CS134 Lecture 33: Sorting

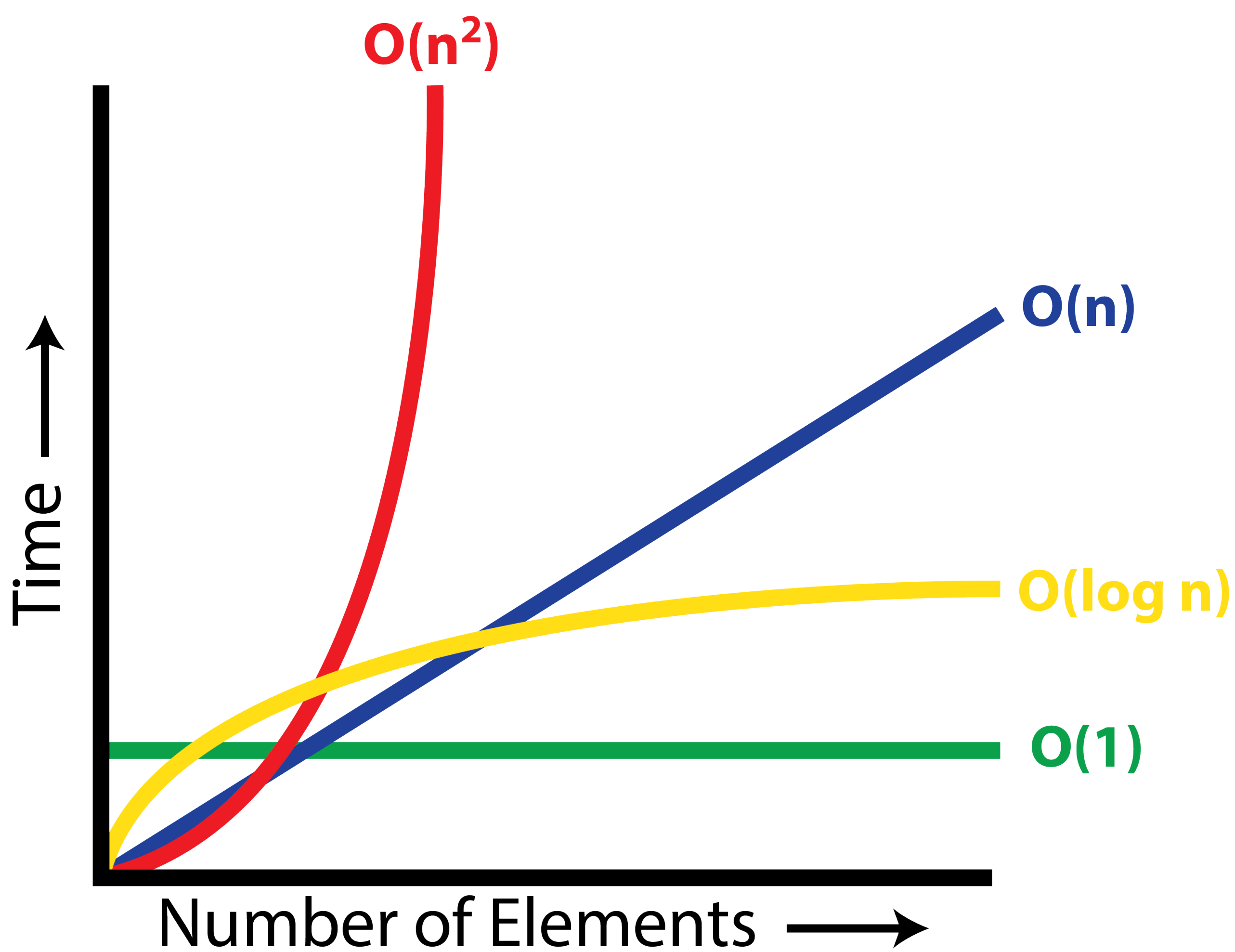
Announcements & Logistics

- **HW 10** due today @ 10 pm
    - Last HW!
- **Lab 10** starts (and hopefully finishes) in this week's labs
    - Very short lab on searching and sorting (today's lecture)
    - No prelab
    - Individual lab but can discuss strategies with lab mate
- CS134 Scheduled Final: **Friday, May 17, 9:30 AM**
    - Room: **TCL 123 (Wege Auditorium) ***

Do You Have Any Questions?

Last Time: Searching & Efficiency

- Searching requires scanning through entire list in the worst case
  - O(n) where n is the size of the list
- We can do better if the list is sorted!
  - O(log n) by using binary search



Today: Sorting

- Discuss some classic sorting algorithms:
    - ***Selection sorting*** in $O(n^2)$ time
    - A brief (high level) discussion of how we can improve sorting to $O(n \log n)$
        - Overview of recursive ***merge sort*** algorithm

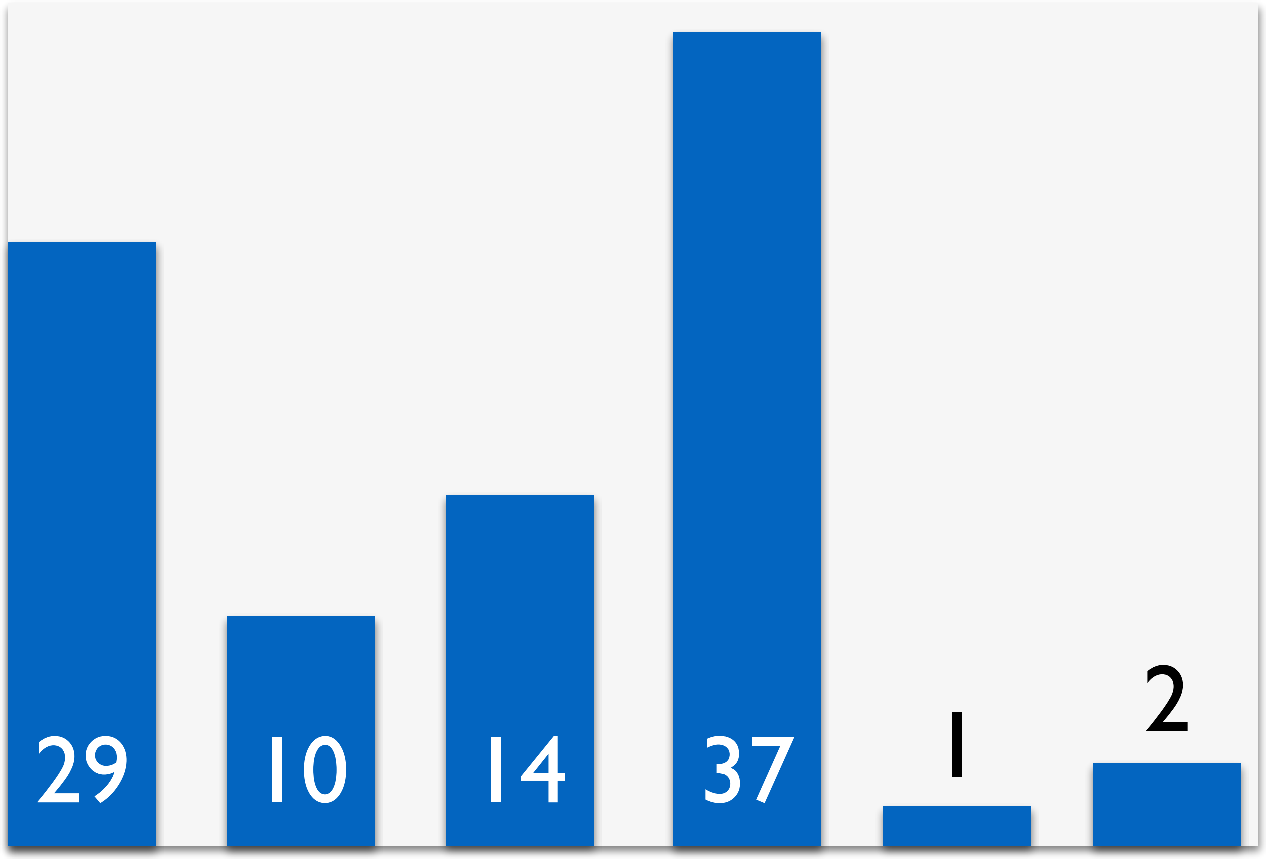
Sorting

Sorting

- **Problem:** Given a sequence of unordered elements, we need to sort the elements in ascending order.
- There are many ways to solve this problem!
- Built-in sorting functions/methods in Python
  - `sorted()`: ***function*** that returns a new sorted list
  - `sort()`: ***list method*** that mutates and sorts the list
- **Today:** how do we design our own sorting algorithm?
- **Question:** What is the best (most efficient) way to sort n items?
- We will use Big-O to find out!

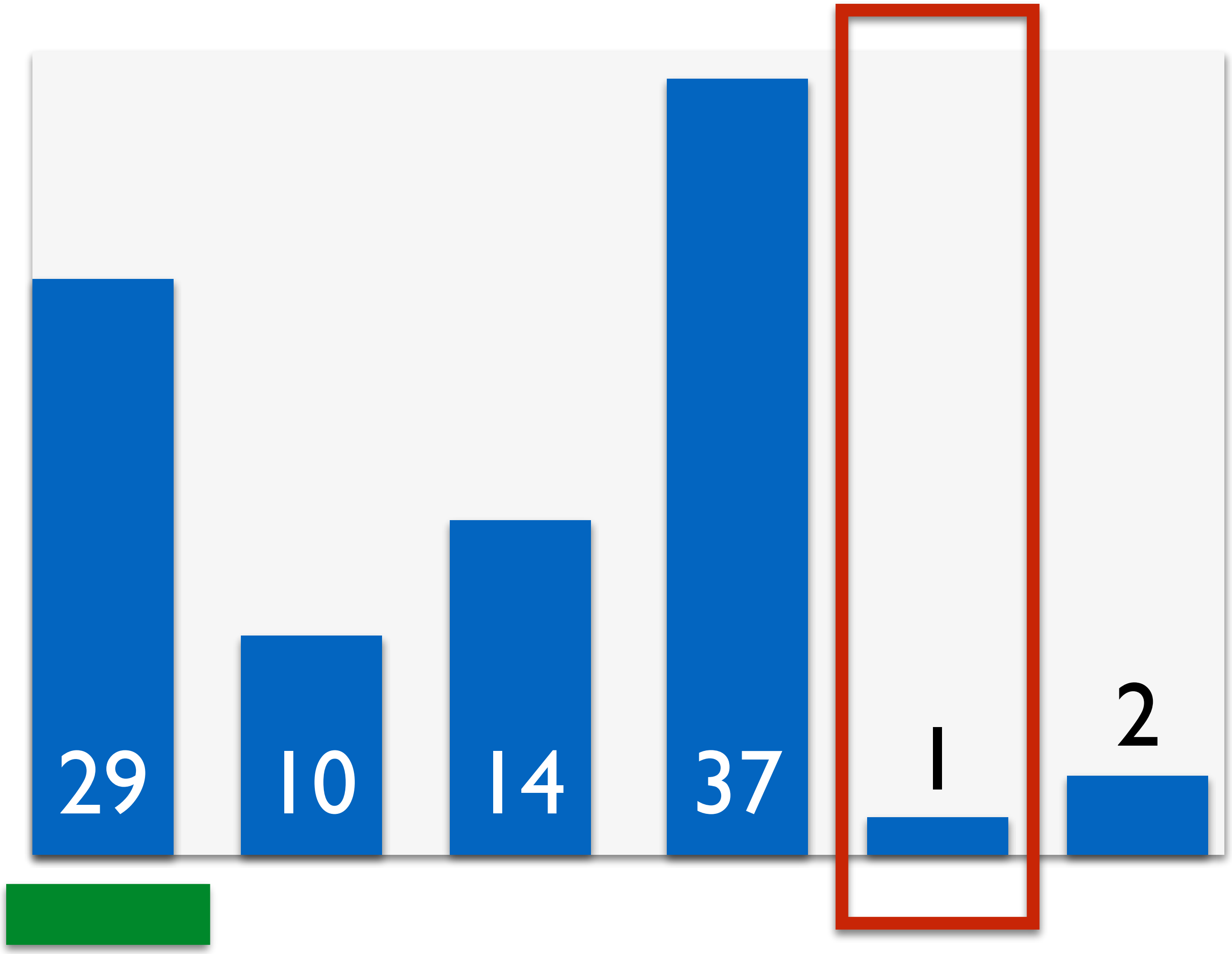
Selection Sort

- A possible approach to sorting elements in a list/array:
  - Find the smallest element and move (swap) it to the first position
  - *Repeat*: find the second-smallest element and move it to the second position, and so on



