CS134 Lecture:
Searching & Sorting

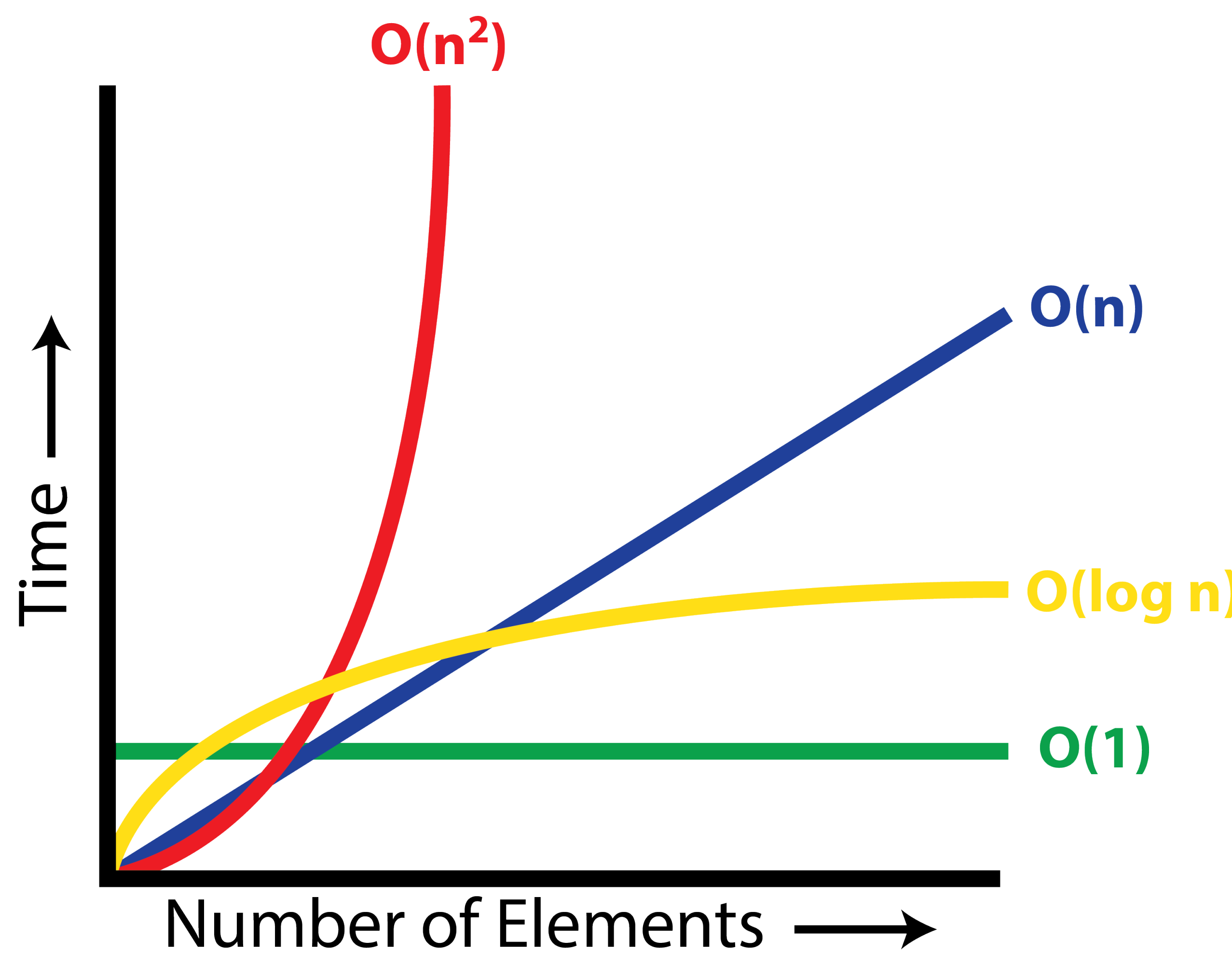
Announcements & Logistics

- **Lab 9** Parts 1 and 2 due today/tomorrow
  - Any questions?
- **HW 11**: Released today, due next Monday
- No Lab next week (Enjoy Thanksgiving break!!!)
  - Practice Final Exam w/ sample solutions will be posted in place of lab
- CS134 Scheduled Final: **Wednesday, December 11, 9:30 AM**
  - Room: **Wachenheim B11**

Do You Have Any Questions?

Last Time: Efficiency

- Defined efficiency as the number of steps taken by algorithm on worst-case inputs of a given size
- Introduced Big-O notation: captures the rate at which the number of steps taken by the algorithm grows w.r.t. input size n, "as n gets large"



Today: Searching (and Sorting)

- Discuss recursive implementation of binary search
- 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

Searching in a Sequence

Search Warm-ups

- **Search Q1:** Given a random input sequence `seq`, search if a given `item` is in the sequence.
- **Input:** a sequence `seq` of n items and a query item, `item`
- **Output:** True if query item is in sequence, else False
- **Search Q2:** Given a random input sequence `seq`, determine if any item in the sequence is a duplicate.
- **Input:** a sequence `seq` of n items
- **Output:** True if at least one duplicate pair of items is in sequence, else False
- (Rules; Use loops or recursion; don't use sets/dictionaries.)

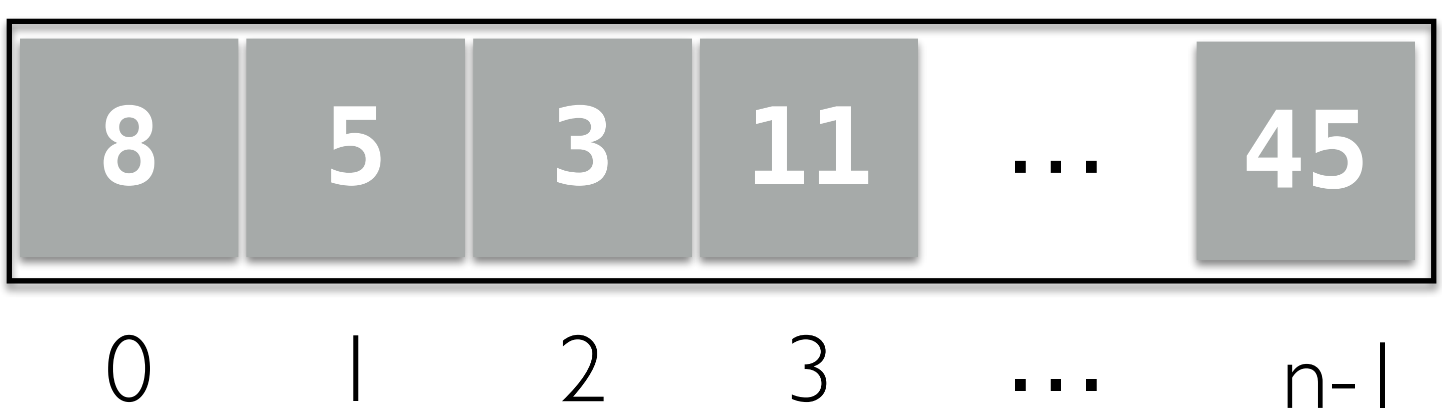
Let's try to write both functions

Searching in a Sequence I

- First algorithm: iterate through the items in sequence and compare each item to query

```
def linear_search(item, seq):
    for elem in seq:
        if elem == item:
            return True
    return False
```

Might return early if item is first elem in seq, but we are interested in the **worst case analysis**; worst case is if item is not in seq at all



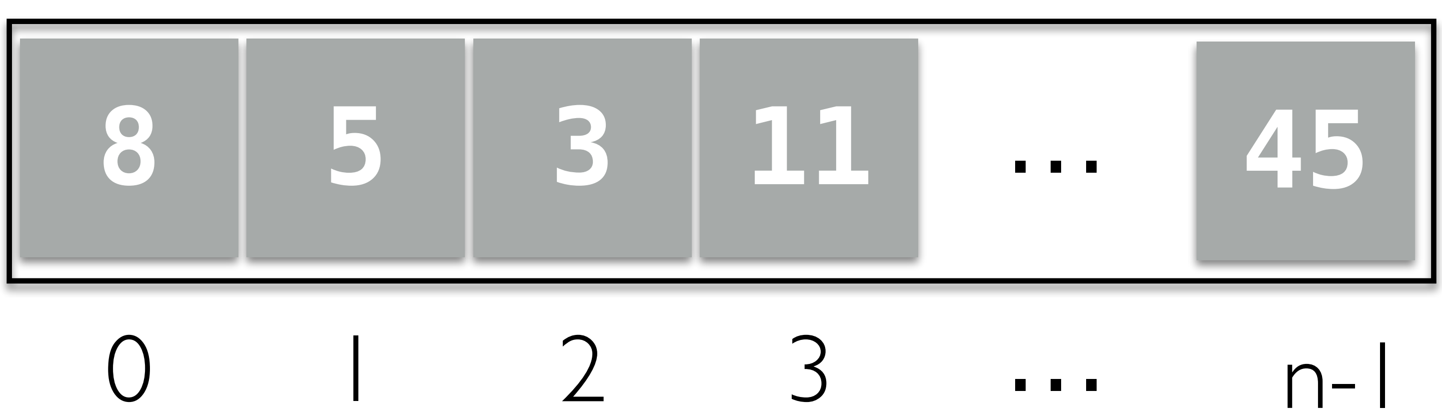
Searching in a Sequence I

- In the worst case, we have to walk through the entire sequence
- Overall, the number of steps is linear in n. We write this as $O(n)$

```
def linear_search(item, seq):
    for elem in seq:
        if elem == item:
            return True
    return False
```

Loop runs n items in worst case

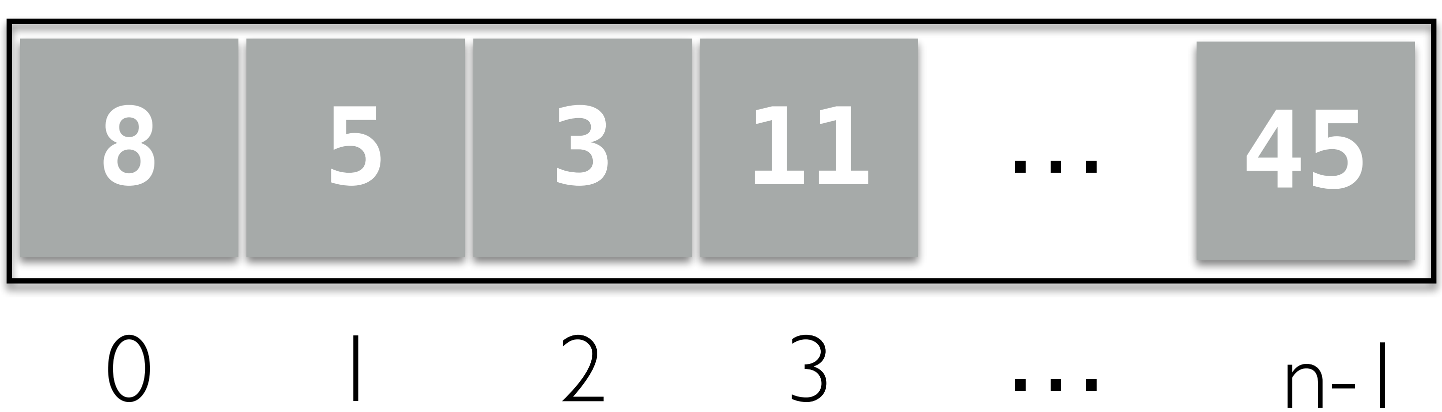
One equality check per iteration: assume comparing elem == item is one step



