Sorting: Merge Sort

Sorting Algorithms

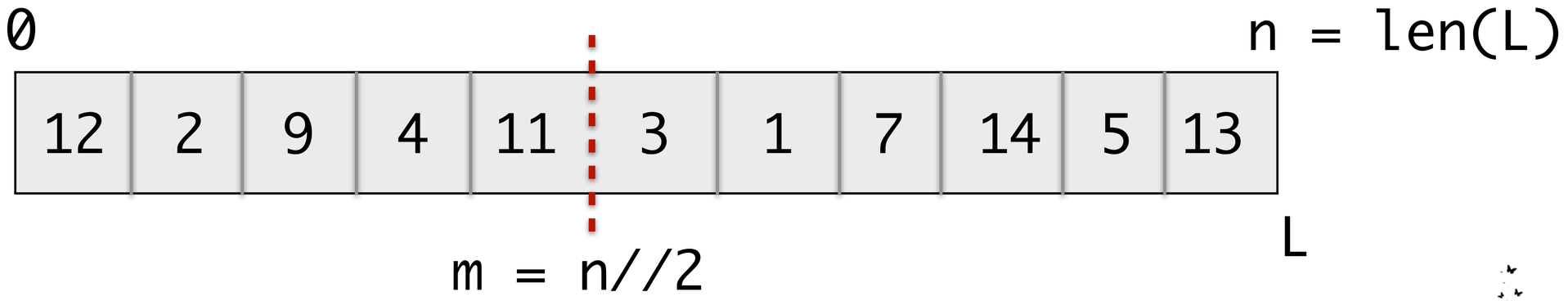
- Last lecture, we implemented and analyzed **Selection Sort**
- Selection sort is natural and intuitive:
  - Finds the min element, puts it in the first spot
  - Finds the second-min element, puts in 2nd spot, etc.
- Selection sort takes $O(n^2)$ steps to sort a list of size n
- Can come up with many other natural sorting algorithms that compare and rearrange a slightly different way
- A lot of these algorithms are $O(n^2)$: intuitively any algorithm that takes k steps to move each item k positions to its sorted location will take at least $O(n^2)$ steps as every element can be $O(n)$ away from its sorted position in the worst case.
- **Question.** Can we beat this? And if so, how?

Towards $O(n \log n)$ Sorting

- Let $d(e)$ denote the distance of e in the list from its location in the list in sorted order
- To do better than much better than n^2, we need to be able to move an item e to its final position in significantly less steps then $d(e)$
    - Essentially, saying that this item is not in the left half, but rather right half efficiently
- We will achieve this by a divide-conquer (recursive) sorting algorithm called **Merge Sort**
- Invented by **John von Neumann** in 1945
- Every elegant idea, that ends up being optimal!

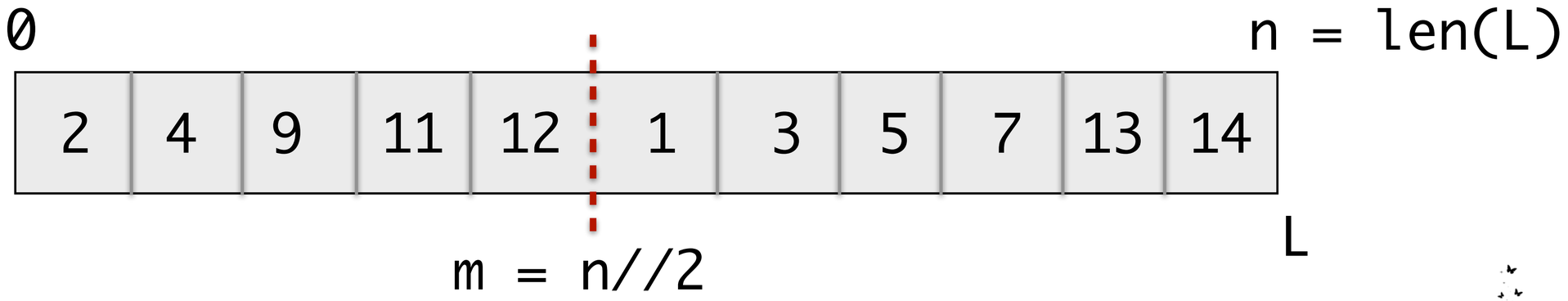
Merge Sort: Idea

- If we split the list in half, sorting the left and right half are smaller versions of the same problem
- **Recursive thinking:** if the recursion fairy sorted the left and right halves of the list, how long does it take to combine them into a single sorted list?



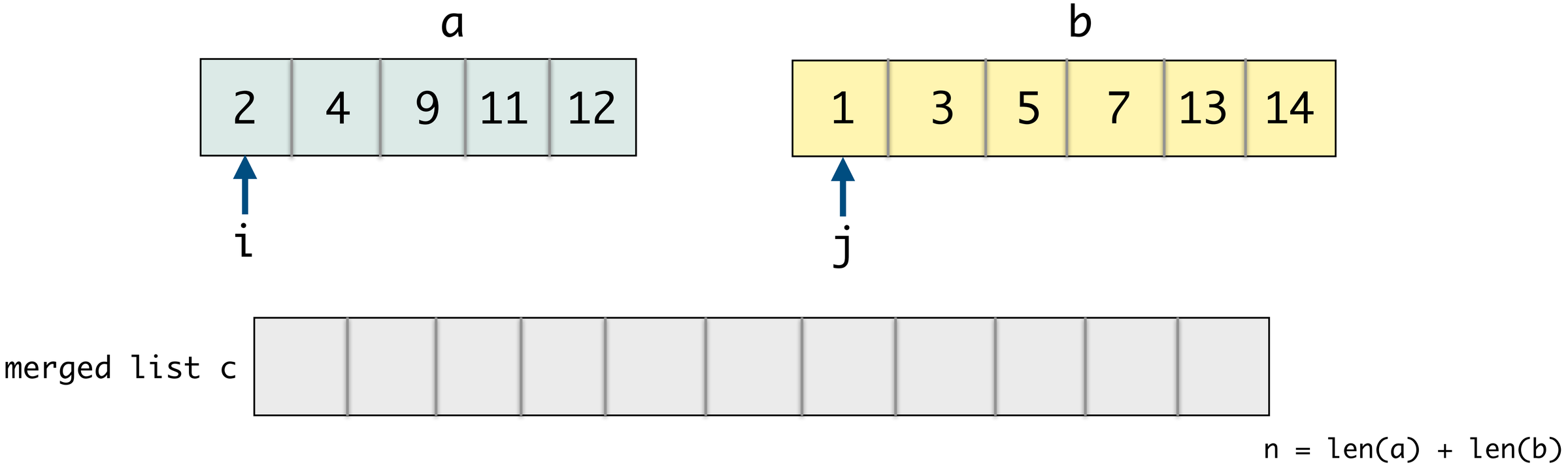
Merge Sort: Idea

- If we split the list in half, sorting the left and right half are smaller versions of the same problem
- **Recursive thinking:** if the recursion fairy sorted the left and right halves of the list, how long does it take to combine them into a single sorted list?



Merging Sorted Lists

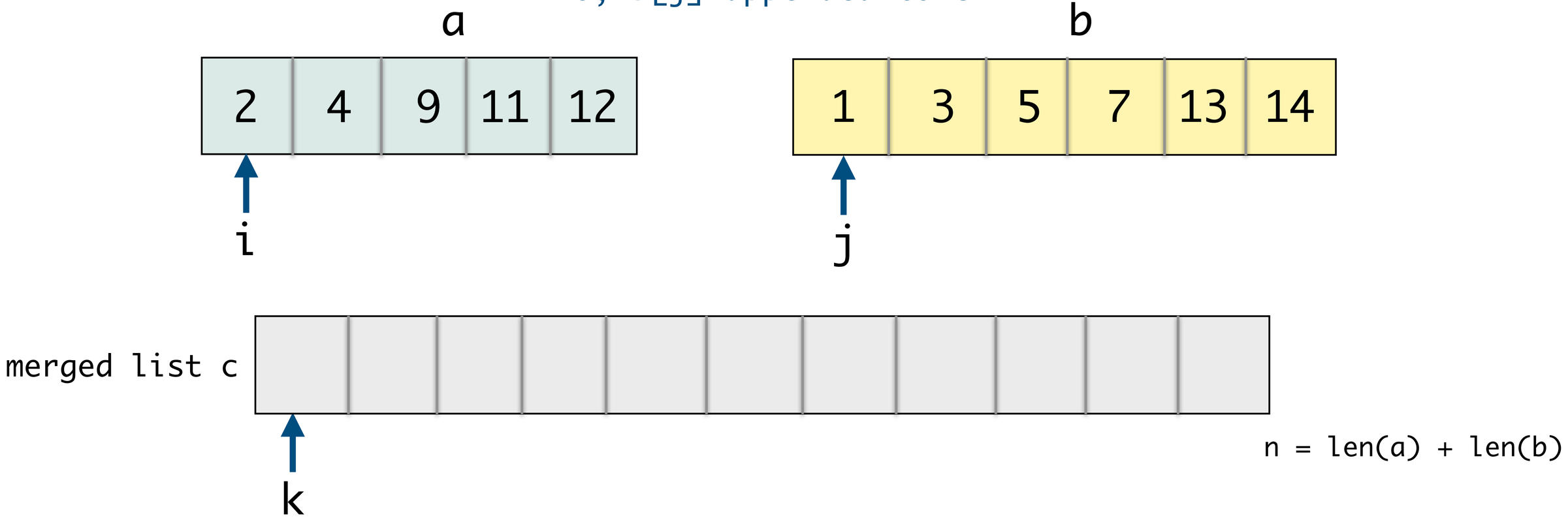
- **Problem.** Given two sorted lists `a` and `b`, how quickly can we merge them into a single sorted list?



