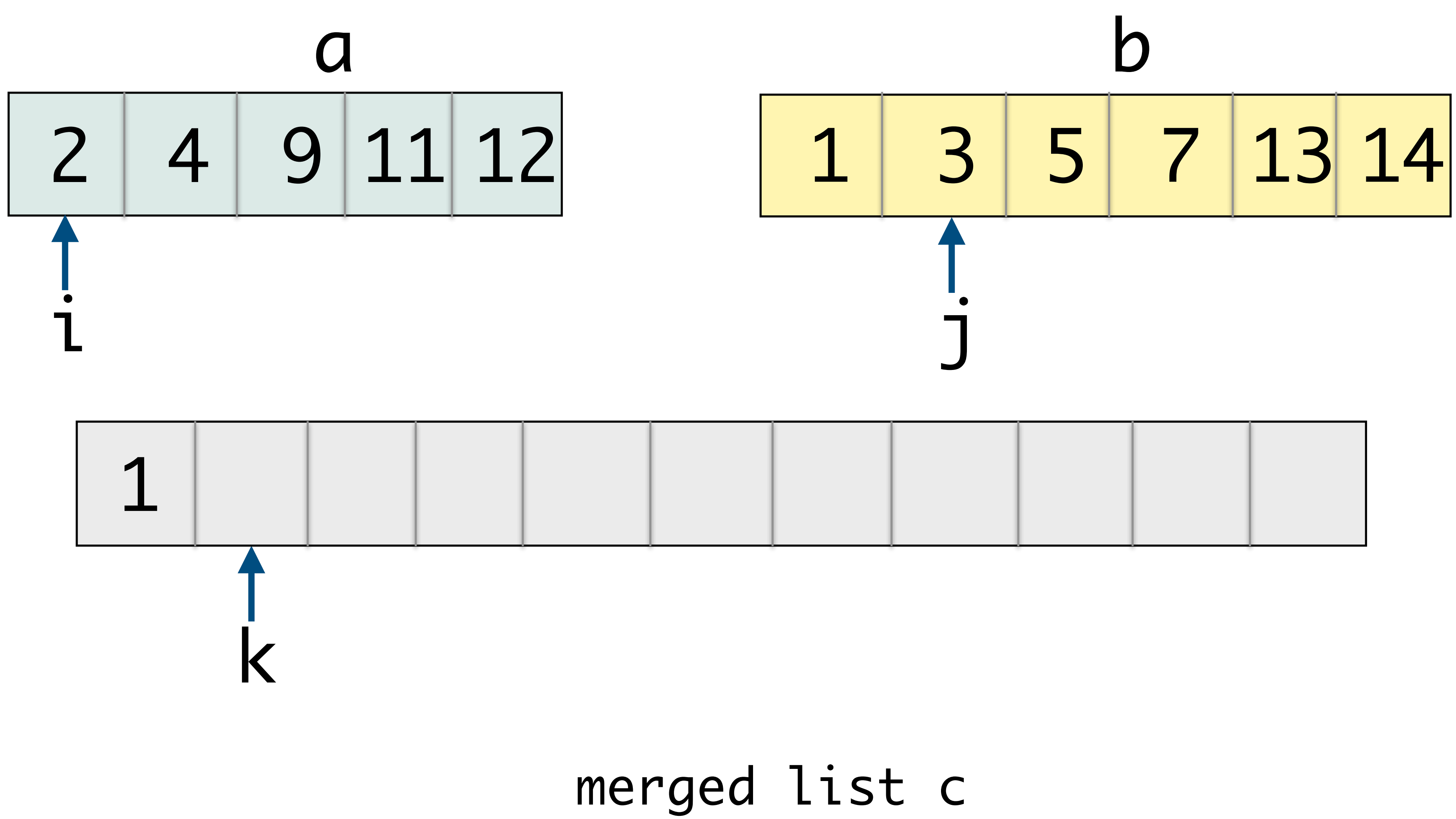


merged list c

Merge Step: $\Theta(n)$

`Is a[i] <= b[j] ?`

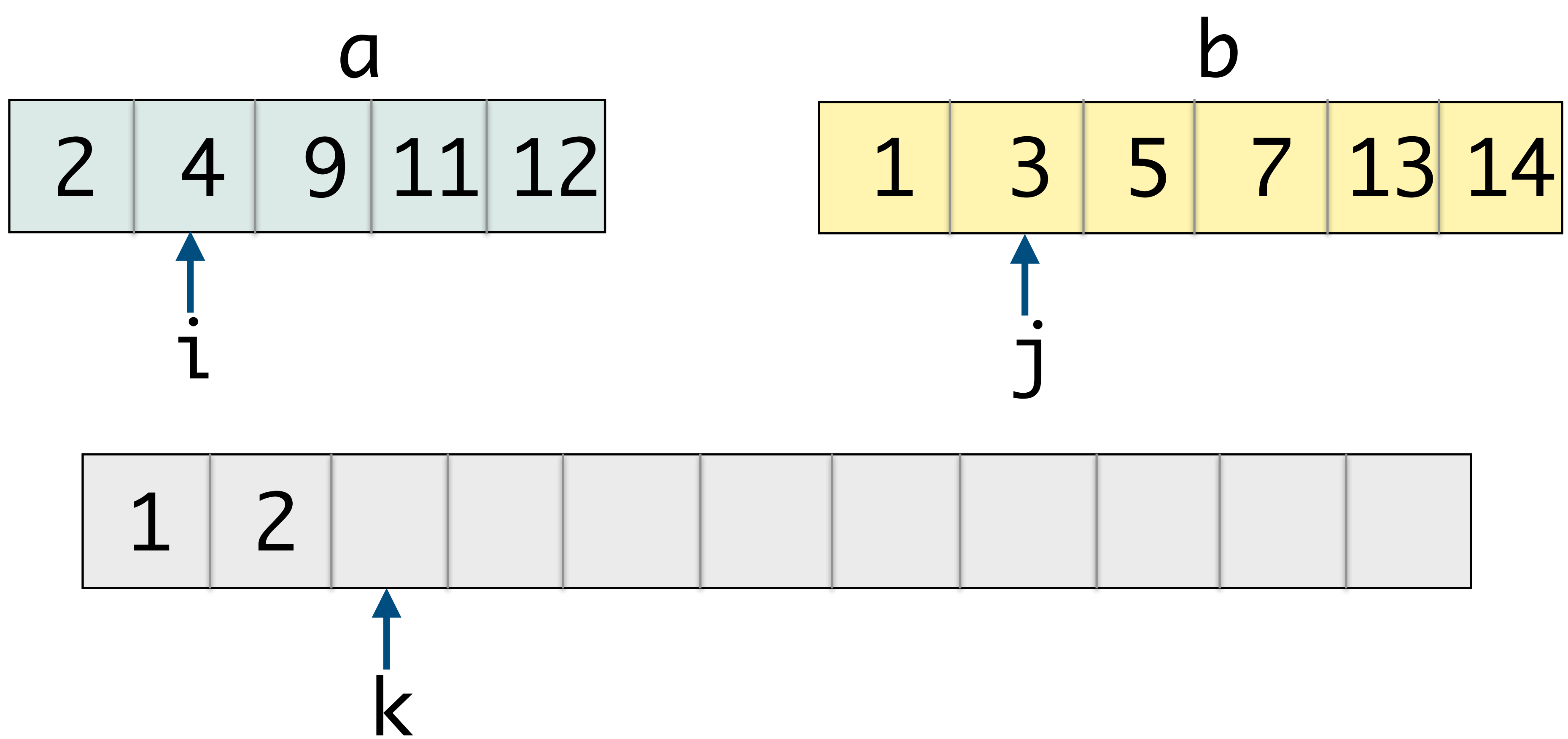
- `Yes, a[i] appended to c`
- `No, b[j] appended to c`



Merge Step: $\Theta(n)$

Is a[i] <= b[j] ?