Searching in a Sequence 2

- Second algorithm: nested loop!
    - Outer loop iterates through the items in `seq` and for each item, inner loop iterates though the items in `seq`
    - Note that the code does not compare any item to itself

```python
def has_duplicates(seq):
    for i in range(len(seq)):
        for j in range(len(seq)):
            if i != j and seq[i] == seq[j]:
                return True
    return False
```



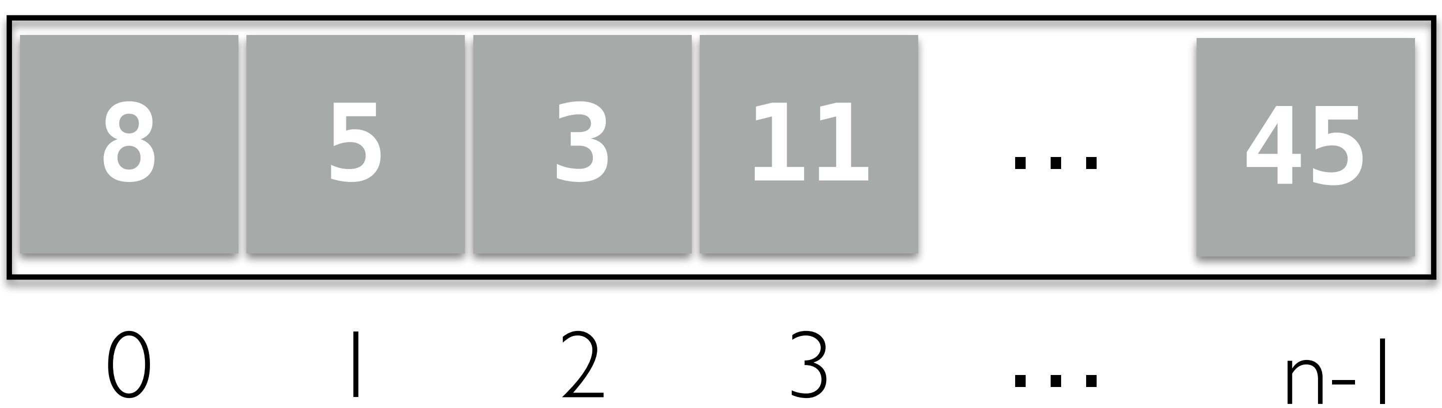
Searching in a Sequence 2

- In the worst case, we have to walk through the entire sequence (n items) once *for each item* in the sequence (n items).
- Overall, the number of steps is quadratic in n. We write this as $O(n^2)$

```
def has_duplicates(seq):
    for i in range(len(seq)):
        for j in range(len(seq)):
            if i != j and seq[i] == seq[j]:
                return True
    return False
```

Each loop runs n times in worst case

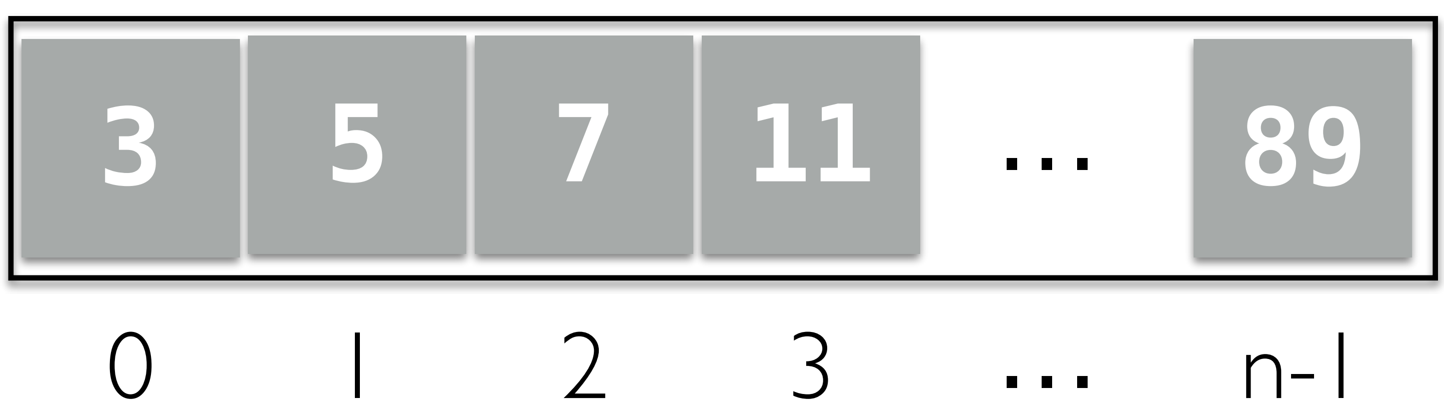
That's n comparisons for each of the n items in the worst case.



Searching in a **Sorted** Array

- If the list is in sorted order, we can do better than a linear scan. We've seen that last class.
  - Think back to our "guessing game": we want to rule out half of the remaining items each time we guess

How do we search for an item (say 10) in a **sorted** array?



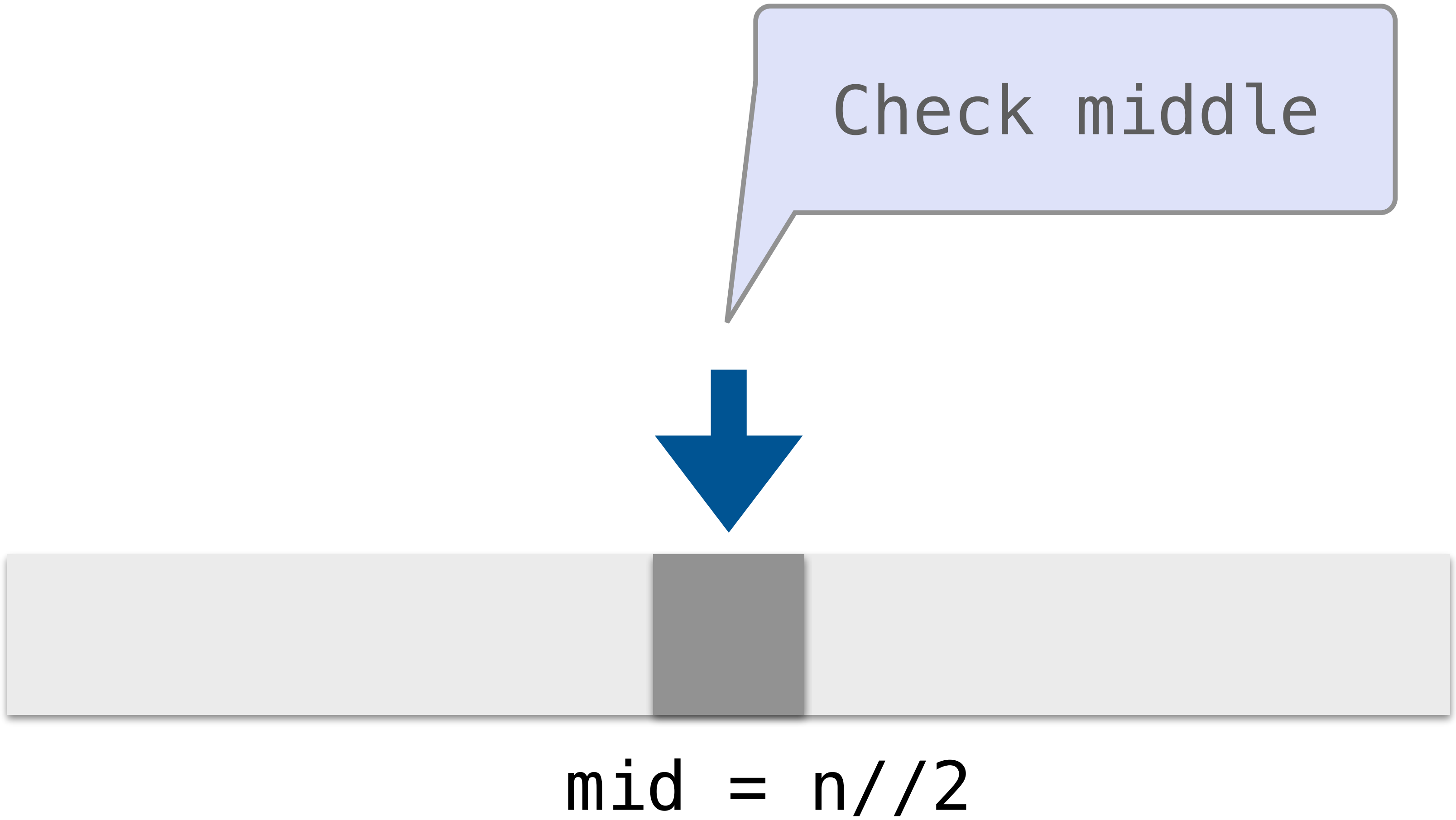
Searching in a **Sorted** Array

- Want to maximize the number of elements we rule out (in the worst case)
  - The best we can do is 50%. Why?
- Basic searching strategy for a sorted sequence is called binary search:
  - Until we find the target (or run out of items to consider), look at the item in the middle of `sequence`
    - If the target is smaller than the item at the middle index, recurse on `sequence[0:mid]`
    - If the target is larger than the item at the middle index, recurse on `sequence[mid+1:]`

Let's develop this algorithm recursively!

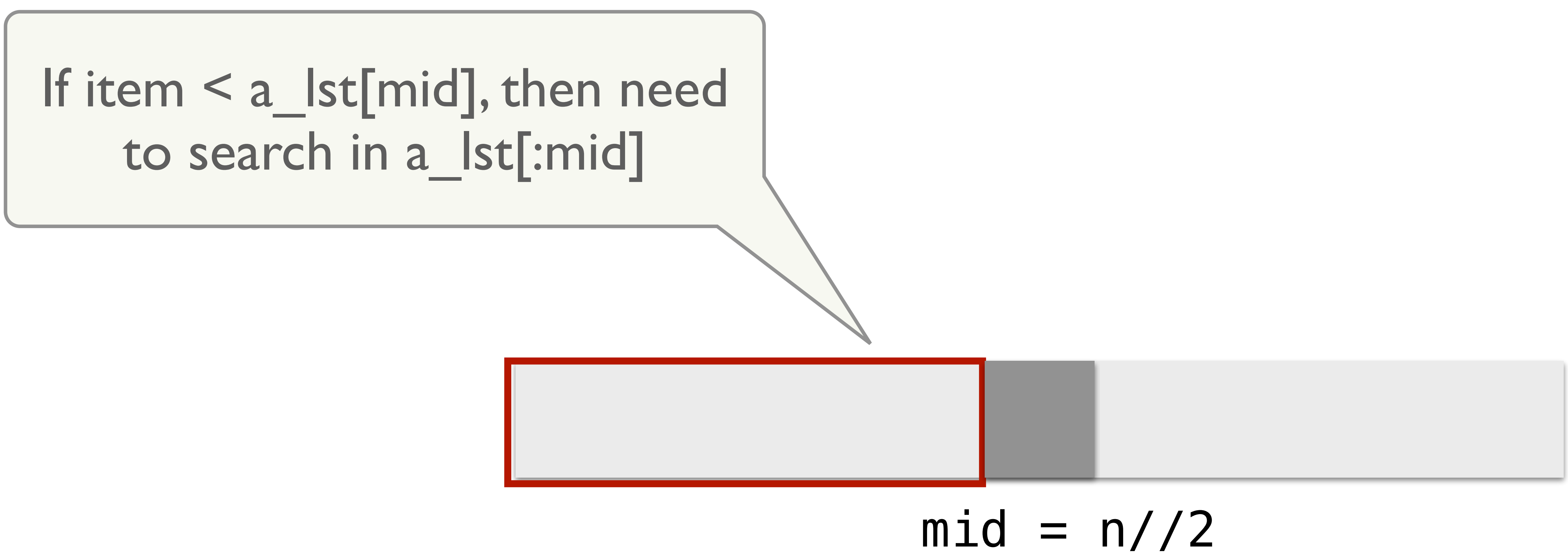
Binary Search

- Base cases? When are we done?
    - If list is too small (or empty) to continue searching, return False
    - If item we're searching for is the middle element, return True