Merging Sorted Lists

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

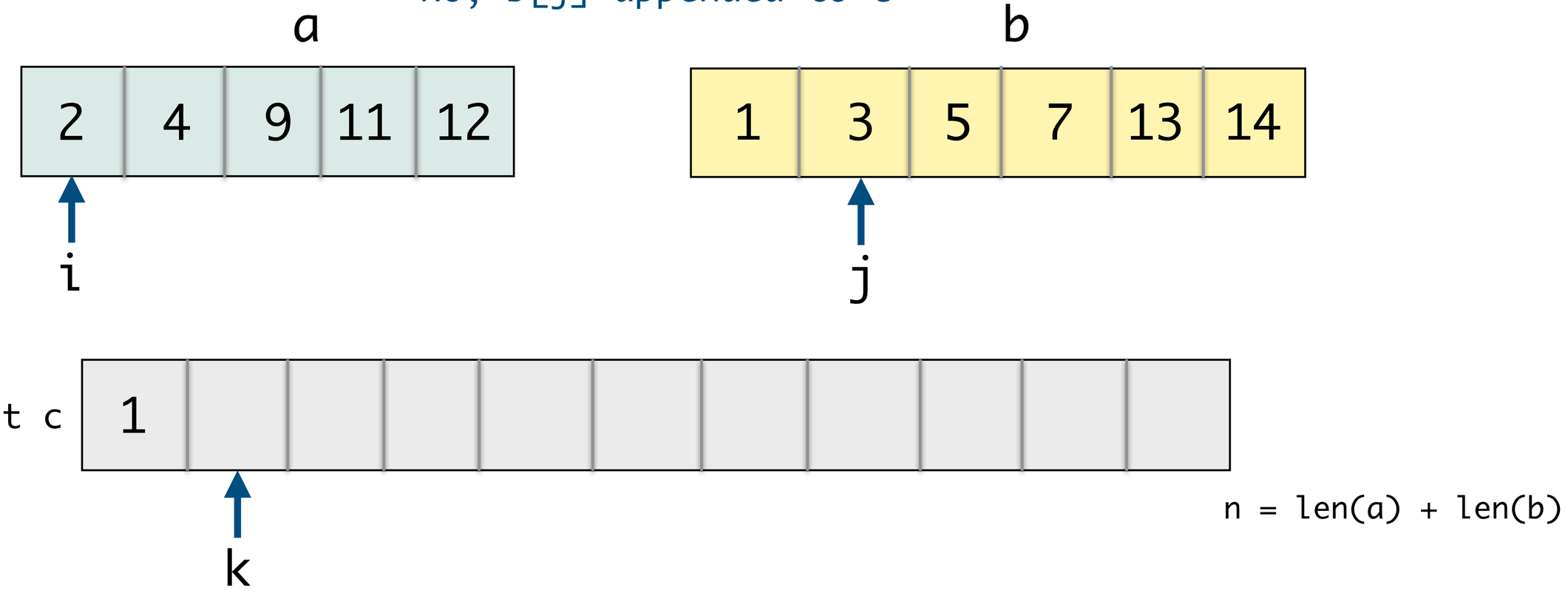
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

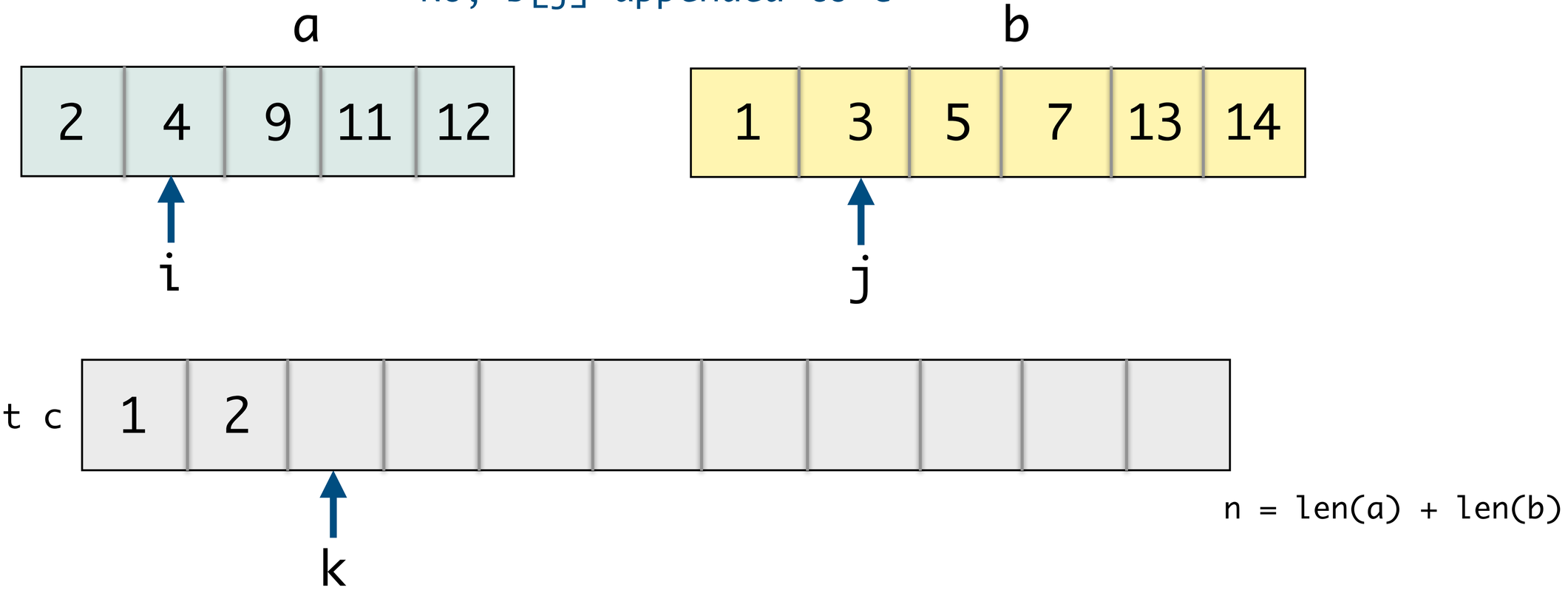
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

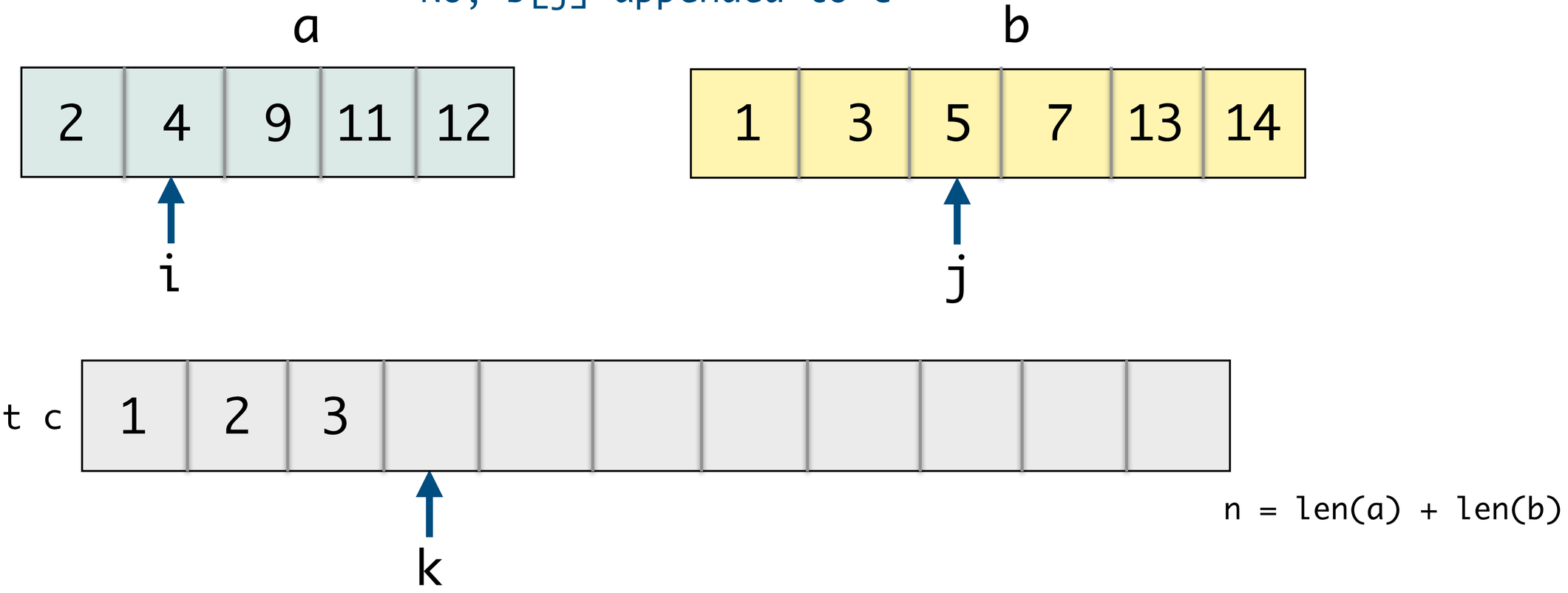
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

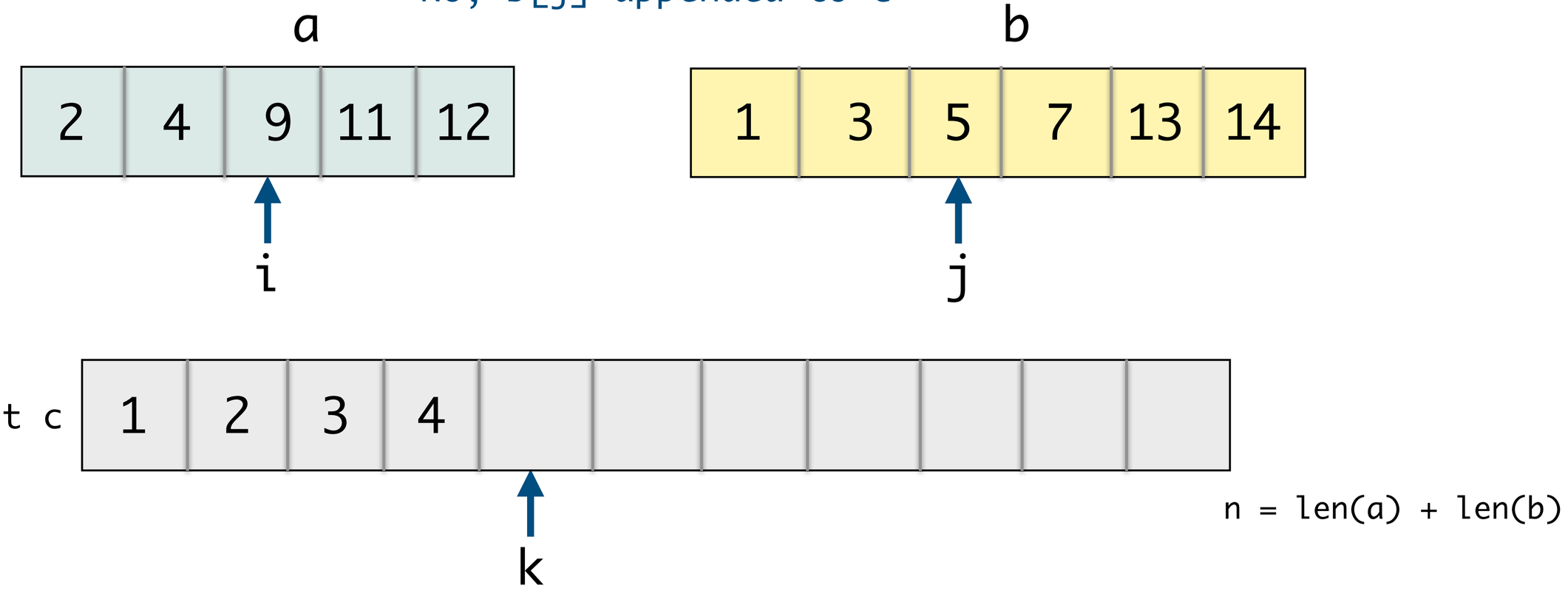
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