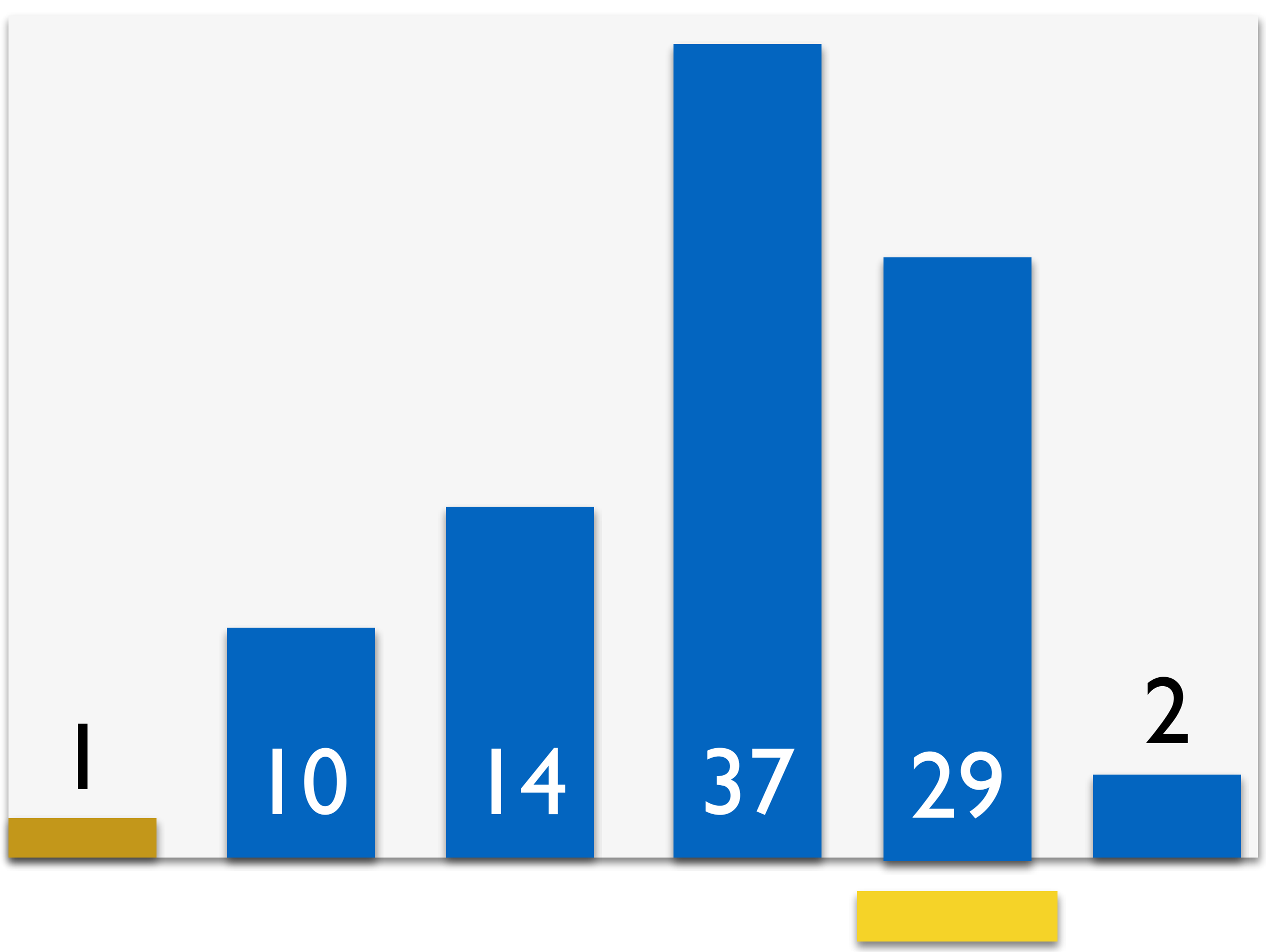
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on



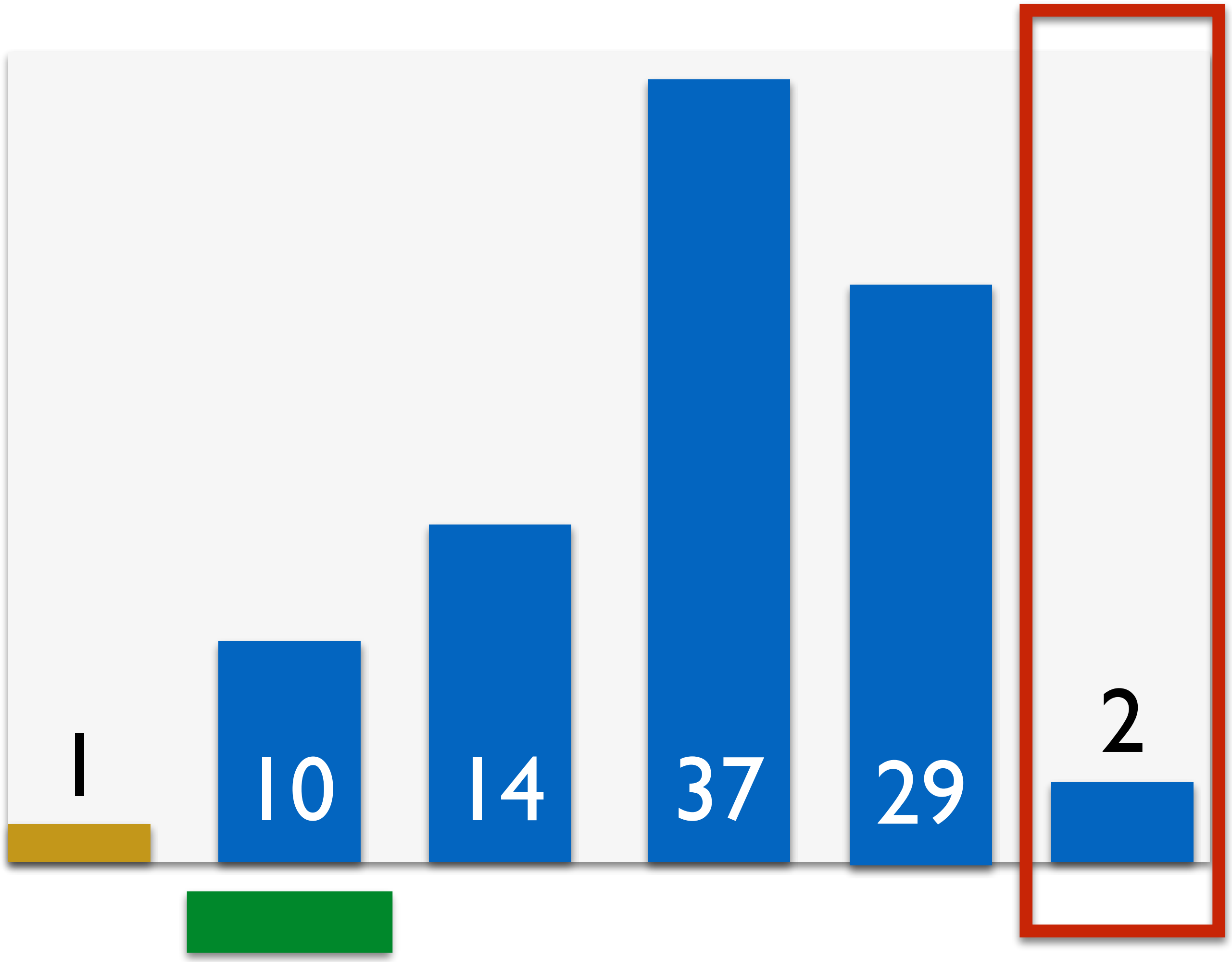
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on



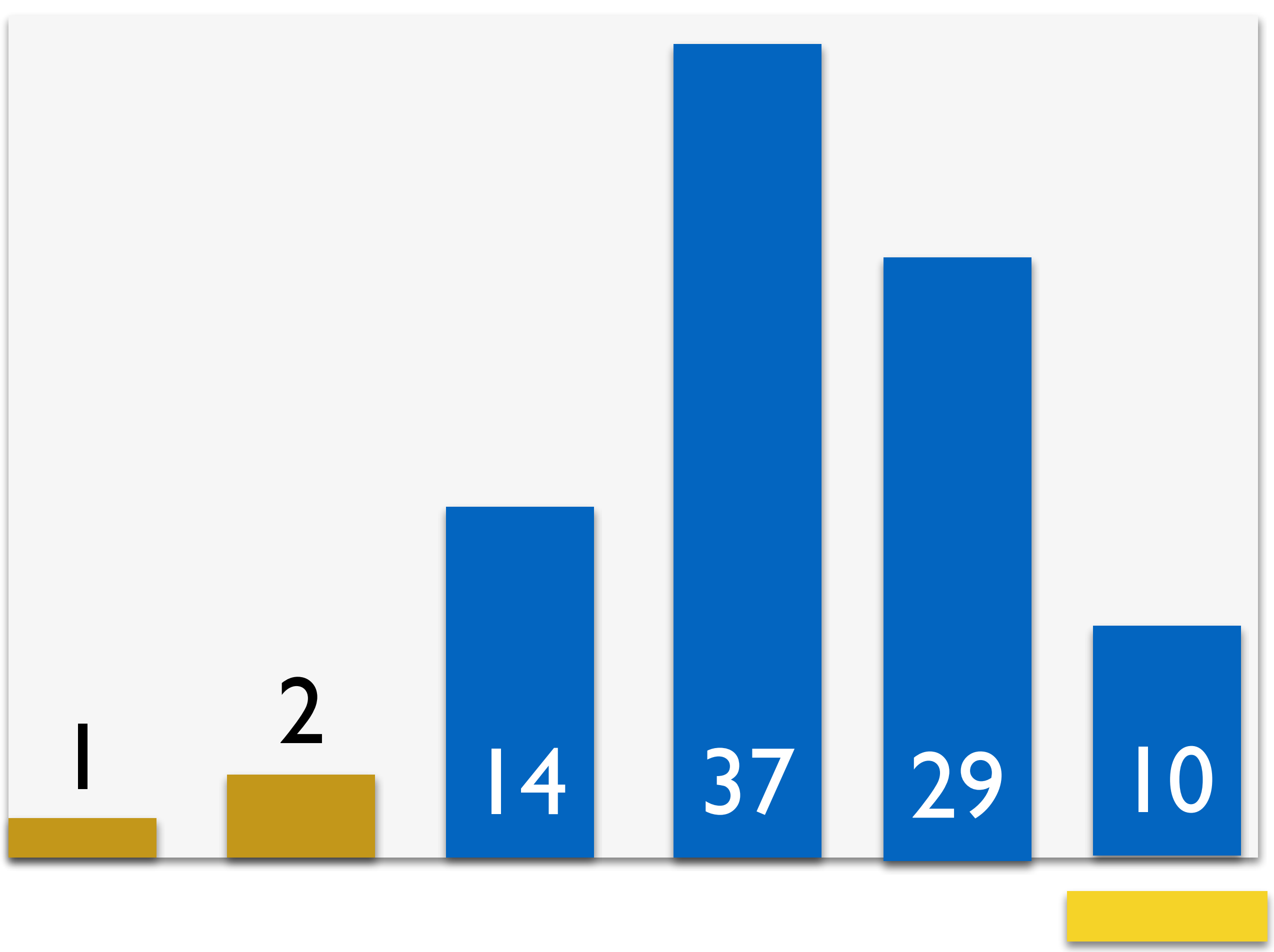
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on