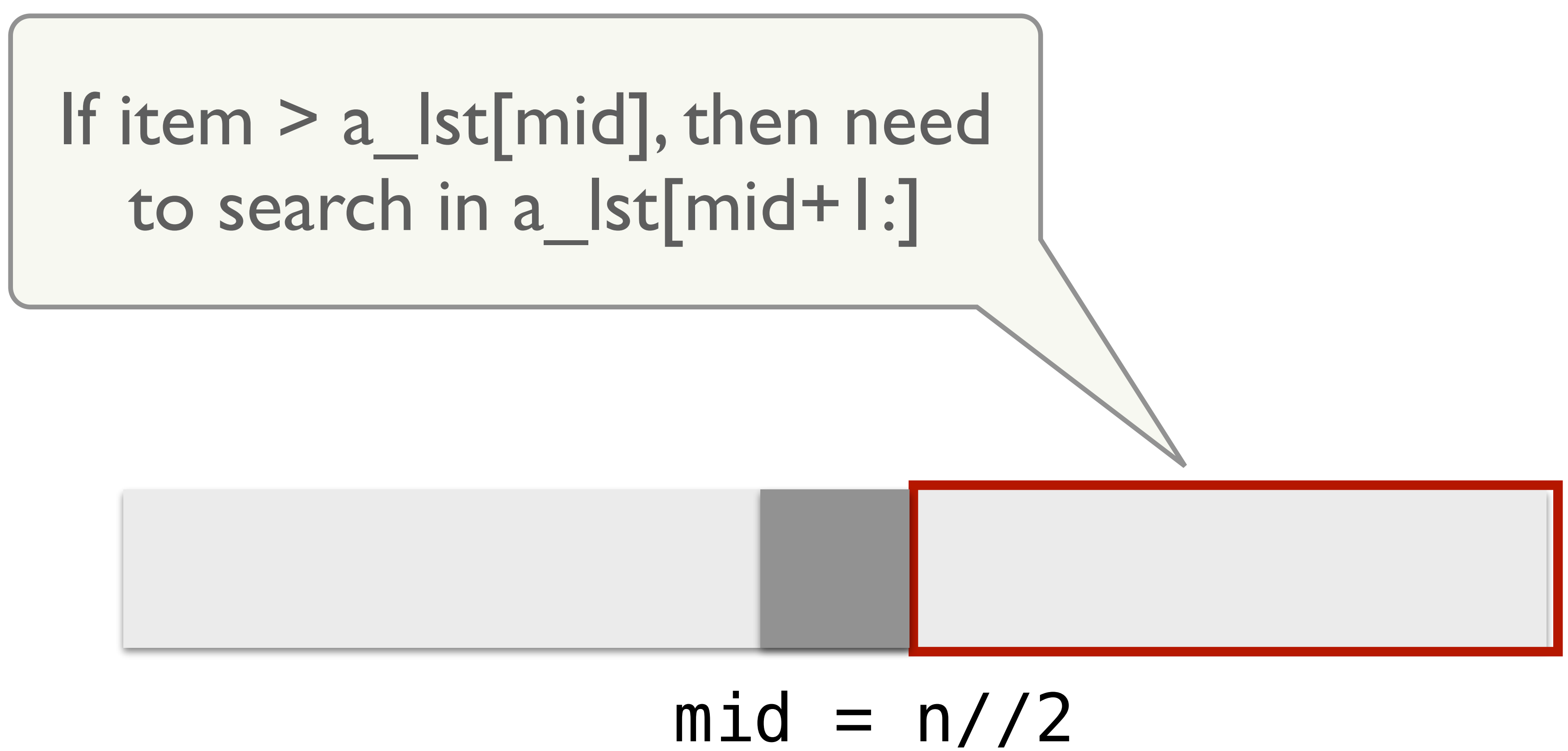
Binary Search

- Recursive case:
    - Recurse on left side if item is smaller than middle
    - Recurse on right side if item is larger than middle



Binary Search

- Recursive case:
    - Recurse on left side if item is smaller than middle
    - Recurse on right side if item is larger than middle



```python
def binary_search(seq, item):
    """Assume seq is sorted. If item is
    in seq, return True; else return False."""

    n = len(seq)

    # base case 1
    if n == 0:
        return False

    mid = n // 2
    mid_elem = seq[mid]

    # base case 2
    if item == mid_elem:
        return True

    # recurse on left
    elif item < mid_elem:
        left = seq[:mid]
        return binary_search(left, item)

    # recurse on right
    else:
        right = seq[mid+1:]
        return binary_search(right, item)
```

Technically, there is one *small* problem with our implementation. List splicing is actually O(n)!

Binary Search: Improved

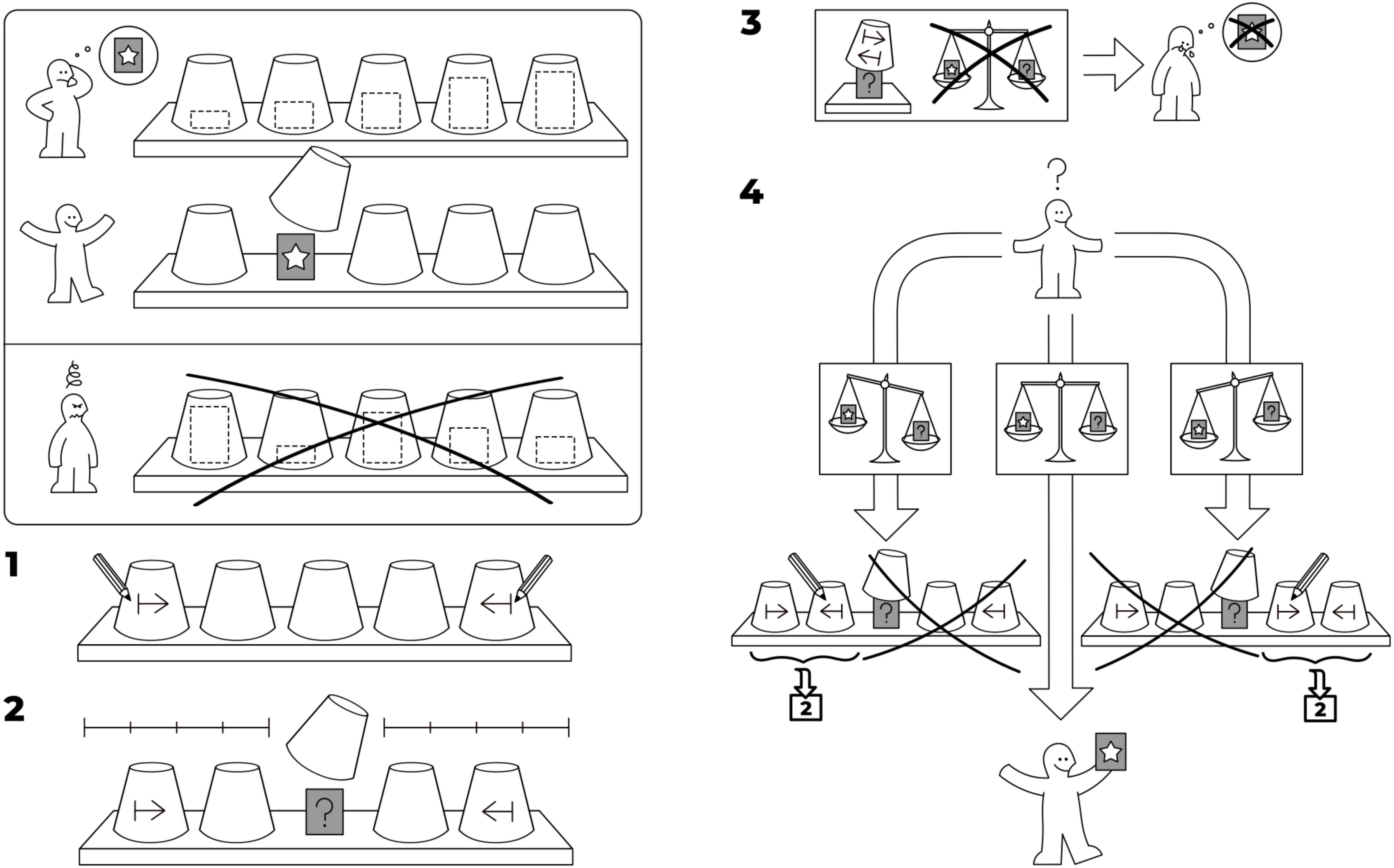
```
def binary_search_helper(seq, item, start, end):
    '''Recursive helper function used in binary search'''

    # base case 1
    if start > end:
        return False

    mid = (start + end) // 2
    mid_elem = seq[mid]

    if item == mid_elem:
        return True

    # recurse on left
    elif item < mid_elem:
        return binary_search_helper(seq, item, start, mid-1)

    # recurse on right
    else:
        return binary_search_helper(seq, item, mid+1, end)

def binary_search_improved(seq, item):

    return binary_search_helper(seq, item, 0, len(seq)-1)
```

Passing start/end indices as arguments avoids the need to splice!

