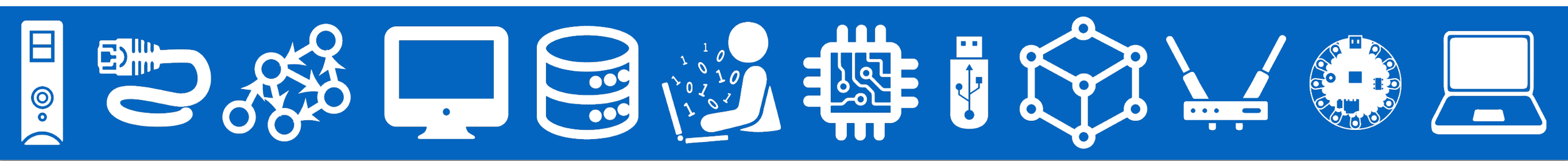
CS134 Lecture:
Sorting



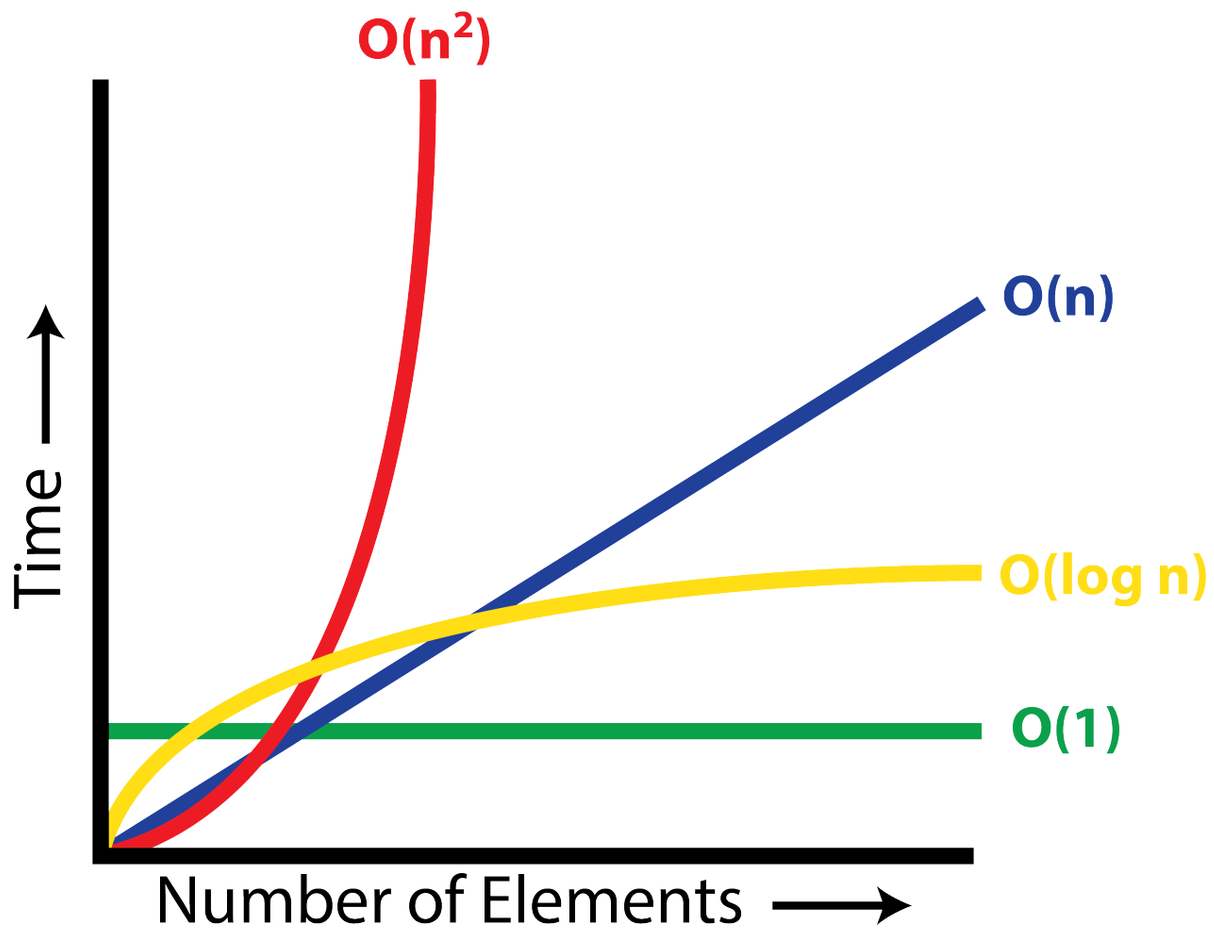
Announcements & Logistics

- **HW 11**: due Monday
- No Lab next week (Enjoy Thanksgiving break!!!)
  - Practice Final Exam w/ sample solutions will be available
- CS134 Scheduled Final: **Wednesday, December11, 9:30 AM**
  - Room: **Wachenheim B11**

Do You Have Any Questions?

Last Time: Efficiency and Search

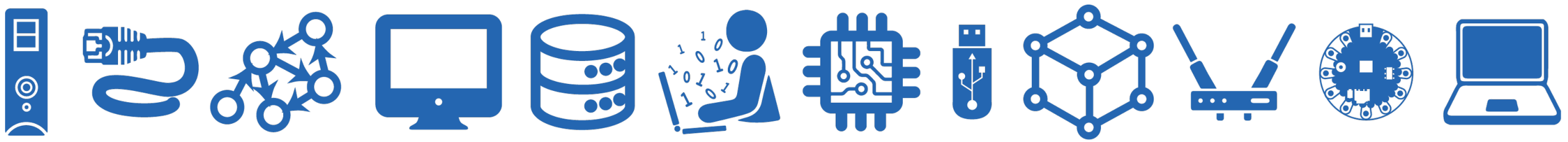
- Explored searching and Big-O notation
  - What common patterns lead to which Big-O performance?



Today: Searching (and Sorting)

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

Sorting

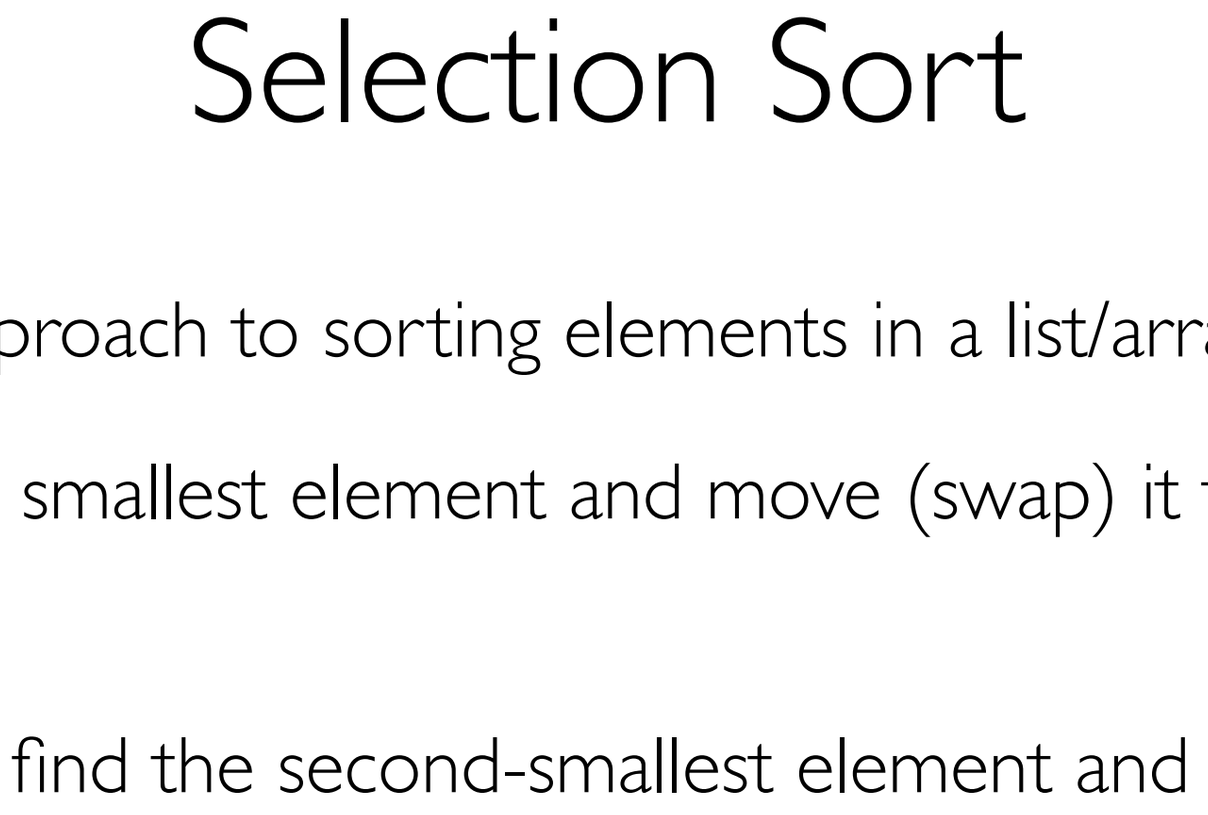


Sorting

- **Problem:** Given a sequence of unordered elements, we want 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 (clone)
    - `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!