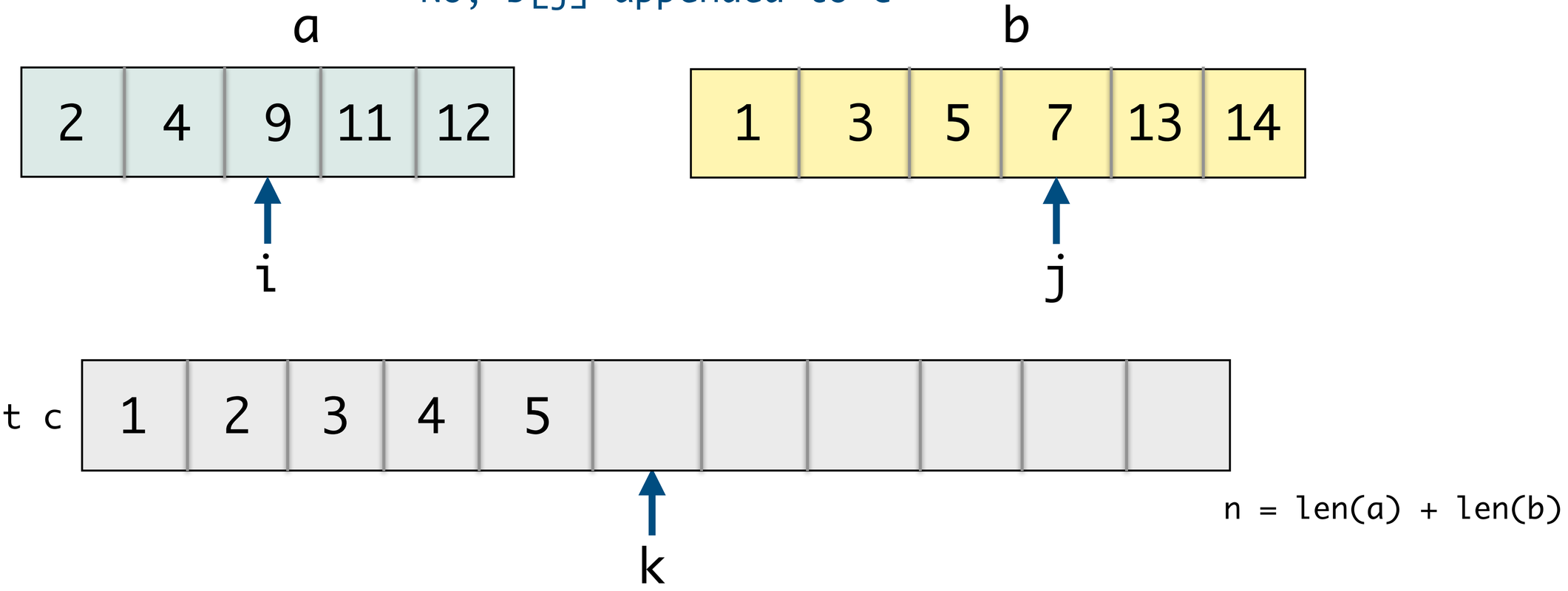
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

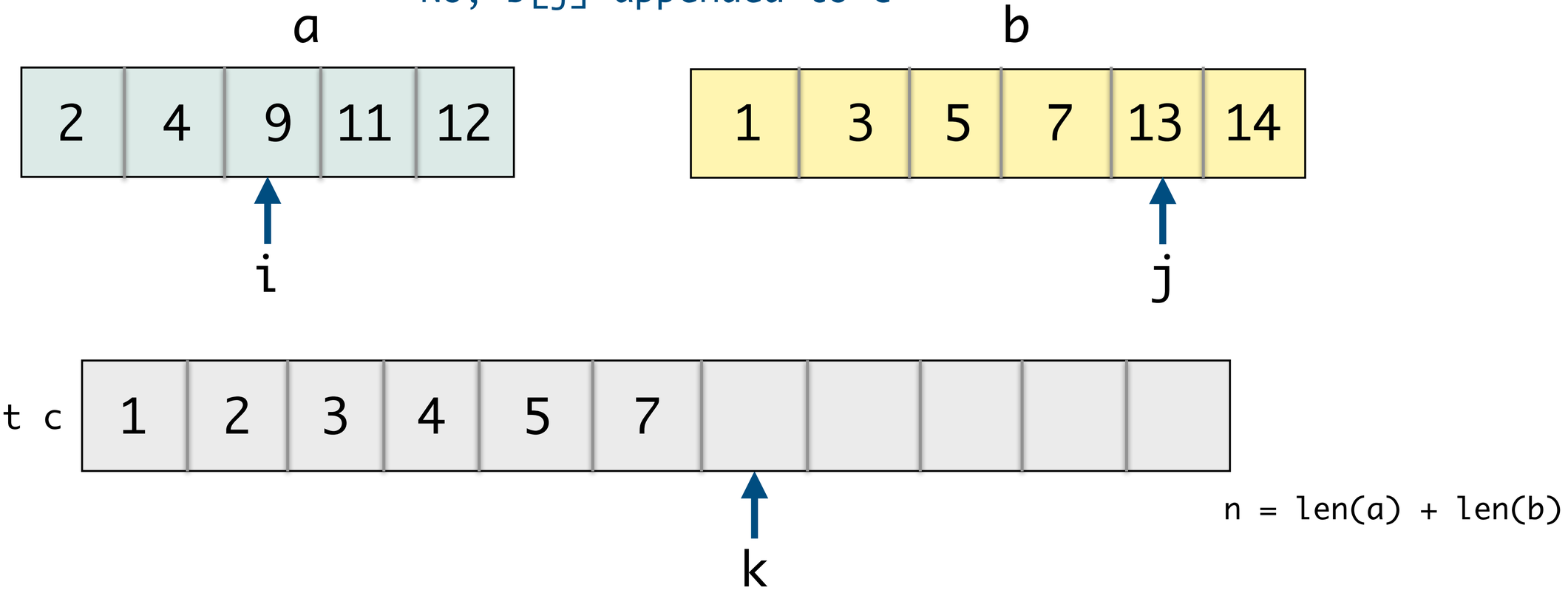
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

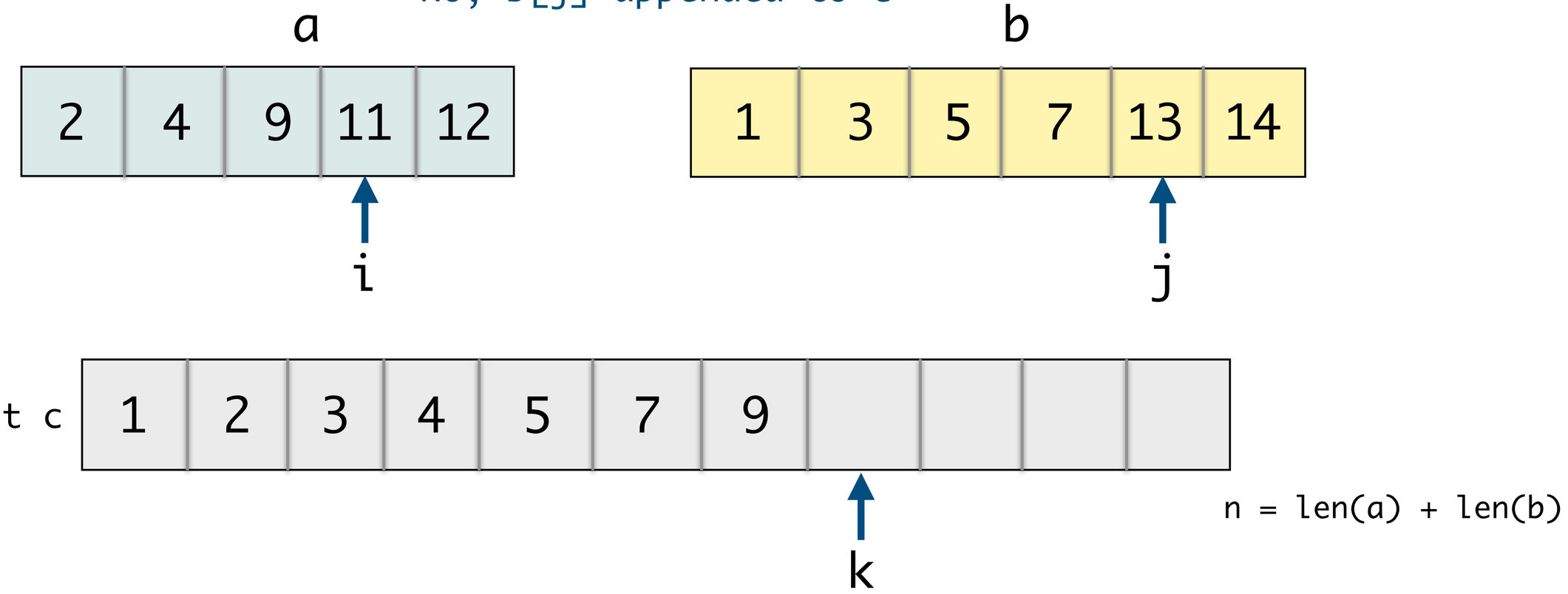
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