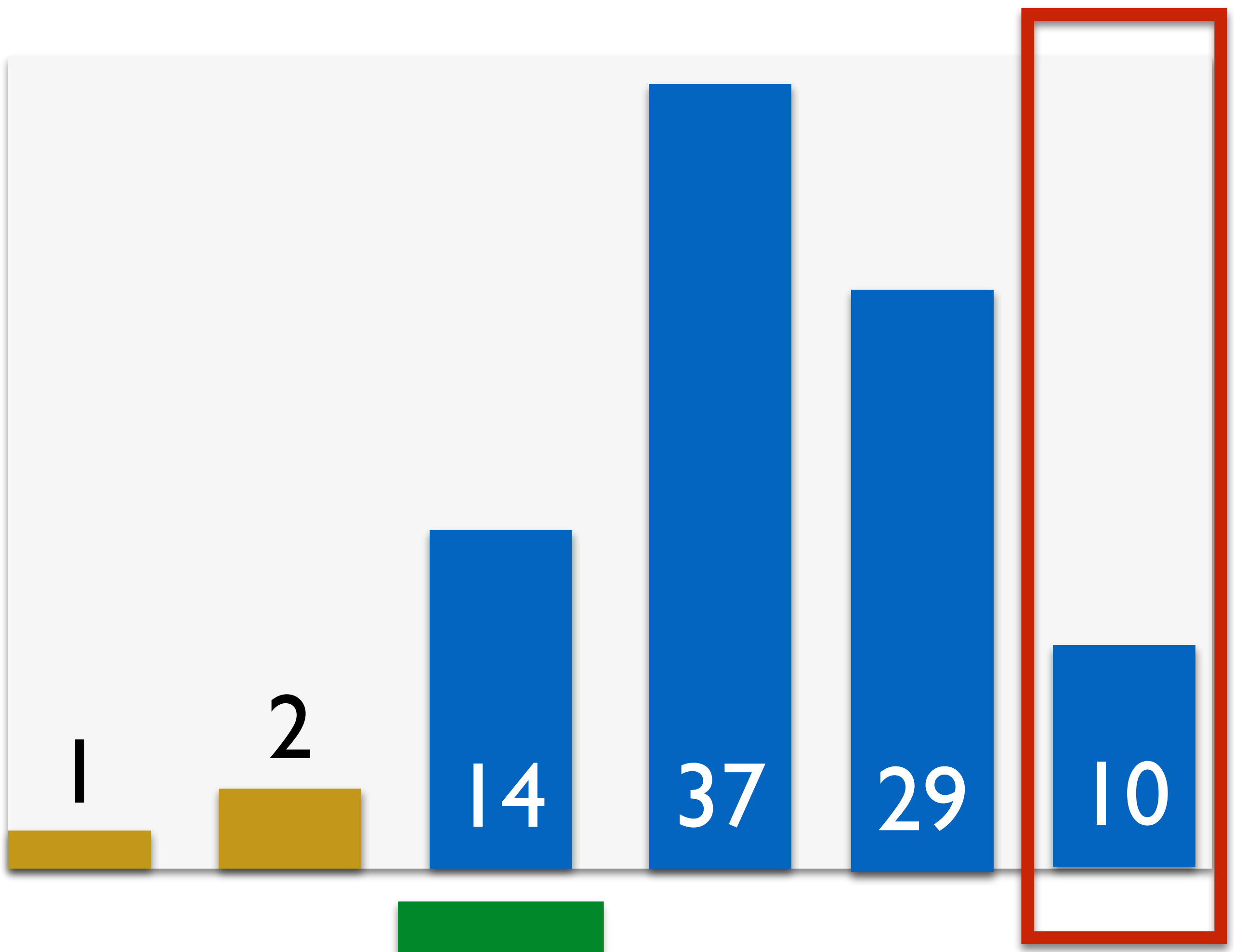
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



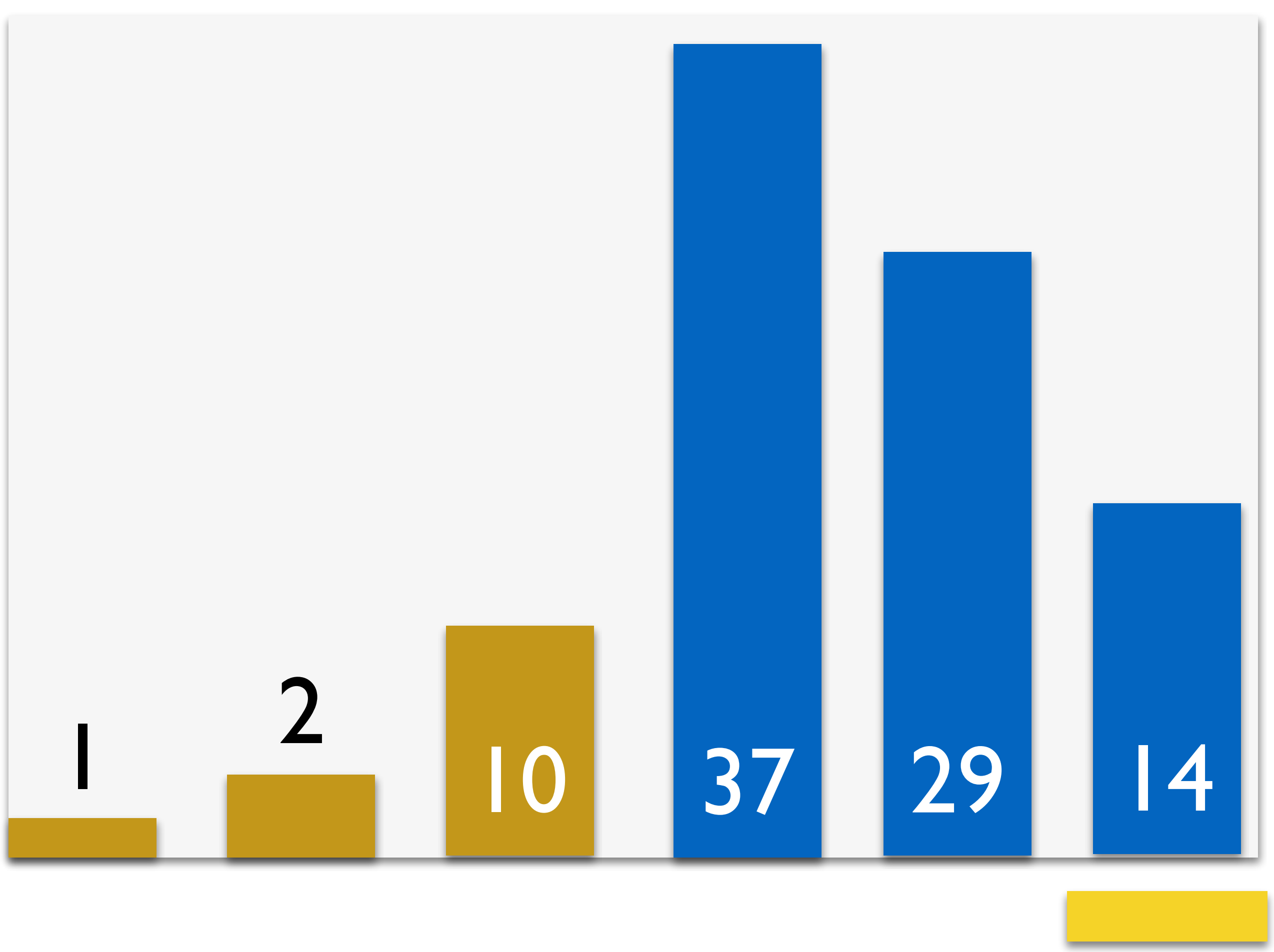
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



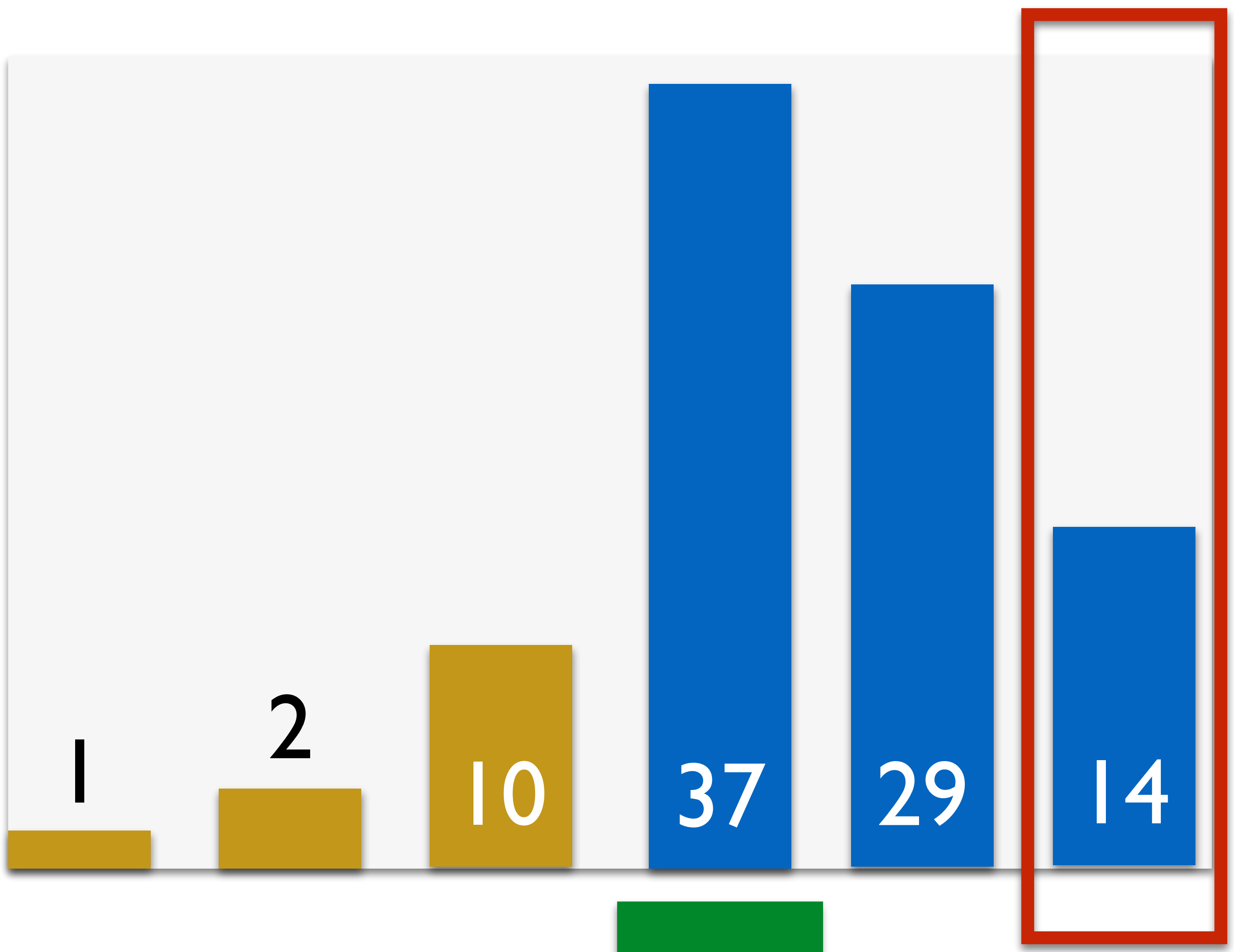
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



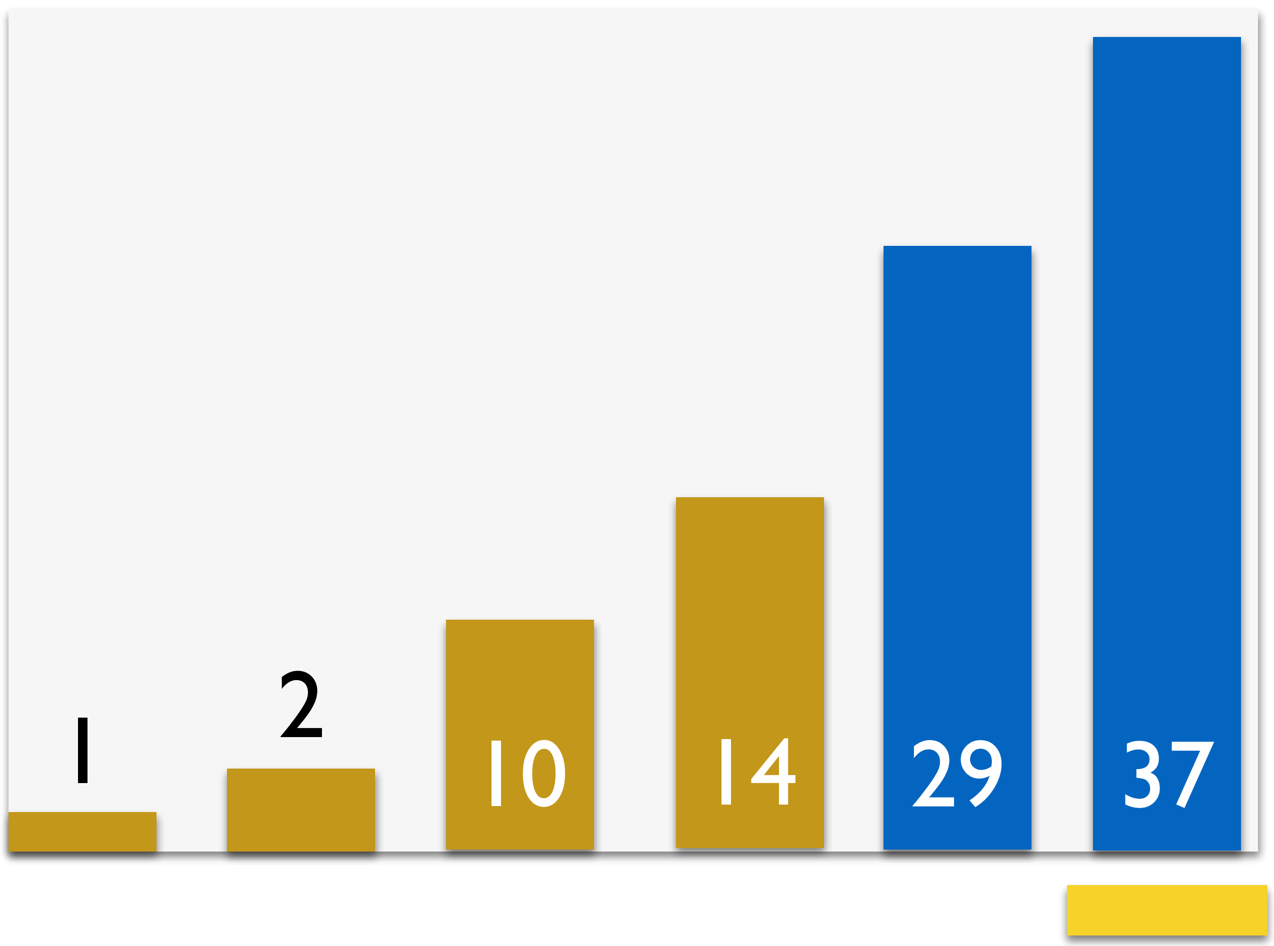
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