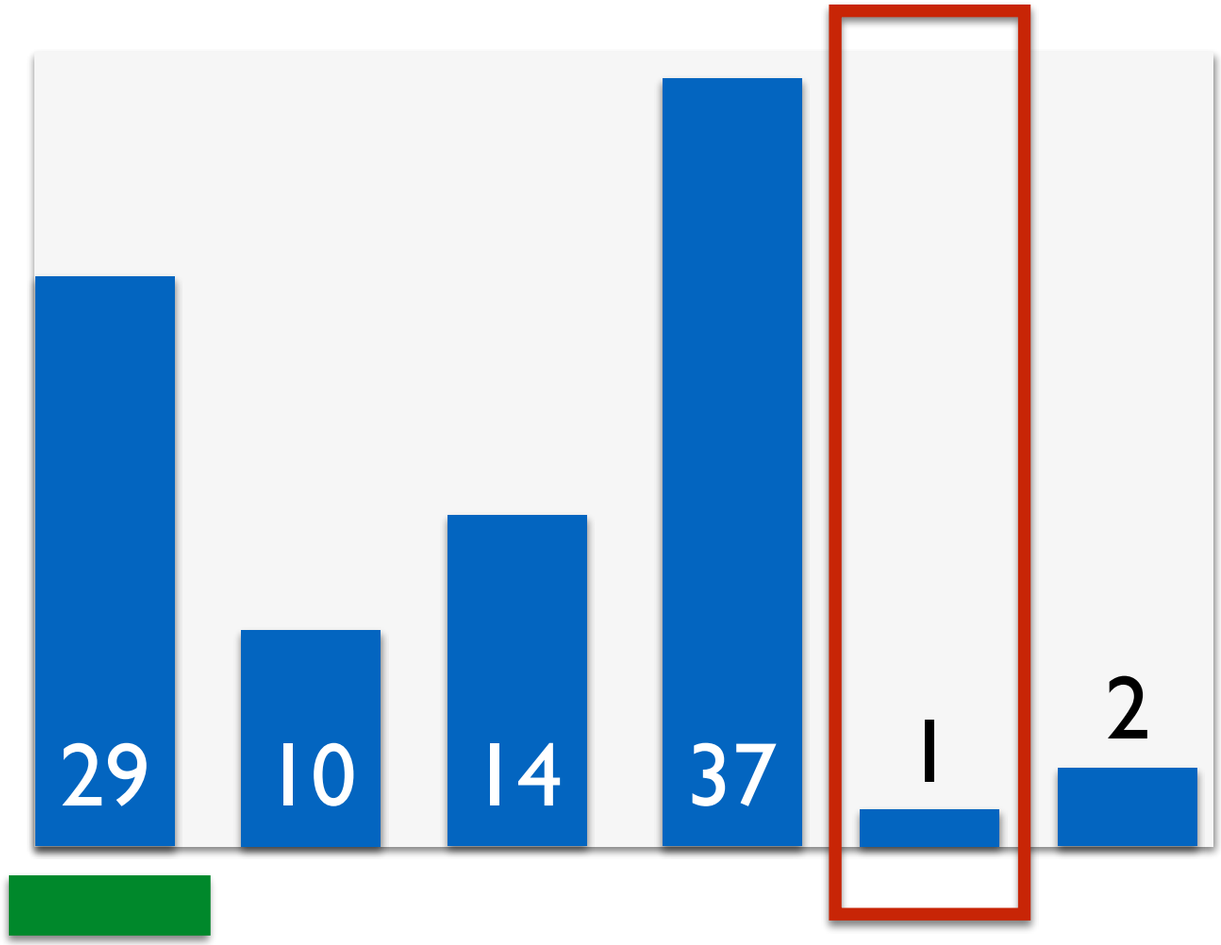
Selection Sort

- 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



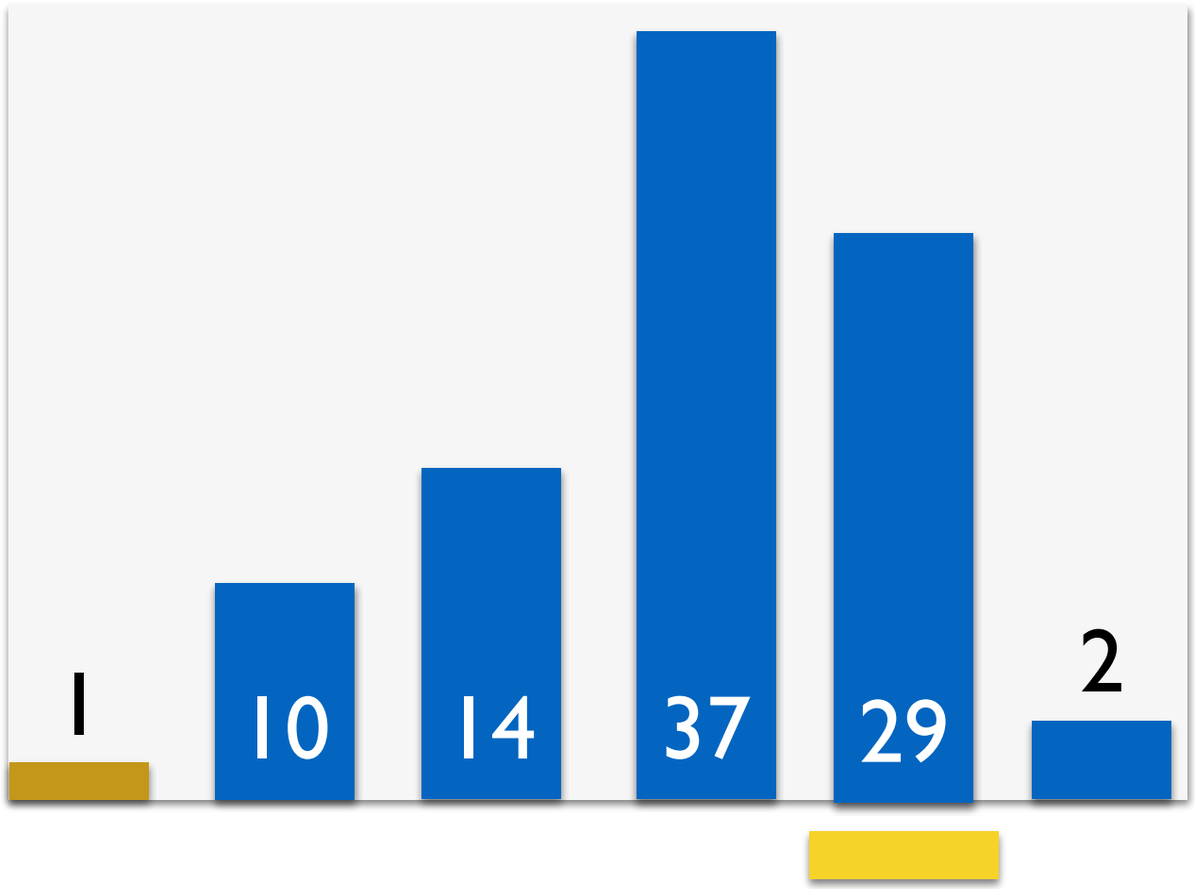
Selection Sort

- 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



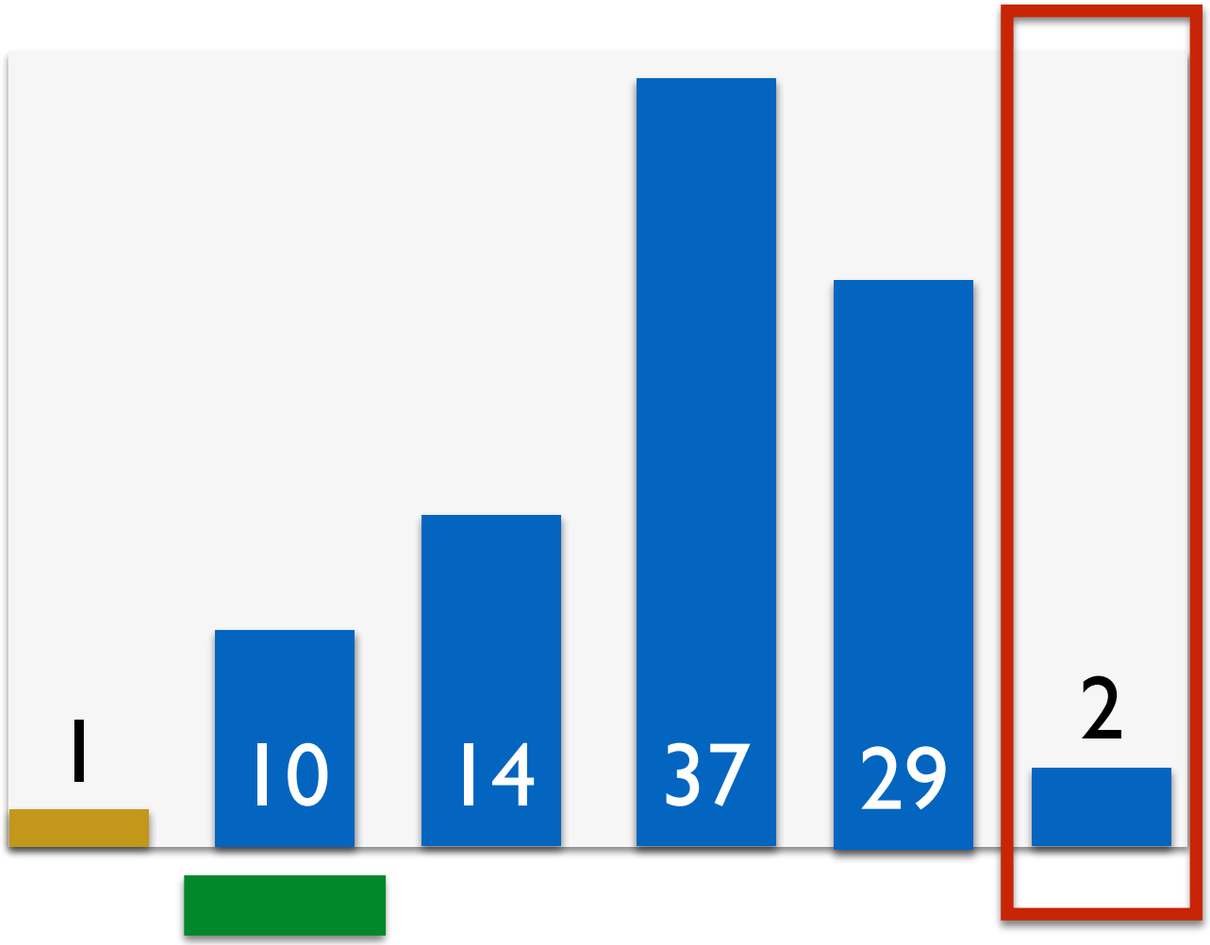
Selection Sort

- 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



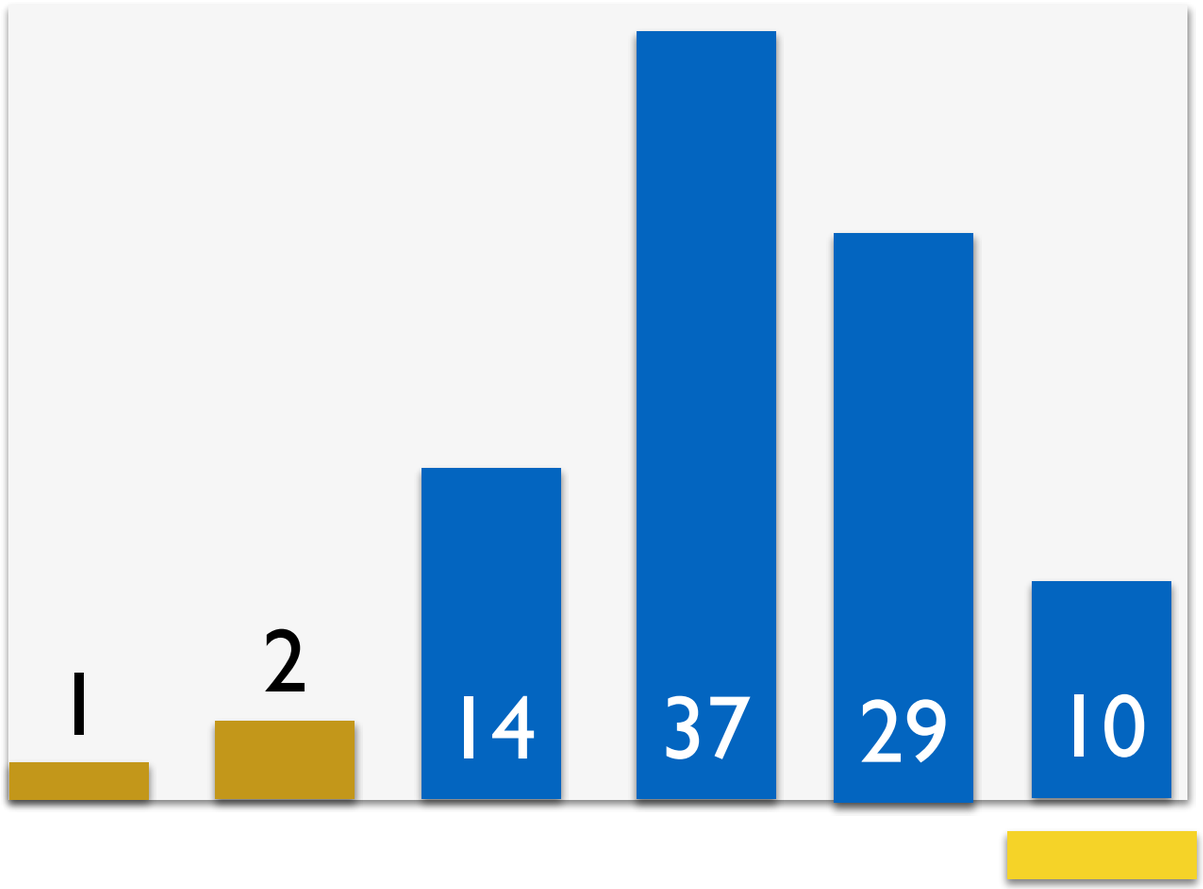
Selection Sort

- 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



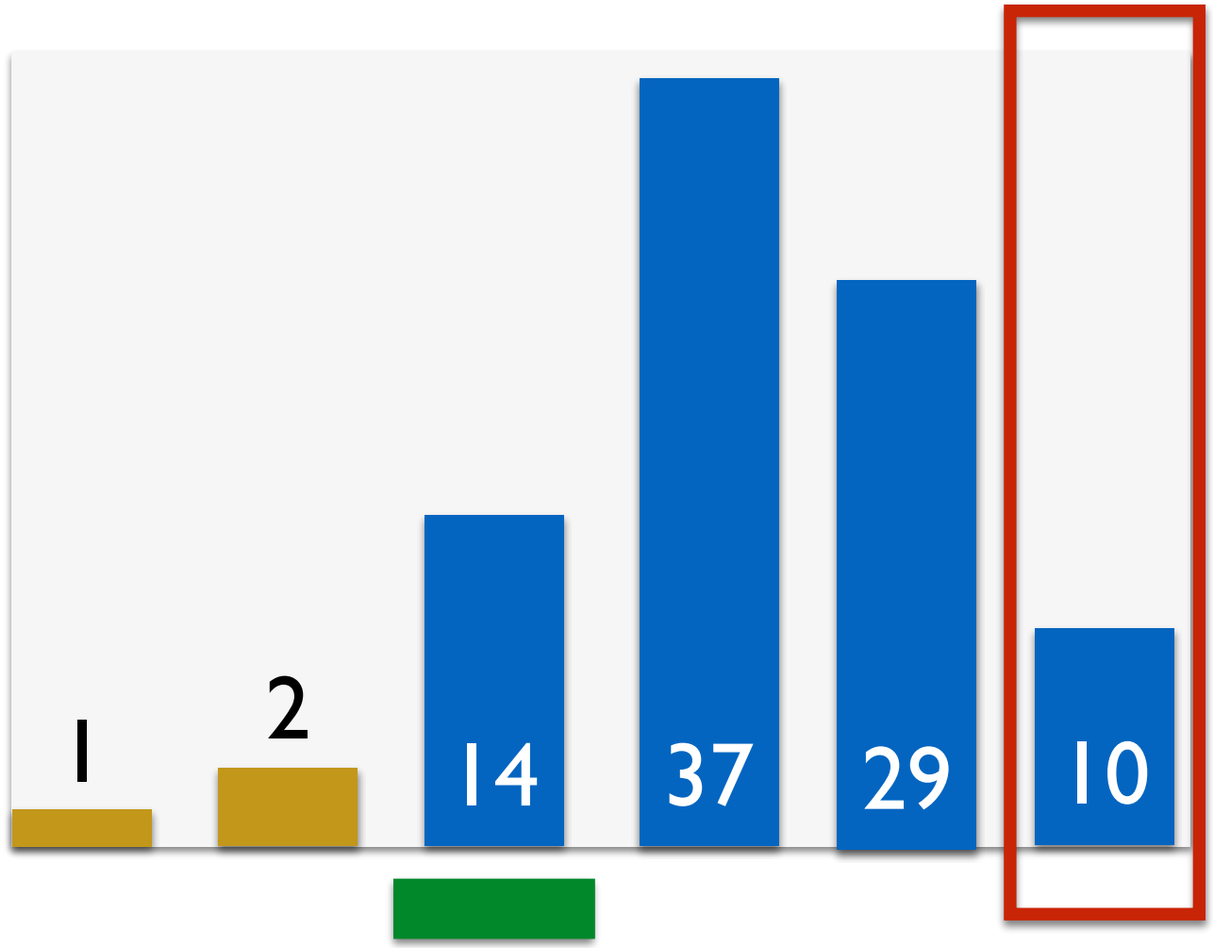
Selection Sort

- 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.



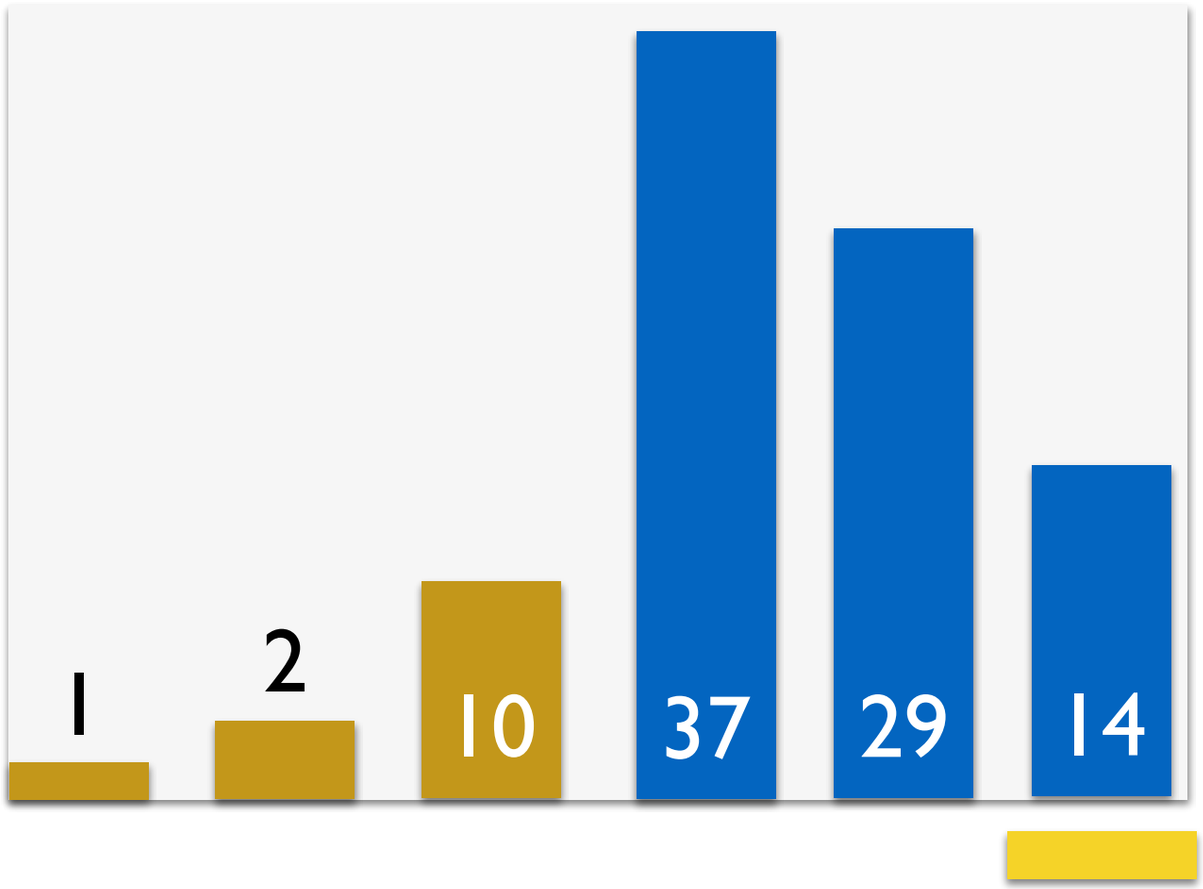
Selection Sort

- 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.