- Yes, a[i] appended to c
- No, b[j] appended to c

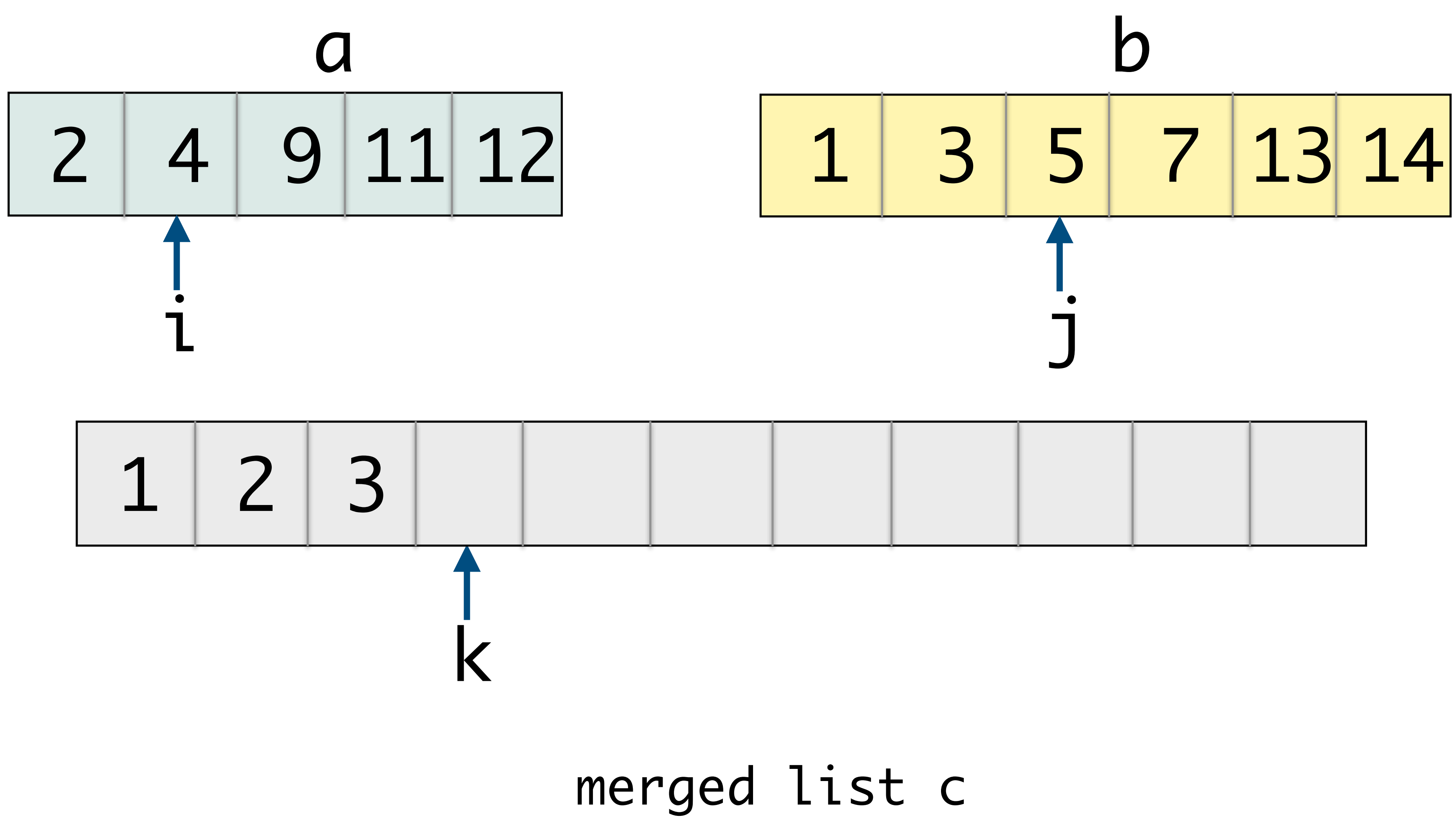


merged list c

Merge Step: $\Theta(n)$

Is a[i] <= b[j] ?

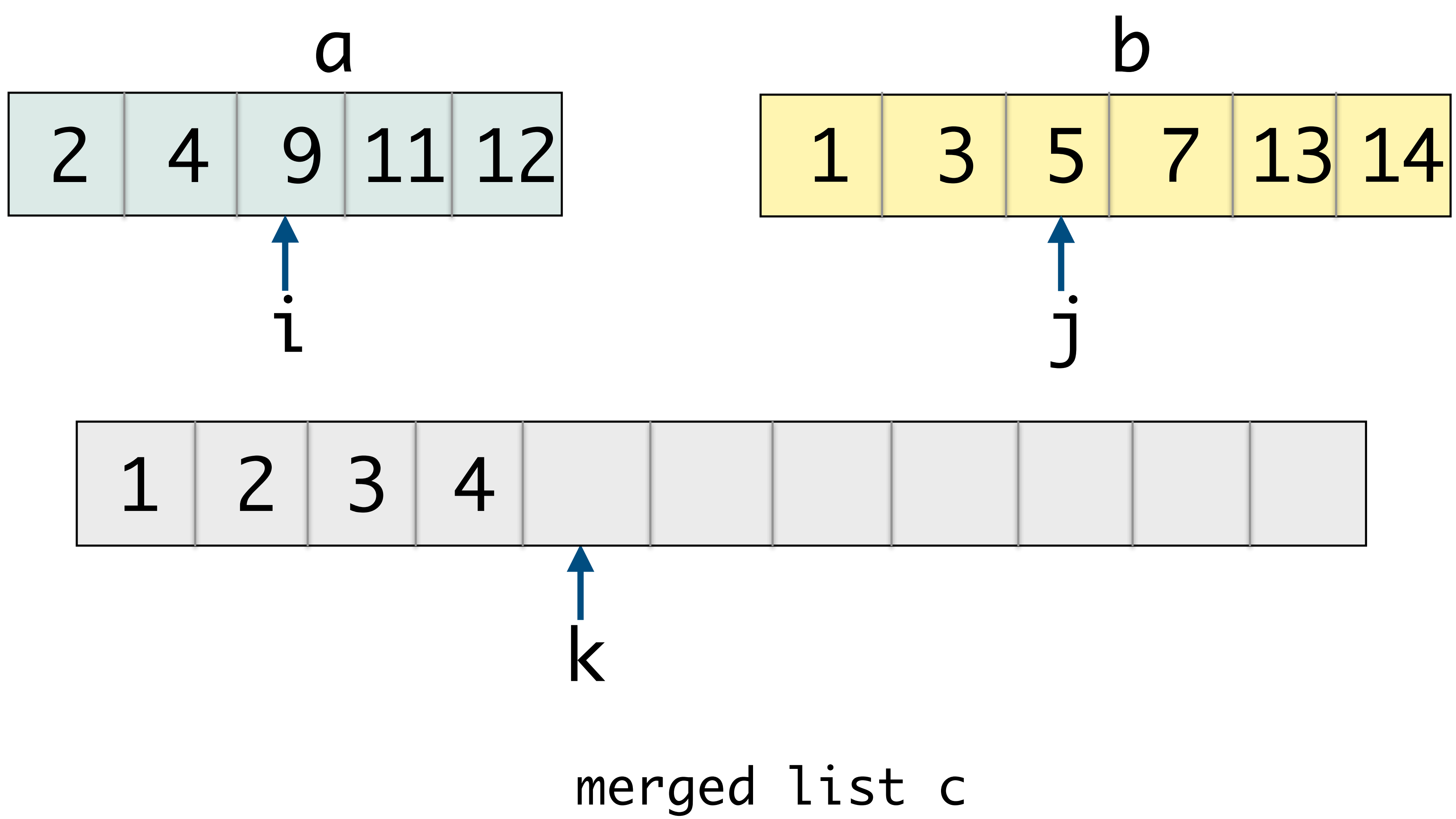
- Yes, a[i] appended to c
- No, b[j] appended to c



Merge Step: $\Theta(n)$

Is a[i] <= b[j] ?