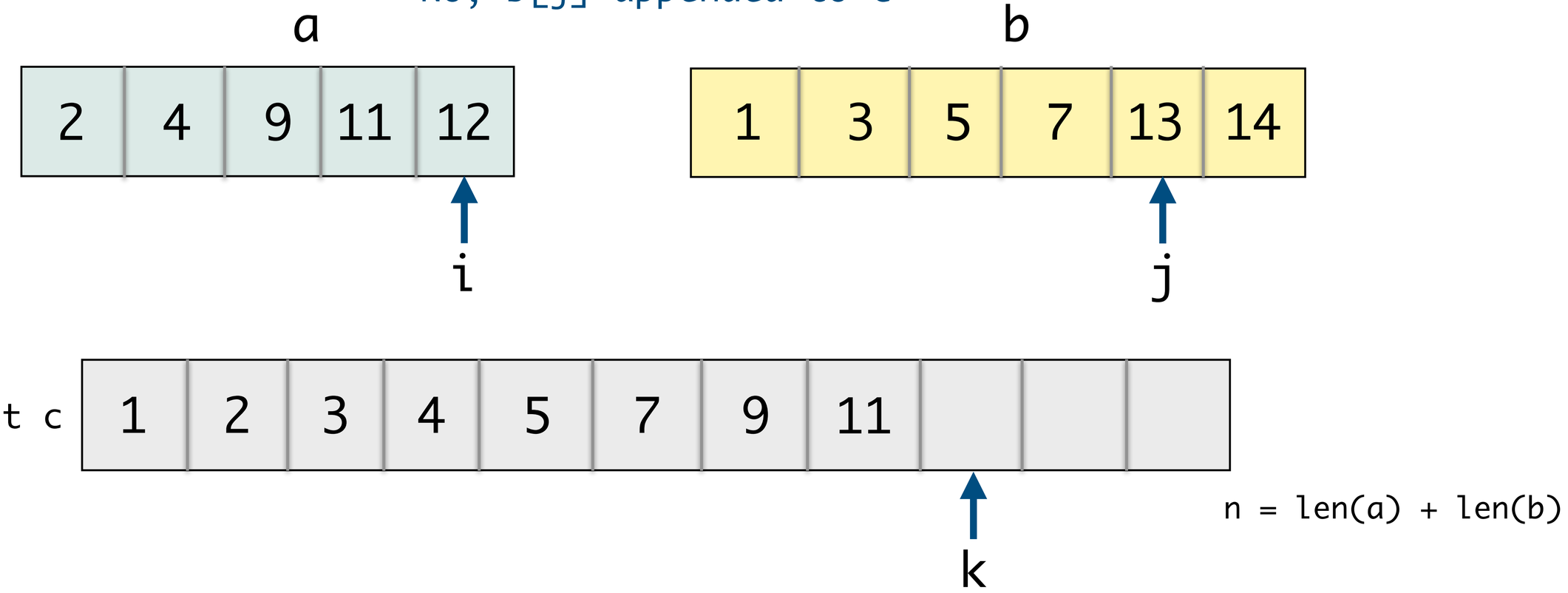
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

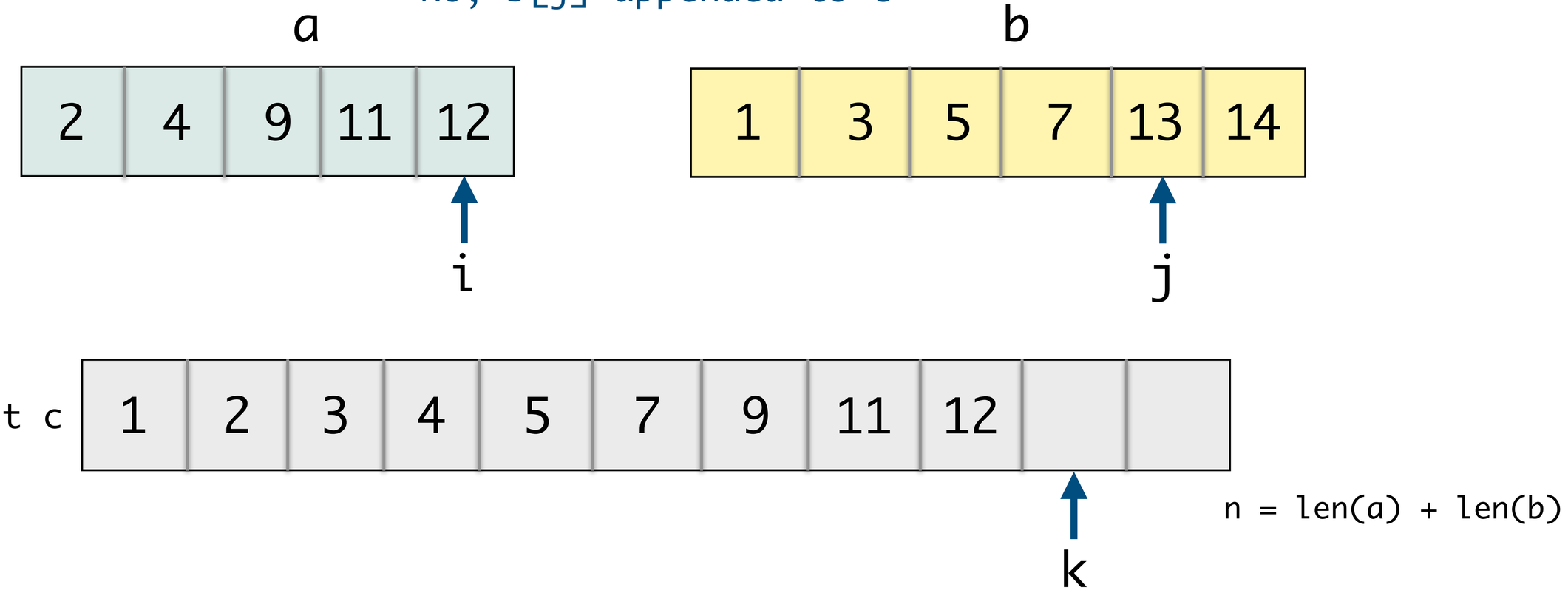
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

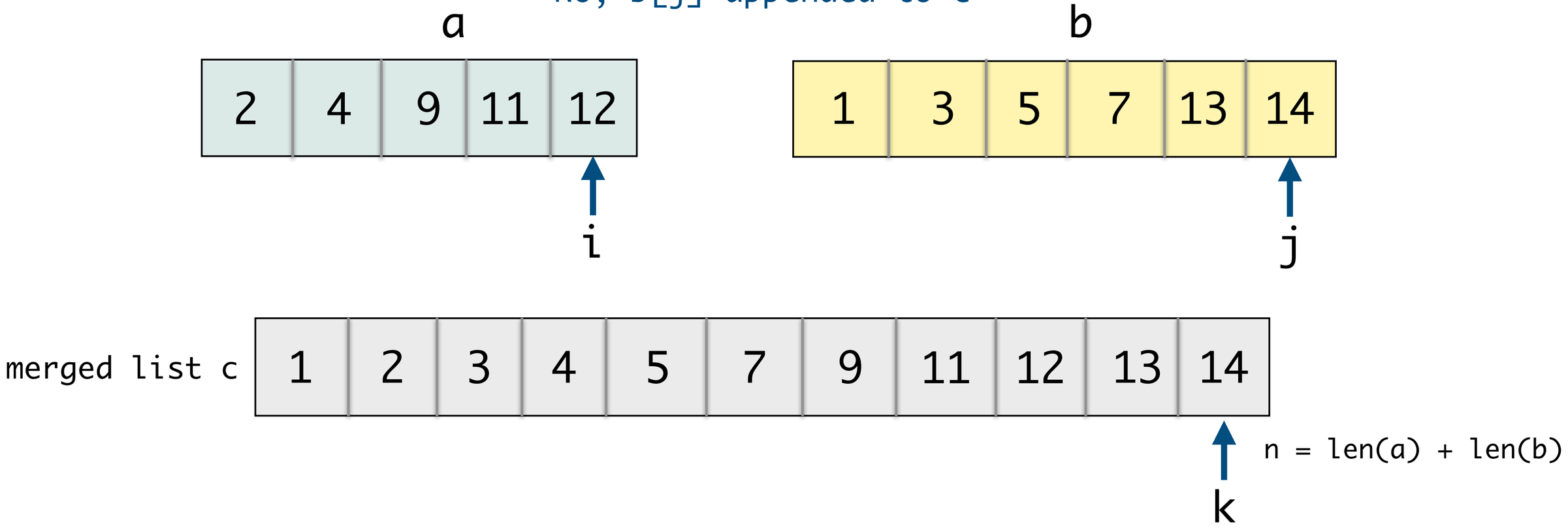
- Yes, a[i] appended to c
- No, b[j] appended to c



Merging Sorted Lists

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

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



Merging Sorted Lists

- Algorithm:
  - Walk through lists a, b, c maintaining current position of indices i, j, k
  - Compare $a[i]$ and $b[j]$, whichever is smaller gets put in the spot of $c[k]$
  - Simple loop that runs $n =$ `len(a) + len(b)` times
- **Takeaway**:
  - Merging two sorted lists into one is an $O(n)$ step algorithm!
- We can use this merge procedure to design our recursive merge sort algorithm