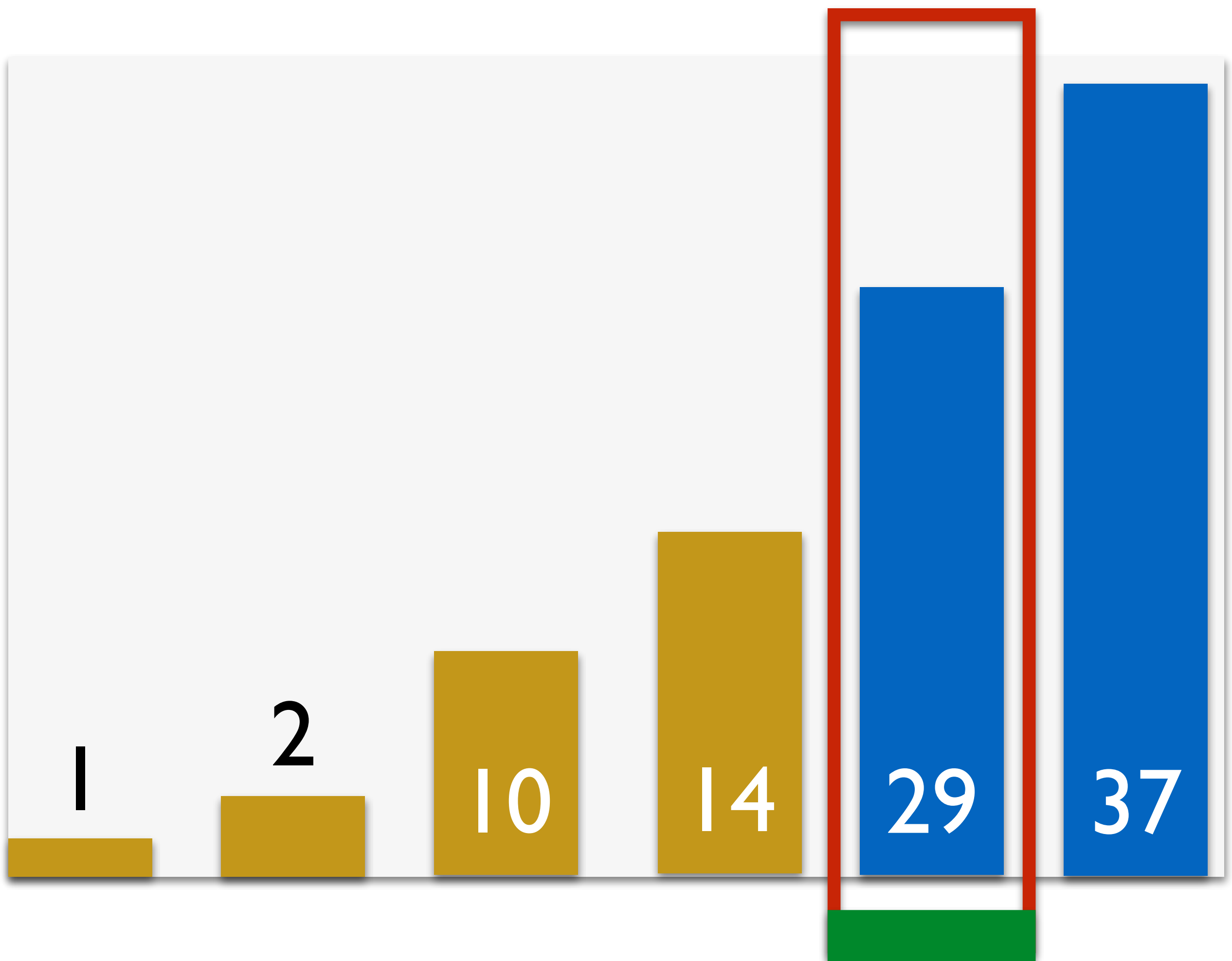
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



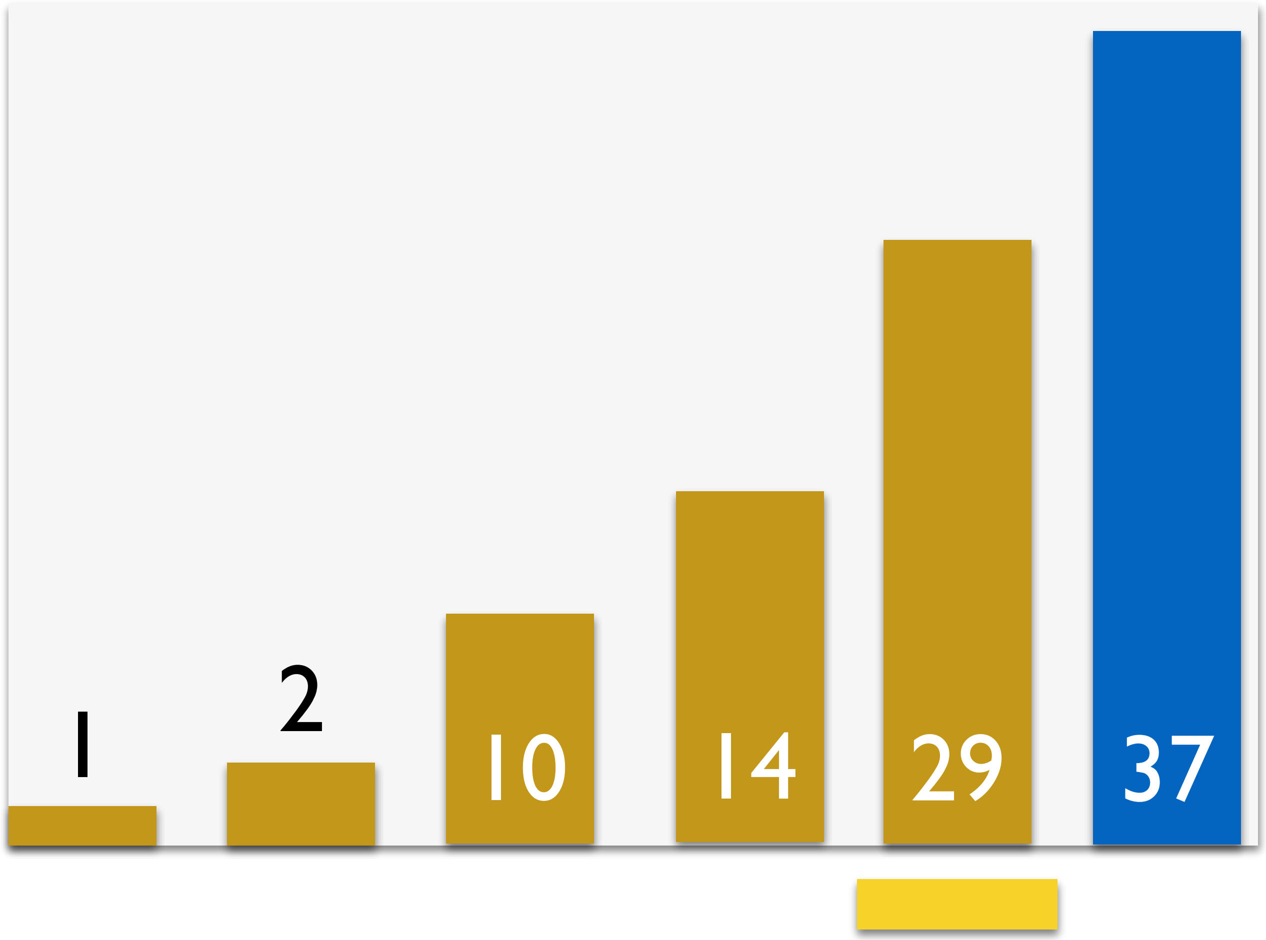
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



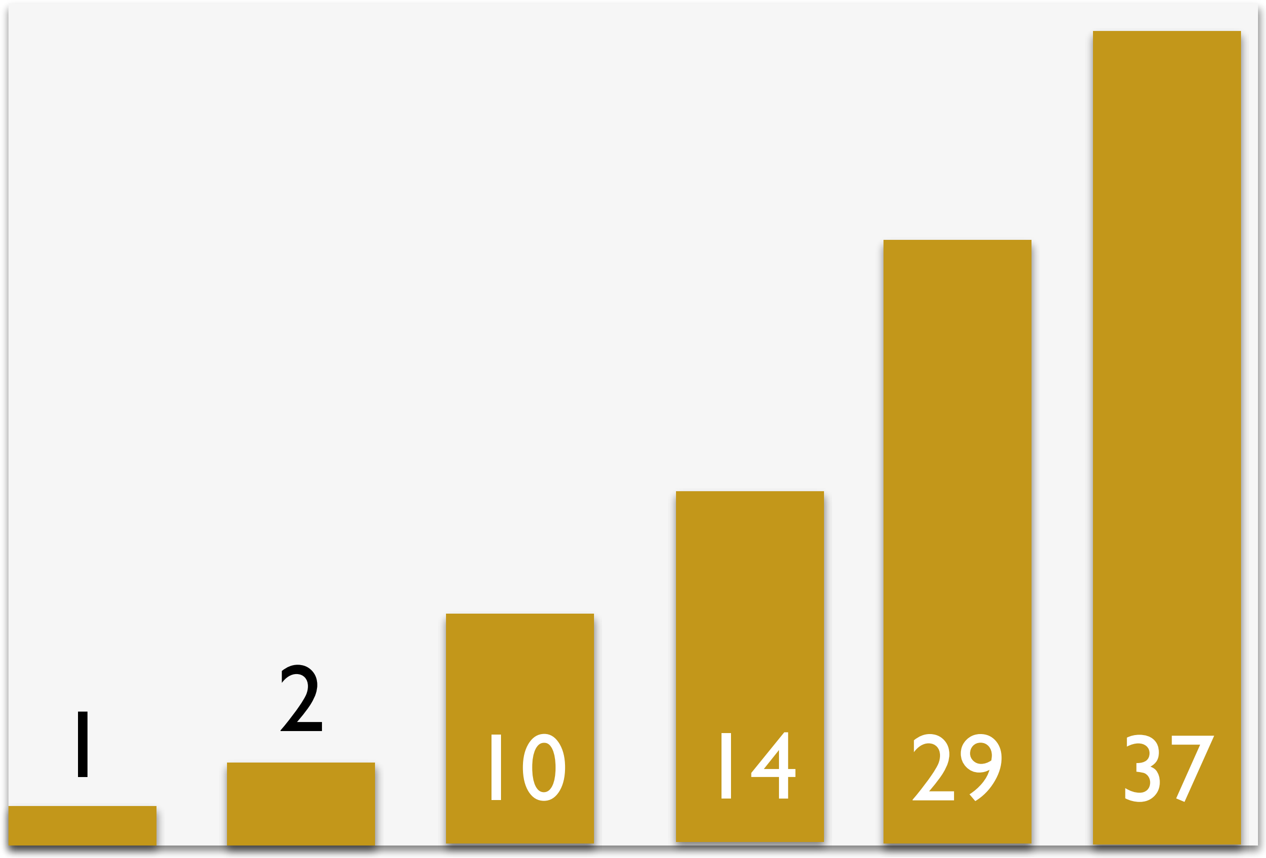
Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



Selection Sort

- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



And now we're finally done!

Selection Sort

- Generalize: For each index i in the list `lst`, we need to find the **min** item in `lst[i:]` so we can replace `lst[i]` with that item
- In fact we need to find the position `min_index` of the item that is the minimum in `lst[i:]`
- **Reminder:** how to swap values of variables `a` and `b`?
  - in-line swapping: `a, b = b, a`
- How do we implement this algorithm?