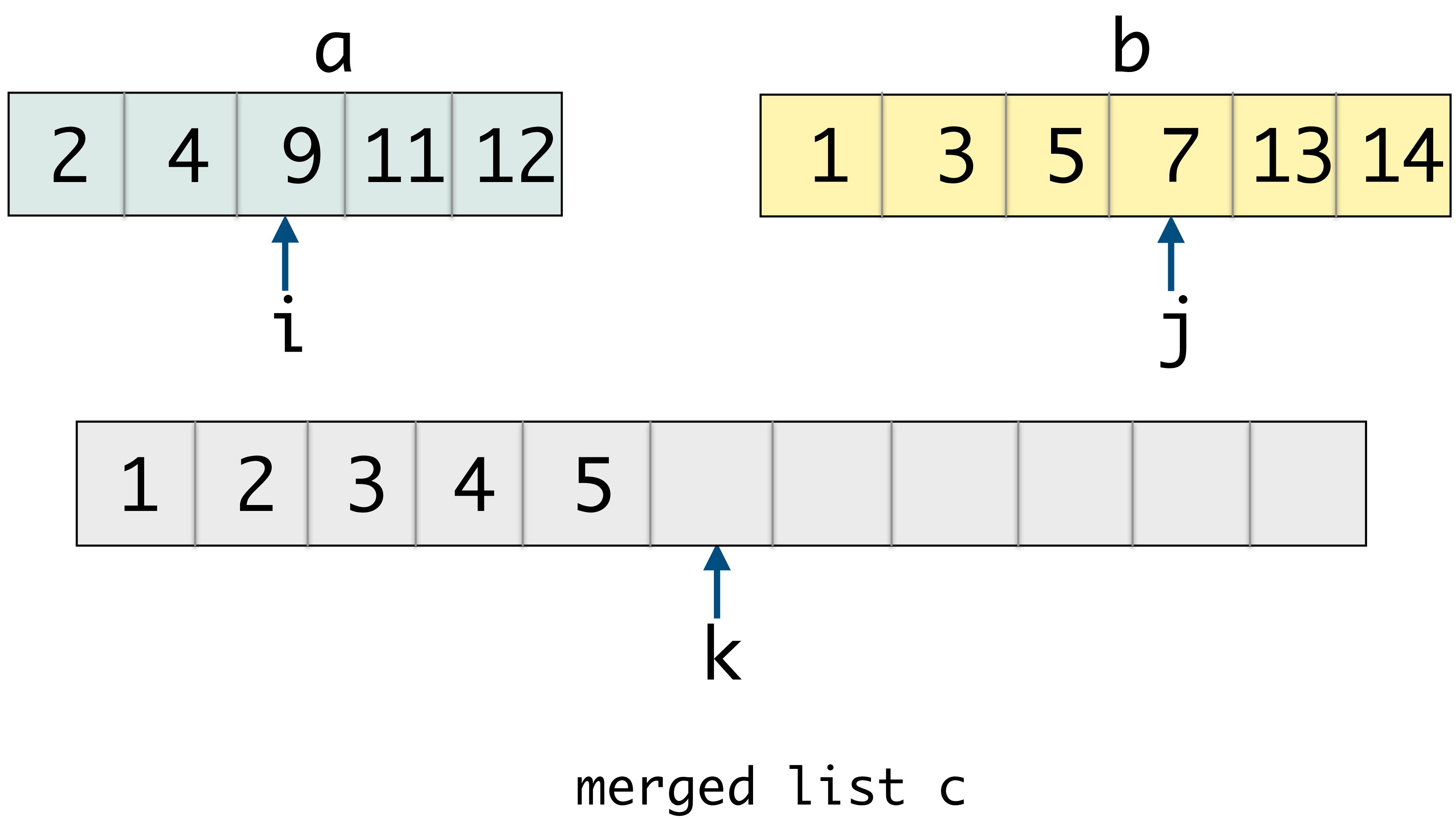
- Yes, a[i] appended to c
- No, b[j] appended to c



Merge Step: $\Theta(n)$

`Is a[i] <= b[j] ?`

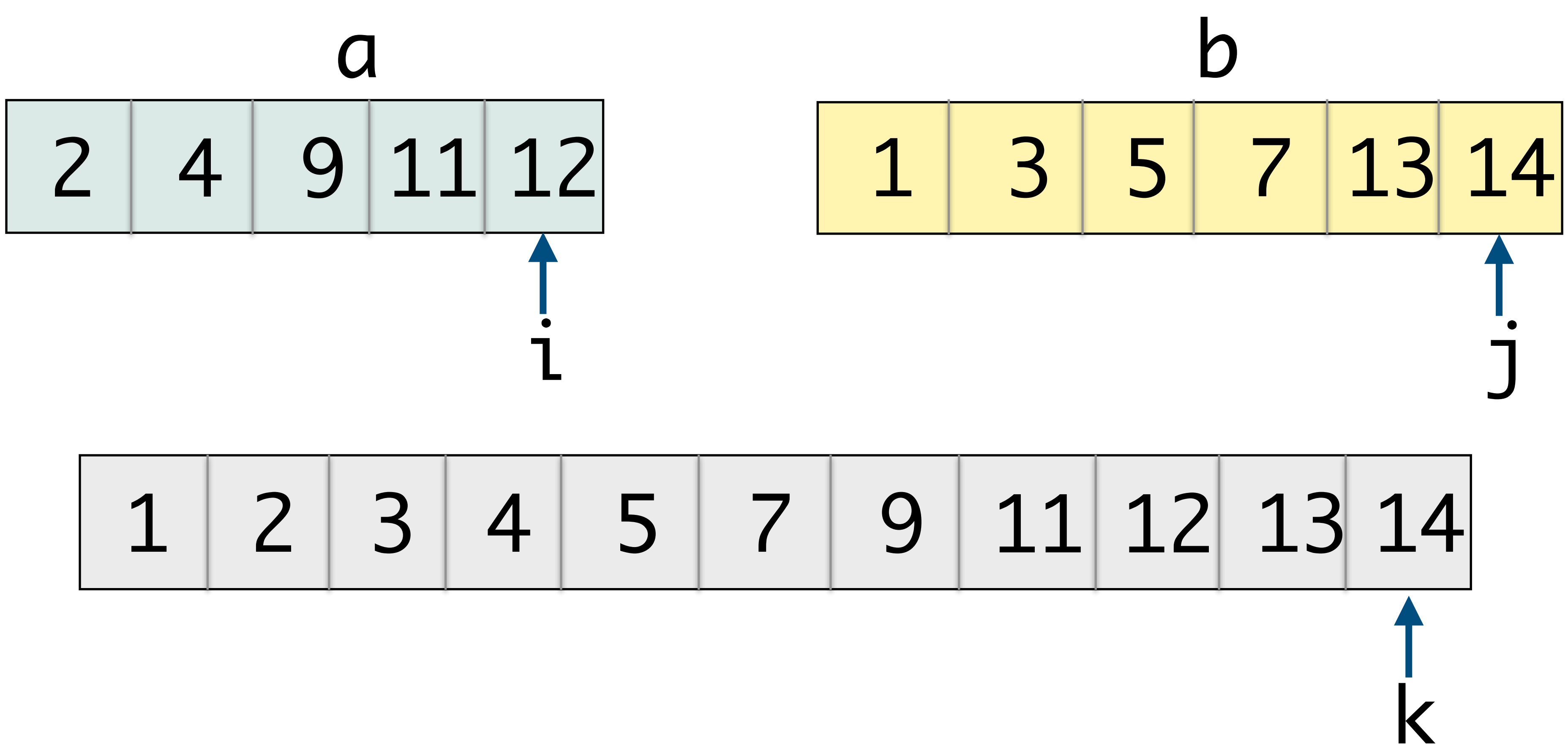
- `Yes, a[i] appended to c`
- `No, b[j] appended to c`



Merge Step: $\Theta(n)$

`Is a[i] <= b[j] ?`

- `Yes, a[i] appended to c`
- `No, b[j] appended to c`



`merged list c`

Correctness: D&C Algorithms

- **Correctness proof pattern:**
  - Natural proof by induction pattern
  - Show base case holds
  - Assume your recursive calls return the correct solution (induction hypothesis)
  - Inductive step: crux on the proof; show that the solutions returned by the recursive calls are **“combined”** correctly