BINÄRY SEARCH

idea-instructions.com/binary-search/

IDEA



More on Big Oh

Understanding Big-O

- Notation: n often denotes the number of elements (size)
- **Constant time** or $O(1)$: when an operation does not depend on the number of elements, e.g.
  - Addition/subtraction/multiplication of two values, or defining a variable etc are all constant time
- **Linear time** or $O(n)$: when an operation requires time proportional to the number of elements, e.g.:

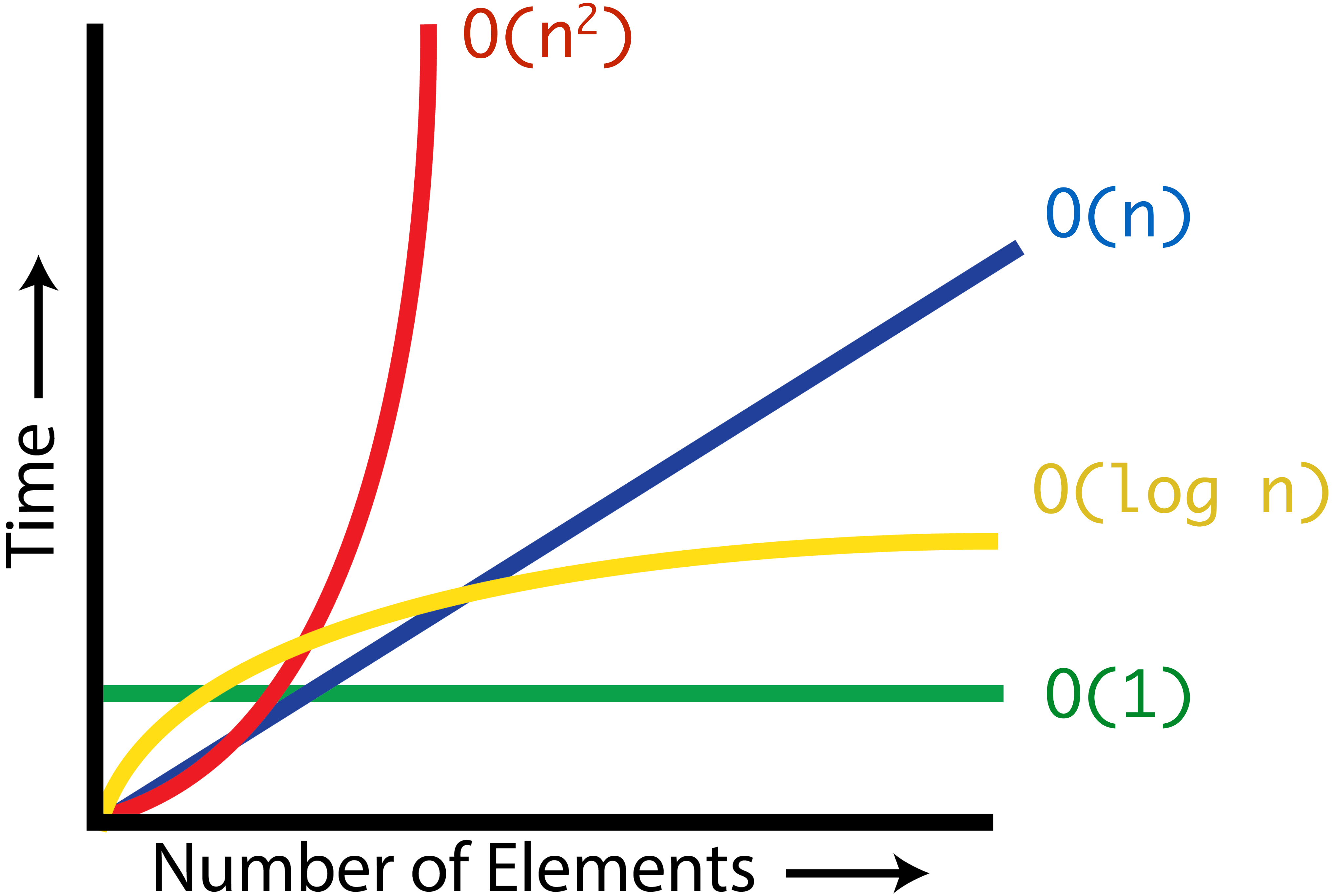
```
for item in seq:
   <do something>
```

- **Quadratic time** or $O(n^2)$: nested loops are often quadratic, e.g.,

```
for i in range(n):
   for j in range(n):
        <do something>
```

Big-O: Common Functions

- Notation: n often denotes the number of elements (size)
- Our goal: understand efficiency of some algorithms at a high level



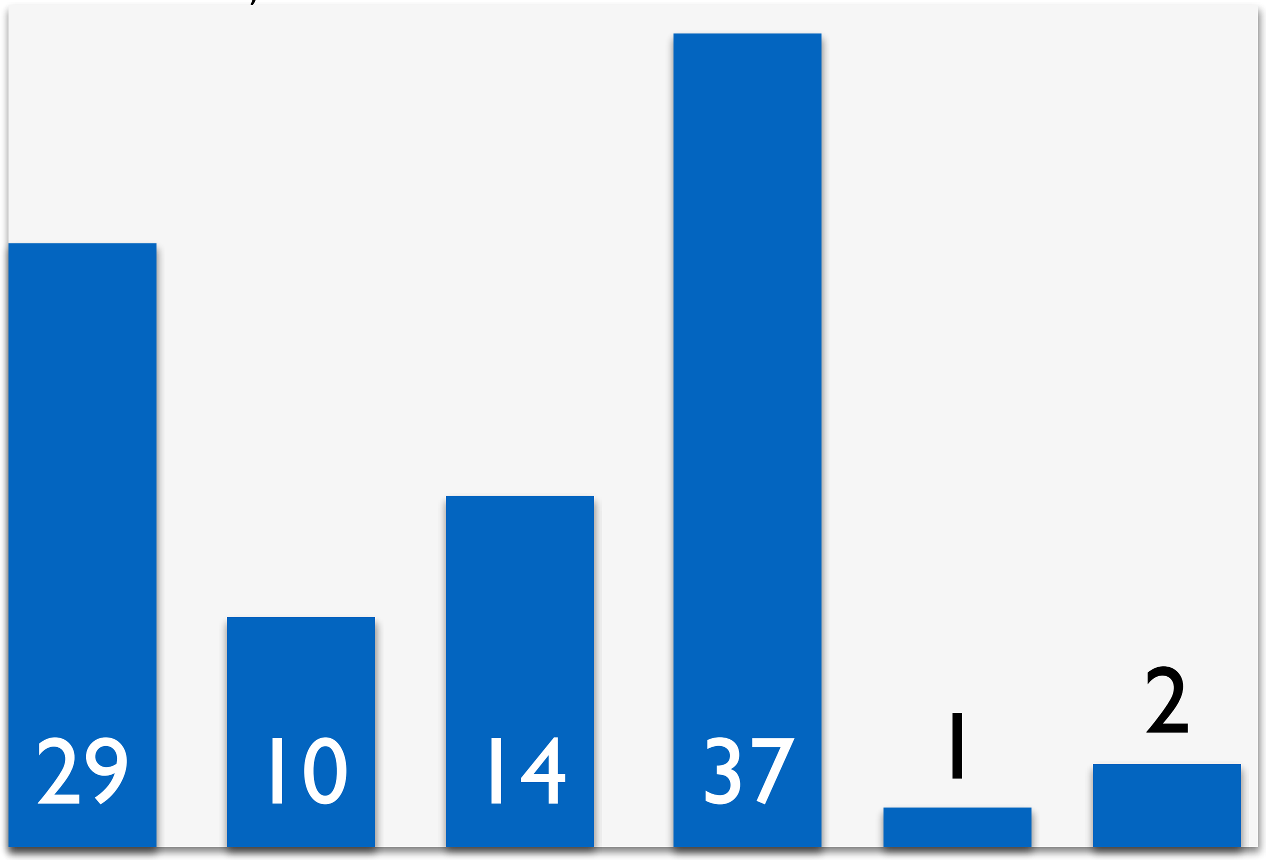
Sorting

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!