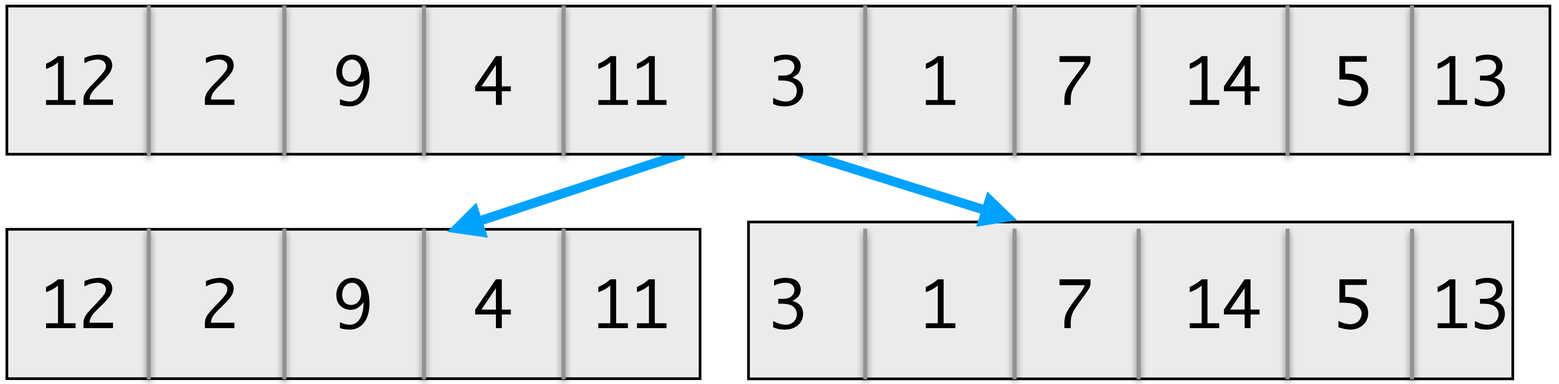
Merge Sort

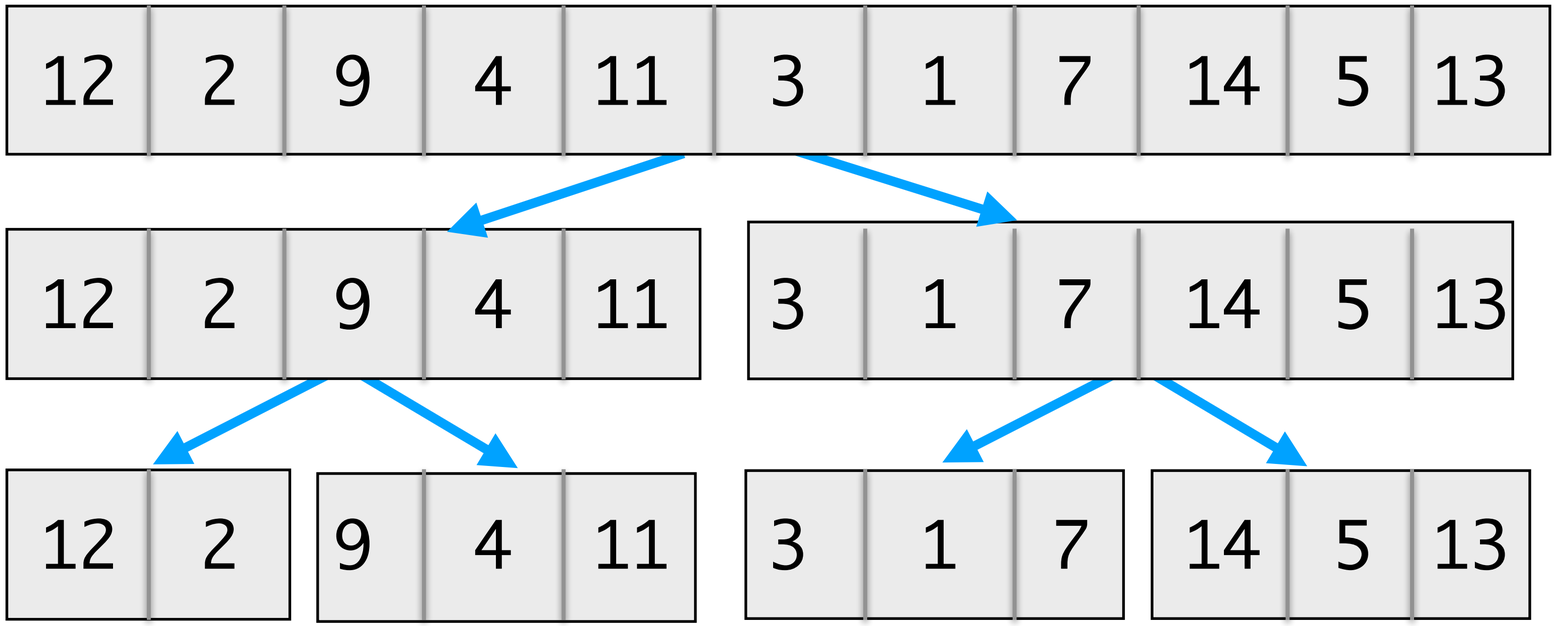
- Base case:
  - If list is empty or contains a single element: its already sorted! We can return it as is.
- Recursive case: `n = len(L) and m = n//2`
  - Recursively sort left and right halves: `L[:m]` and `L[m:]` (take the recursively leap of faith)
  - Merge the sorted lists into a single list and return it
- Things you may be thinking:
  - What?
  - Where is the sorting happening?
  - Is this magic?