Correctness: Merge Sort

- **Claim. (Combine step.)** Merge subroutine correctly merges two sorted subarrays $A[1, \ldots i]$ and $B[1, \ldots, j]$ where $i + j = n$.
- Prove that for the first k iterations of the loop correctly merge A and B for $k = 0$ to n.
- **Invariant.** Merged array is sorted after every iteration.
- Base case: $k = 0$
    - Algorithm correctly merges two empty subarrays
- For inductive step, there are several cases based on whether $a_i \leq b_j$ or $a_i > b_j$, show that newly added element maintains sorted-ness

Analyzing Running Time

- For this topic, our main focus will be on analysis of running time
- We analyze the running time of recursive functions by:
  - **Making recursive calls**: consider the number of calls made and the size of input to each call
    - e.g., merge sort on an input of size n makes two recursive calls each on an input of size $n/2$
  - **Time spent "combining"** solutions ("non-recursive work") returned by recursive calls
    - e.g. merge step combines the sorted arrays in $\Theta(n)$ time
- Using the two, we typically write a **running time recurrence**

Running Time Recurrence

- Let $T(n)$ represent the worst-case running time of merge sort on an input of size n
- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$
- **Base case:** $T(1) = 1$; often ignored
- We will ignore the floors and ceilings (turns out it doesn't matter for asymptotic bounds; we'll show this later)
- So the recurrence simplifies to:
  - $T(n) = 2T(n/2) + O(n)$
  - The answer to this ends up being $T(n) = O(n \log n)$
  - Today we will learn different ways to derive it

Recurrences: Unfolding

Method 1. Unfolding the recurrence

- Assume $n = 2^\ell$ (that is, $\ell = \log n$)
- Because we don't care about constant factors and are only upper-bounding, we can always choose smallest power of 2 greater than that is, $n < n' = 2^\ell < 2n$

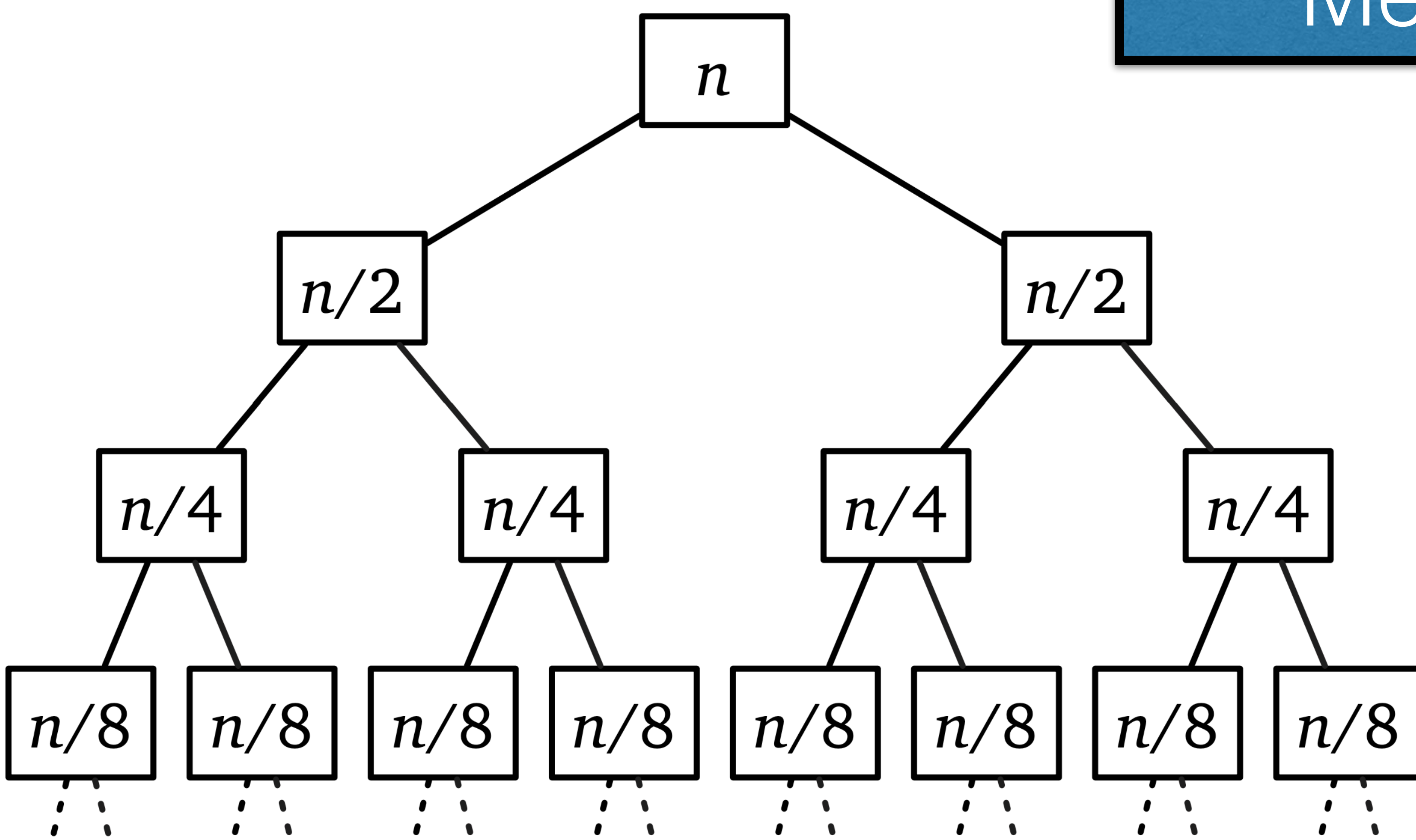
$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2T(2^{\ell-1}) + c2^\ell \\
&= 2(2T(2^{\ell-2}) + c2^{\ell-1}) + c2^\ell = 2^2T(2^{\ell-2}) + 2 \cdot c2^\ell \\
&= 2^3T(2^{\ell-3}) + 3 \cdot 2^\ell \\
&= \ldots = 2^\ell T(2^0) + c\ell 2^\ell = O(n \log n)
\end{aligned}
$$

Recurrences: Recursion Tree

Method 2. Recursion Trees

- Work done at each level $2^i \cdot (n/2^i) = n$
- Total $\log_2 n$ levels

Recommended Method!



Recurrences: Recursion Tree

- This is really a method of visualization
- Very similar to unrolling, but much easier to keep track of what’s going on
- It’s not (quite) a proof, but generally it is sufficient for running times in this class
  - “Solve the recurrence” can be done by drawing the recursion tree and explaining the solution

Recurrences: Guess & Verify

Method 3. Guess and Verify

- Eyeball recurrence and make a guess
- Verify guess using induction

- More on this later…

Counting Inversions

- Way to compare two different rankings
- Or a way to measure how far an array is from sorted
- Let $a_1, a_2, \ldots, a_n$ be an ordering of n numbers
- We say two indices $i < j$ form an **inversion** if $a_i > a_j$

- Example: How many inversions in $2,4,1,3,5$?
  - $2,1$ is an inversion
  - $4,1$ and $4,3$ is an inversion
  - 3 inversions total

Counting Inversions

- Way to compare two different rankings
- Or a way to measure how far an array is from sorted
- Let $a_1, a_2, \ldots, a_n$ be an ordering of n numbers
- We say two indices $i < j$ form an **inversion** if $a_i > a_j$
- Counting all inversions in a naive way:
    - Comparing every pair is $\Theta(n^2)$
- **Can we do better** by using a divide and conquer approach?

Count Inversions Recursively

- **Divide:** break array into two halves A and B
- **Conquer:** recursively count number of inversions in both
- **Combine:** count number of inversions of the type (a, b) where $a \in A, b \in B$ and return total

- How do combine in $O(n)$ time?
- **Idea:** easy if A and B are sorted!
  - Sorting is the process of "fixing inversions", so why not count them while sorting?