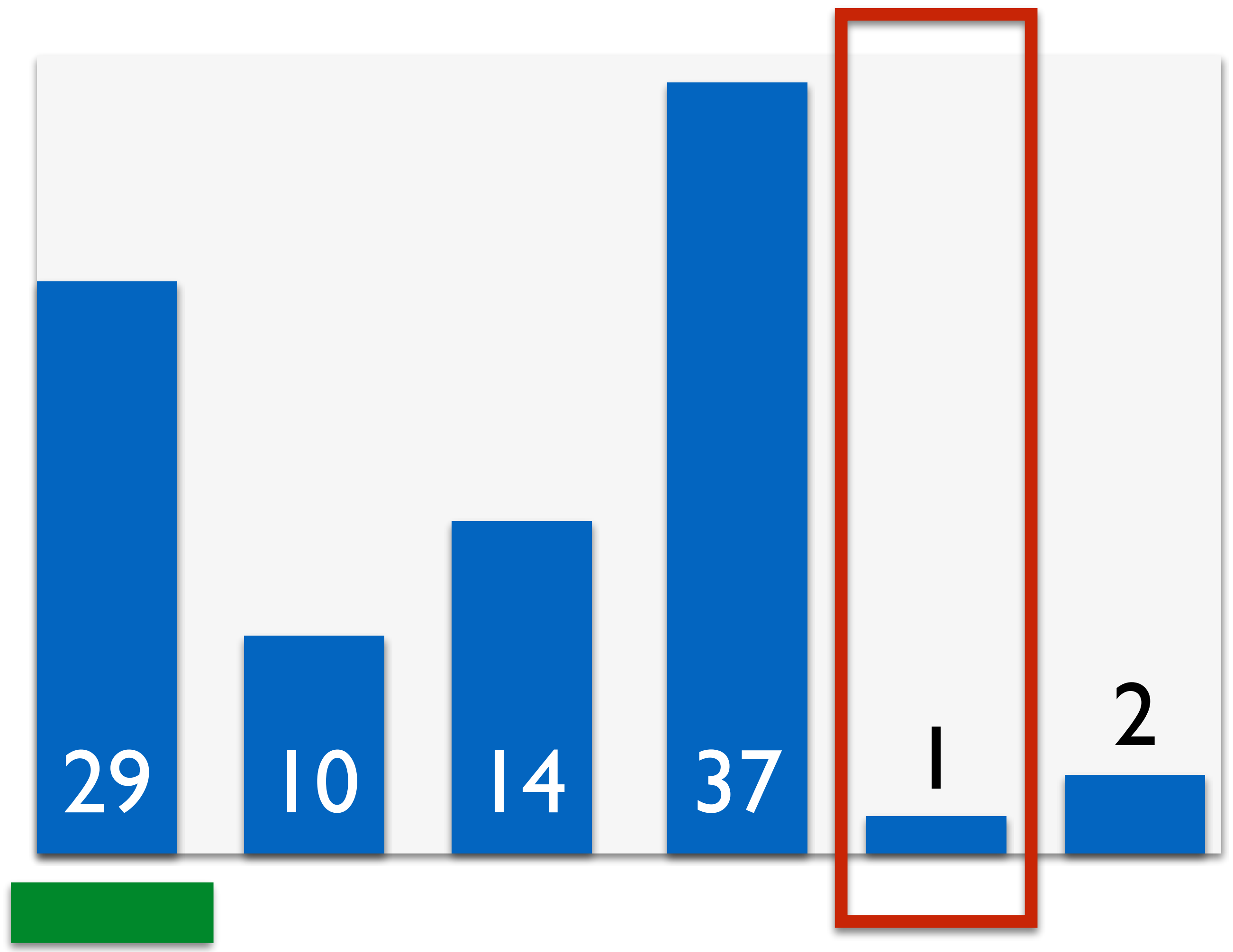
Selection Sort

- 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



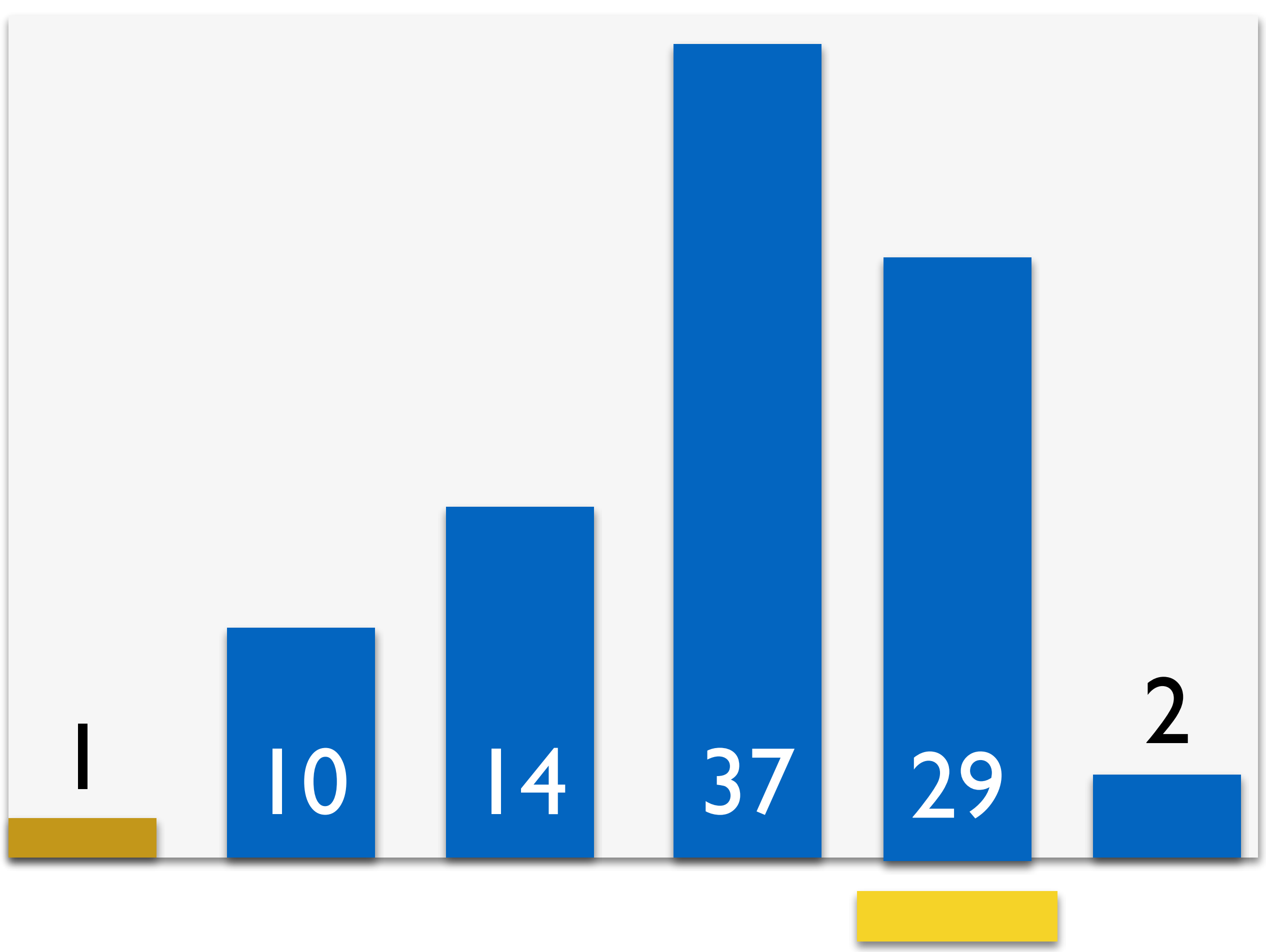
Selection Sort

- 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



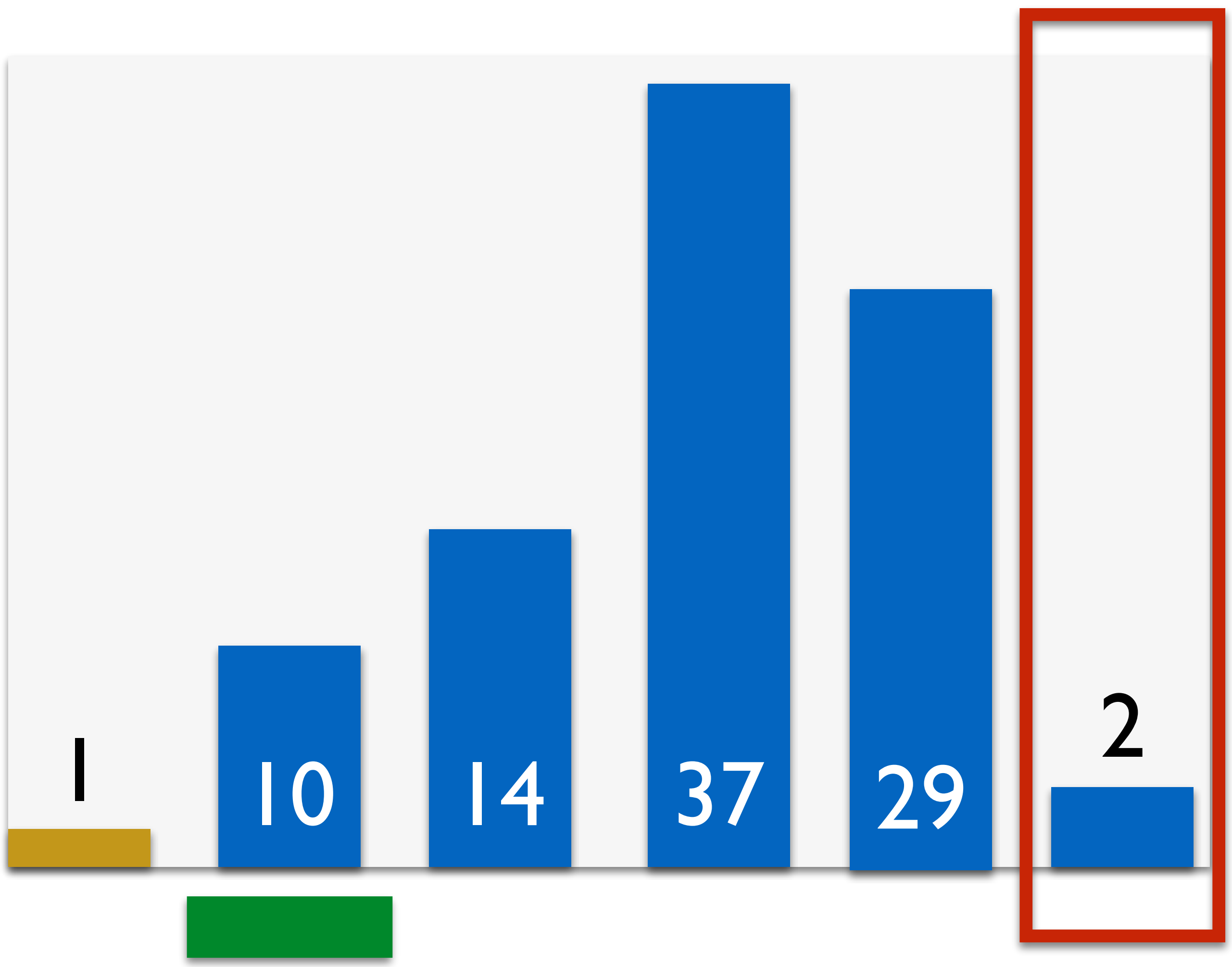
Selection Sort

- 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



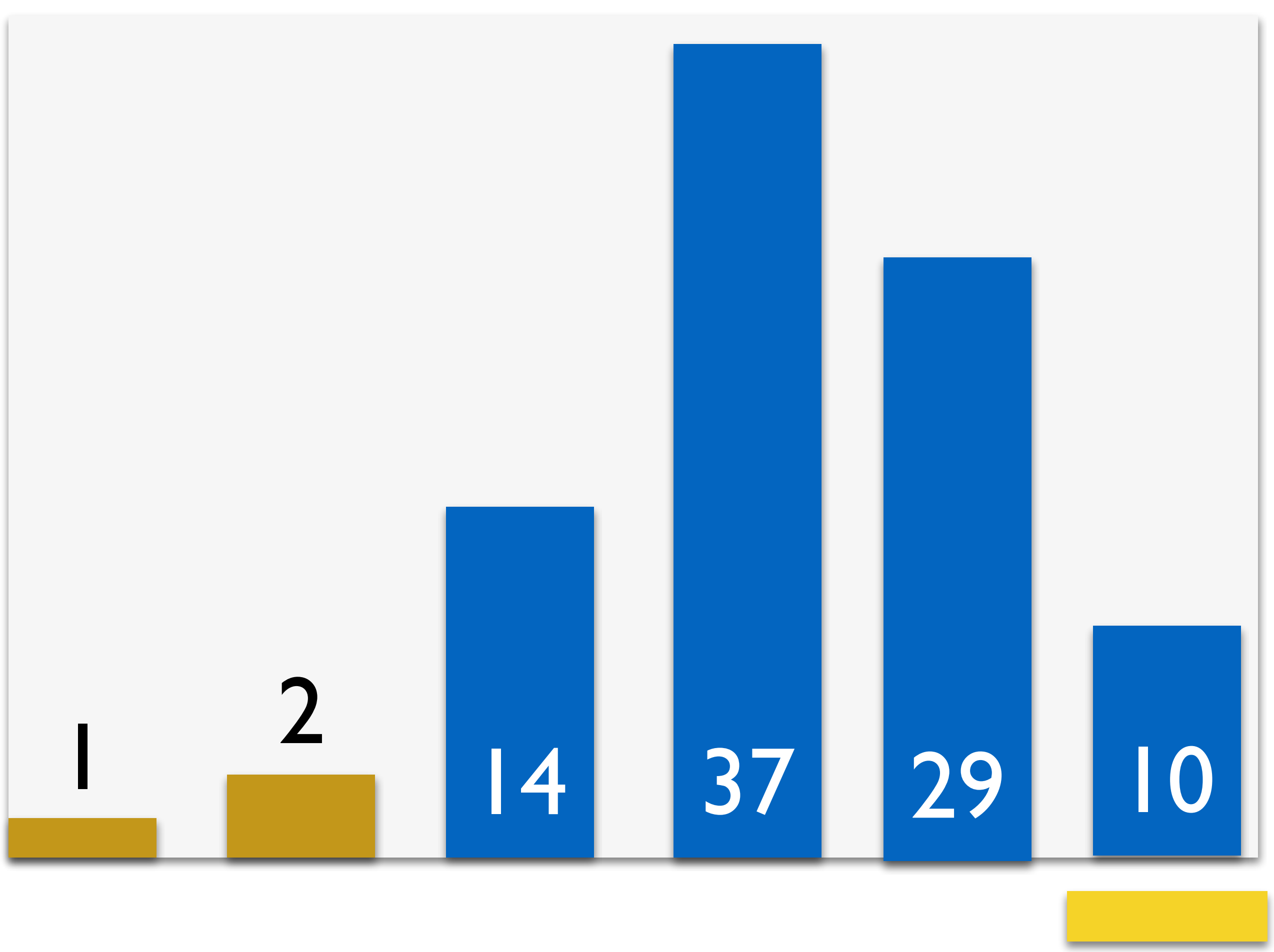
Selection Sort