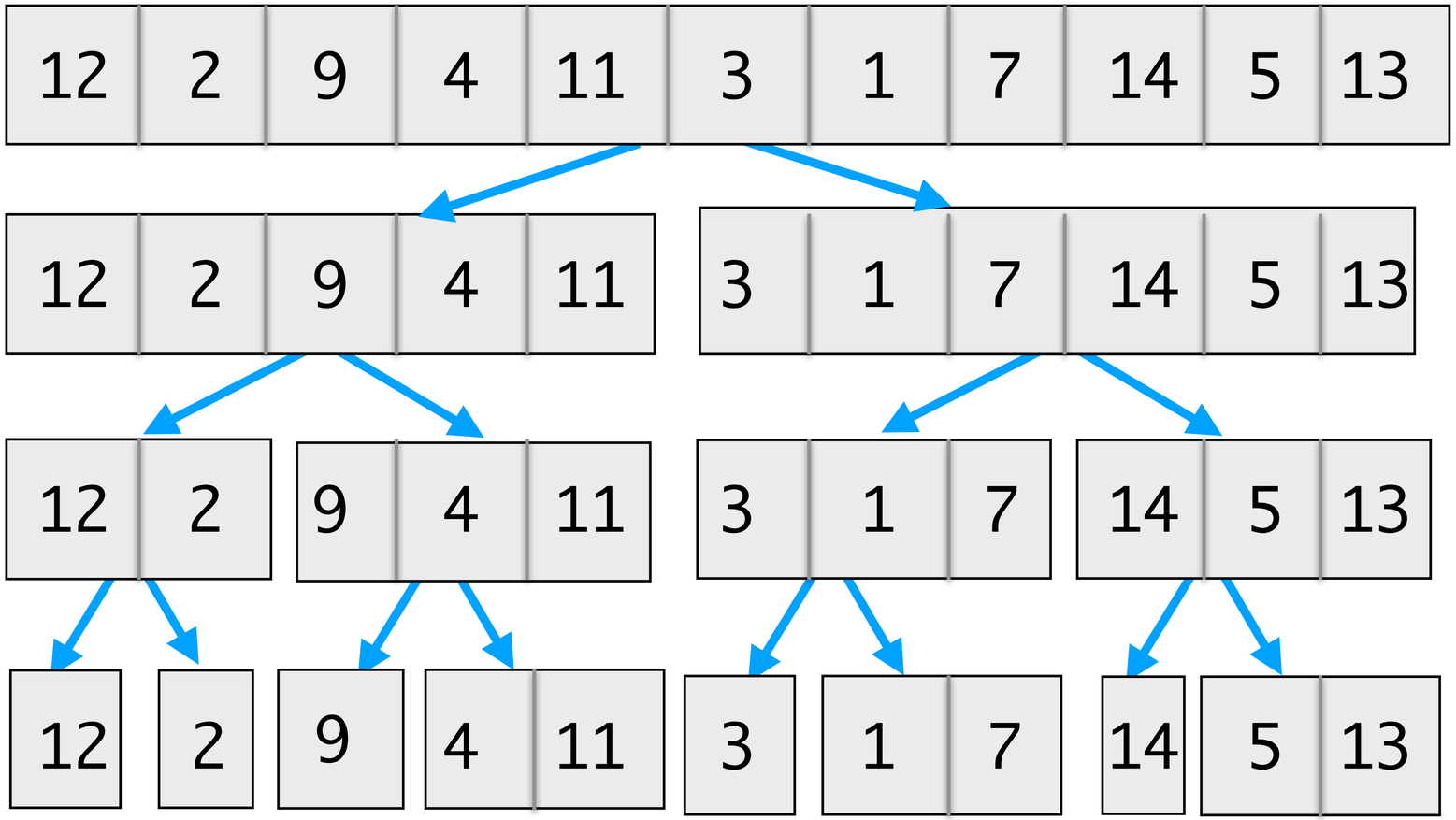
Merge Sort

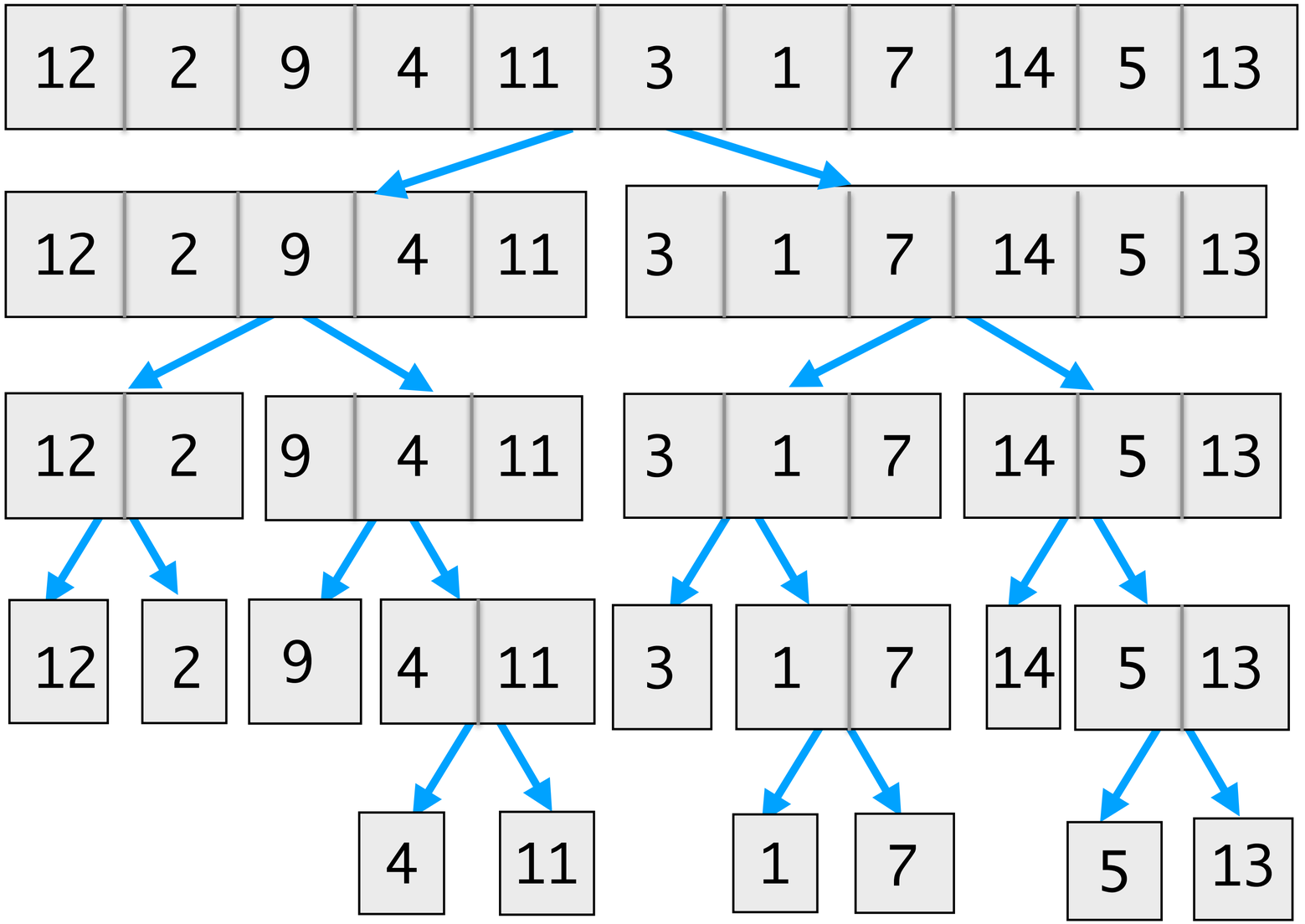
- Base case:
  - If list is empty or contains a single element: its already sorted! We can return it as is.
- Recursive case: `n = len(L) and m = n//2`
  - Recursively sort left and right halves: `L[:m]` and `L[m:]` (take the recursively leap of faith)
  - Merge the sorted lists into a single list and return it
- Things you may be thinking:
  - What?
  - Where is the sorting happening?
  - Is this magic?

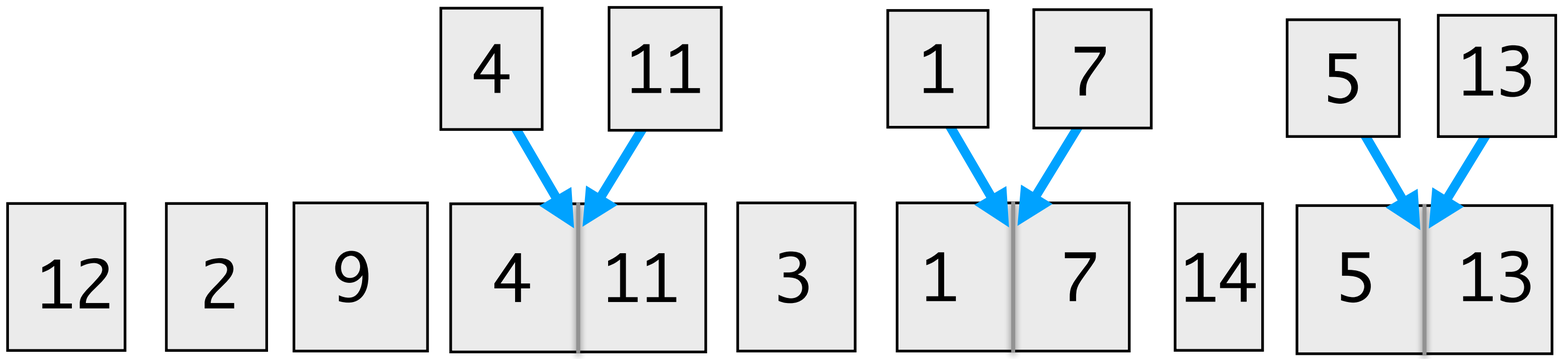
Let's Play Out the Recursion to Understand