Counting Inversions

- Counting inversions: (a, b) where $a \in A, b \in B$ when A, B are sorted
- Scan both from left to right
- Compare a_i and b_j

count inversions (a, b) with a ∈ A and b ∈ B

3	7	10	a_i	18
			↑	

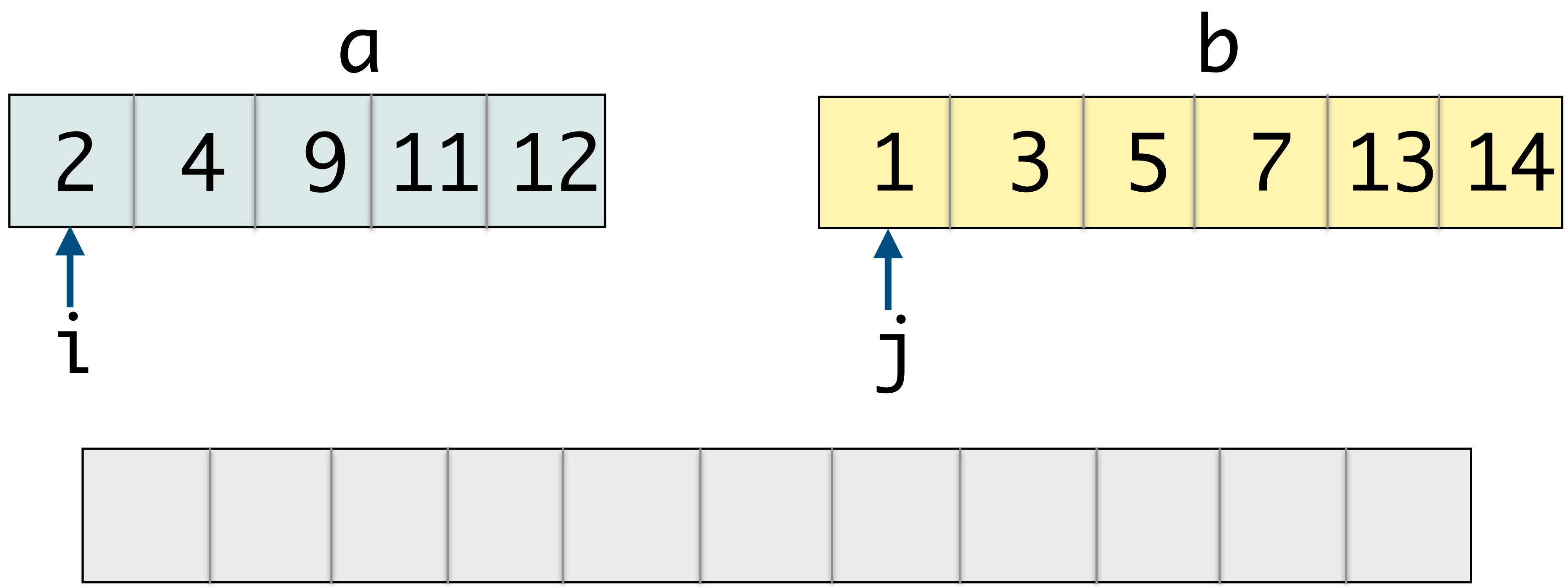
2	11	b_j	20	23
5	2	↑		

merge to form sorted list C

2	3	7	10	11					
					↑				

Merge Step: Count Inversions

- Scan both arrays from left to right
- Compare a_i and b_j, when is there an inversion?

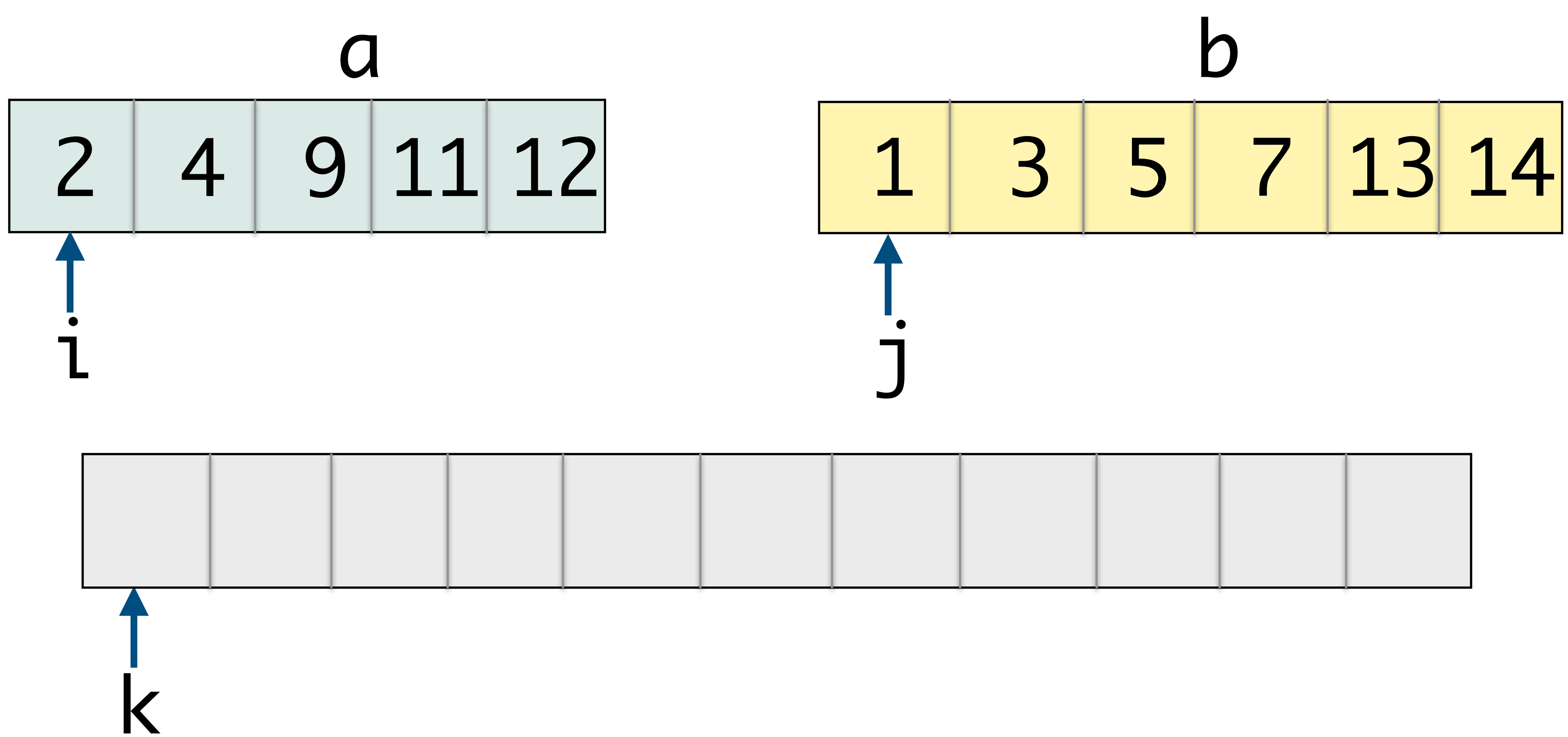


Merge Step: Count Inversions

Is a[i] <= b[j] ?

- Yes, a[i] appended to c
- No, b[j] appended to c

Inversion! How many?