Selection Sort

```
def selection_sort(my_lst):
    """Selection sort of a given mutable sequence my_lst,
    sorts my_lst by mutating it.  Uses selection sort."""


    # find size
    n = len(my_lst)

    # traverse through all elements
    for i in range(n):

        # find min element in the sublist from index i+1 to end

        min_index = get_min_index(my_lst, i)

        # swap min element with current element at i
        my_lst[i], my_lst[min_index] = my_lst[min_index], my_lst[i]
```

You will work on this helper function in Lab 10

Selection Sort

```
def selection_sort(my_lst):
    """Selection sort of a given mutable sequence my_lst,
    sorts my_lst by mutating it.  Uses selection sort."""


    # find size
    n = len(my_lst)

    # traverse through all elements
    for i in range(n):

        # find min element in the sublist from index i+1 to end

        min_index = get_min_index(my_lst, i)

        # swap min element with current element at i
        my_lst[i], my_lst[min_index] = my_lst[min_index], my_lst[i]
```

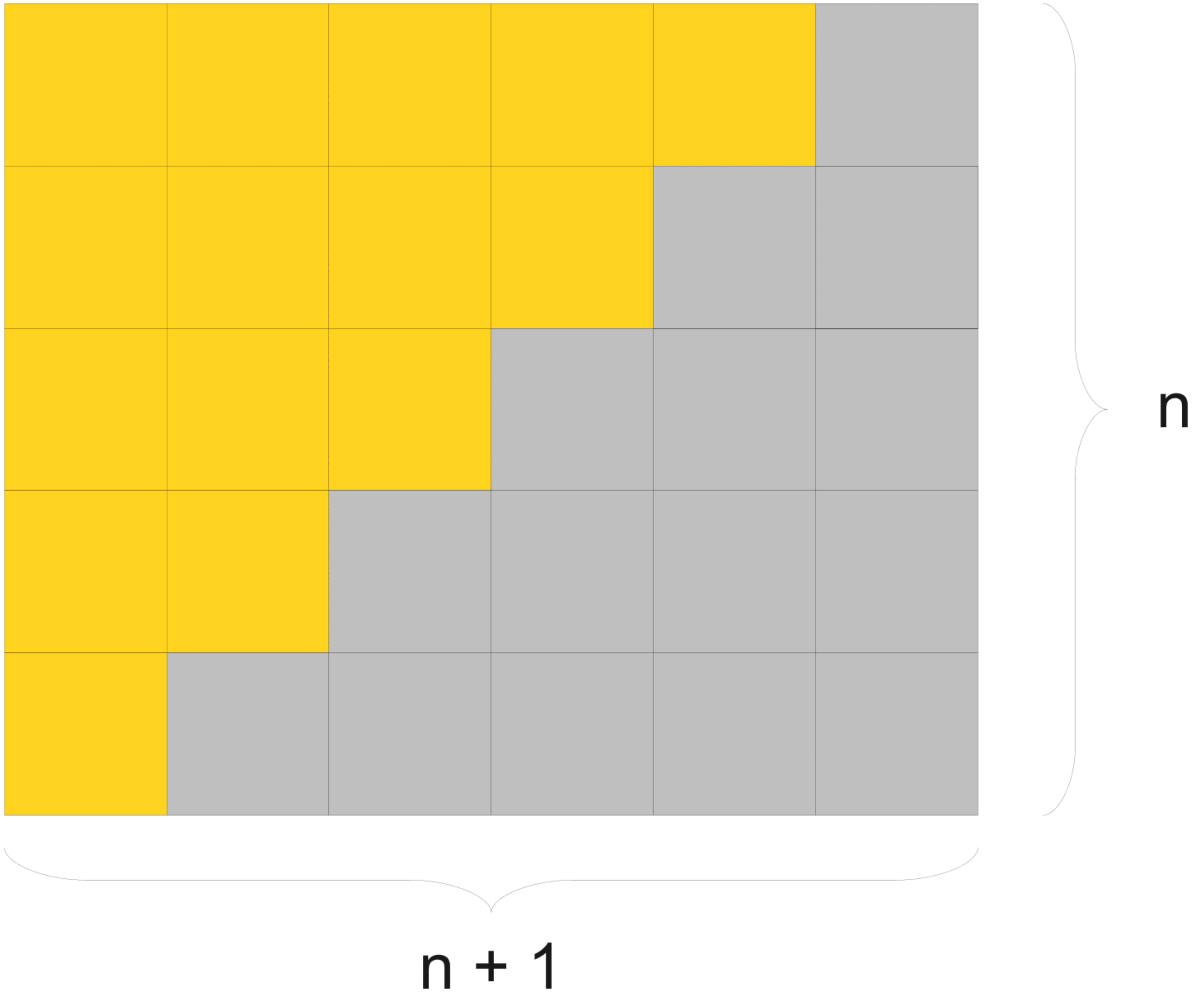
Even without an implementation, can we guess how many steps does this function need to take?

Selection Sort Analysis

- The helper function `get_min_index` must iterate through index `i` to `n` to find the min item
  - When `i = 0` this is `n` steps
  - When `i = 1` this is `n-1` steps
  - When `i = 2` this is `n-2` steps
  - And so on, until `i = n-1` this is `1` step
- Thus overall number of steps is sum of inner loop steps $(n-1)+(n-2)+\cdots+0 \leq n+(n-1)+(n-2)+\cdots+1$
- What is this sum? (You will see this in MATH 200 if you take it.)

Selection Sort Analysis: Visual

$$n + (n-1) + \ldots + 2 + 1 = n(n+1) / 2$$



Selection Sort Analysis: Algebraic

$$S = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1$$

$$+ \quad S = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$$

$$2S = (n+1) + (n+1) + \cdots + (n+1) + (n+1) + (n+1)$$

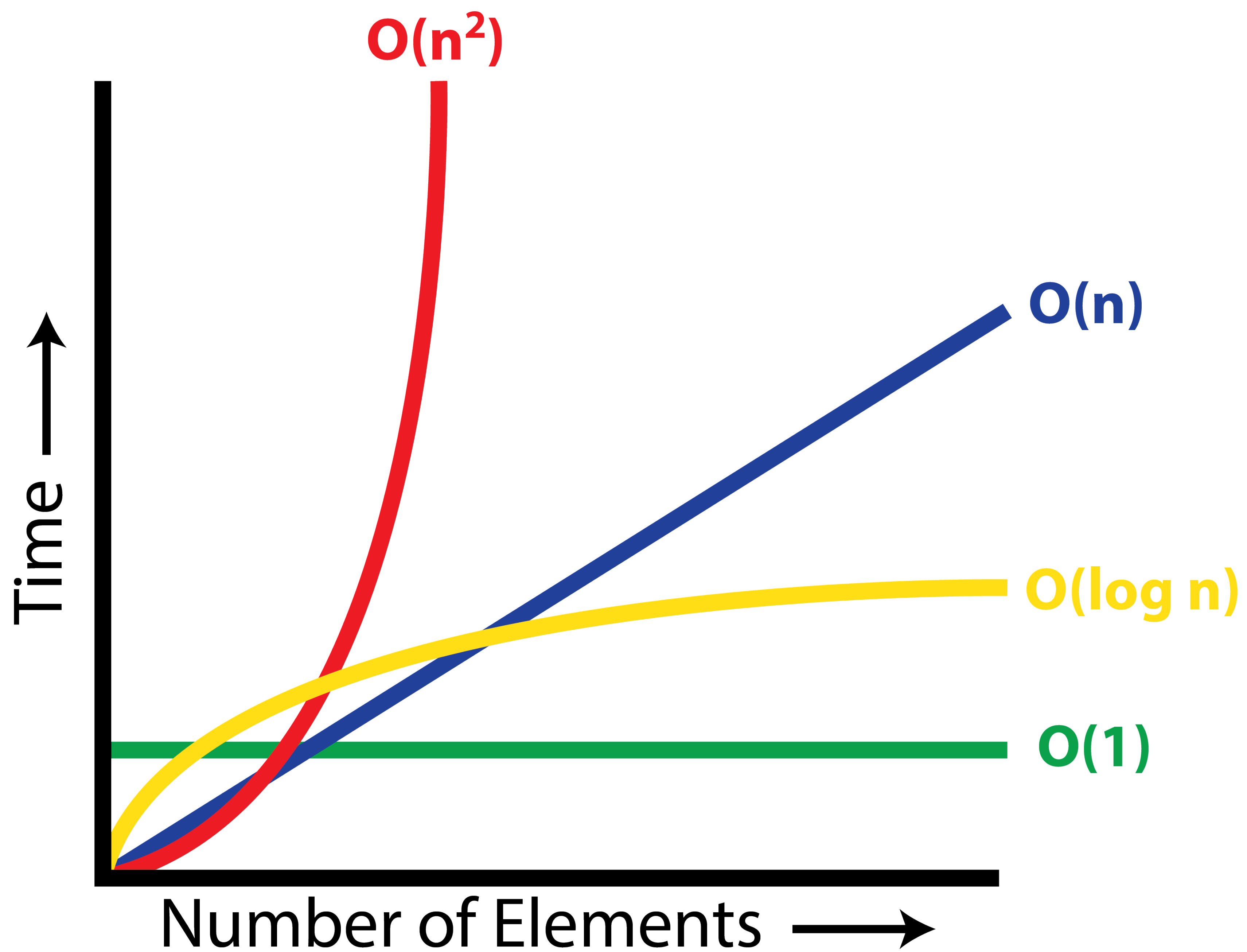
$$2S = (n+1) \cdot n$$

$$S = (n+1) \cdot n \cdot 1/2$$

- Total number of steps taken by selection sort is thus:
  - $O(n(n+1)/2) \ = O(n(n+1)) \ = O(n^2 + n) \ = O(n^2)$

How Fast Is Selection Sort?

- Selection sort takes approximately n^2 steps!



More Efficient Sorting:

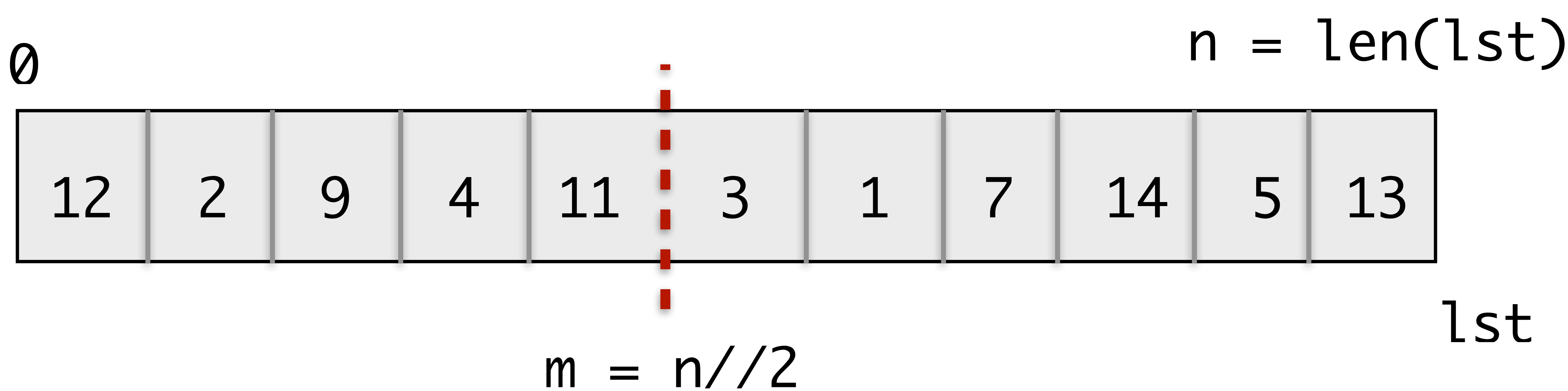
Merge Sort

Towards an $O(n \log n)$ Algorithm

- There are other sorting algorithms that compare and rearrange elements in different ways, but are still $O(n^2)$ steps
    - Any algorithm that takes n steps to move each item n positions (in the worst case) will take at least $O(n^2)$ steps
    - To do better than n^2, we need to move an item in fewer than n steps
- We can sort in $O(n \log n)$ time if we are clever: ***Merge sort algorithm*** (Invented by John von Neumann in 1945)