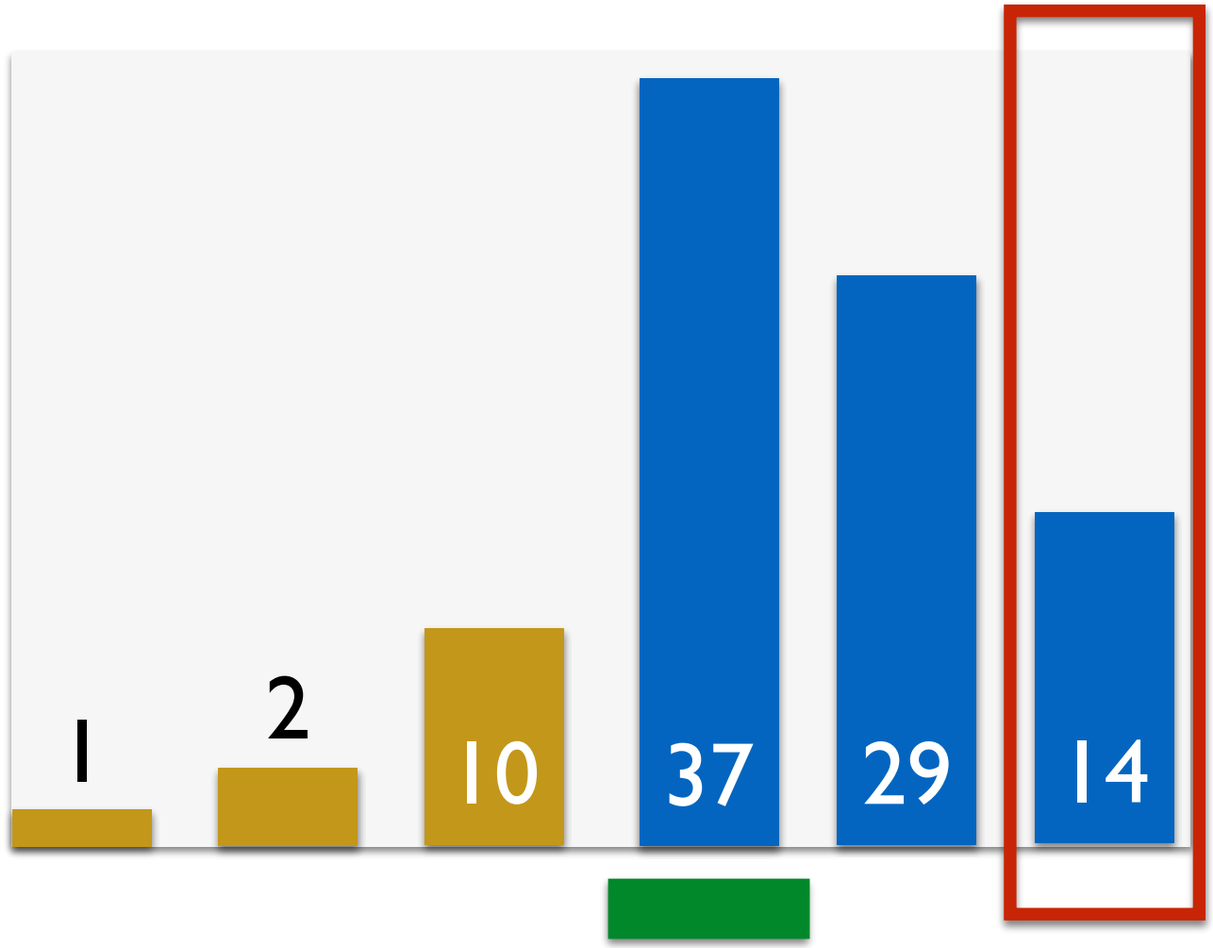
Selection Sort

- 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.



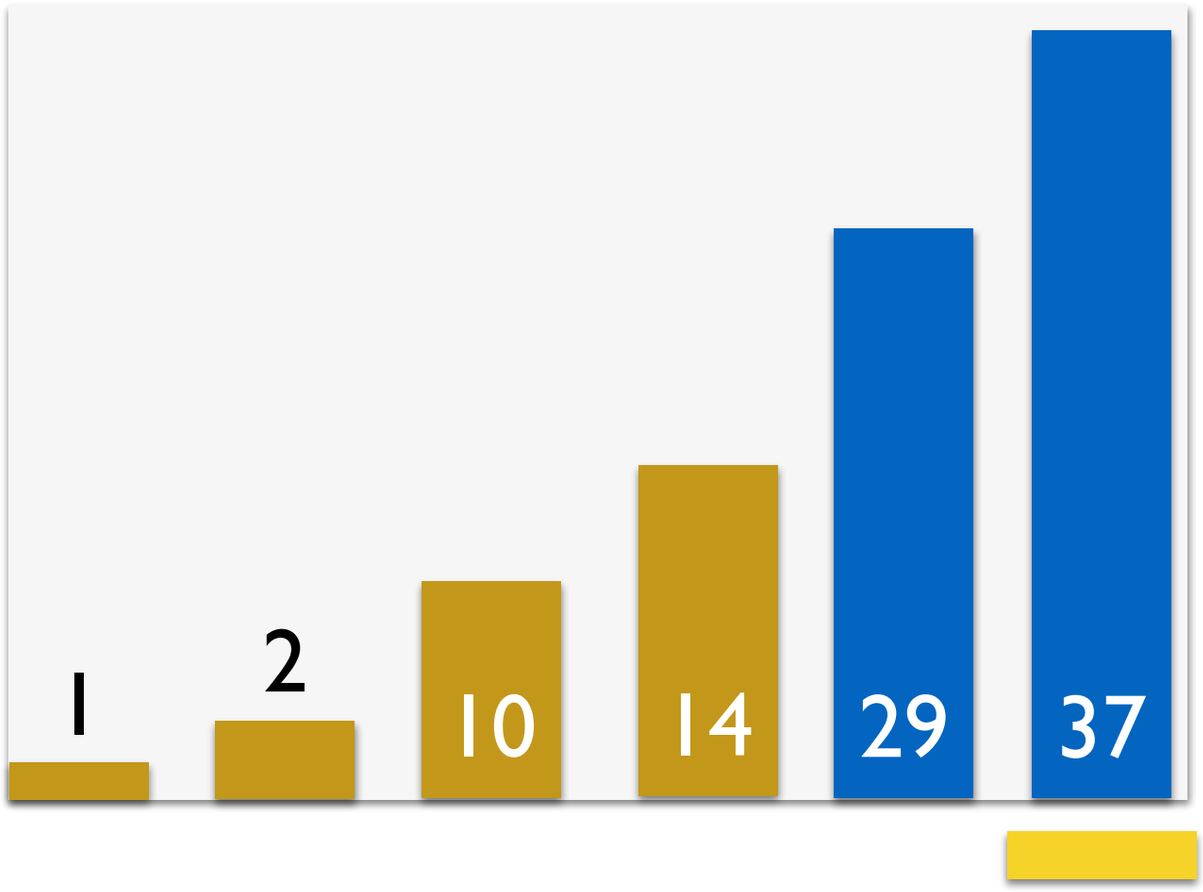
Selection Sort

- 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.



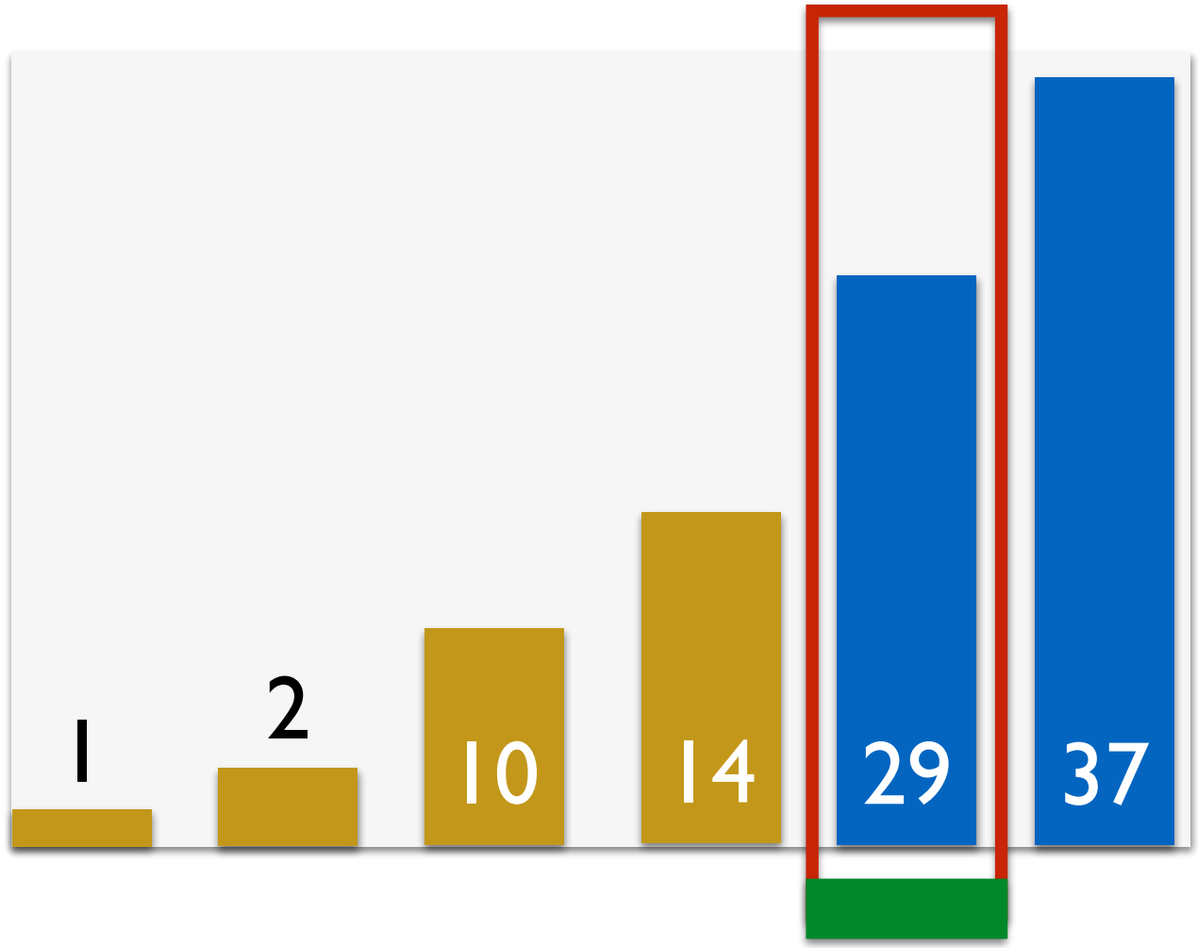
Selection Sort

- 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.



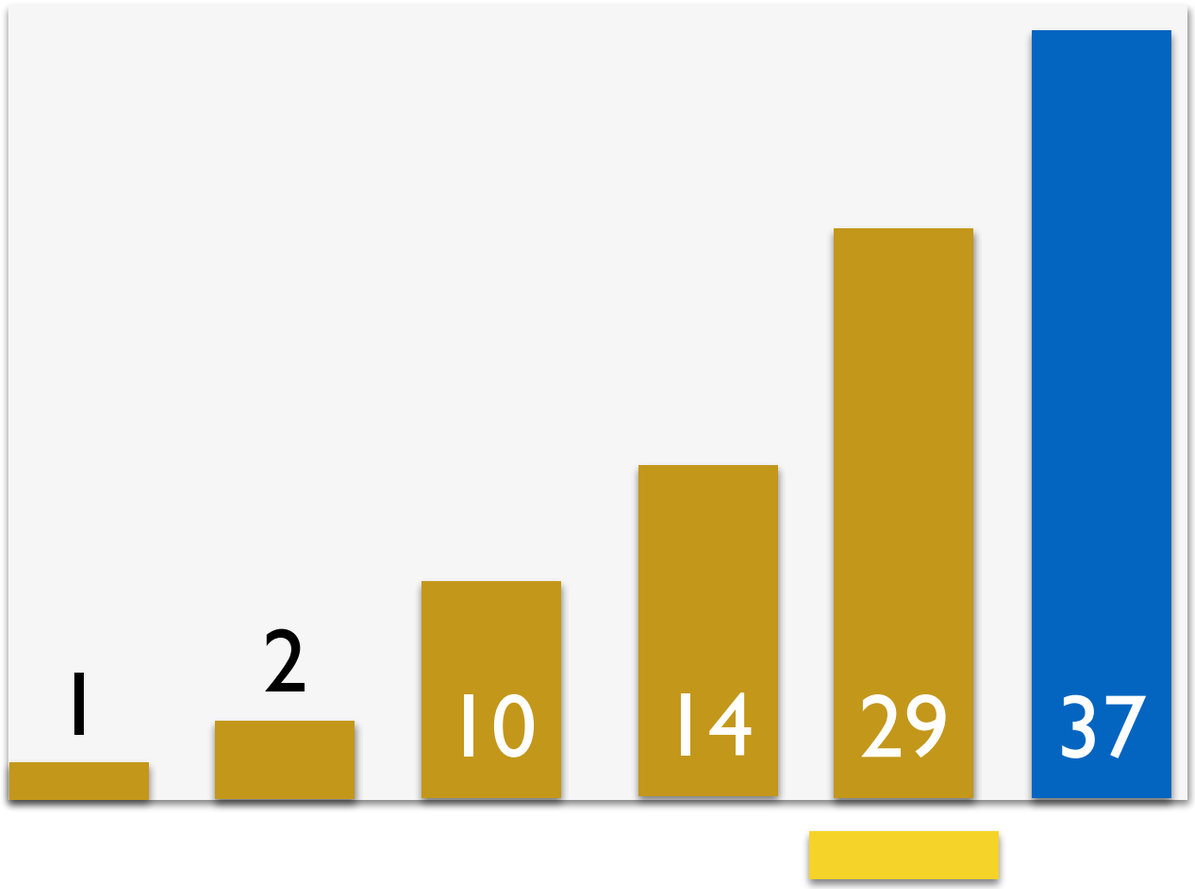
Selection Sort

- 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.



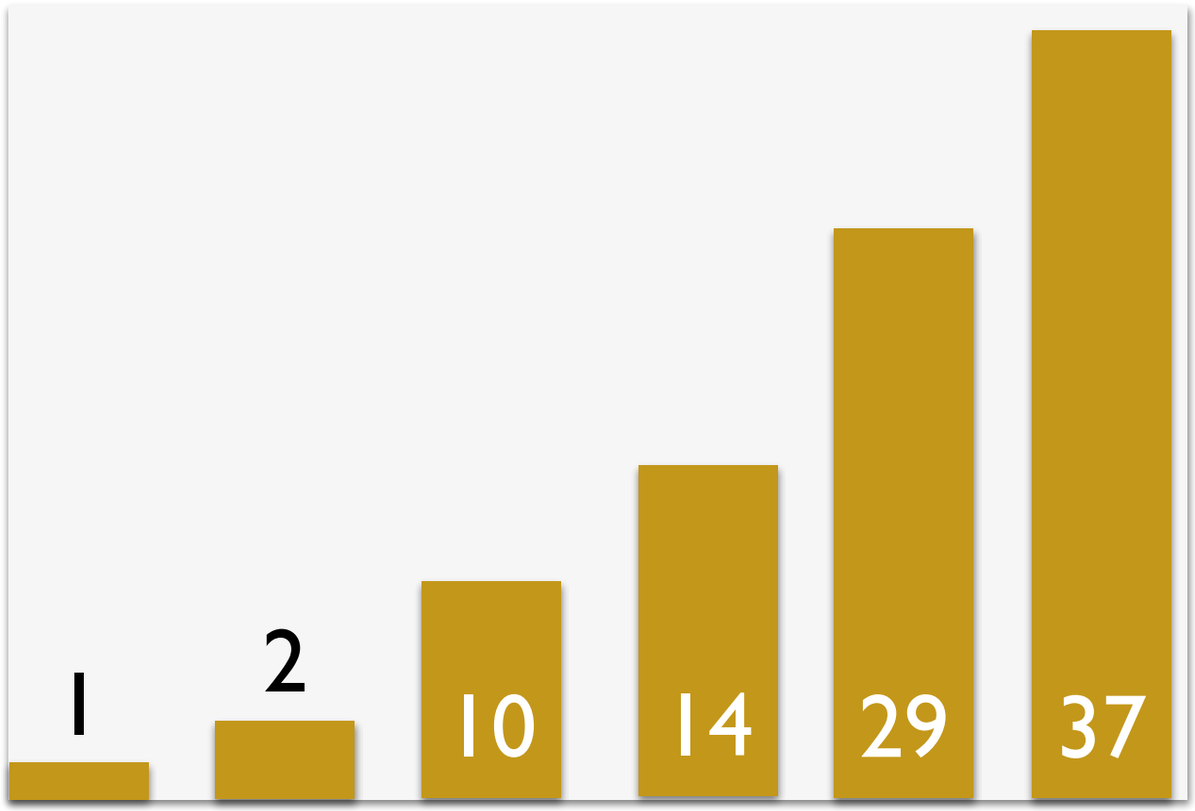
Selection Sort

- 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

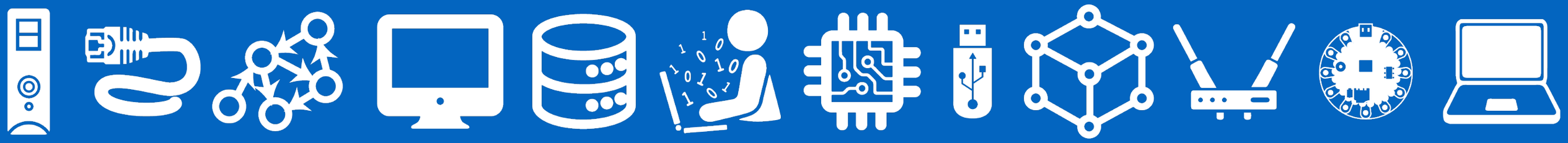
- Find the smallest element and move (swap) it to the first position
- *Repeat*: find the second-smallest element and move it to the second position, and so on
- The gold bars represent the sorted portion of the list.