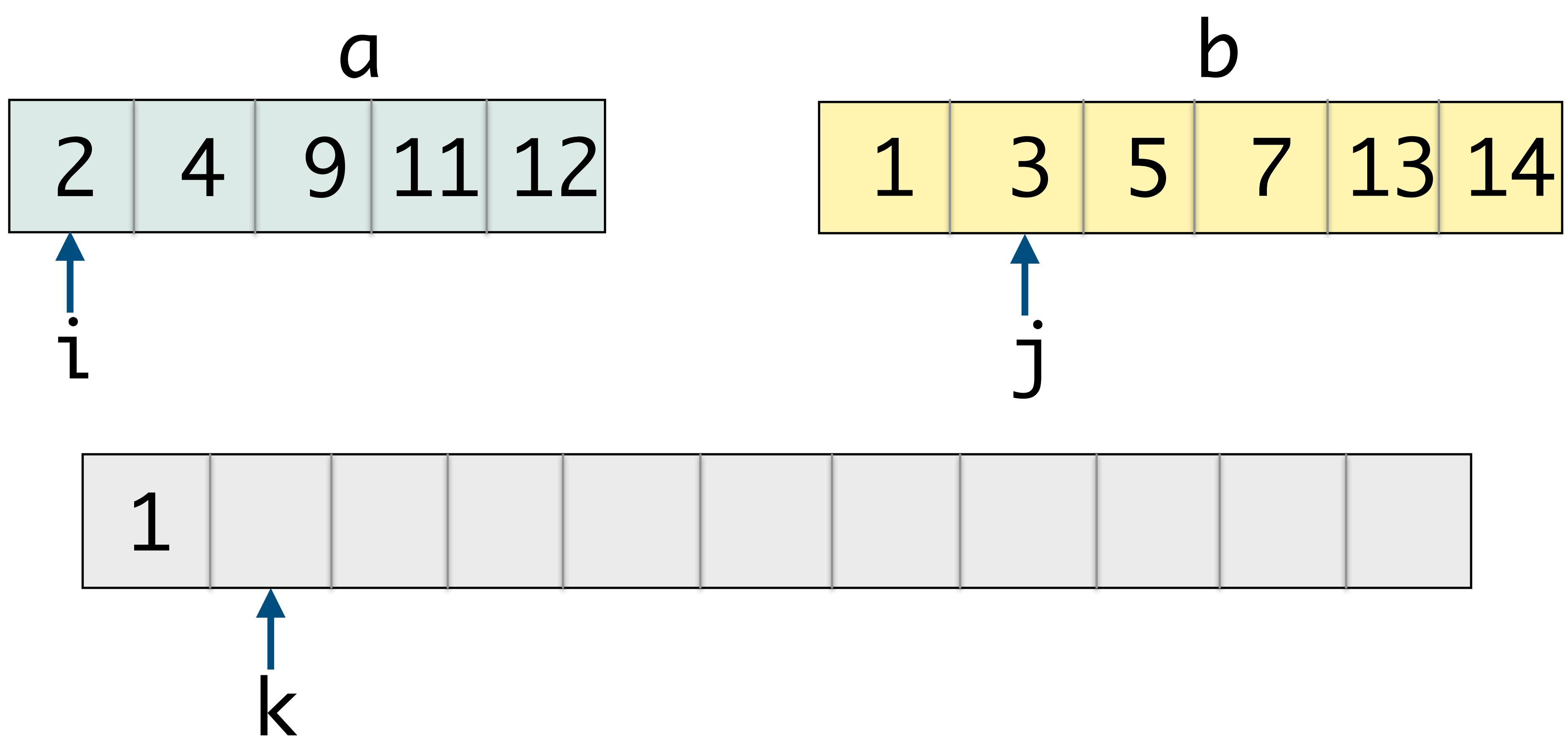
merged list c

Merge Step: Count Inversions

Is a[i] <= b[j] ?

- Yes, a[i] appended to c
- No, b[j] appended to c

Inversion! How many?