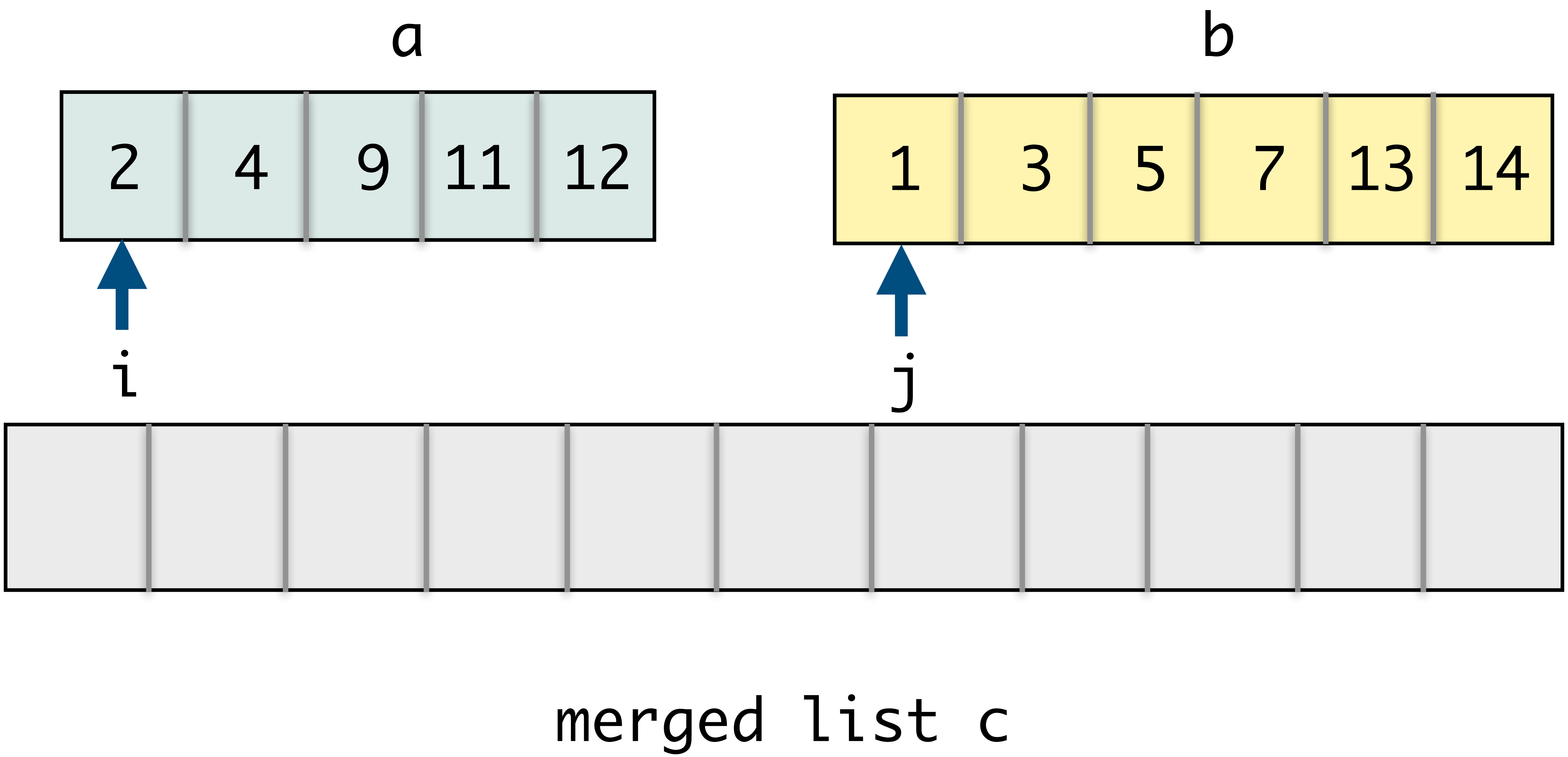
Merge Sort: Basic Idea

- If we split the list in half, sorting the left and right half are smaller versions of the same problem
- **Algorithm:**
  - **(Divide)** Recursively sort left and right half ($O(\log n)$)
  - **(Unite)** Merge the sorted halves into a single sorted list ($O(n)$)



Merging Sorted Lists

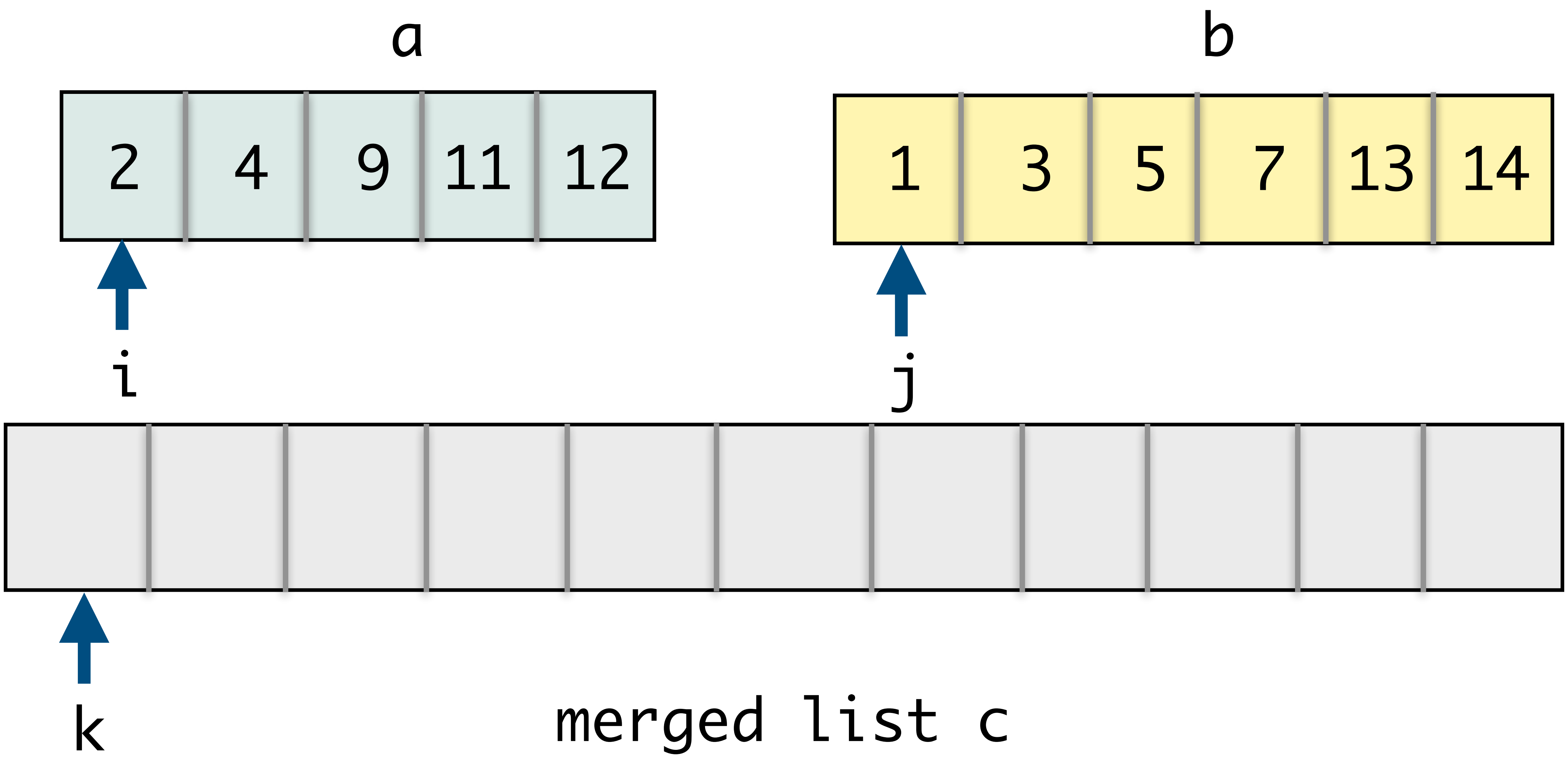
- **Problem.** Given two sorted lists `a` and `b`, how quickly can we merge them into a single sorted list?



Merging Sorted Lists

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

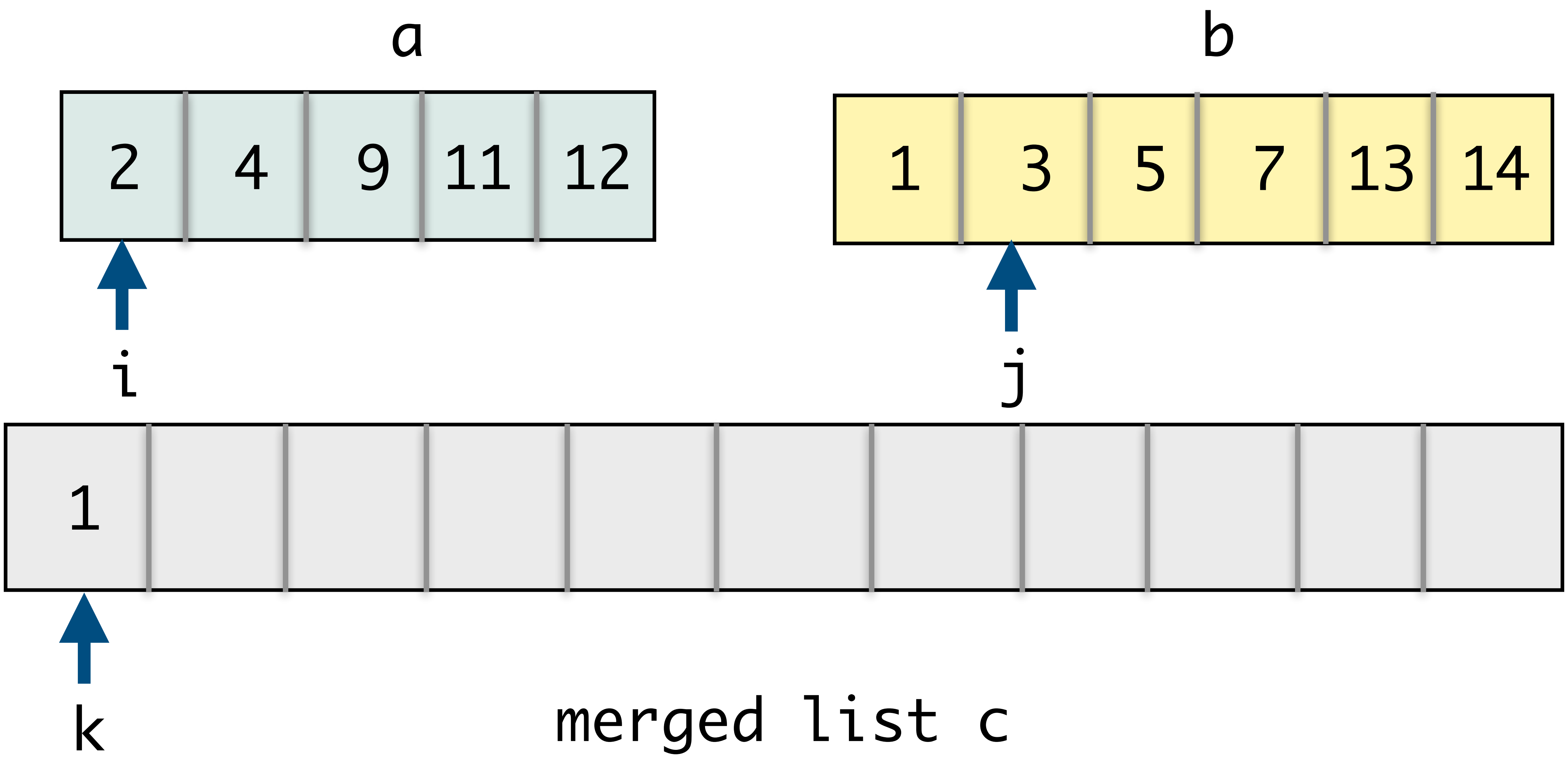
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