And now we're finally done!

Selection Sort
Roma Folk Dance

- https://www.youtube.com/watch?v=Ns4TPTC8whw



Selection Sort

- Strategy: 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
    - To do this, we need to find the position `min_index` of the item that is the minimum in `lst[i:]`
- **Neat trick:** how to swap values of variables `a` and `b` in one line?
    - in-line "tuple" swapping: `a, b = b, a`
- What about swapping the values at list index `i` and `j` in one line?
    - `lst[i],list[j] = lst[j],lst[i]`

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 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 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?

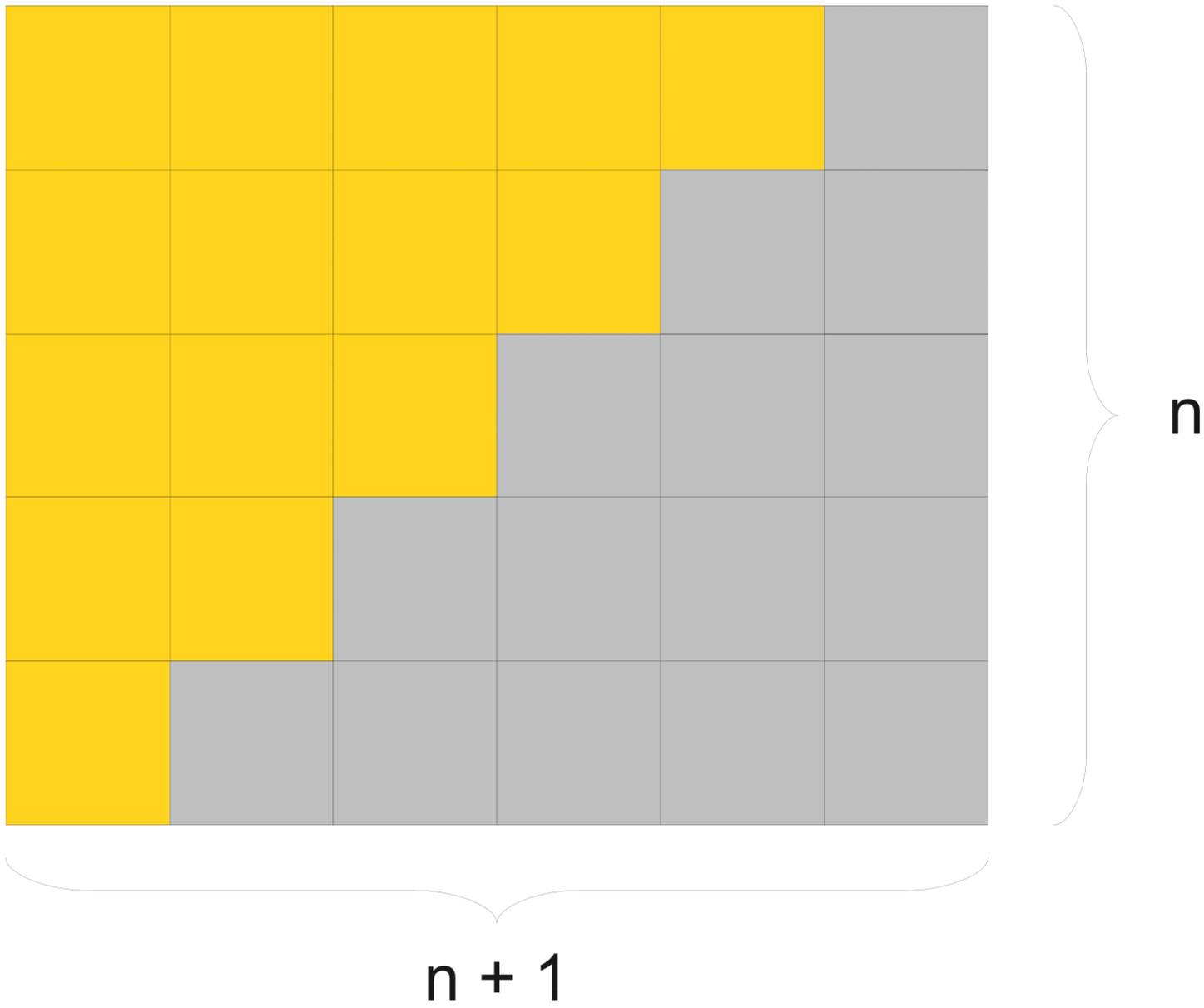
Selection Sort Analysis

- The helper function `get_min_index` must iterate through indexes `i` to `n-1` to find the min item in that range
  - 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.)

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



Selection Sort Analysis: Algebraic

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

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

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

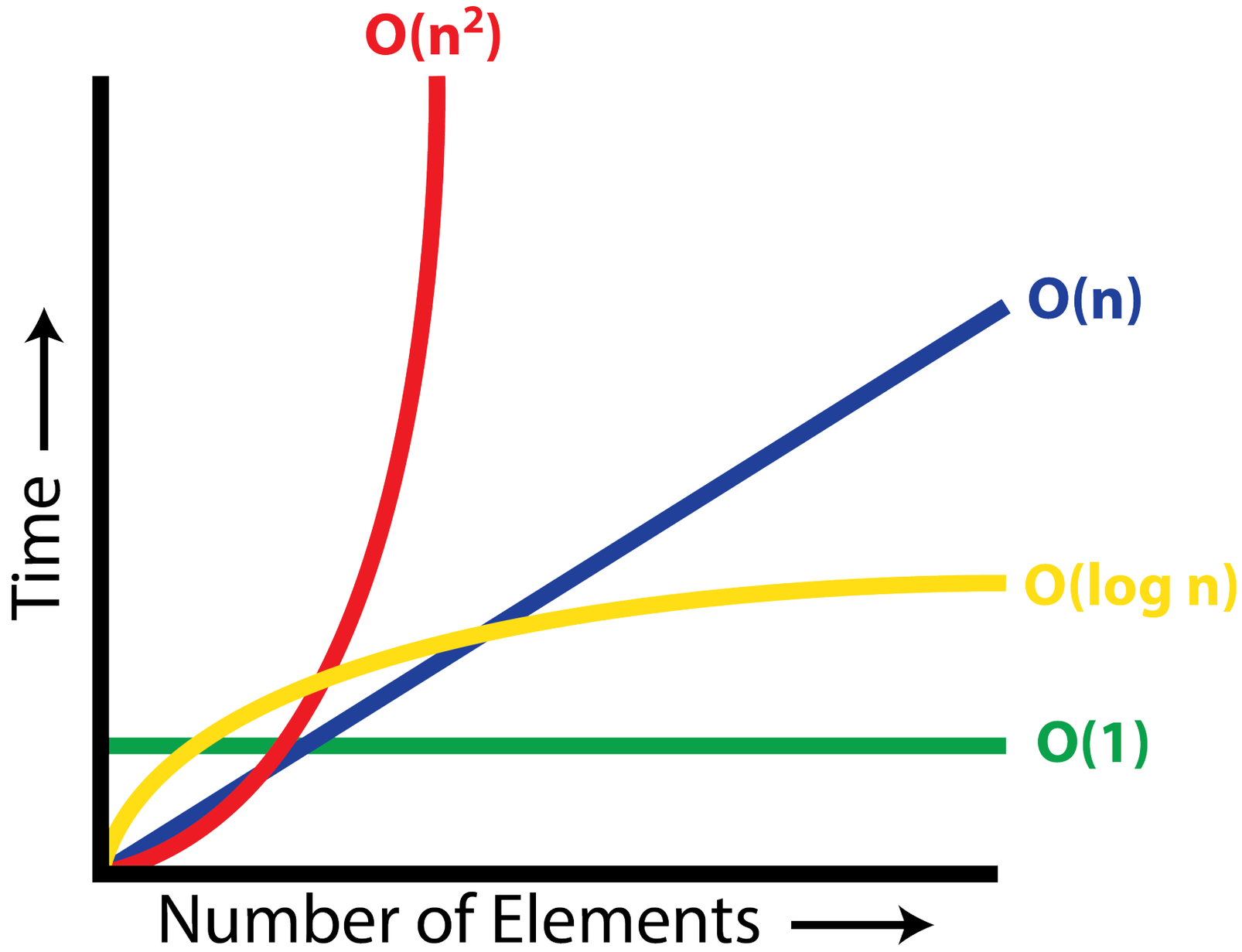
$$2S = (n+1) \cdot n$$

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

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

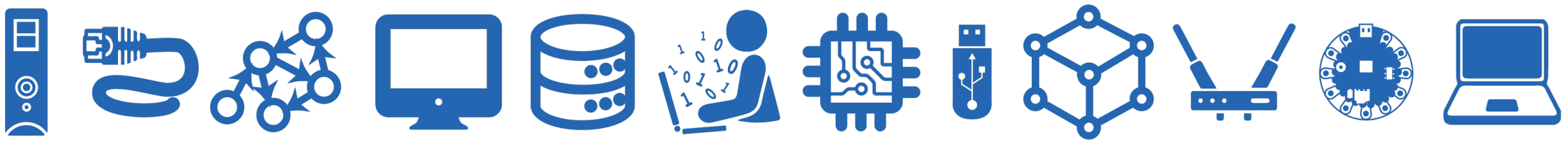
How Fast Is Selection Sort?

- Selection sort takes approximately n^2 steps!



More Efficient Sorting:

Merge Sort

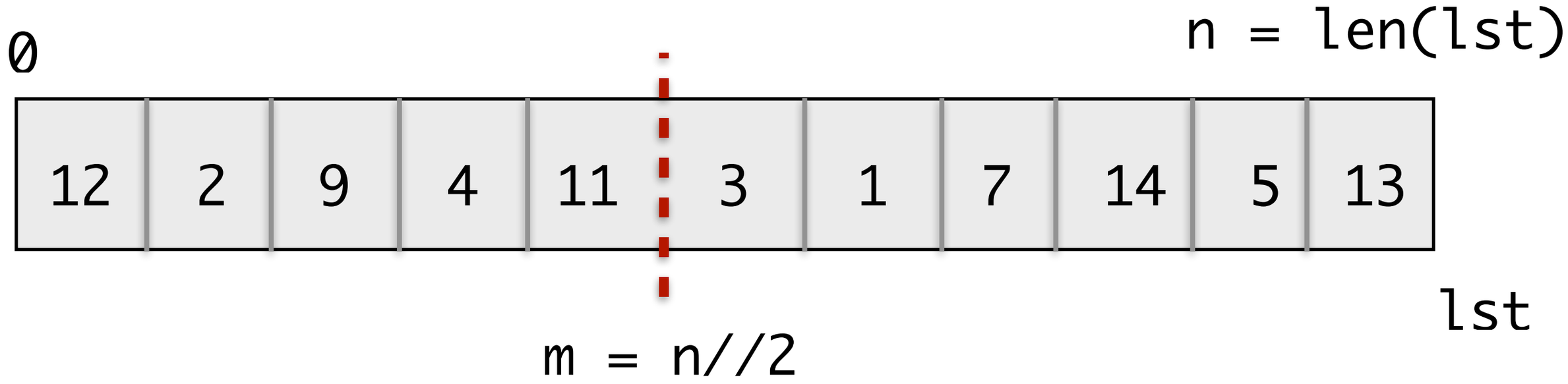


Towards an $O(n \log n)$ Algorithm

- There are other sorting algorithms that compare and rearrange elements in different ways, but are still $O(n^2)$ steps
    - They have fun names like "bubble sort" and "insertion sort", but they share the general approach of iterating through the unsorted region of the list to determine the final location for a single element
    - You will discuss (and implement them!) if you take CSCI 136
- Today we want to explore an $O(n \log n)$ time sorting algorithm: ***Merge sort*** (Invented by John von Neumann in 1945)
    - $O(n \log n)$ is the best we can do in a comparison-based sort!

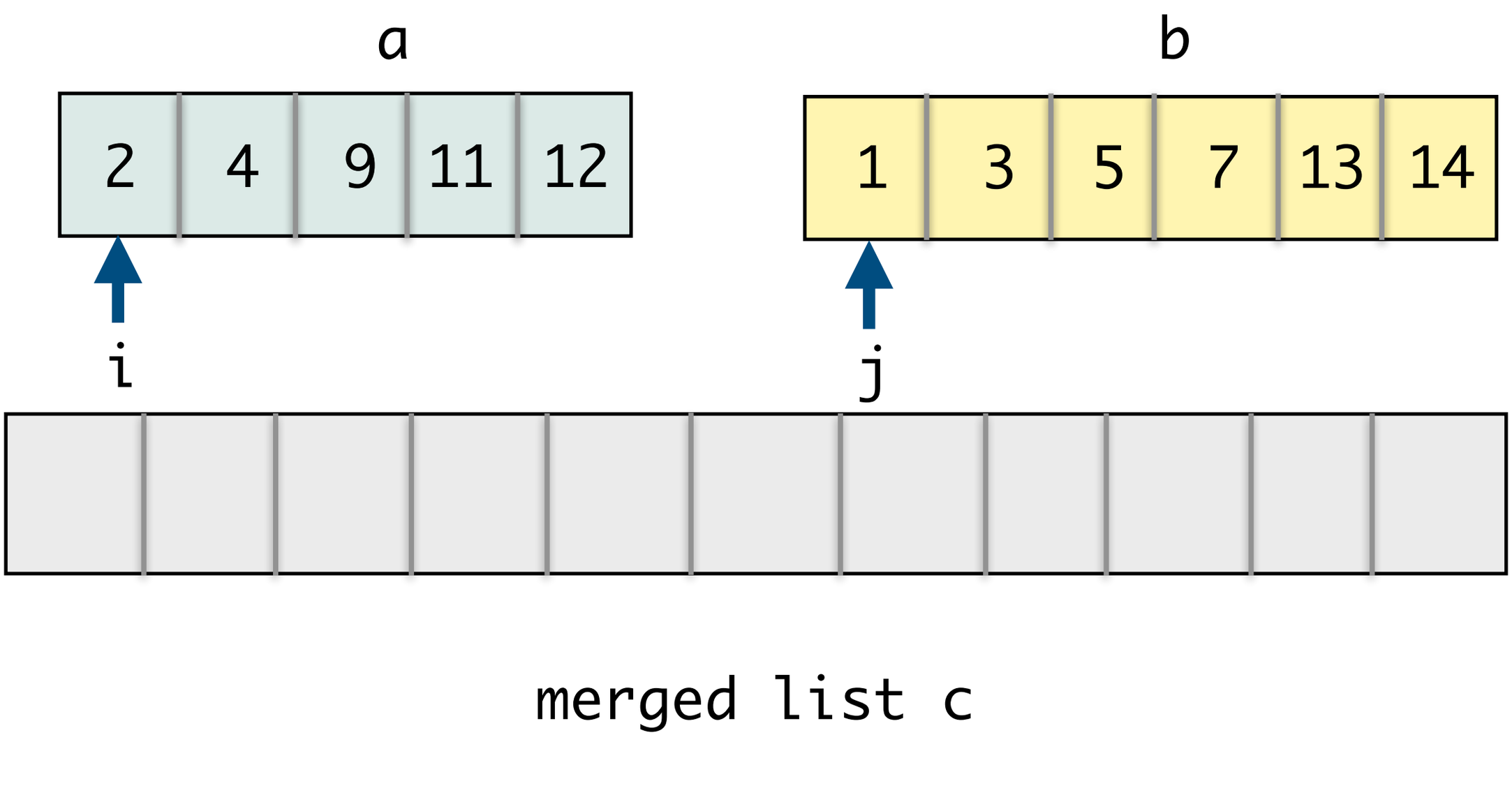
Merge Sort: Basic Idea

- If we split the list in half, sorting the left and right half are smaller versions of the same problem
- To solve the larger problem (sort the full list), we need to combine the solutions to the two smaller problems (sorted half lists)
- **Algorithm:**
  - **(Divide)** Recursively sort left and right half
  - **(Unite)** Merge the sorted halves into a single sorted list