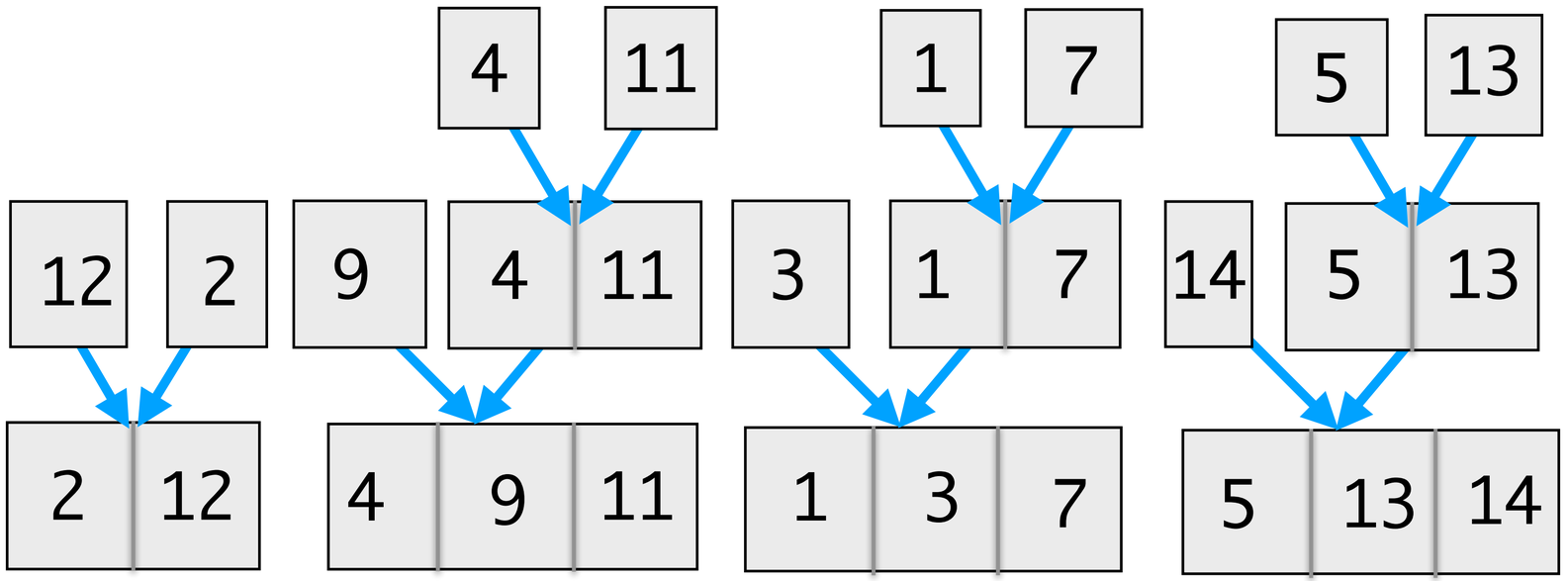

12 2 9 4 11 3 1 7 14 5 13
12 2 9 4 11
3 1 7 14 5 13

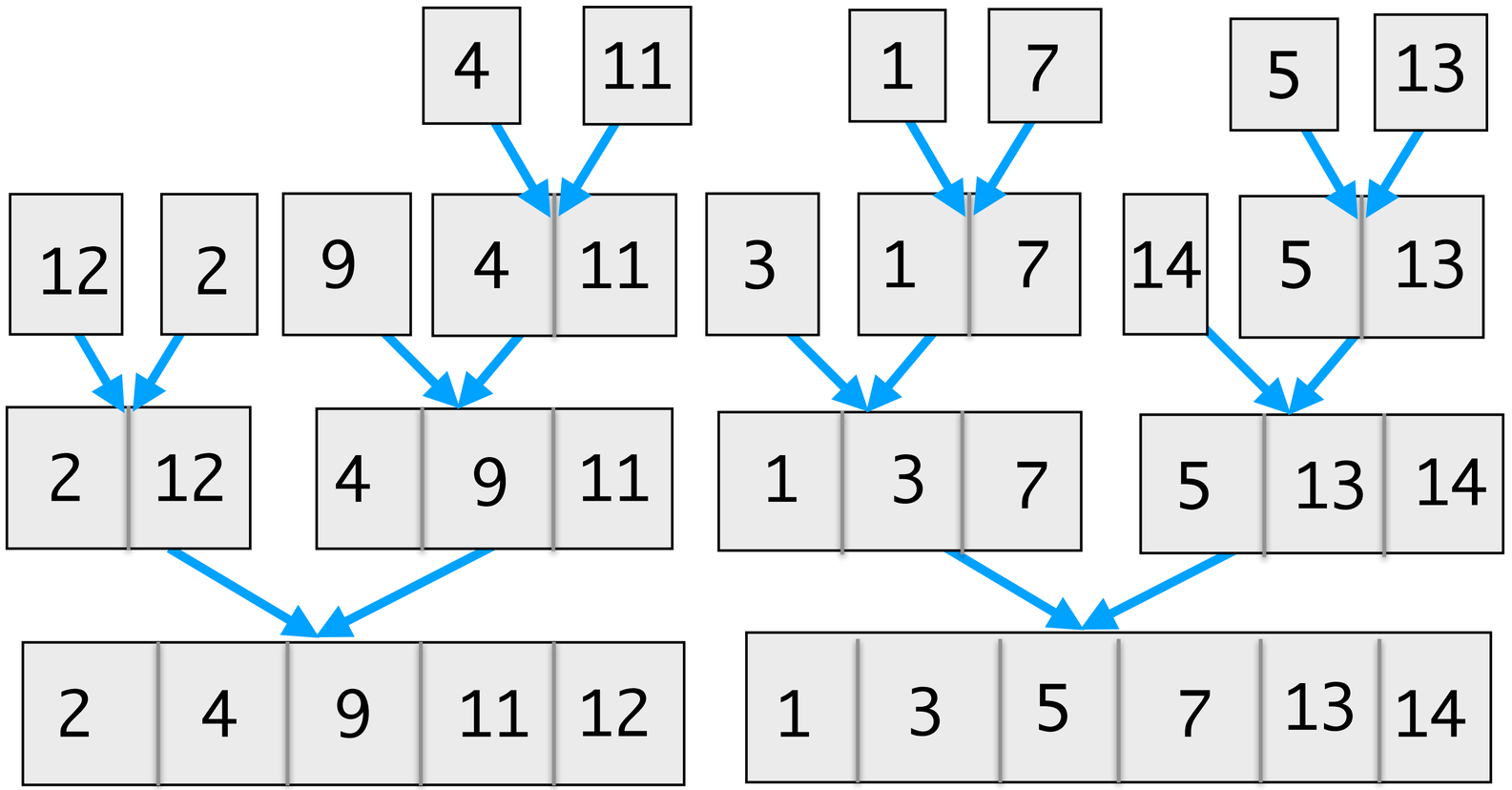

12 2 9 4 11 3 1 7 14 5 13
12 2 9 4 11
3 1 7 14 5 13
12 2
9 4 11
3 1 7
14 5 13

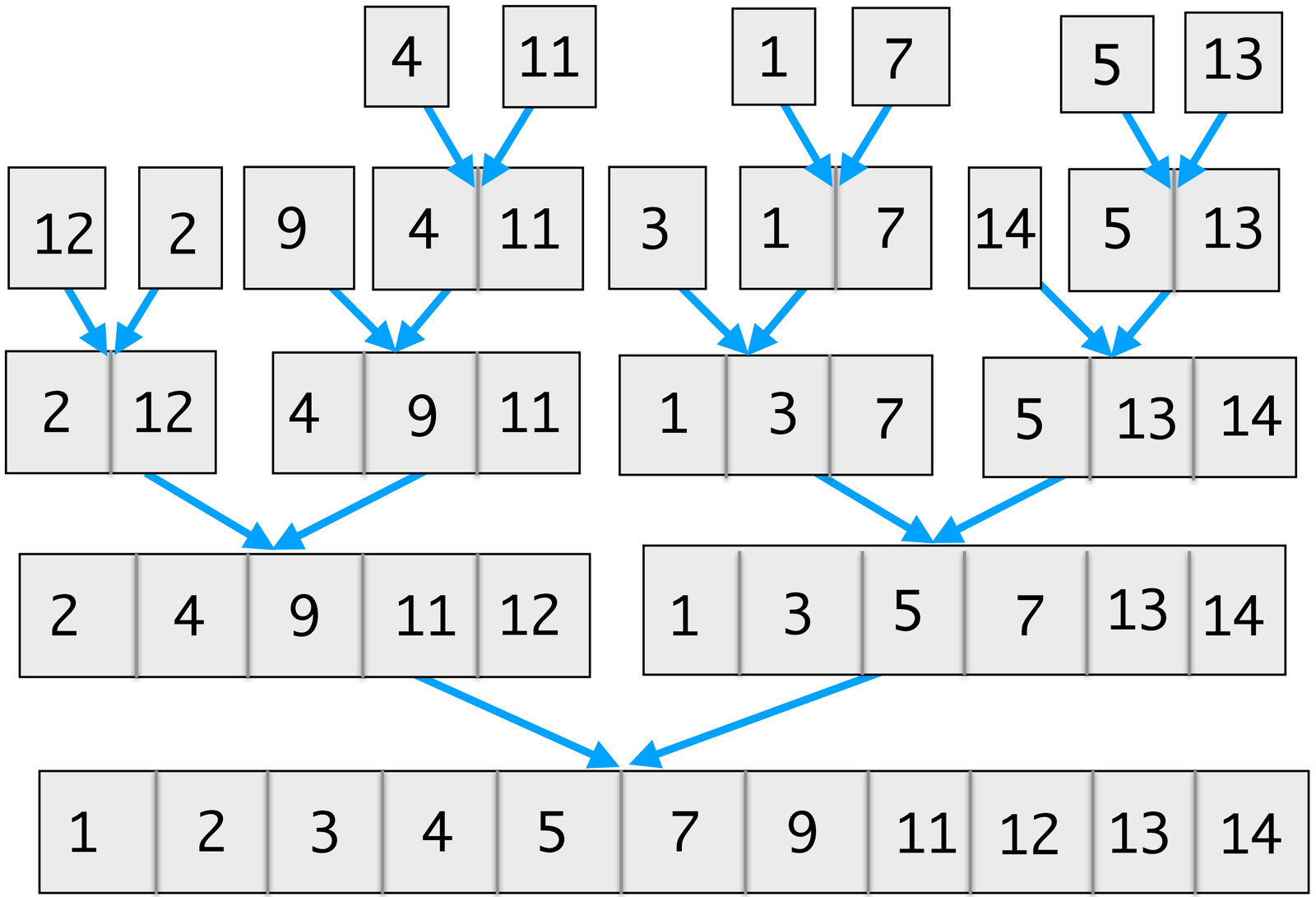

12 2 9 4 11 3 1 7 14 5 13
12 2 9 4 11
3 1 7 14 5 13
12 2
9 4 11
3 1 7
14 5 13
12
2
9
4 11
3
1 7
14
5 13


12 2 9 4 11 3 1 7 14 5 13
12 2 9 4 11
3 1 7 14 5 13
12 2
9 4 11
3 1 7
14 5 13
12
2
9
4 11
3
1 7
14
5 13
4
11
1
7
5
13


4
11
1
7
5
13
12
2
9
4
11
3
1
7
14
5
13


4
11
1
7
5
13
12
2
9
4
11
3
1
7
14
5
13
2
12
4
9
11
1
3
7
5
13
14


4
11
1
7
5
13
12
2
9
4
11
3
1
7
14
5
13
2
12
4
9
11
1
3
7
5
13
14
2
4
9
11
12
1
3
5
7
13
14

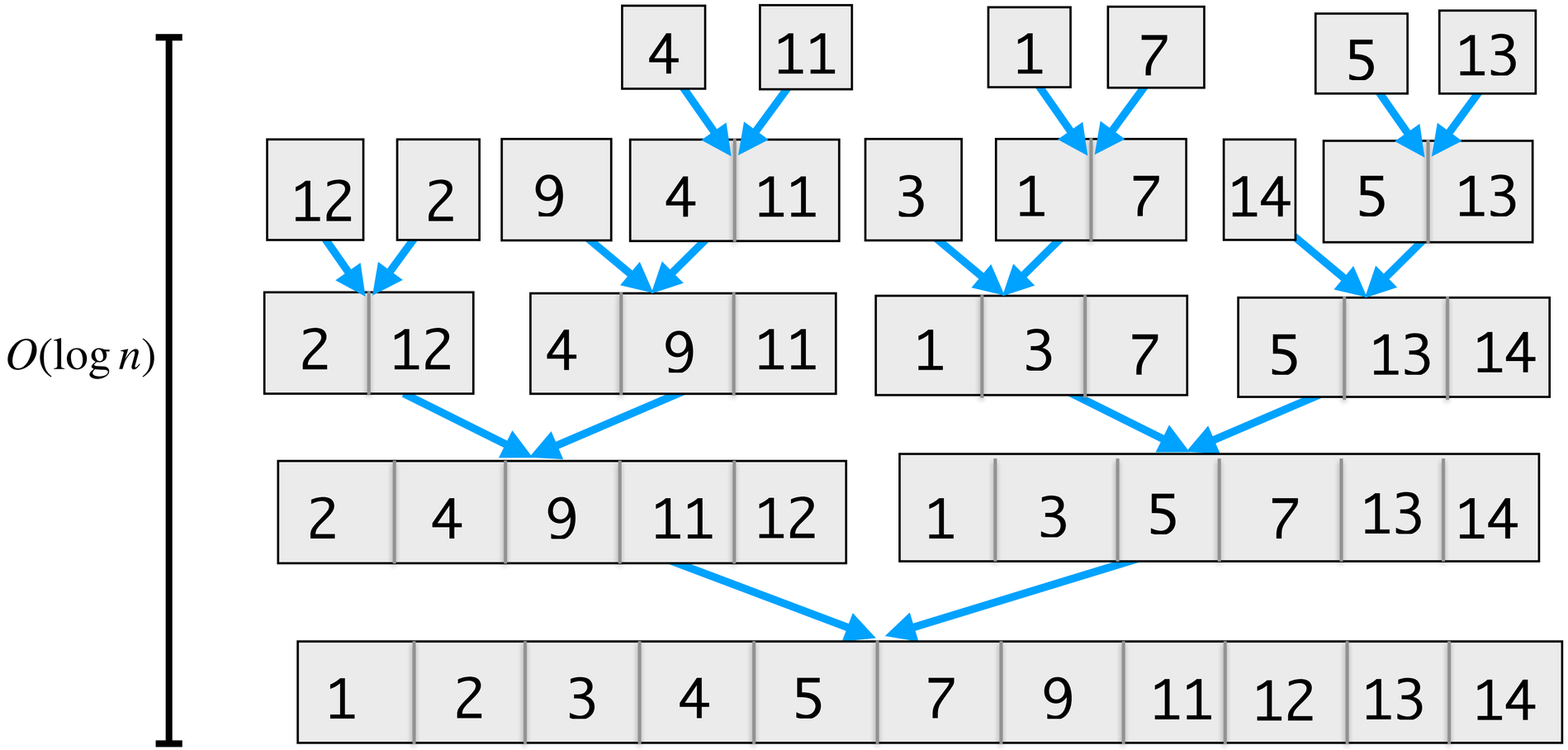

4
11
1
7
5
13
12
2
9
4
11
3
1
7
14
5
13
2
12
4
9
11
1
3
7
5
13
14
2
4
9
11
12
1
3
5
7
13
14
1
2
3
4
5
7
9
11
12
13
14

Merge Sort Takeaways

- All the sorting happens after the recursive calls hit their base case and "merge" on the way back
- Merging is cheap: cost is proportional to the size of the merged list
- How deep can the recursion be?
    - At each step, the size of the lists go down by half
    - Level 1: n
    - Level 2: $n/2$
    - ....
    - Level i: $n/2^i$
- When does size become 1: $n/2^i = 1$, level $i = \log_2 n$
- Thus, at most $\lceil \log_2 n \rceil + 1$ levels until we're done

Merge Sort Analysis

$O(n)$ steps per level, we have $O(\log n)$ levels, thus overall $O(n \log n)$ algorithm



Let's Implement Merge Sort!