- 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



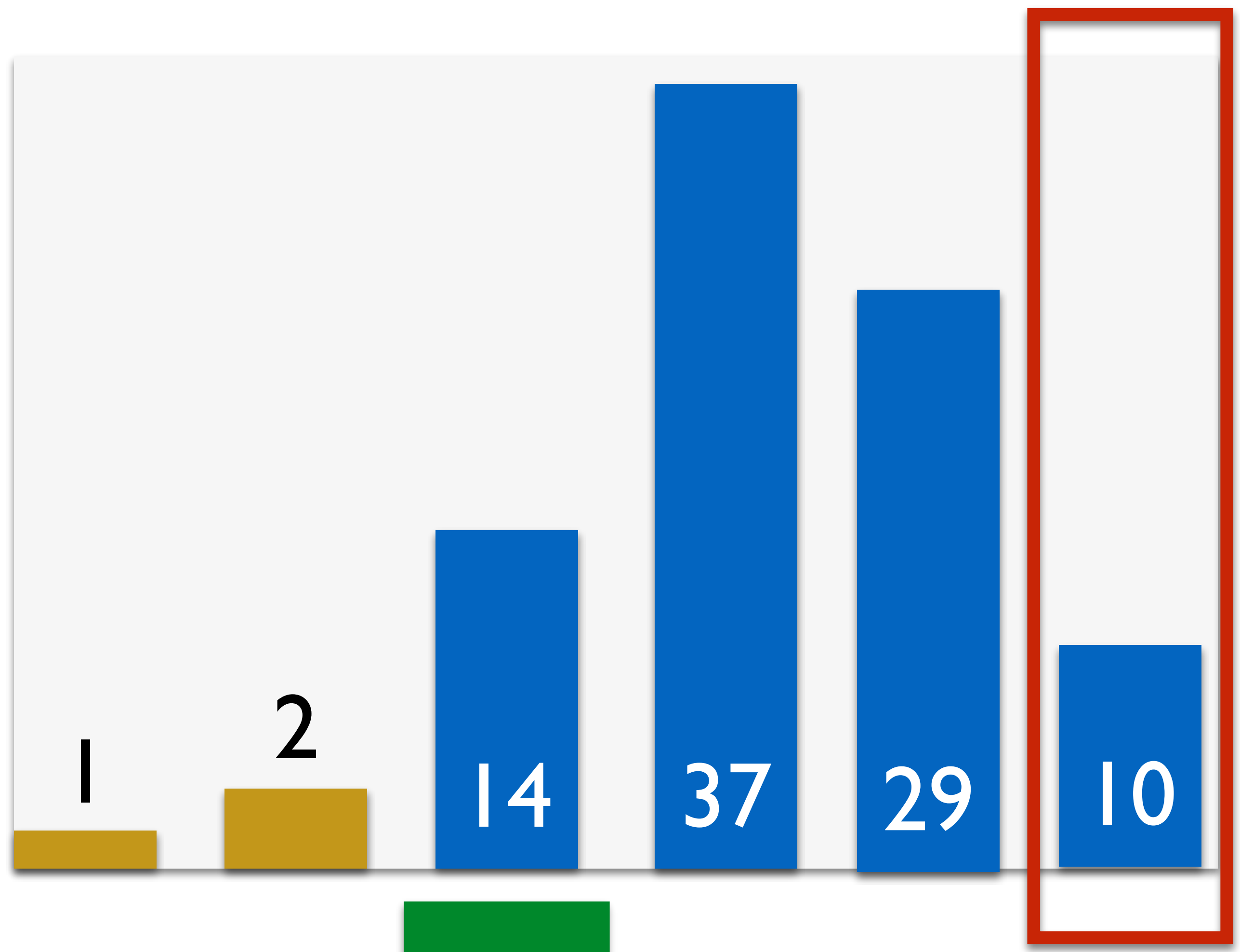
Selection Sort

- 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.



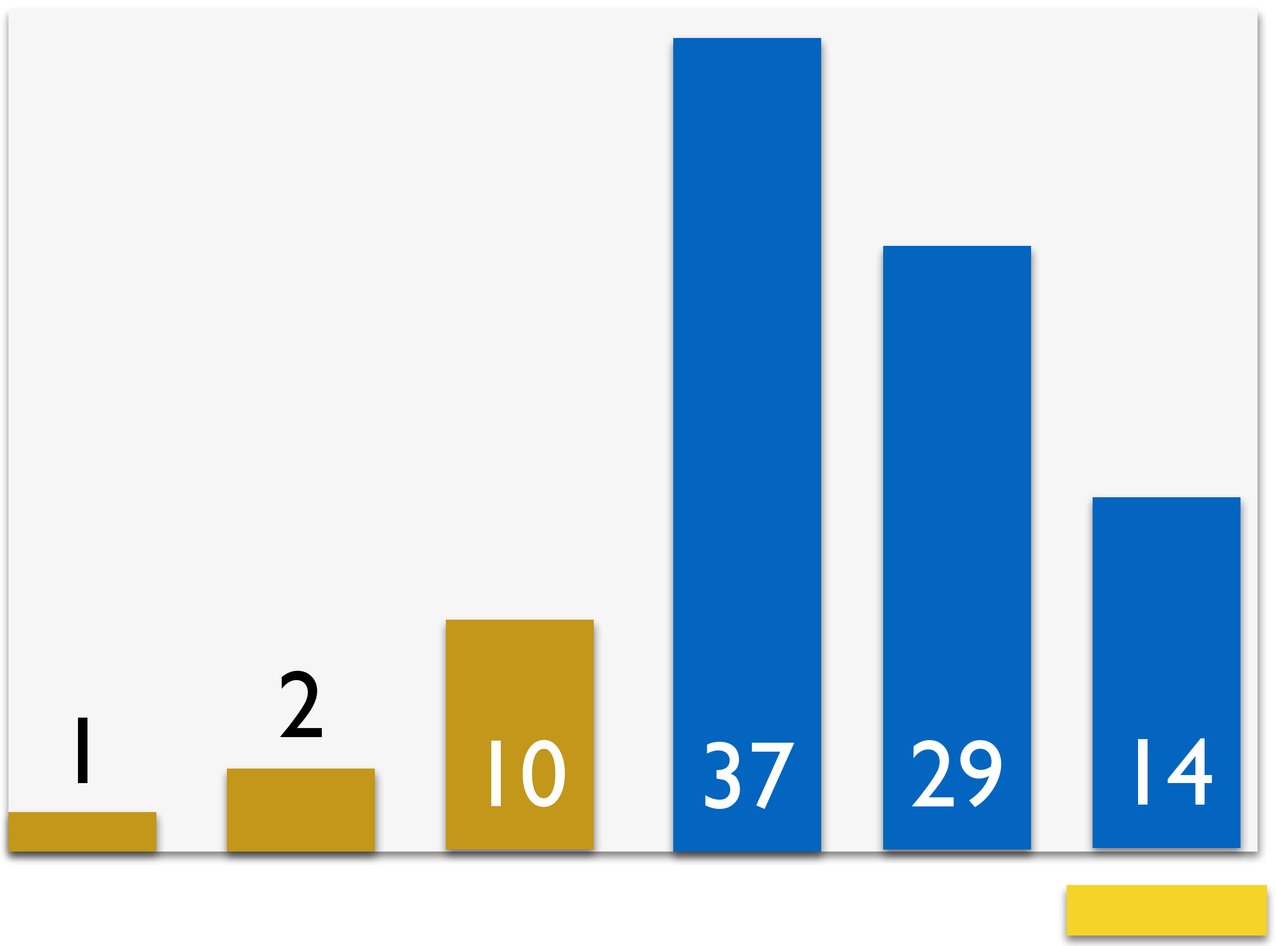
Selection Sort

- 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.



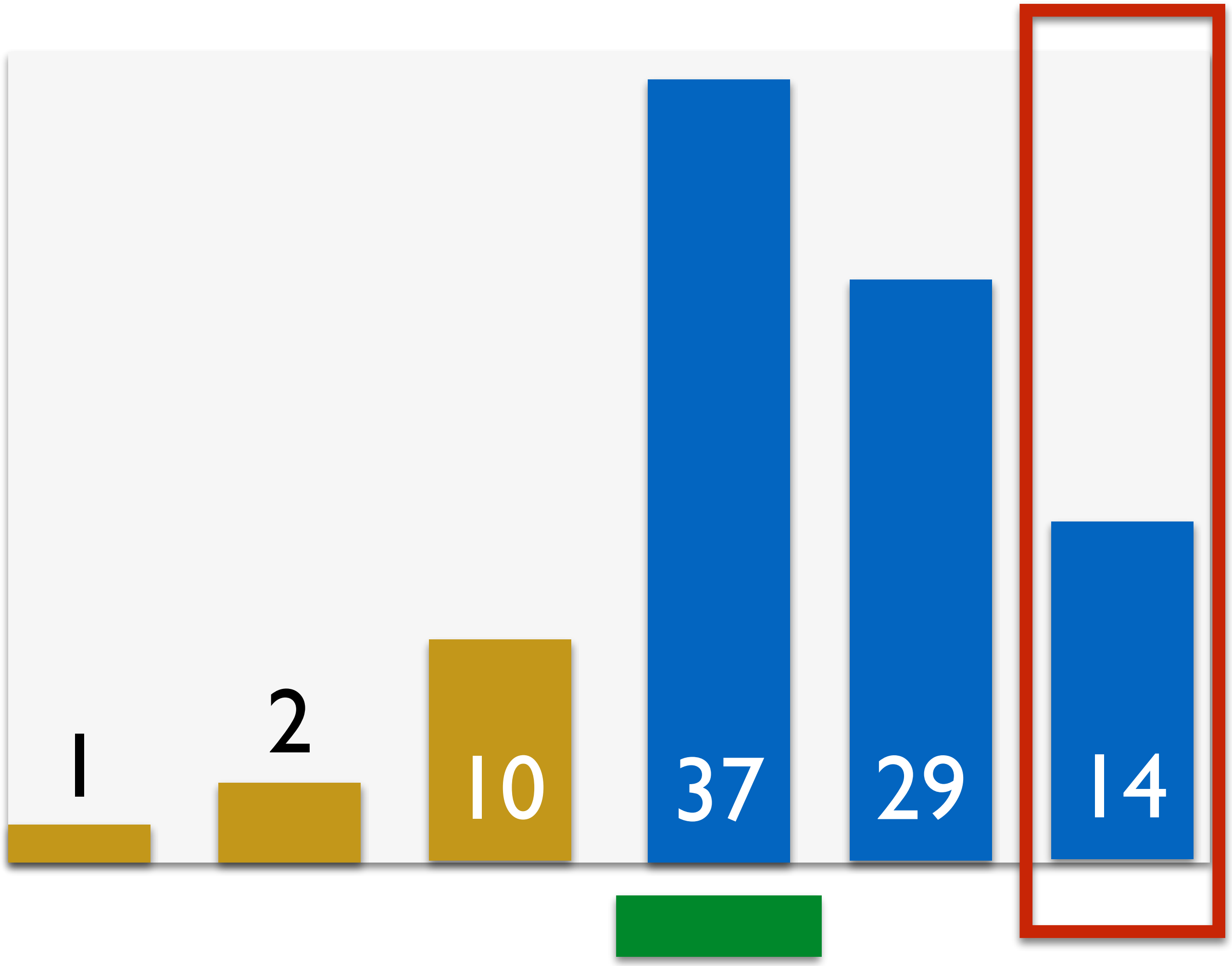
Selection Sort

- 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.