Merging Sorted Lists

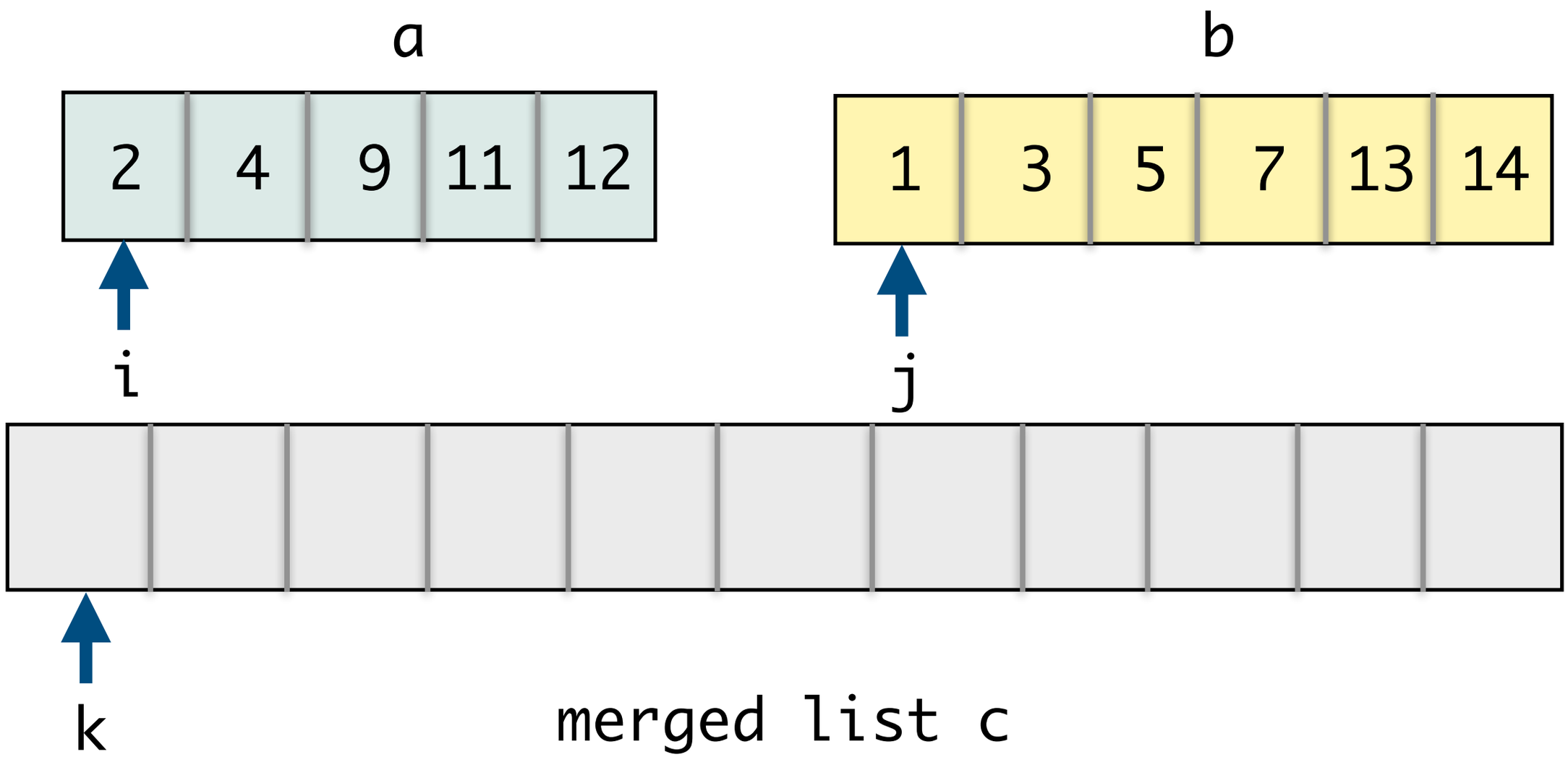
- **Problem.** Given two sorted lists `a` and `b`, how quickly can we merge them into a single sorted list?



Merging Sorted Lists

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

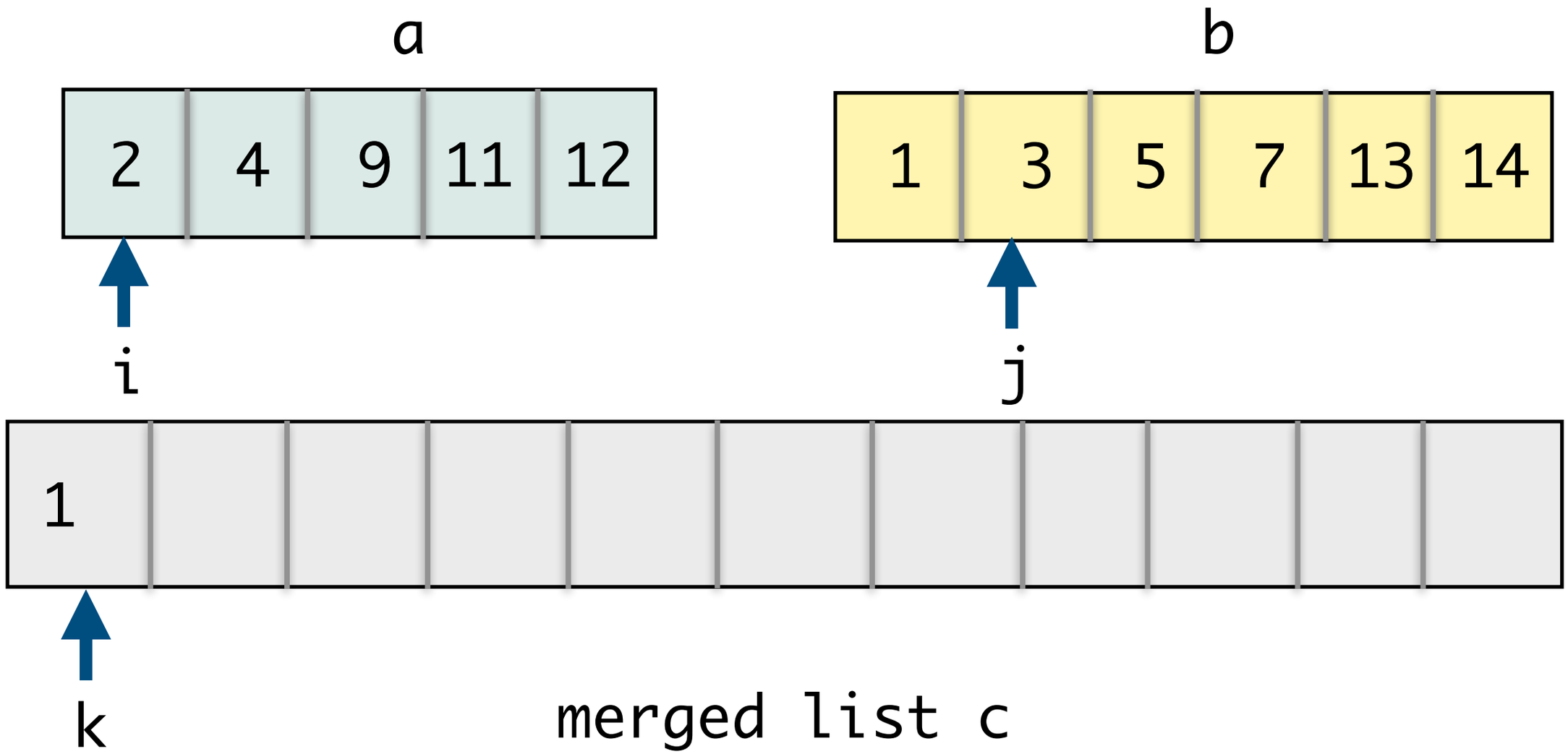
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

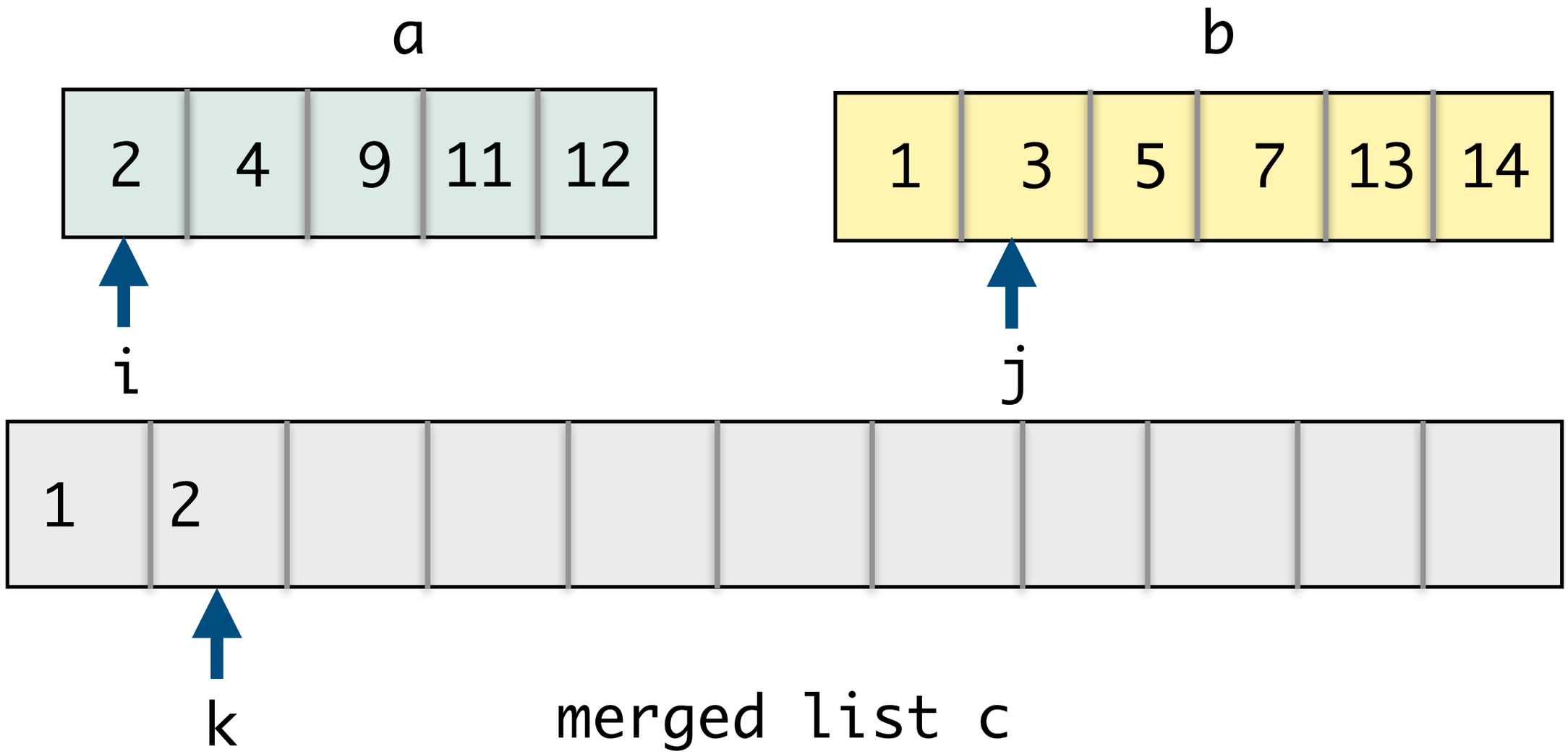
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

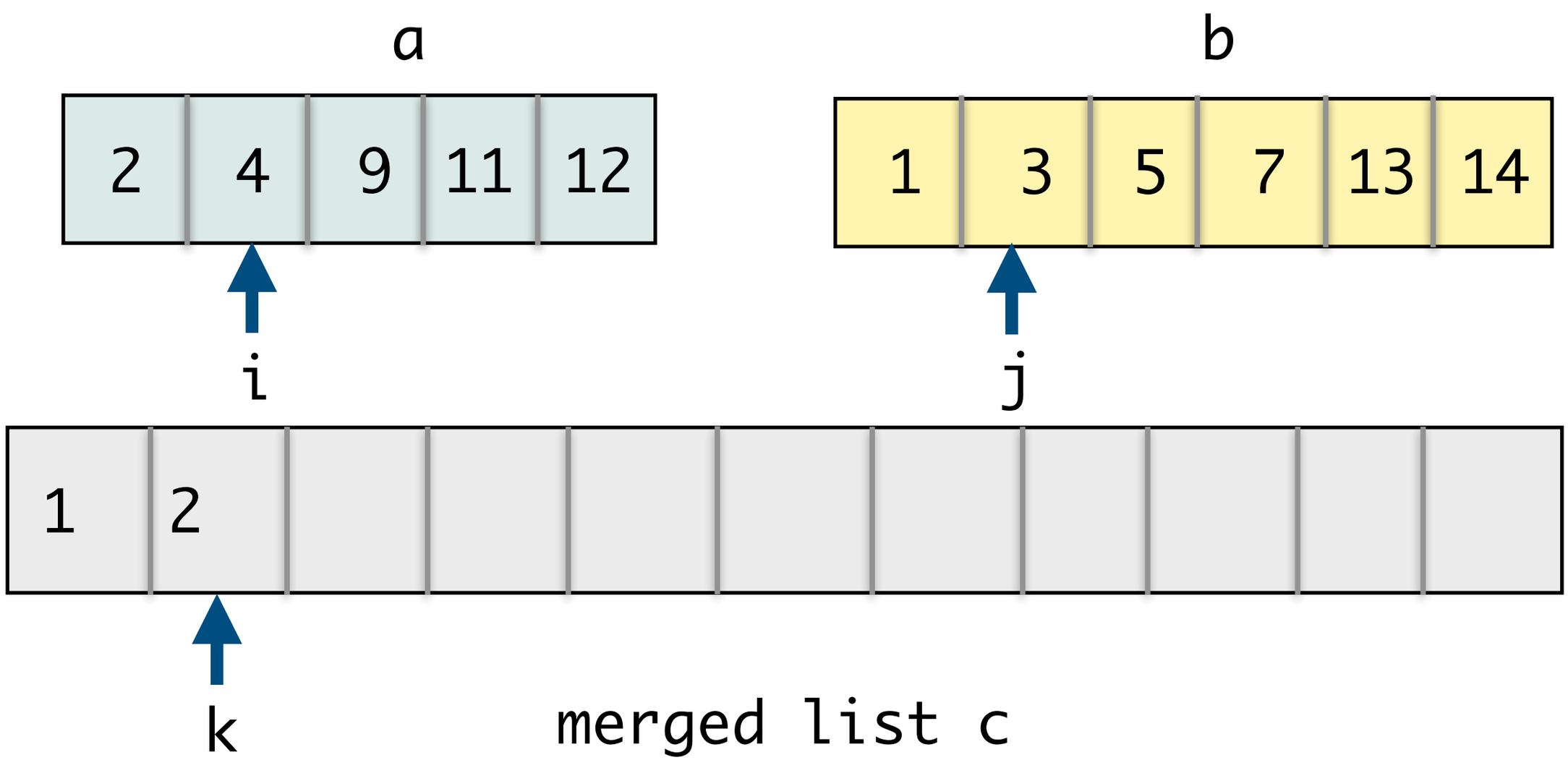
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