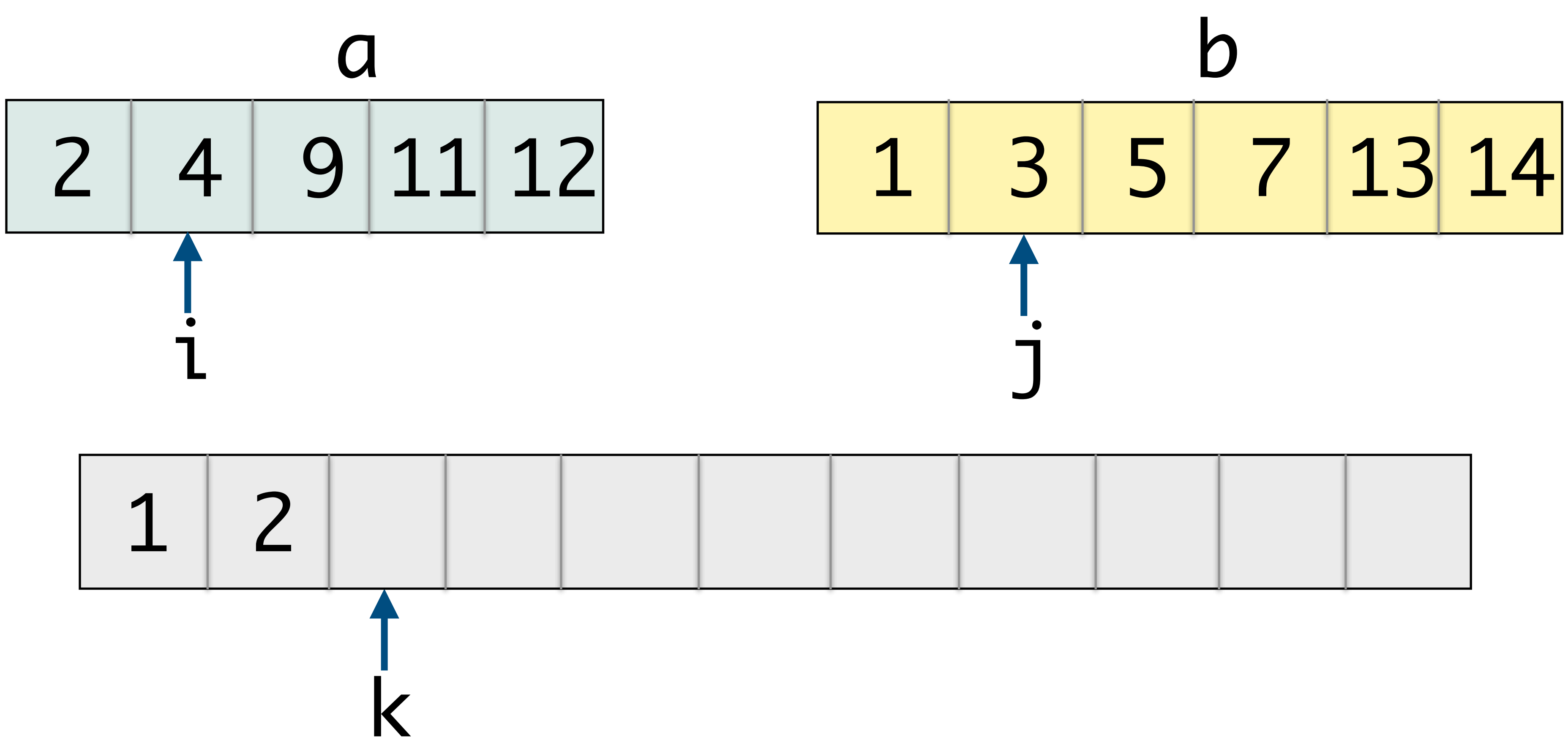


merged list c

Merge Step: Count Inversions

`Is a[i] <= b[j] ?`

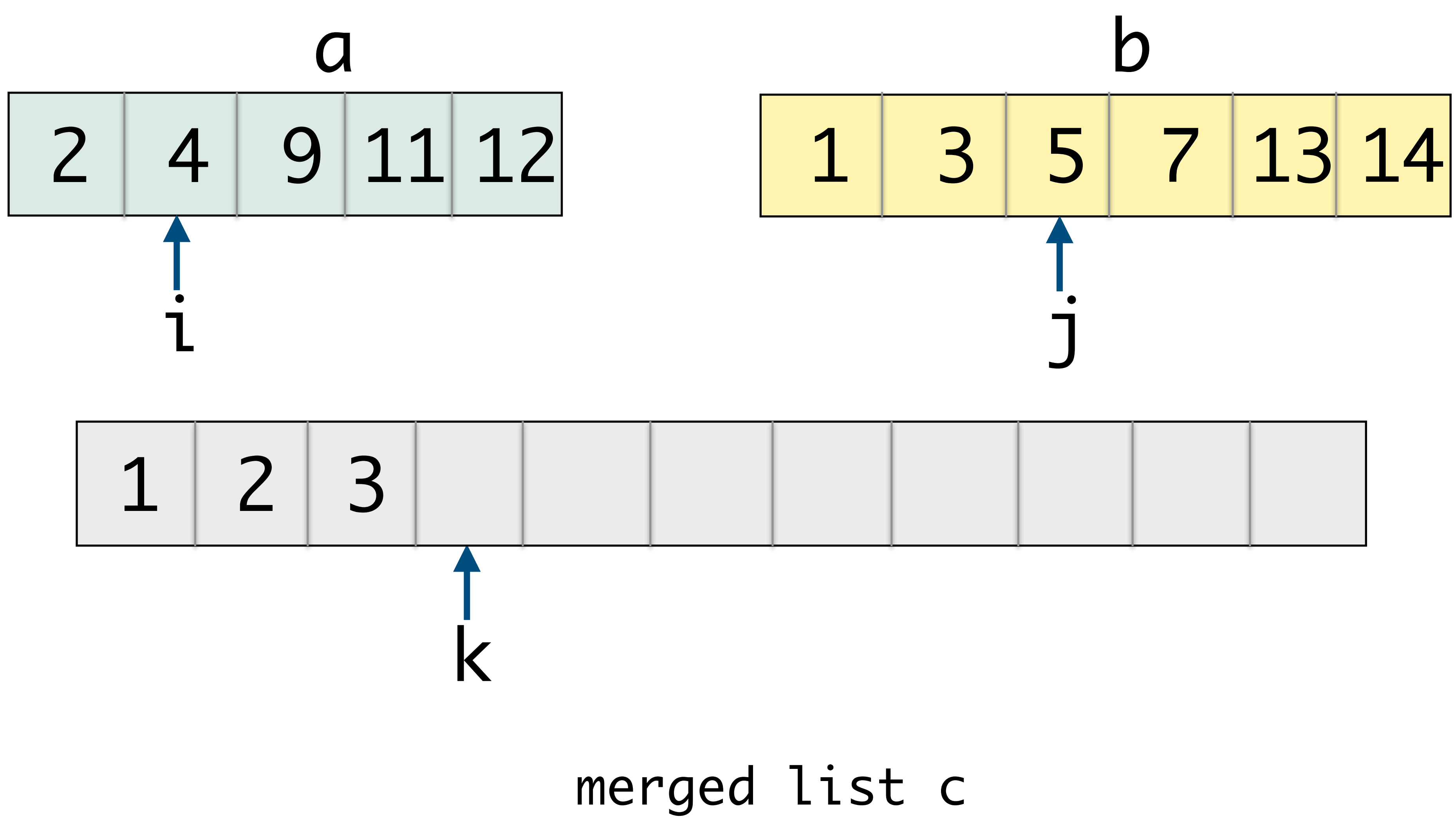
- `Yes, a[i] appended to c`
- `No, b[j] appended to c`



`merged list c`

Merge Step: Count Inversions

```
Is a[i] <= b[j] ?
• Yes, a[i] appended to c
• No, b[j] appended to c
```

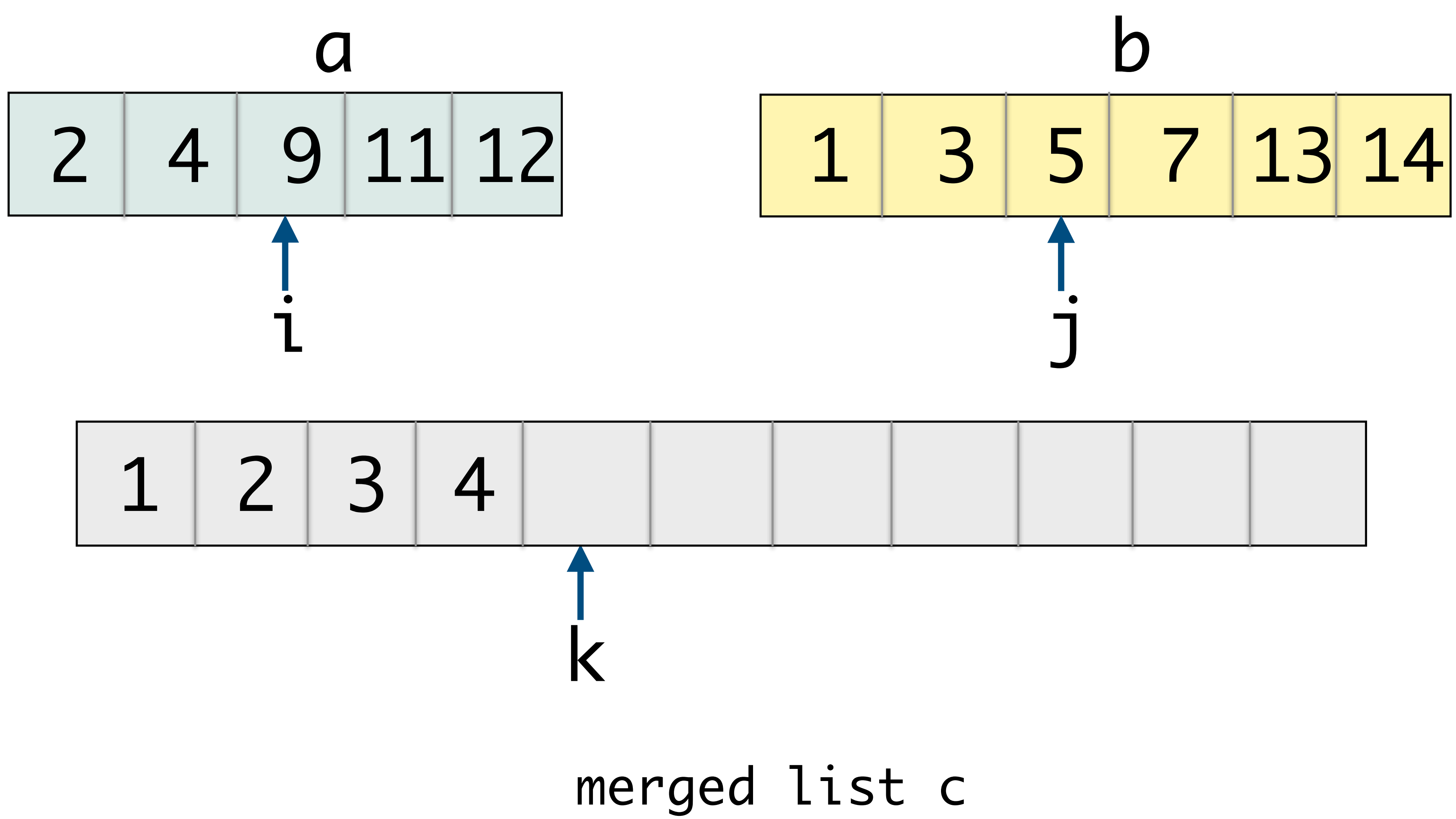


Merge Step: Count Inversions

`Is a[i] <= b[j] ?`

- `Yes, a[i] appended to c`
- `No, b[j] appended to c`

Inversion! How many?