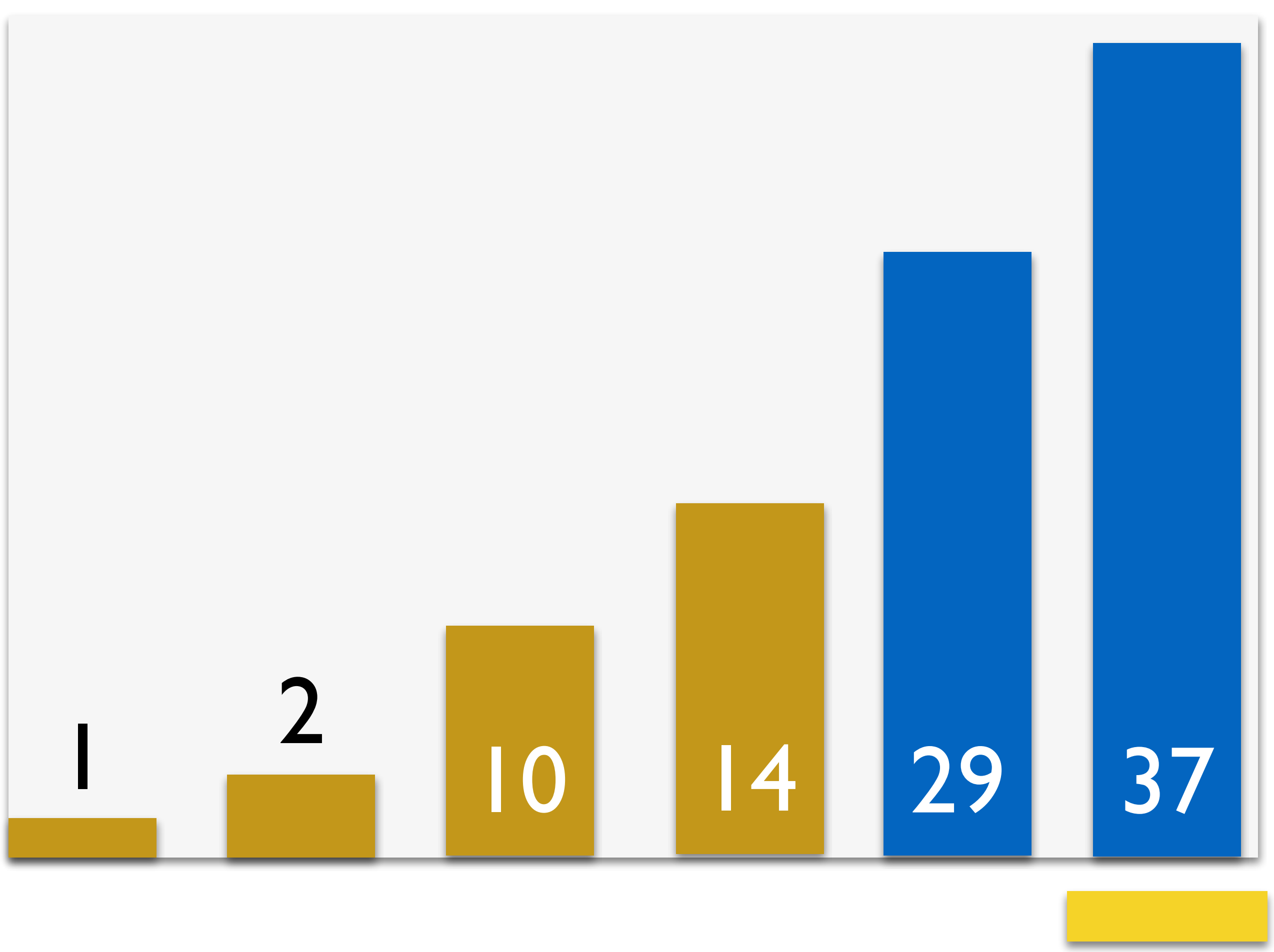
Selection Sort

- 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.



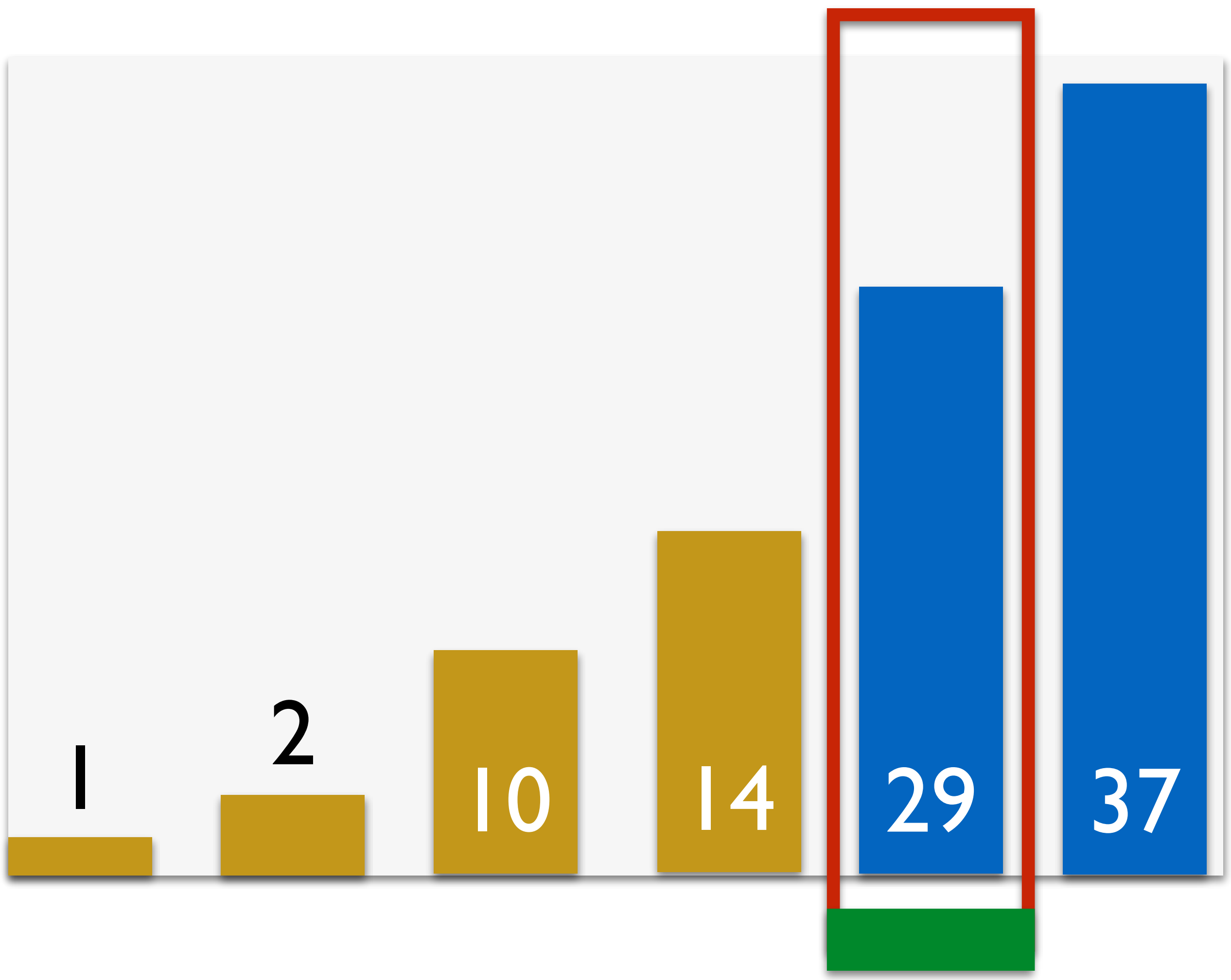
Selection Sort

- 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.



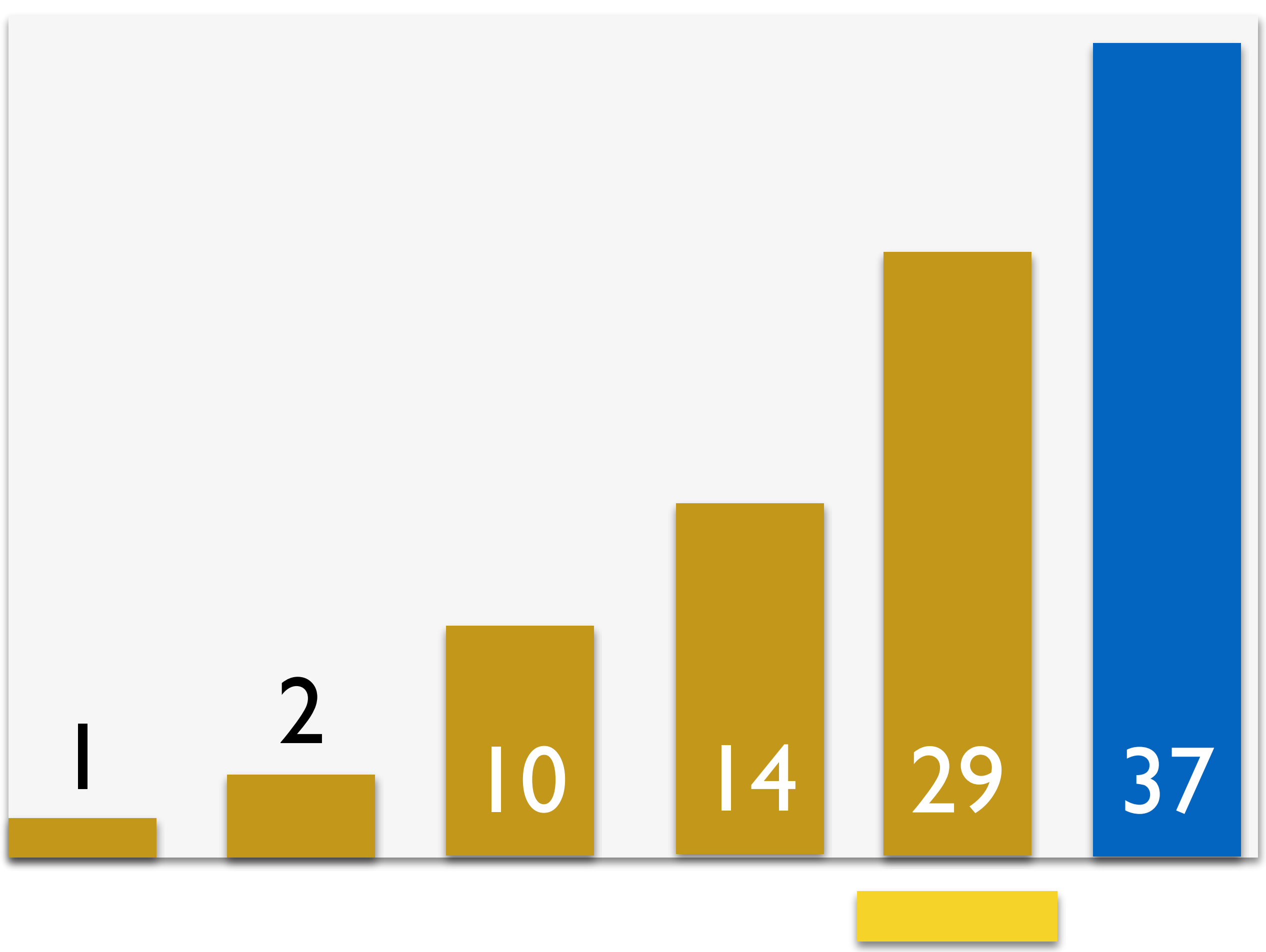
Selection Sort

- 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.