See Jupyter Notebook

Overview of Algorithms

- We have seen algorithms that are
  - $O(\log n)$: binary search
  - $O(n)$: searching in an unsorted list
  - $O(n \log n)$: merge sort
  - $O(n^2)$: selection sort
- What about exponential time algorithms?
  - $O(2^n)$
- If we are not careful we can end up designing such an algorithm unintentionally
  - Example: recursive Fibonacci

Exponential Recursive Fibonacci:
Stupid Recursion

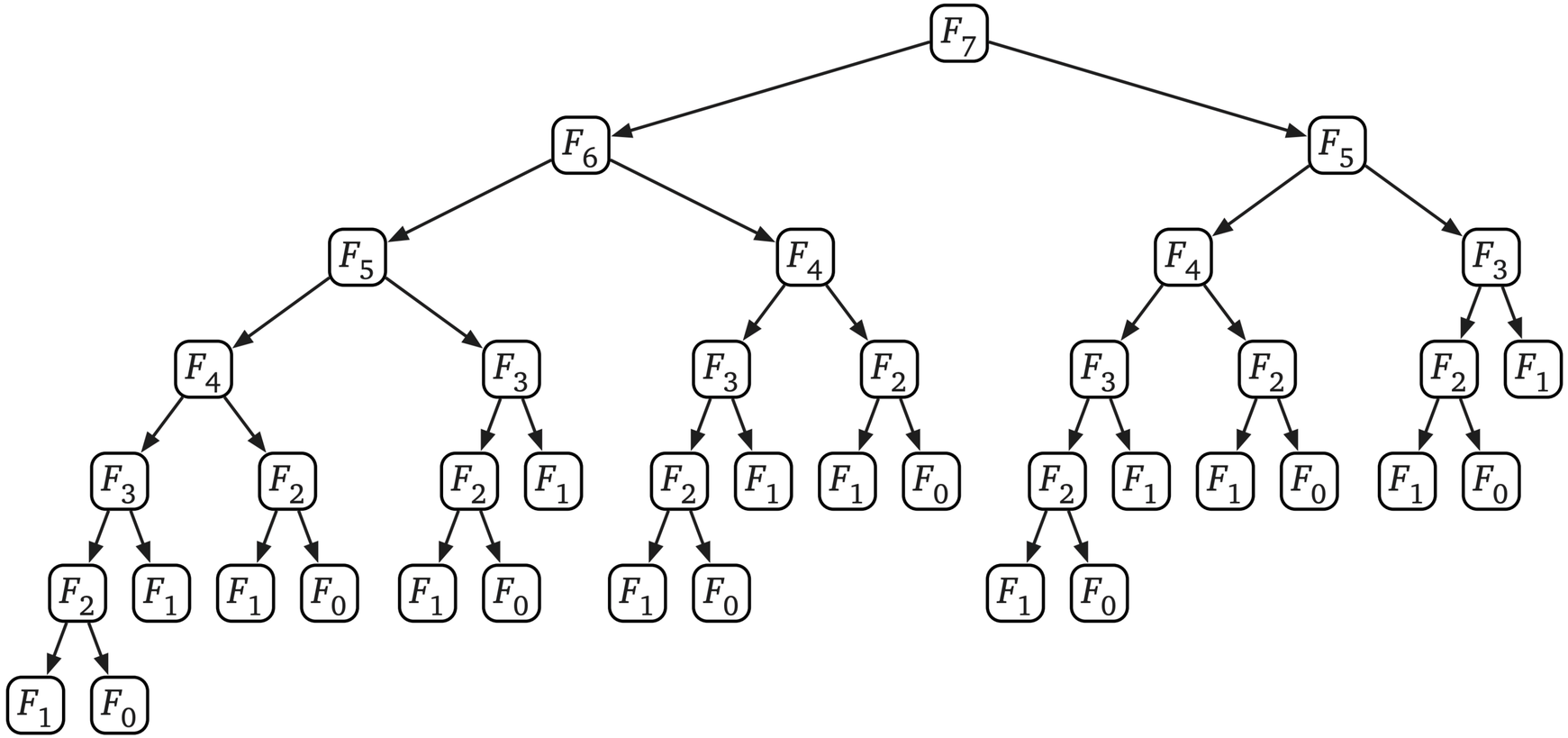
Fibonacci Sequence

- The fibonacci numbers F_n form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,
- $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-2} + F_{n-1}$ for all $n \geq 2$.
- Named after mathematician Pisa (later called Fibonacci), although it appears in early Indian mathematical texts
- Recursive Fibonacci is slow because it computes the same values over and over again

```
RecFibo(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        return RecFibo(n − 1) + RecFibo(n − 2)
```

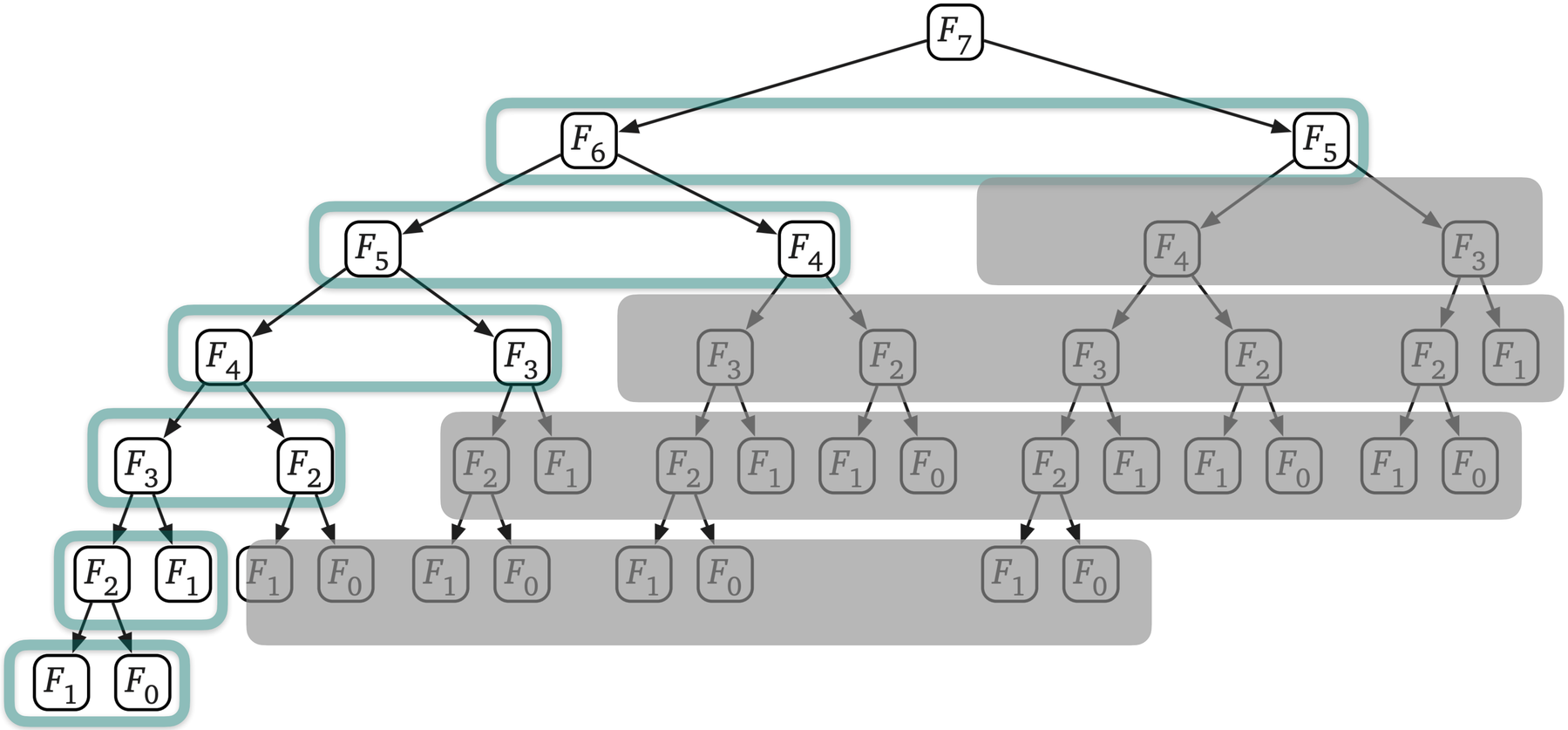
Recursive Fibonacci

- $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-2} + F_{n-1}$ for all $n \geq 2$.
- Number of recursive calls made are exponential in n



Fibonacci Loop

- Remembering the $(n-1)$th and $(n-2)$th number is sufficient to compute the nth Fibonacci number



Wrapping Up: Final Words

What We Have Learnt

- How to use Python as a computational tool
- Built toolboxes to manipulate strings, files
- Visualize data
- Abstraction
    - Choosing the right abstraction: functions, classes, etc.
    - Hiding the details from the user
- Automation
    - No need to write repetitive code
    - Iterative and recursive strategies
- **Problem solving strategies**/ Computational thinking
- Data Structures, recursive thinking

Where To Go from Here

- **CS 136:** Data Structure and Advanced Programming
- Discrete Mathematics (**Math 200**)

Course Description

This course combines work on program design, analysis, and verification with an introduction to the study of data structures. Data structures capture common ways in which to store and manipulate data, and they are important in the construction of sophisticated computer programs. We will use the Java programming language in class and for the assignments.

You will be expected to write several programs, ranging from the short and simple to the more complex and challenging as the semester progressess. Since one of our goals in this course is to help you learn how to write large, reliable programs composed from reusable pieces, we will be emphasizing the development of clear, modular programs that are easy to read, debug, verify, analyze, and modify.

DIY Problem Solving

About Project Euler

What is Project Euler?

Project Euler is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.

The motivation for starting Project Euler, and its continuation, is to provide a platform for the inquiring mind to delve into unfamiliar areas and learn new concepts in a fun and recreational context.

Who are the problems aimed at?

The intended audience include students for whom the basic curriculum is not feeding their hunger to learn, adults whose background was not primarily mathematics but had an interest in things mathematical, and professionals who want to keep their problem solving and mathematics on the cutting edge.

Can anyone solve the problems?

The problems range in difficulty and for many the experience is inductive chain learning. That is, by solving one problem it will expose you to a new concept that allows you to undertake a previously inaccessible problem. So the determined participant will slowly but surely work his/her way through every problem.

"Project Euler exists to encourage, challenge, and develop the skills and enjoyment of anyone with an interest in the fascinating world of mathematics."

Source: https://projecteuler.net/

Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/
- Selection sort images from: https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/lectures/11/Slides11.pdf