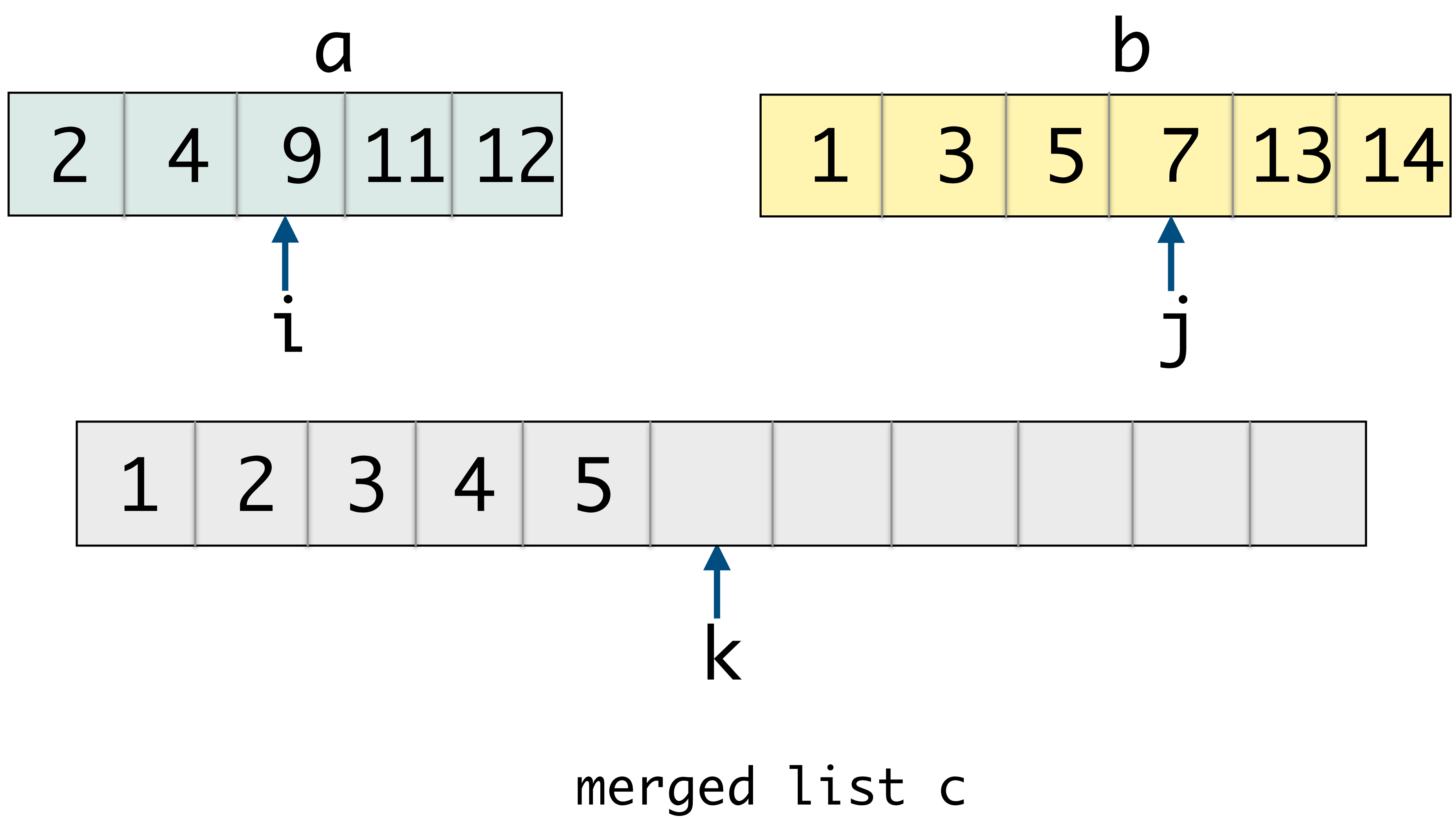


Merge Step: Count Inversions

Is a[i] <= b[j] ?

- Yes, a[i] appended to c
- No, b[j] appended to c

Inversion! How many?



Counting Inversion

- Count inversions of the type (a, b) where $a \in A$ and $b \in B$ and both A, B are sorted
- Scan both arrays from left to right
- Compare a_i and b_j
- If $a_i < b_j$,
    - a_i is not inverted wrt all remaining elements in B
- If $a_i > b_j$
    - b_j is **inverted with respect to every element left in** A
- Append smaller element to sorted list C

Counting Inversions: Divide & Conquer

SORT-AND-COUNT(L)

IF (list L has one element)

  RETURN $(0, L)$.

Divide the list into two halves A and B.

$(r_A,\ A) \leftarrow$ SORT-AND-COUNT(A). $\longleftarrow T(n/2)$

$(r_B,\ B) \leftarrow$ SORT-AND-COUNT(B). $\longleftarrow T(n/2)$

$(r_{AB},\ L) \leftarrow$ MERGE-AND-COUNT(A, B). $\longleftarrow \Theta(n)$

RETURN $(r_A + r_B + r_{AB},\ L)$.

Combine Step

Running Time

- Same as merge sort
- $O(n)$ time to merge and count (non-recursive)
- Two subproblems of half the size

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

Correctness

$\text{SORT-AND-COUNT}(L)$

IF (list L has one element)

  RETURN $(0, L)$.

Divide the list into two halves A and B.

$(r_A,\ A) \leftarrow \text{SORT-AND-COUNT}(A)$. $\longleftarrow T(n/2)$

$(r_B,\ B) \leftarrow \text{SORT-AND-COUNT}(B)$. $\longleftarrow T(n/2)$

$(r_{AB},\ L) \leftarrow \text{MERGE-AND-COUNT}(A, B)$. $\longleftarrow \Theta(n)$

RETURN $(r_A + r_B + r_{AB},\ L)$.

Assume is correct by strong induction

Show is correct

Correctness

- Induction on the size of the array n
- Base case: $n = 1$, no inversions
- Assume that your algorithm is correct on all subproblems of size $< n$ (thus your recursive call return the correct solution) and show that the combine step is correct
- Let A, B the the subarrays returned by the recursive call
- By inductive hypothesis all inversions of the type (i, j) where $i, j \in A$ or $i, j \in B$ have been counted correctly
- Thus, need to argue that combine step correctly counts all inversions of the type (i, j) where $i \in A$ and $j \in B$
    - Another induction similar to merge step of merge sort

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last subarrays

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	$\boxed{\text{P}}$	L
Partition:	A	G	O	E	I	N	L	M	$\boxed{\text{P}}$	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	$\boxed{\text{P}}$	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	$\boxed{\text{P}}$	R	S	T	X

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last subarrays
- **Description.** (Divide and conquer): often the cleanest way to present is **short and clean pseudocode** with high level explanation
- **Correctness proof.** Induction and showing that partition step correctly partitions the array.

```
QuickSort(A[1 .. n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1 .. r − 1])    ⟨⟨Recurse!⟩⟩
      QuickSort(A[r + 1 .. n])    ⟨⟨Recurse!⟩⟩
```

Quick Sort Analysis

- Partition takes $O(n)$ time
- Size of the subproblems depends pivot; let r be the rank of the pivot, then:
- $T(n) = T(r-1) + T(n-r) + O(n), \;\; T(1) = 1$
- Let us analyze some cases for r
  - **Best case:** r is the median: $r = \lfloor n/2 \rfloor$ (we will learn how to compute the median in $O(n)$ time)
  - **Worst case:** $r = 1$ or $r = n$
  - **In between:** say $n/10 \leq r \leq 9n/10$
- Note in the worst-case analysis, we only consider the worst case for r. We are looking at the difference cases, just to get a sense for it.

Quick Sort: Cases

- Suppose $r = n/2$ (pivot is the median element), then
  - $T(n) = 2T(n/2) + O(n)$, $T(1) = 1$
  - We have already solved this recurrence
  - $T(n) = O(n \log n)$

- Suppose $r = 1$ or $r = n - 1$, then
  - $T(n) = T(n-1) + T(1) + 1$
  - What running time would this recurrence lead to?
  - $T(n) = \Theta(n^2)$ (notice: this is tight!)

Quick Sort: Cases

- Suppose $r = n/10$ (that is, you get a one-tenth, nine-tenths split
- $T(n) = T(n/10) + T(9n/10) + O(n)$
- Let's look at the recursion tree for this recurrence
- We get $T(n) = O(n \log n)$, in fact, we get $\Theta(n \log n)$
- In general, the following holds (we'll show it later):
- $T(n) = T(\alpha n) + T(\beta n) + O(n)$
    - If $\alpha + \beta < 1 : T(n) = O(n)$
    - If $\alpha + \beta = 1, T(n) = O(n \log n)$

Quick Sort: Theory and Practice

- We can find the **median element in $\Theta(n)$** time
    - Using divide and conquer! we'll learn how in next lecture
- In practice, the constants hidden in the Oh notation for median finding are too large to use for sorting
- Common heuristic
    - Median of three (pick elements from the start, middle and end and take their median)
- If the pivot is chosen **uniformly at random**
    - quick sort runs in time $O(n \log n)$ in expectation and *with high probability*
    - We will prove this in the second half of the class

Challenge Recurrence

- Solve the following recurrence:

$$T(n) = \sqrt{n}T(\sqrt{n}) + n$$

- **Hint.** Try some change of variables

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)