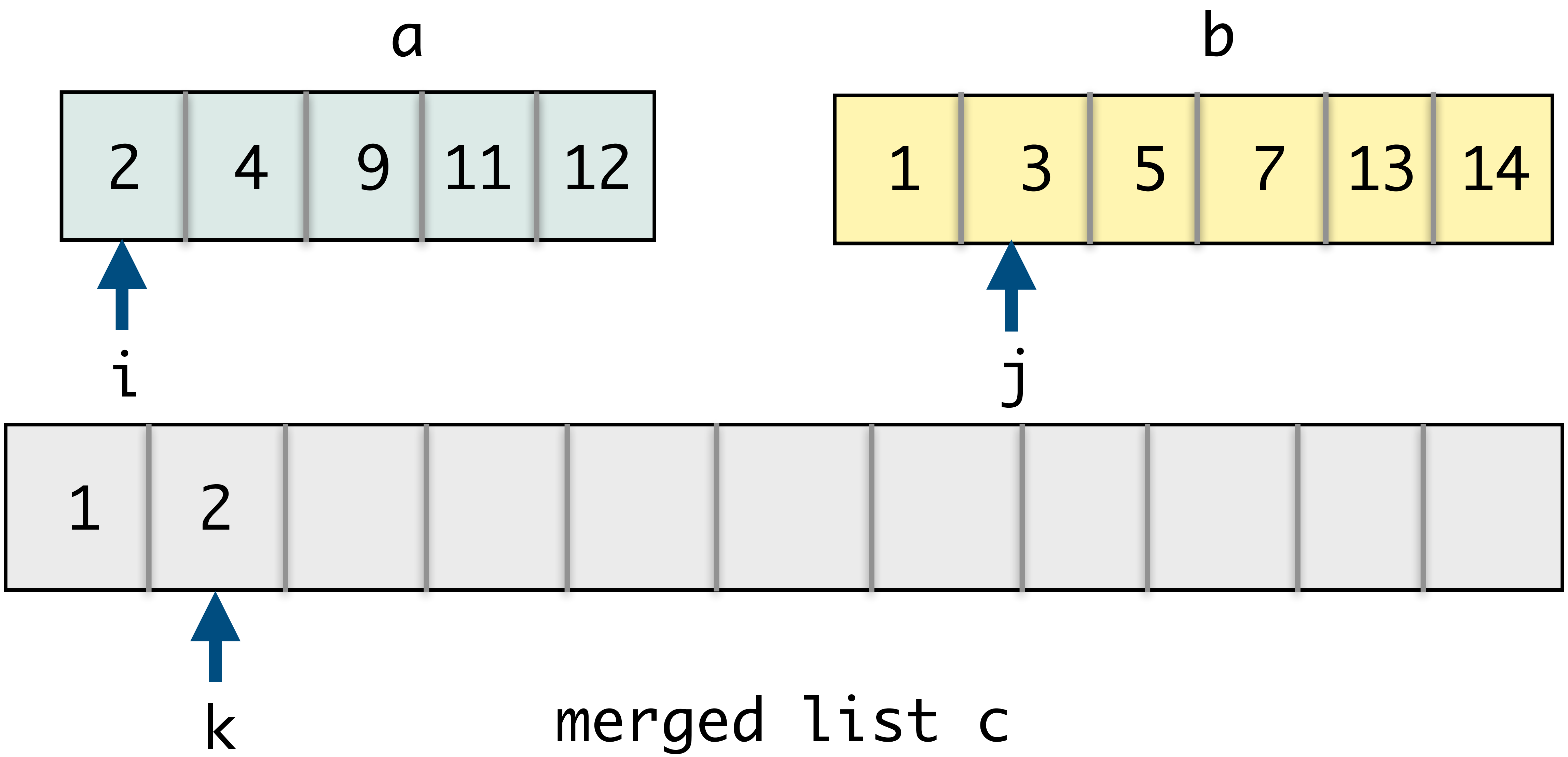
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

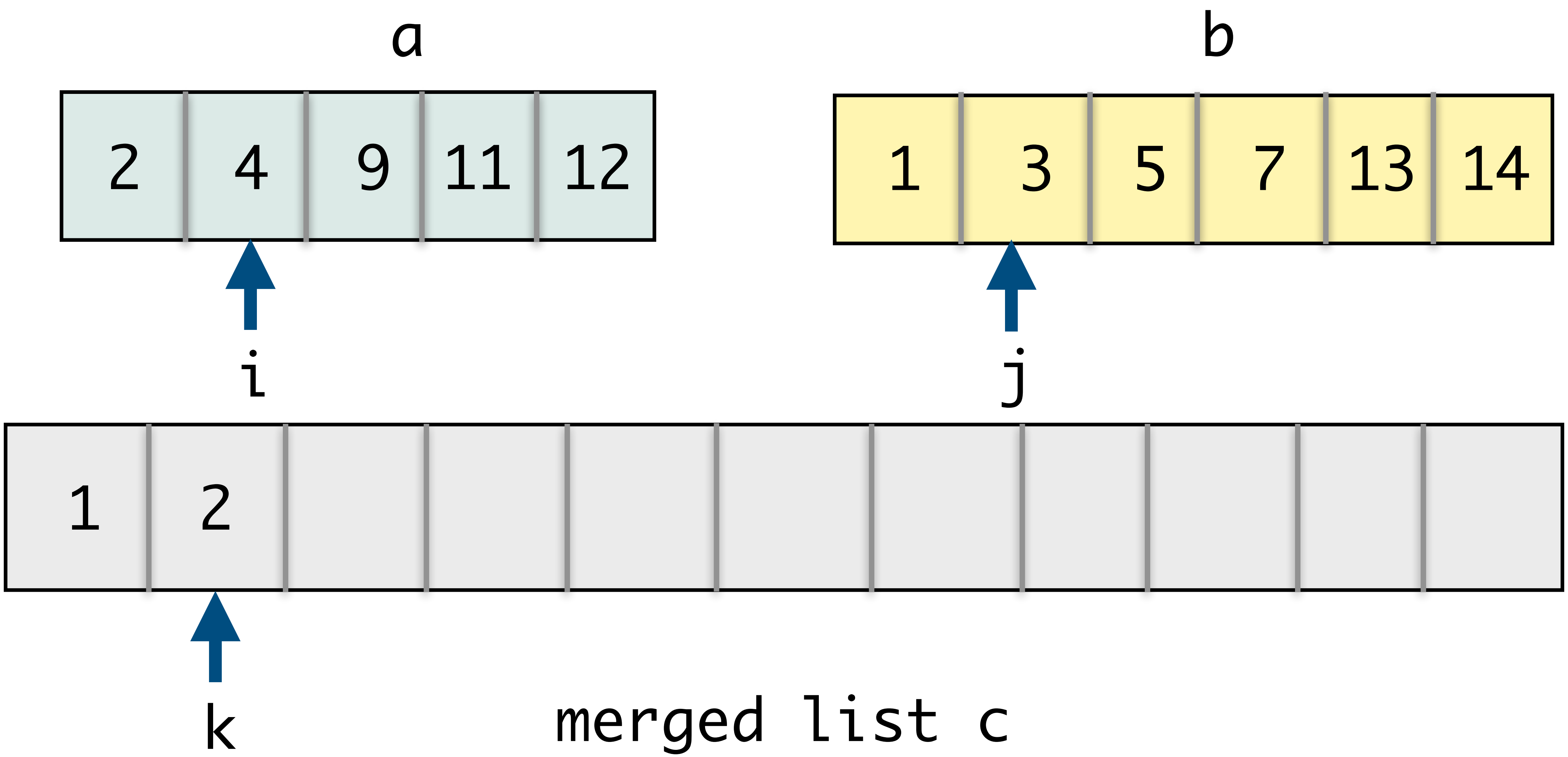
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

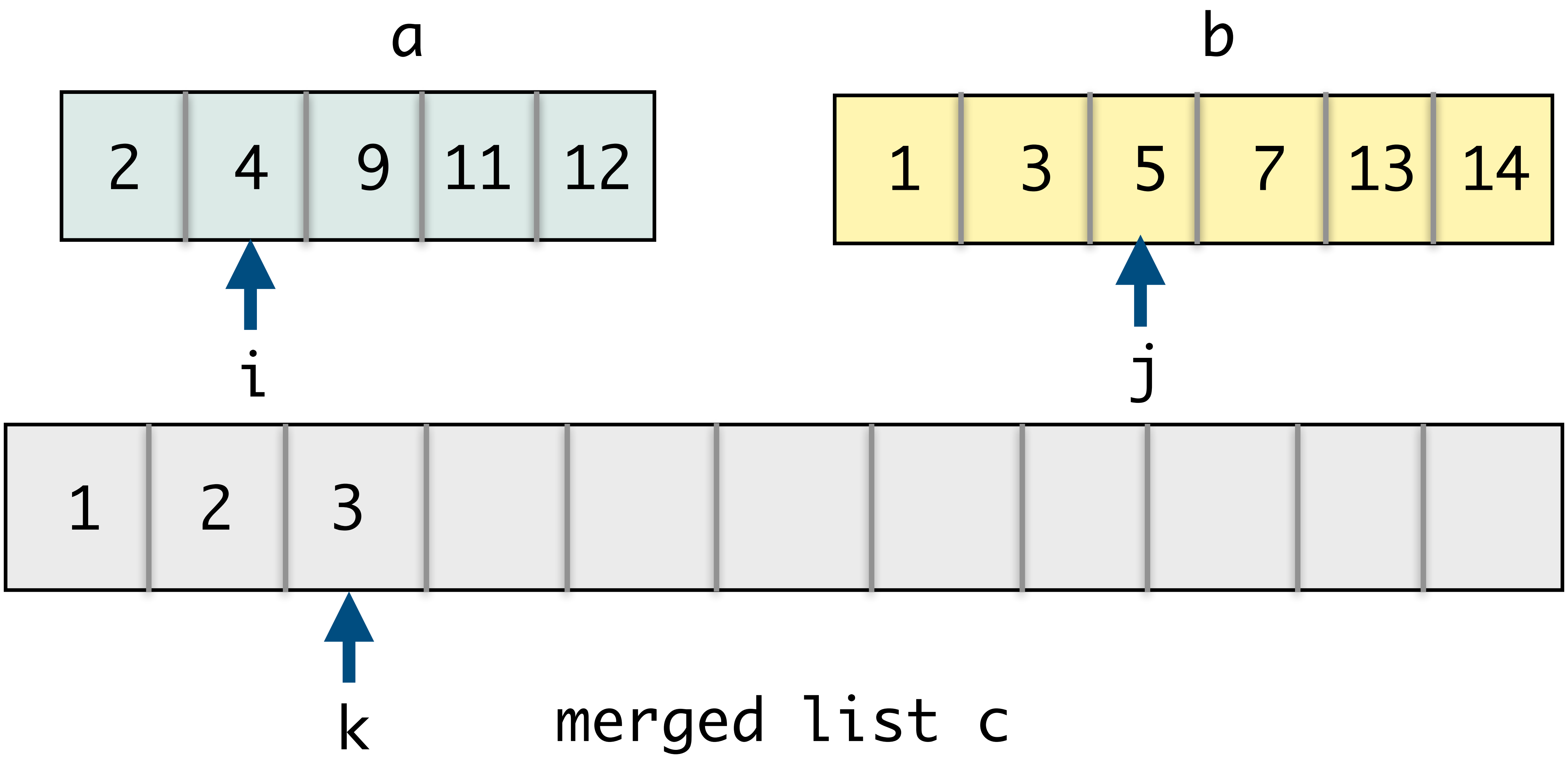
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

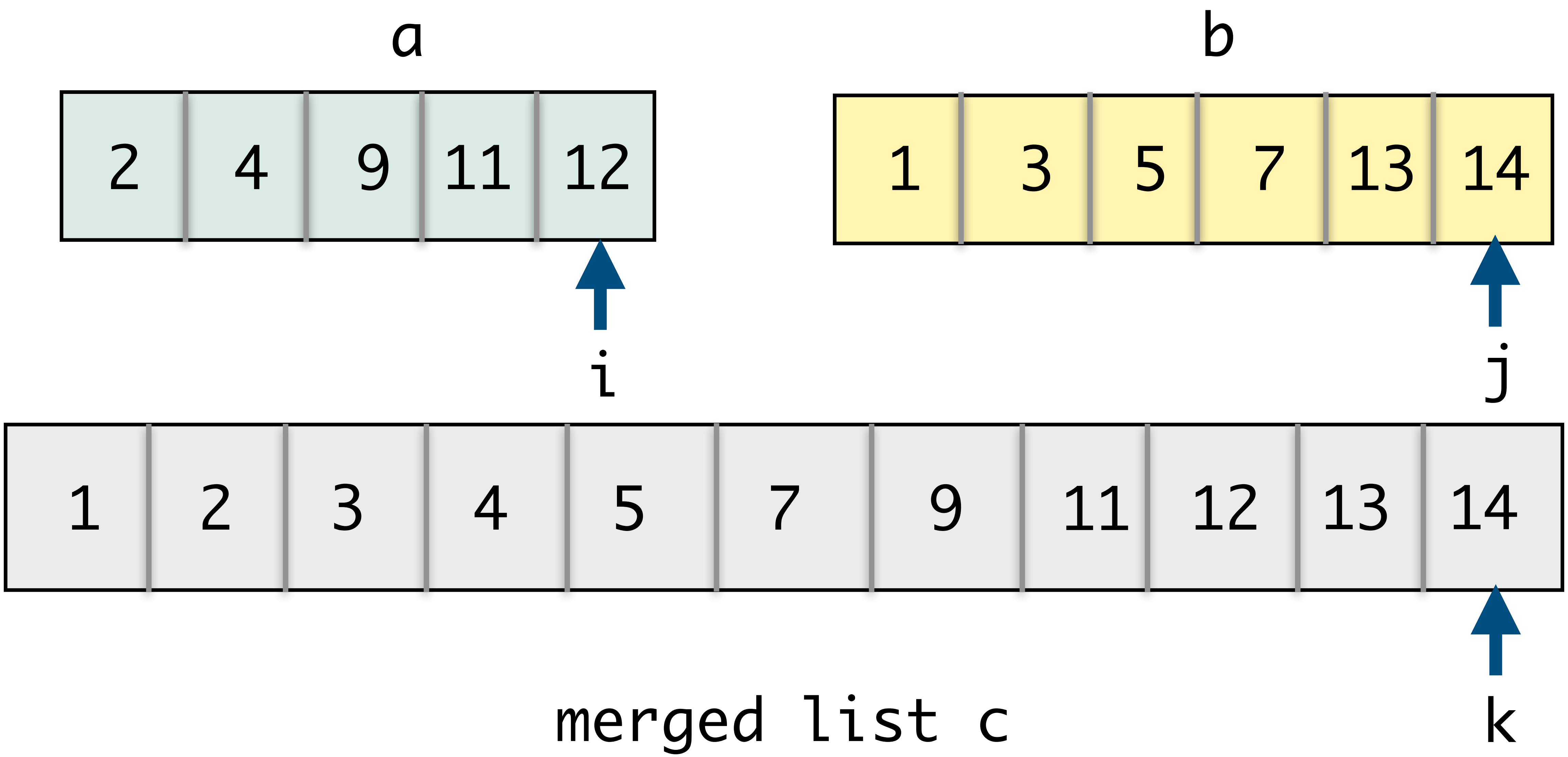
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

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



Merging Sorted Lists

- 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 $c[k]$
- Merging two sorted lists into one is an $O(n)$ step algorithm!
- Can use this merge procedure to design our recursive merge sort algorithm!