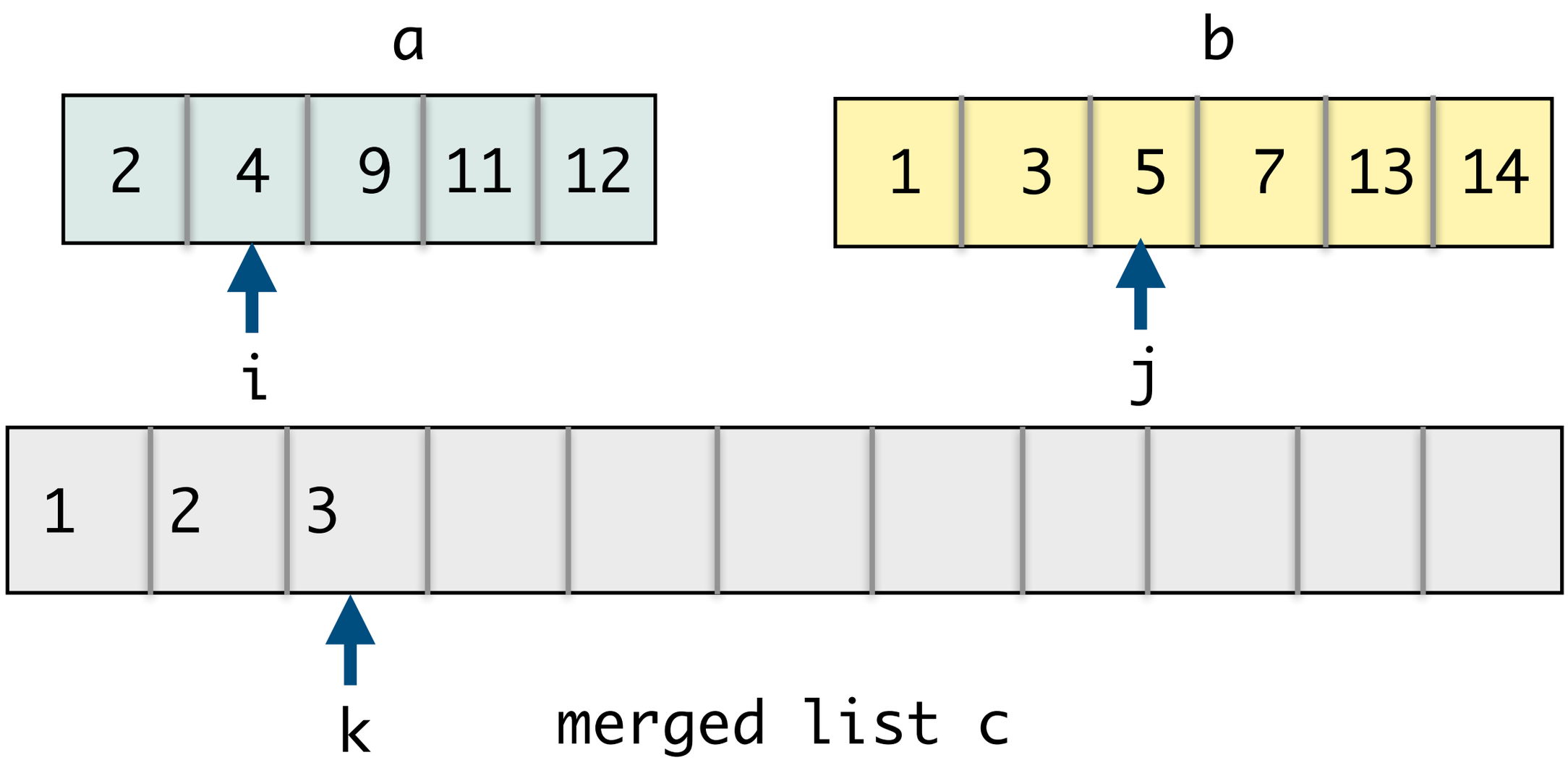
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

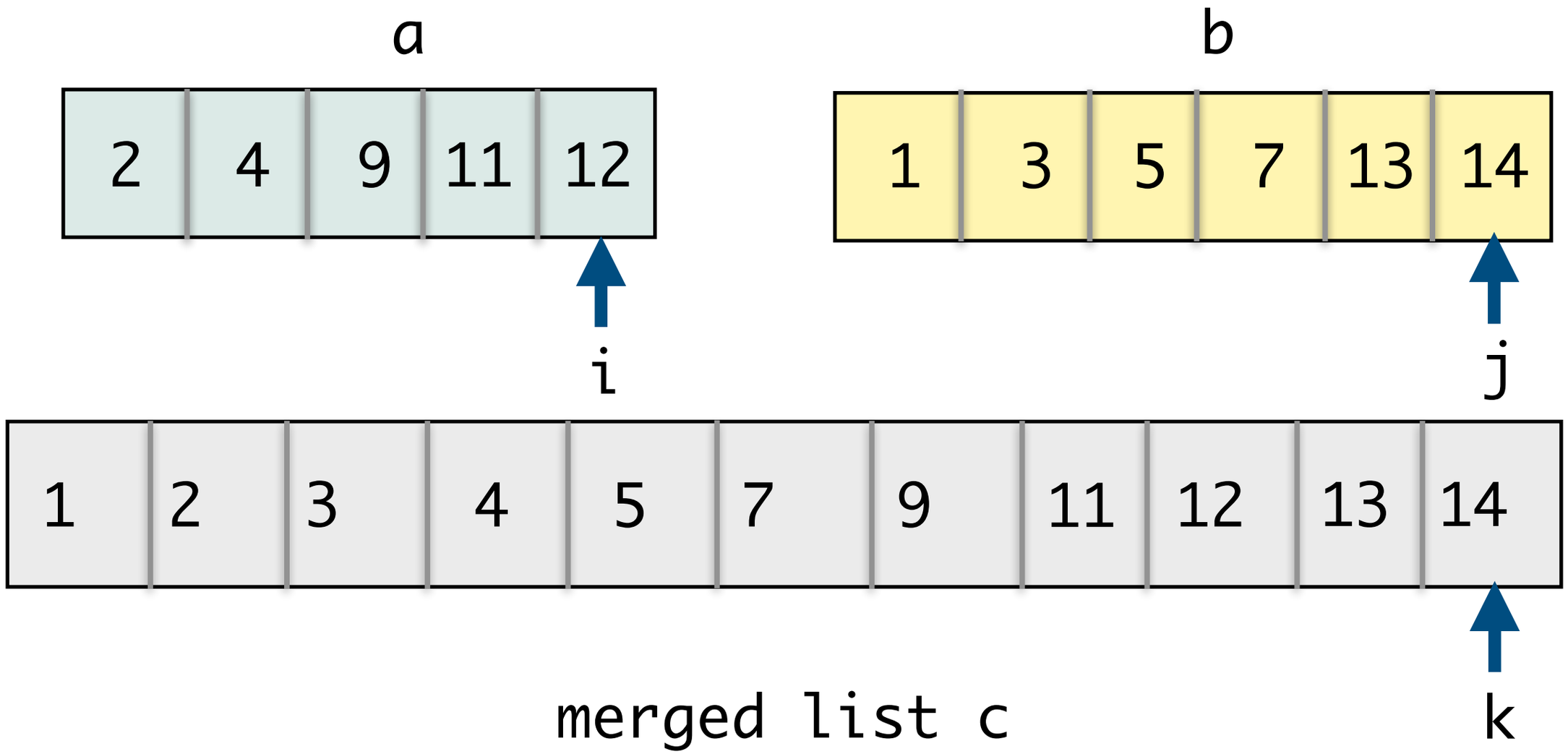
- Yes, `a[i]` appended to `c`
- No, `b[j]` appended to `c`



Merging Sorted Lists

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

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



Merging Sorted Lists

- 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!

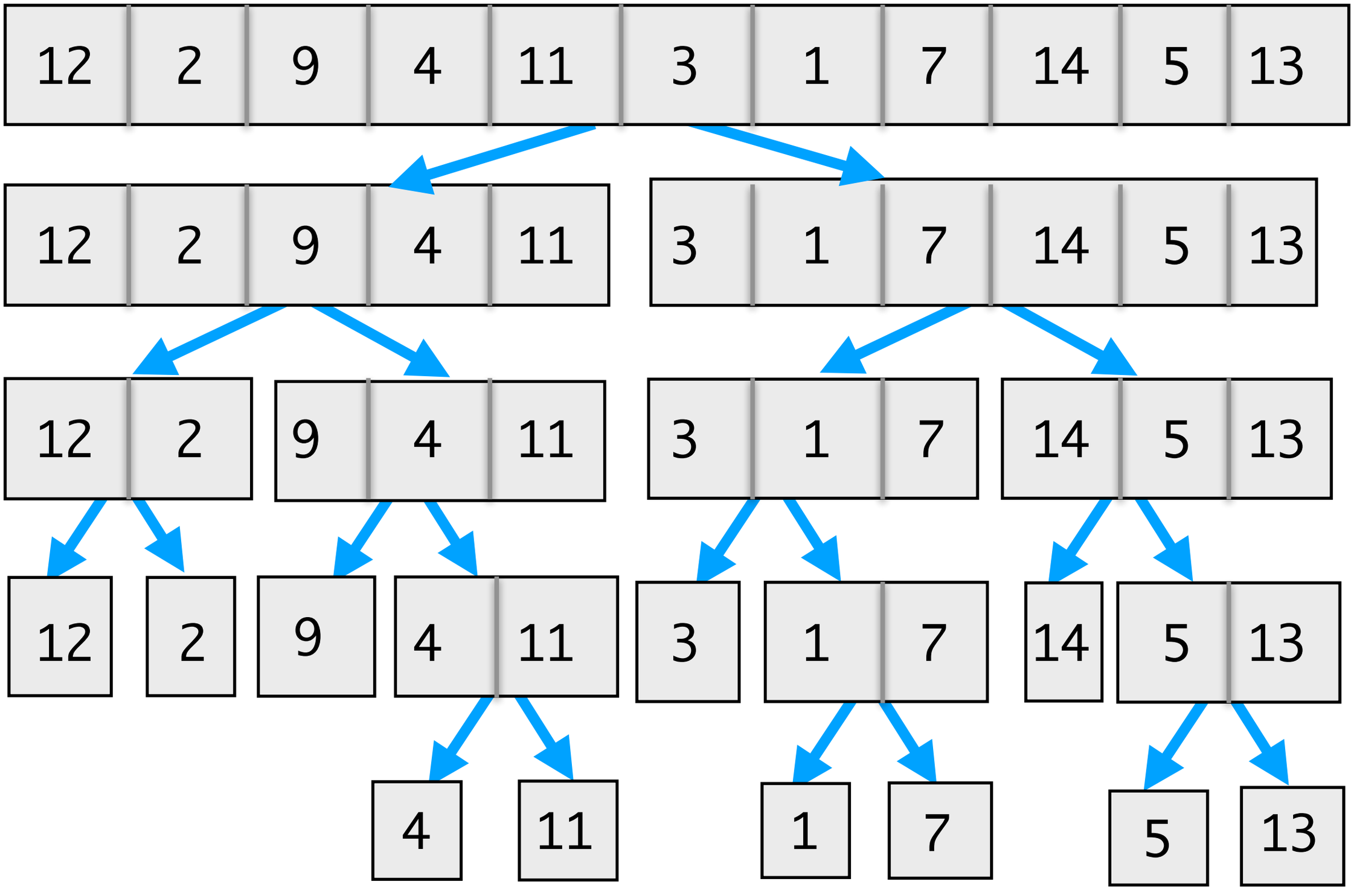
```
def merge(a, b):
    """Merges two sorted lists a and b,
    and returns new merged list c"""
    # initialize variables
    i, j, k = 0, 0, 0
    len_a, len_b = len(a), len(b)
    c = []
    # traverse and populate new list
while i < len_a and j < len_b:

    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

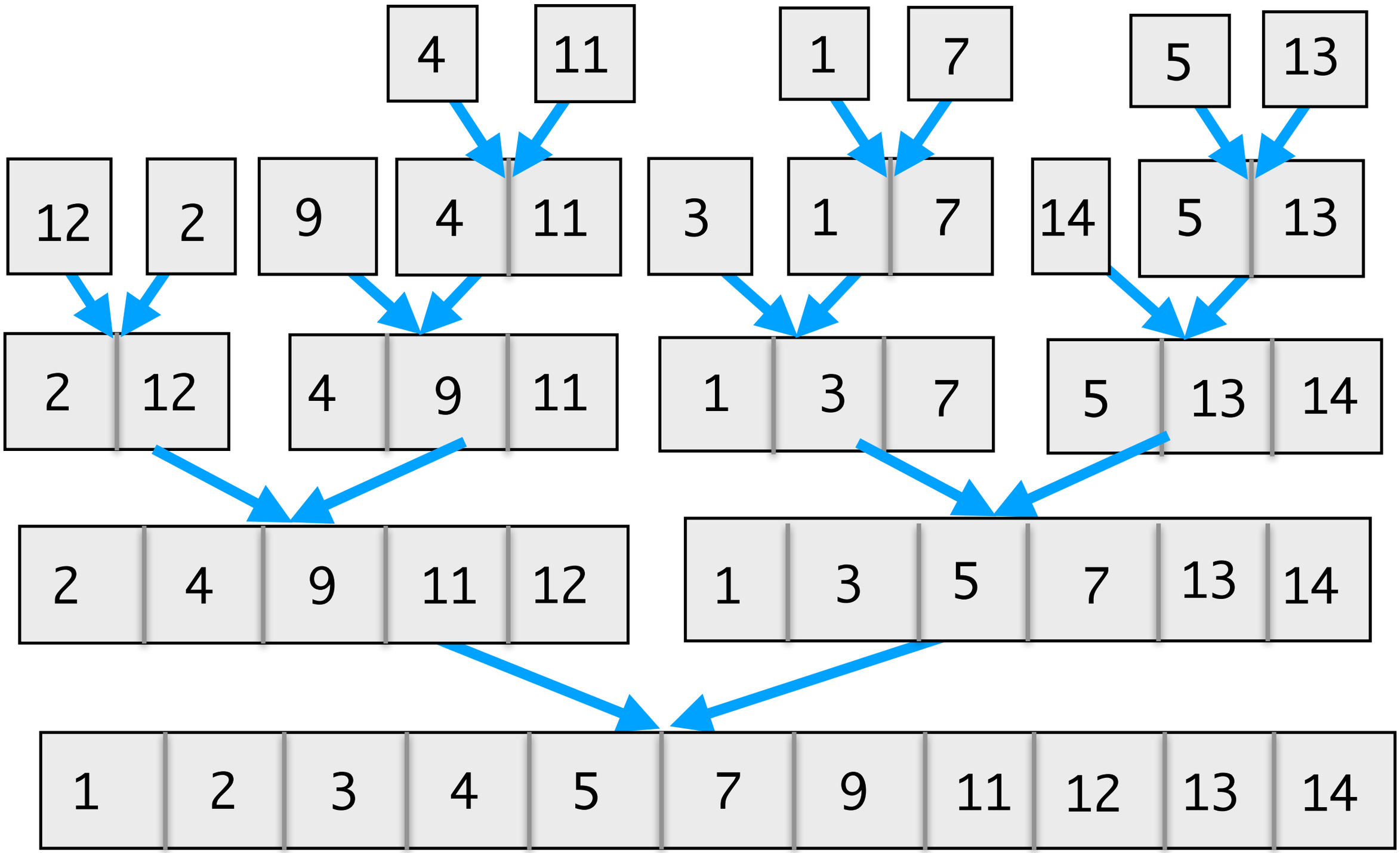
- **Base case:** If list is empty or contains a single element: it is already sorted
- **Recursive case:**
  - Recursively sort left and right halves
  - Merge the sorted lists into a single list and return it
- **Question:**
  - Where is the ***sorting*** actually taking place?