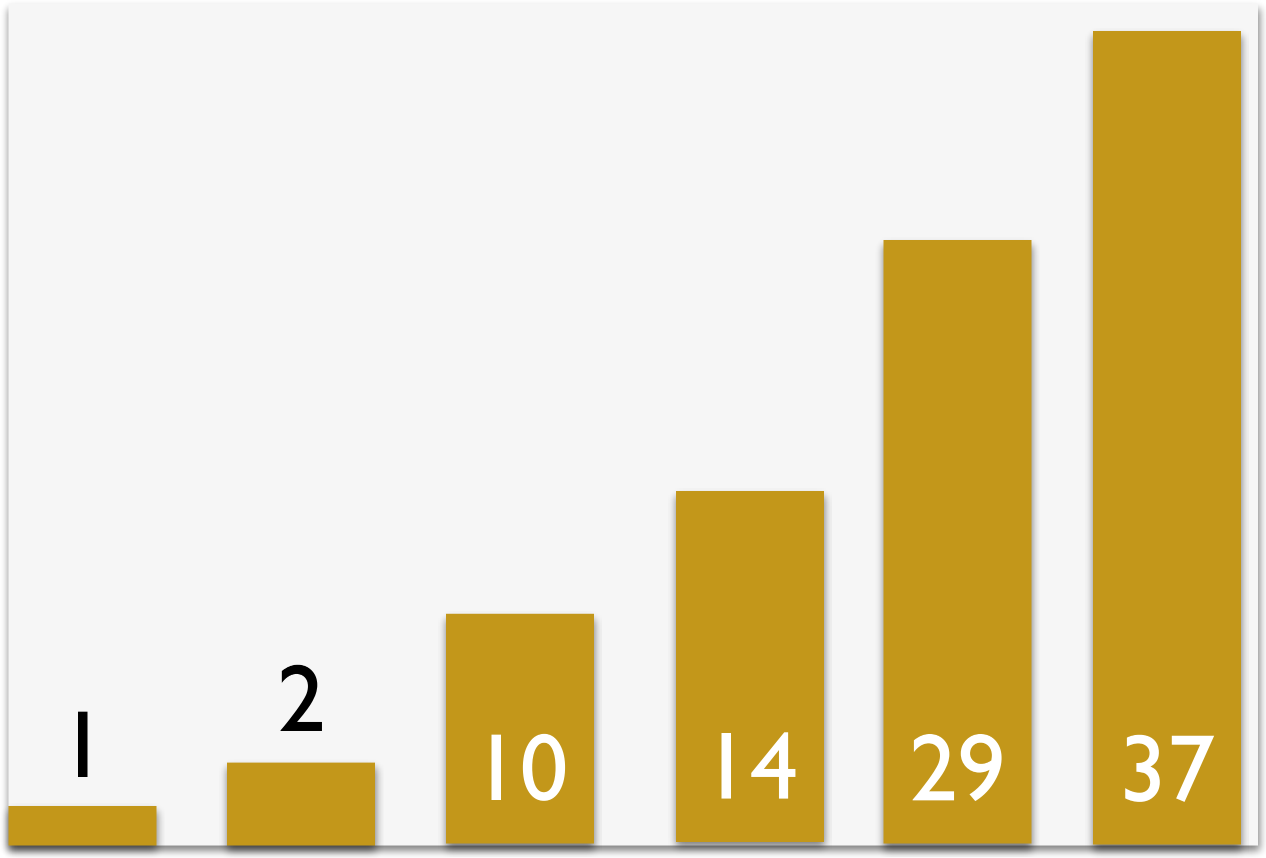
Selection Sort

- 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
Roma Folk Dance

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

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:]`
- **Neat trick:** how to swap values of variables `a` and `b` in one line?
    - in-line "tuple" 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

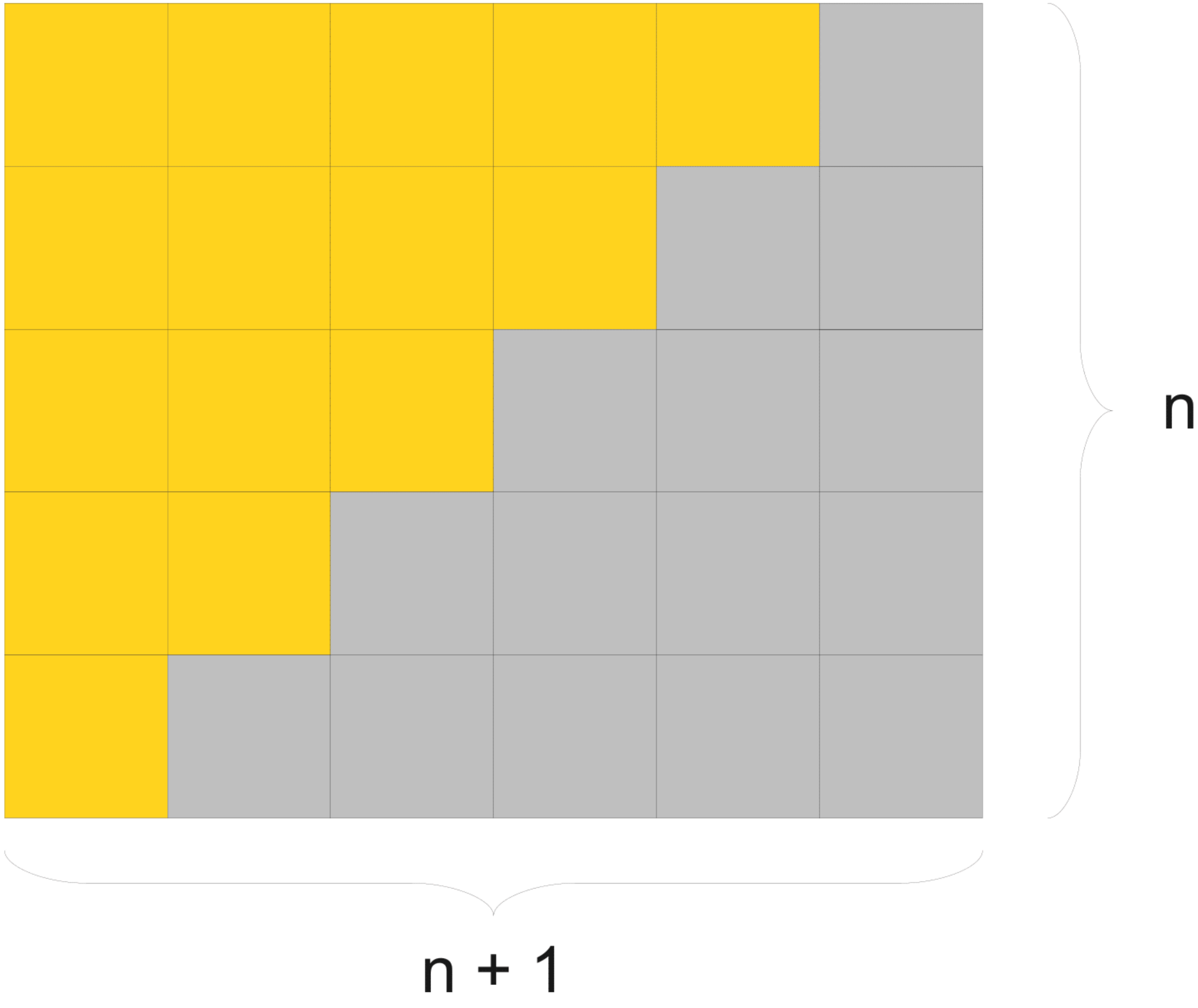
```
def selection_sort(my_lst):
    """Selection sort of a given mutable sequence my_lst,
    sorts my_lst by mutating it.  Uses selection sort."""


    # 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.)

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



Selection Sort Analysis: Algebraic

$$
\begin{aligned}
& S = n + (n-1) + (n-2) + \cdots + 2 + 1 \\
+ \quad & S = 1 + 2 + \cdots + (n-2) + (n-1) + n \\
\hline
& 2S = (n+1) + (n+1) + \cdots + (n+1) + (n+1) + (n+1)
\end{aligned}
$$

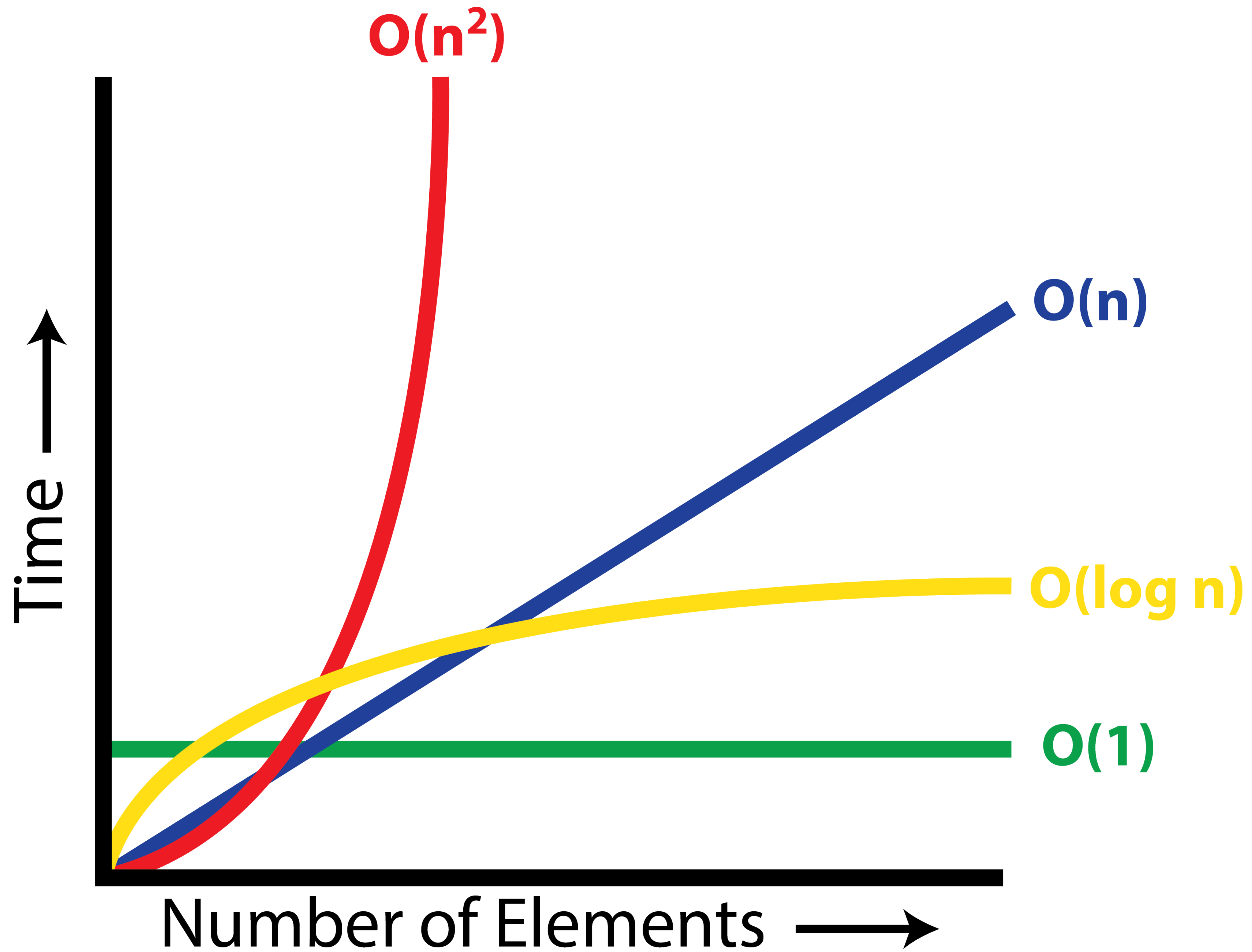
$$2S = (n+1) \cdot n$$

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

- 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!



The end!