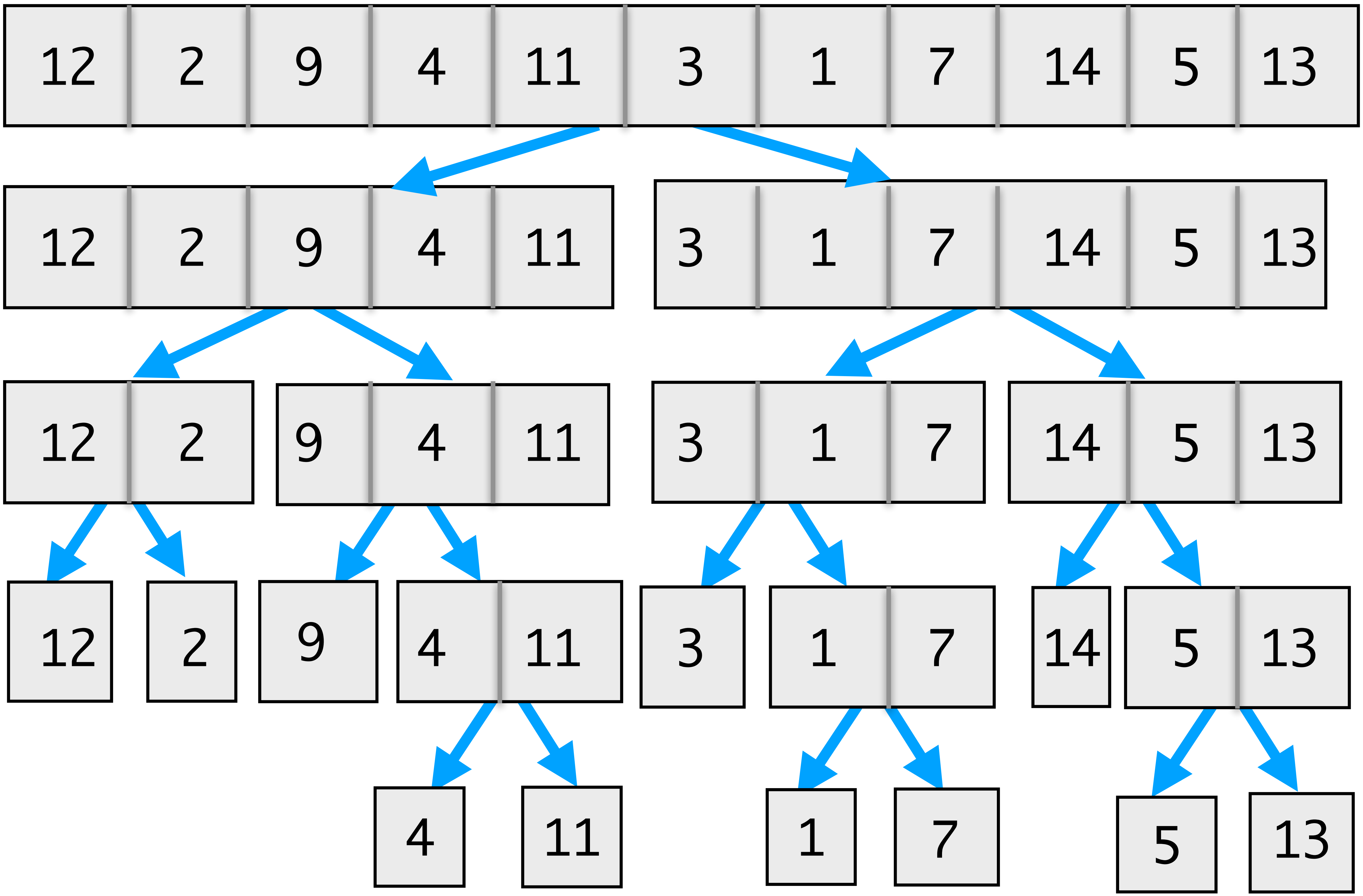
```python
def merge(a, b):
    """Merges two sorted lists a and b,
    and returns new merged list c"""
    # initialize variables
    i, j, k = 0, 0, 0
    len_a, len_b = len(a), len(b)
    c = []
    # traverse and populate new list
    while i < len_a and j < len_b:

        if a[i] <= b[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1

    # handle remaining values
    if i < len_a:
        c.extend(a[i:])

    elif j < len_b:
        c.extend(b[j:])

    return c
```

Merge Sort Algorithm

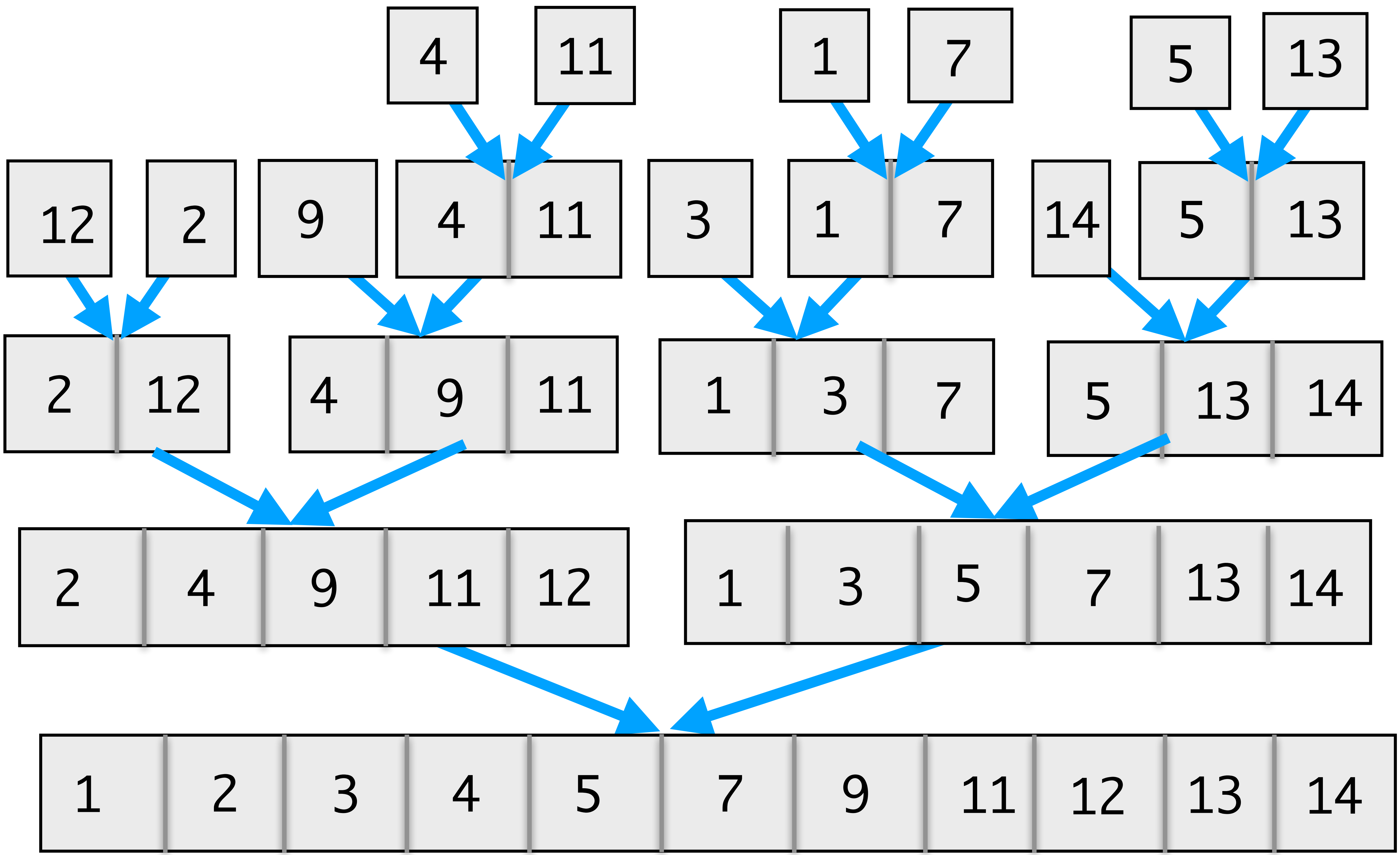
- **Base case:** If list is empty or contains a single element: it is already sorted
- **Recursive case:**
  - Recursively sort left and right halves
  - Merge the sorted lists into a single list and return it
- **Question:**
  - Where is the ***sorting*** actually taking place?

```python
def merge_sort(lst):
    """Given a list lst, returns
    a new list that is lst sorted
    in ascending order."""
    n = len(lst)

    # base case
    if n == 0 or n == 1:
        return lst

    else:
        m = n//2 # middle

        # recurse on left & right half
        sort_lt = merge_sort(lst[:m])
        sort_rt = merge_sort(lst[m:])

        # return merged list
        return merge(sort_lt, sort_rt)
```

Merge Sort Example

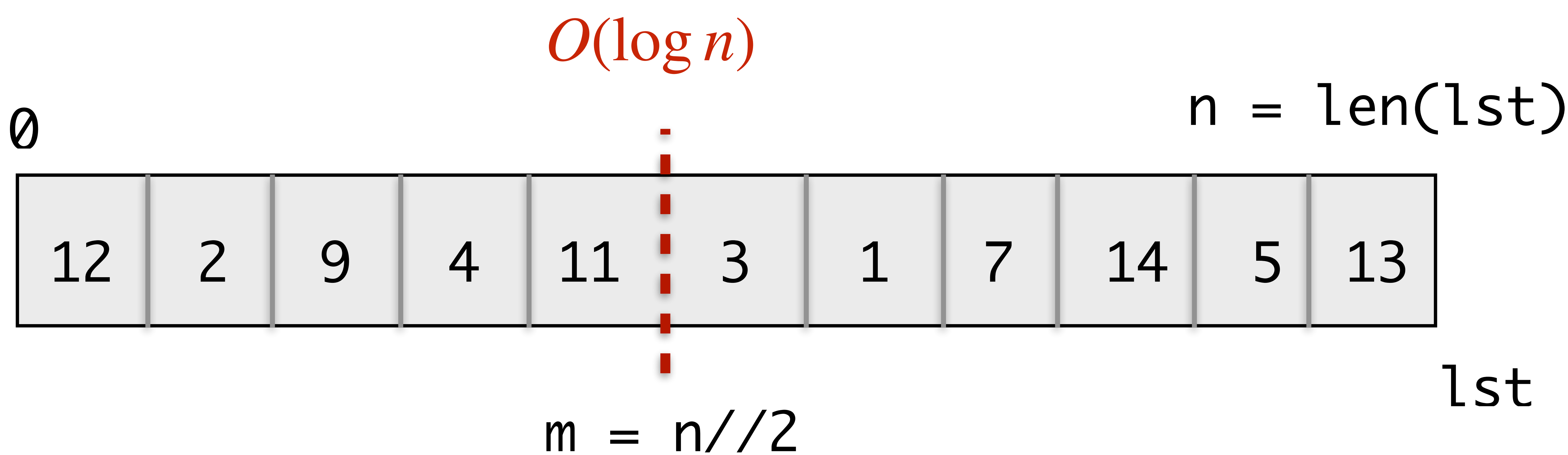


Merge Sort Example



Merge Sort: Basic Idea

- If we split the list in half, sorting the left and right half are smaller versions of the same problem
- **Algorithm:**
  - **(Divide)** Recursively sort left and right half ($O(\log n)$)
  - **(Unite)** Merge the sorted halves into a single sorted list ($O(n)$)



Big Oh Comparisons

- Selection sort: $O(n^2)$
- Merge sort: $O(n \log n)$