```
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

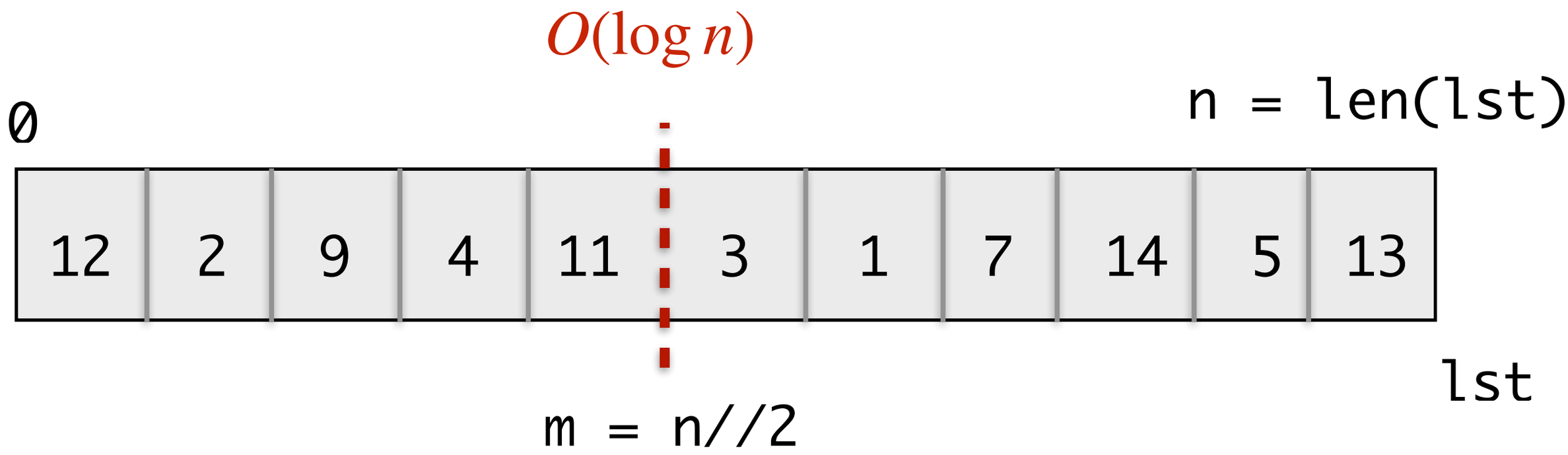


Merge Sort Example



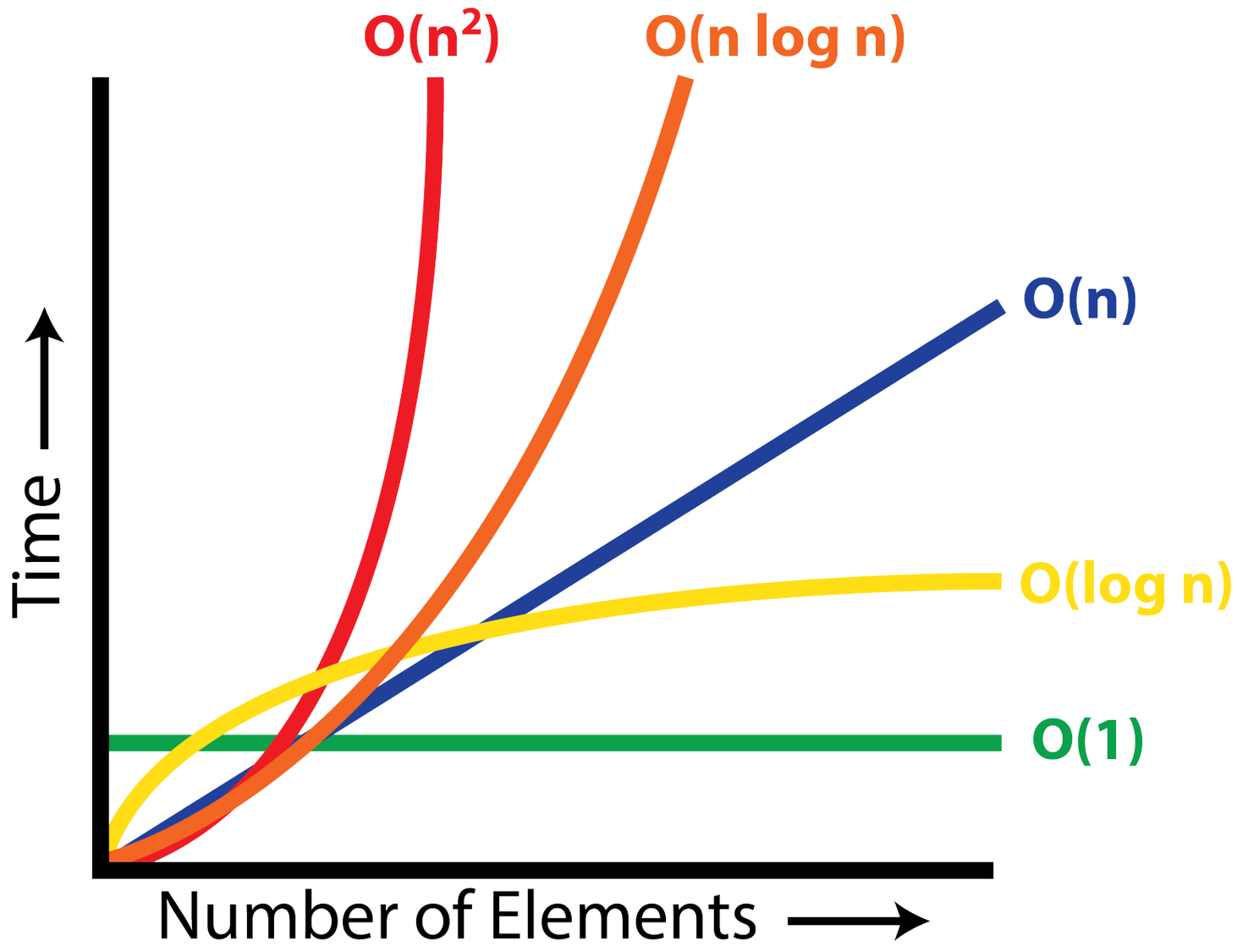
Merge Sort: Basic Idea

- 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 ($O(\log n)$ splits until all lists are length 1)
  - **(Unite)** Merge the sorted halves into a single sorted list ($O(n)$)



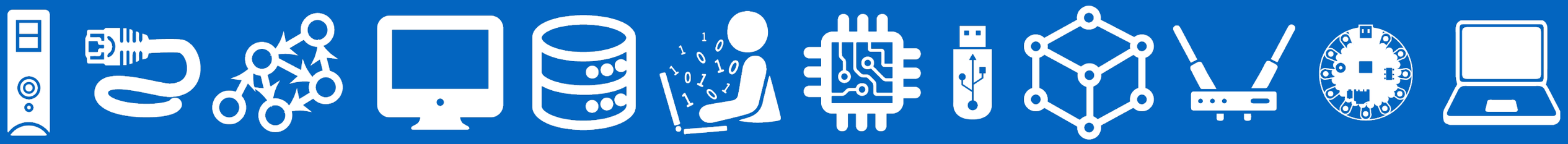
Big Oh Comparisons

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



Merge Sort w Transylvanian-Saxon Folk Dance

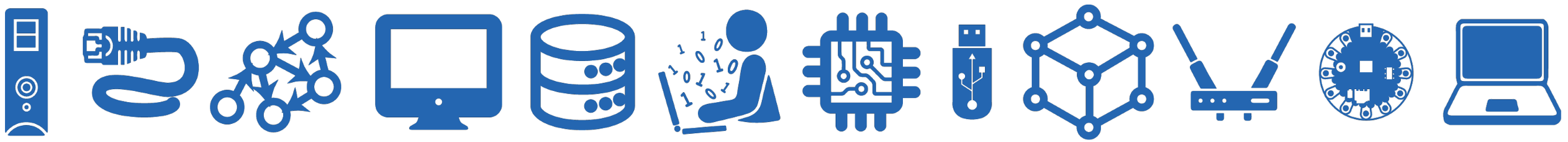
- https://www.youtube.com/watch?v=XaqR3G_NVoo



Keep going!

Until you see "The End"!

Selection Sort
vs. Merge Sort
in practice



Selection vs Merge Sort in Practice

- Selection sort is $O(n^2)$ and merge sort is $O(n \log n)$ time
- How different is the performance in practice?
- Example: `word_lst` is 12,000 words from the book *Pride & Prejudice*
- We'll time our algorithms on the first 100, 500, 1000, 5000, and 10000 words

```
word_lst = []
with open('prideandprejudice.txt') as book:
    for line in book:
        line = line.strip().split()
        word_lst.extend(line)
print(len(word_lst))
```

```
12089
```

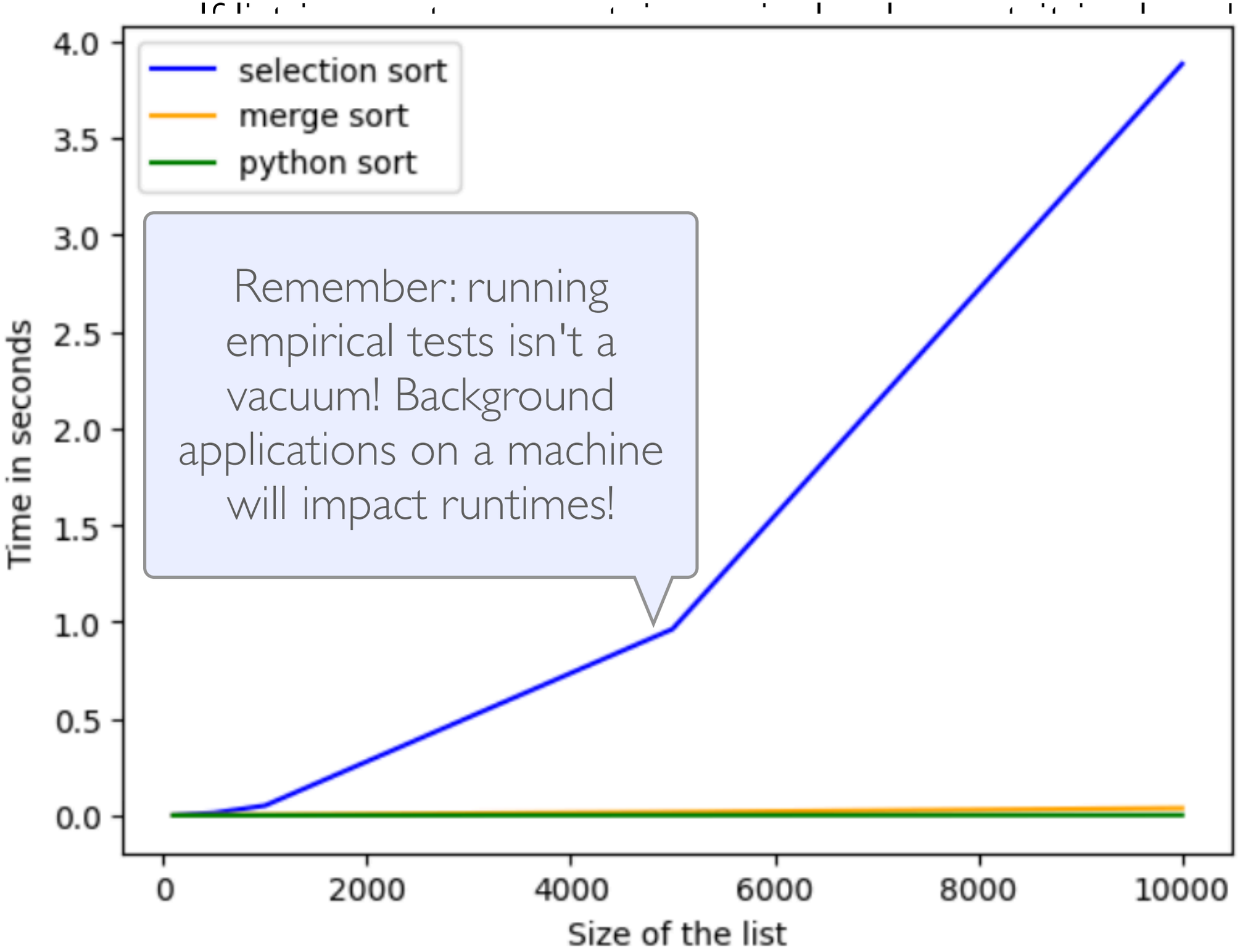
Timing Sort Algorithms

```
from matplotlib import pyplot as plt
from time import time

def compare_sorts(data):
    list_sizes = [100, 500, 1000, 5000, 10000]
    selection_times = []
    merge_times = []
    pythonsort_times = []

    for size in list_sizes:

        print("size:", size) # debugging
        lst_to_sort = data[:size] # compare times
        lst_to_sort1 = lst_to_sort[:]
        lst_to_sort2 = lst_to_sort[:]
        lst_to_sort3 = lst_to_sort[:]

        selection_start = time()
        selection_sort(lst_to_sort1)
        selection_end = time()
        selection_times.append(selection_end - selection_start)

        merge_start = time()
        merge_sort(lst_to_sort2)
        merge_end = time()
        merge_times.append(merge_end - merge_start)

        pythonsort_start = time()
        lst_to_sort3.sort()
        pythonsort_end = time()
        pythonsort_times.append(pythonsort_end - pythonsort_start)

    # plot lines
    plt.plot(list_sizes, selection_times, label = "selection sort")
    plt.plot(list_sizes, merge_times, label = "merge sort")
    plt.plot(list_sizes, pythonsort_times, label = "python sort")
    plt.xlabel("Size of the list")
    plt.ylabel("Time in seconds")
    plt.legend()
    plt.show()
```

Let's also compare python's `.sort()` method

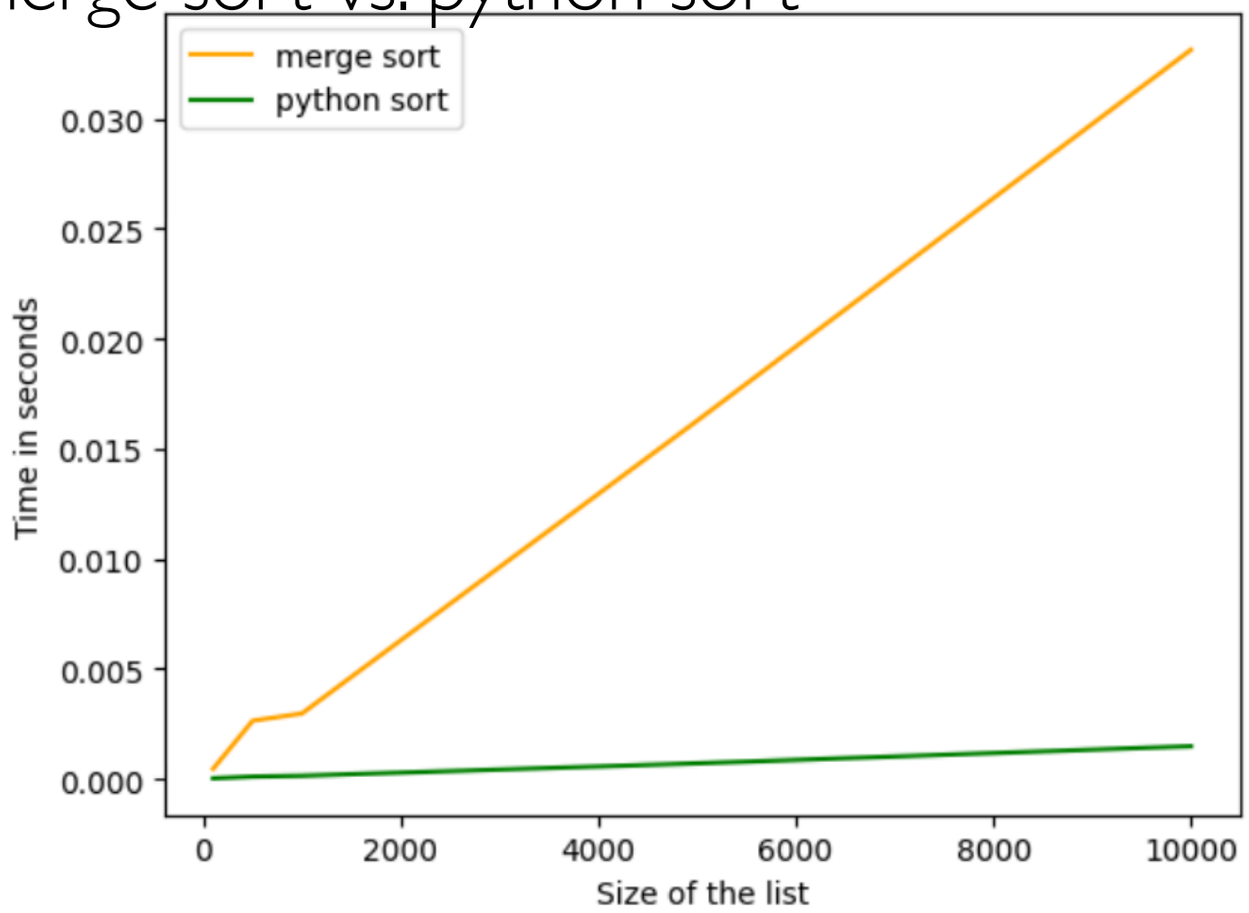
Timing Sort Algorithms

- **B**
  - sc
- **R**
  -
  -
- C
  -



Timing Sort Algorithms

Close-up of merge sort vs. python sort



- Python uses a hybrid of merge sort and insertion sort
- Python *always* optimizes, so built-in libraries typically are the most efficient
    - Often implemented in C, which is faster than python

Searching Algorithms in Practice

- Linear search is $O(n)$ and Binary Search is $O(\log n)$ time
- How different is the performance in practice?
- Example: `num_lst` is 100,000 sorted numbers
- We'll time our algorithms on the first 1000, 5000, 10000, 50000, and 100000 elements

```
num_lst = list(range(100000))
print(num_lst[:5])
```

```
[0, 1, 2, 3, 4]
```

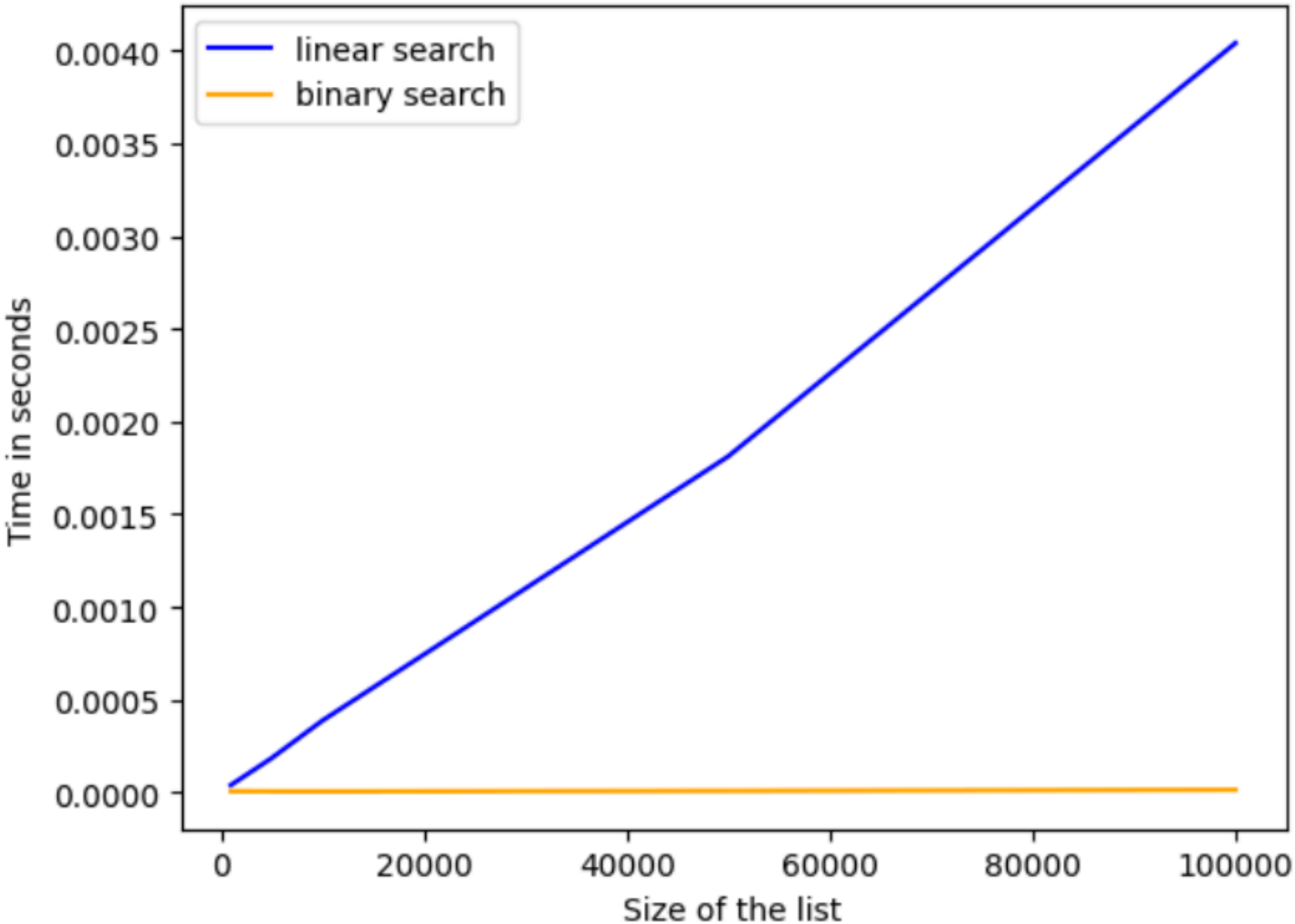
Timing Search Algorithms

```
from matplotlib import pyplot as plt
from time import time

def compare_searches(data, item):
    list_sizes = [1000, 5000, 10000, 50000, 100000]
    linear_times = []
    bsearch_times = []
    pythonin_times = []

    for size in list_sizes:

        print("size:", size) # debugging
        lst_to_search = data[:size] # compare times

        linear_start = time()
        linear_search(lst_to_search, item)
        linear_end = time()
        linear_times.append(linear_end - linear_start)

        bsearch_start = time()
        binary_search_better(lst_to_search, item, 0, size)
        bsearch_end = time()
        bsearch_times.append(bsearch_end - bsearch_start)

        pythonin_start = time()
        item in lst_to_search
        pythonin_end = time()
        pythonin_times.append(pythonin_end - pythonin_start)

    # plot lines
    plt.plot(list_sizes, linear_times, label = "linear search", color="blue")
    plt.plot(list_sizes, bsearch_times, label = "binary search", color="orange")
    plt.plot(list_sizes, pythonin_times, label = "python 'in'", color="green")
    plt.xlabel("Size of the list")
    plt.ylabel("Time in seconds")
    plt.legend()
    plt.show()
```

Timing Search Algorithms

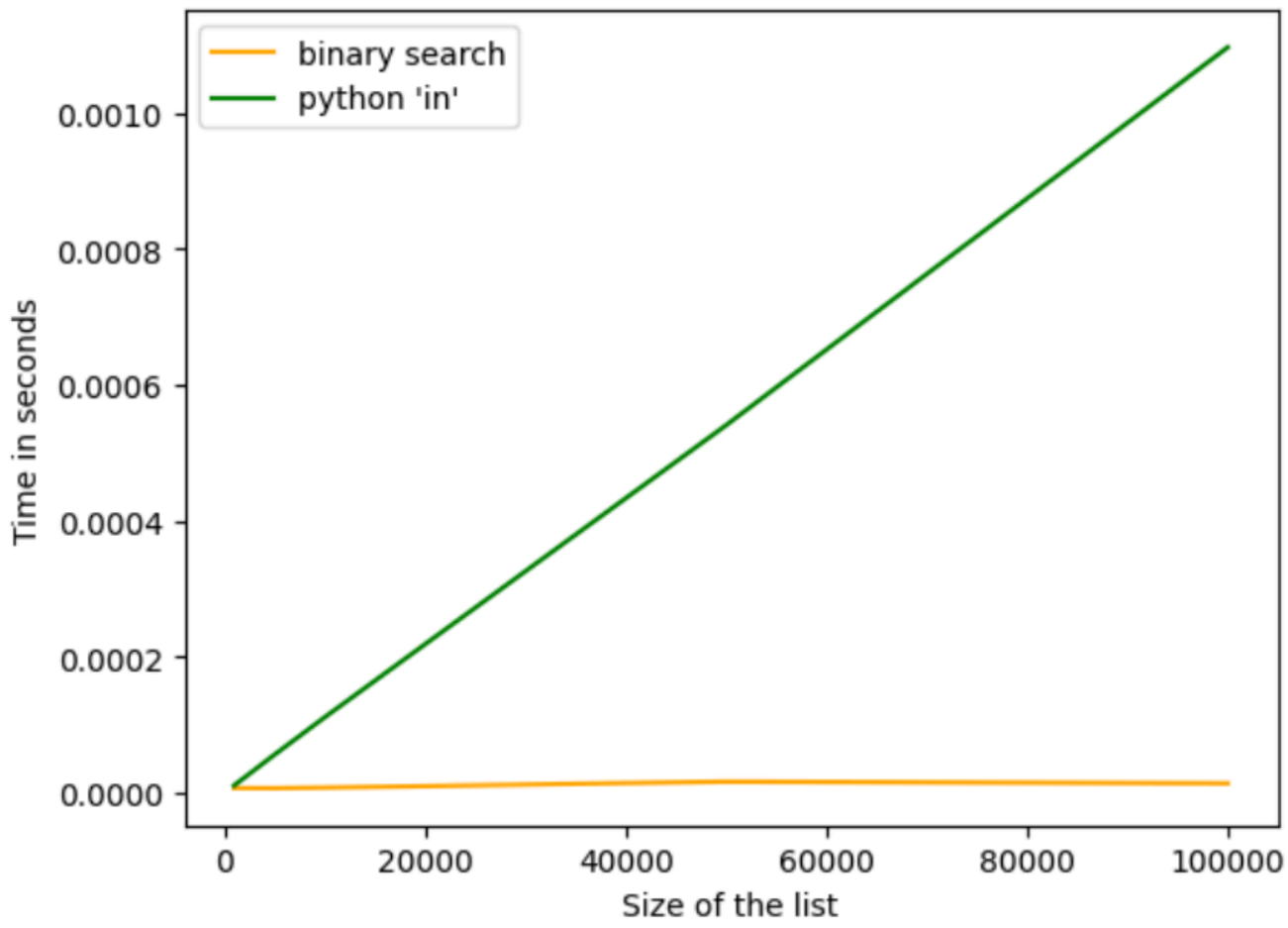
```
# worst case: look for number not in the list
compare_searches(num_lst, -1)
```

- **Base case:** If list is empty or contains a single element: it is already
-
-



Timing Sort Algorithms

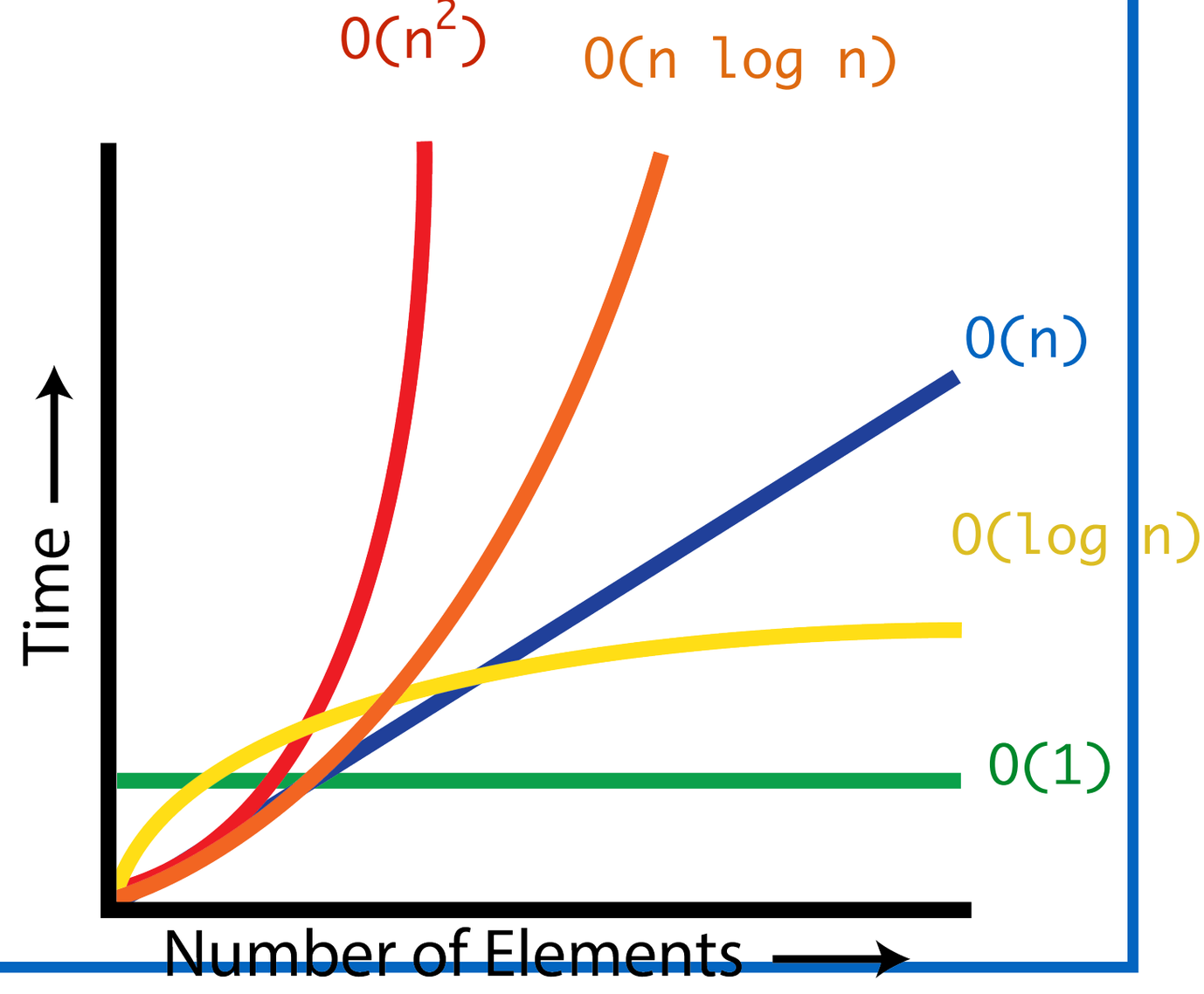
Comparing Binary Search and python `in`



- Python doesn't know when a list is sorted, so `in` *must* use linear search!

Summary: Searching and Sorting

- We have seen algorithms that are
  - $O(\log n)$: binary search in a sorted list
  - $O(n)$: linear searching in an unsorted list
  - $O(n \log n)$: merge sort
  - $O(n^2)$: selection sort
- Important to think about efficiency when writing code!



The end!