Middlebury

CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 17: Searching and Sorting

`Python` uses `sys.maxsize` as the maximum size of a container (and hence maximum index).

On most modern computers, this max size is represented by an integer with 64 bits. So what's `sys.maxsize`?

```
>>> import sys
>>> sys.maxsize
9223372036854775807
```

$$2^{63} - 1$$

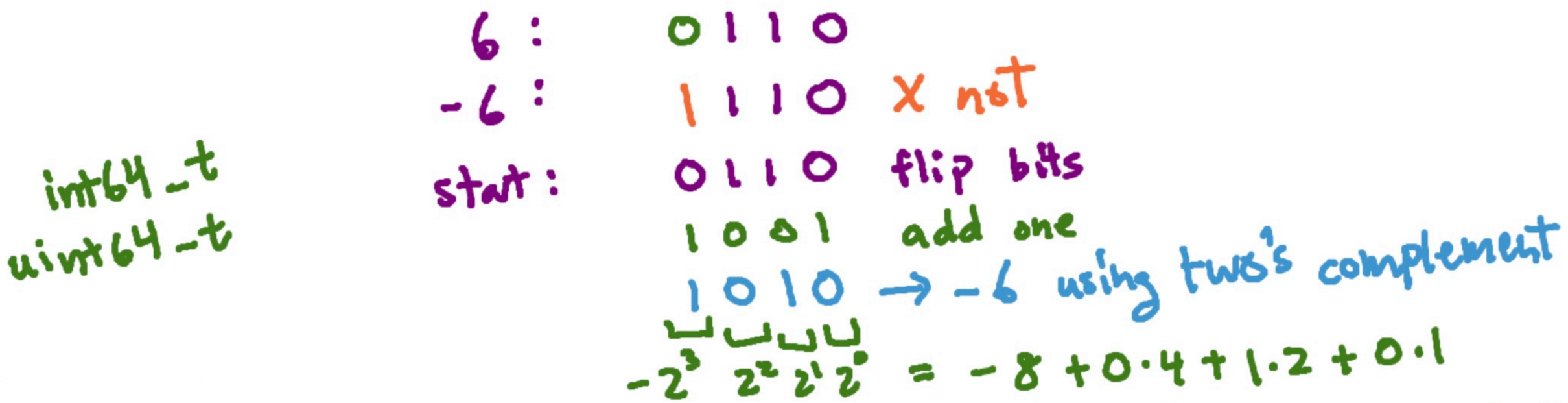
- Account for negative indices.
- 1 bit used for sign.

But computers use "two's complement" to encode negative and positive numbers.

Use the most-significant (leftmost) bit (MSB) to represent the sign (0: positive, 1: negative).

- But we don't just use this sign bit with the unchanged number (e.g. `-6` would not be `1110`).
- Instead, start with absolute value of number (e.g. `6` is `0110`).
- Then flip all the bits: `1001`.
- Add one: `1010`. This is the two's-complement representation of `-6`.

Check? Convert as usual but use `-` on MSB term.



Representing floating-point numbers.

```
>>> 0.1 + 0.2 <= 0.3
False
```

64-bit floating-point number:

- 1 bit for sign s
- 11 bits for exponent e (offset from some bias e_0)
- 52 bits for mantissa m

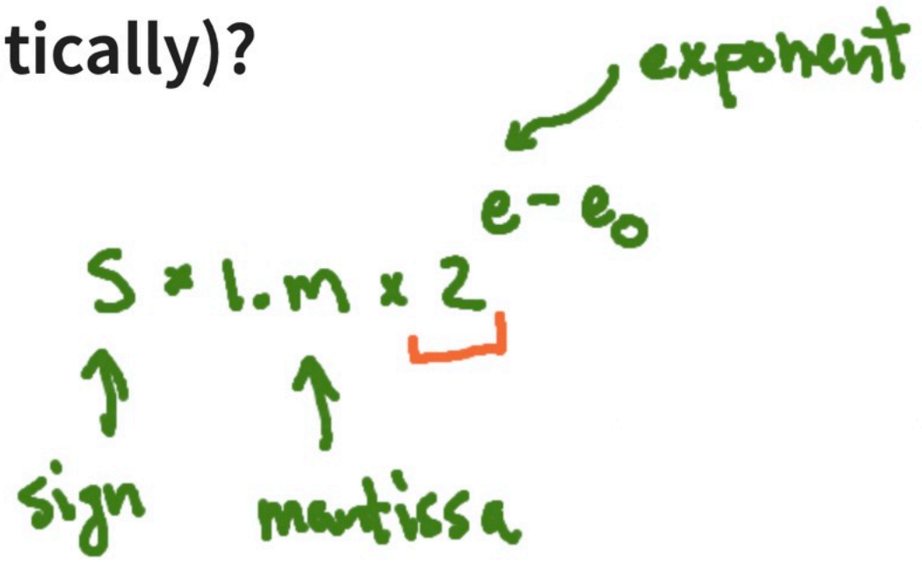
$$s * 1.m * 2^{e-e_0}$$

Question 1: Will `0.125 + 0.25 <= 0.375` evaluate to `True` (as expected mathematically)?

A. Yes

B. Possibly, I'd have to try it out.

C. No



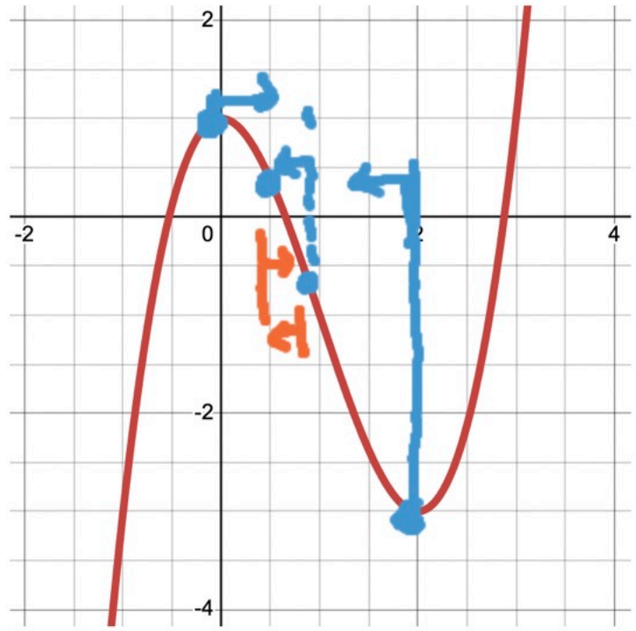
$0.125 \quad 2^{-3}$

$0.25 \quad 2^{-2}$

Goals for today

- Describe algorithms for linear and binary search.
- Describe algorithms for selection sort, insertion sort and merge sort.
- Recall the asymptotic runtime complexity of different search and sort algorithms.

Why is searching important?



bisection method

What if we want the root of $f(x) = x^3 - 3x^2 + 1$ in the interval $[0, 2]$?

Determining the index of an `item` in a list with *linear search*.

```
def linear_search(a_list, item):
    """Return index of item in a_list or None if not found

    Args:
        a_list: A sequence of comparable values
        item: A value to search for in a_list

    Returns:
        Index of item or None is not found
    """
    for i in range(len(a_list)):
        if item == a_list[i]:
          return i
    return None
```

- [31, 98, 80, 87, 94, 22, 32, 35]
- [11, 14, 21, 23, 36, 49, 51, 79]

find 94

find 14

complexity

O(n) worst

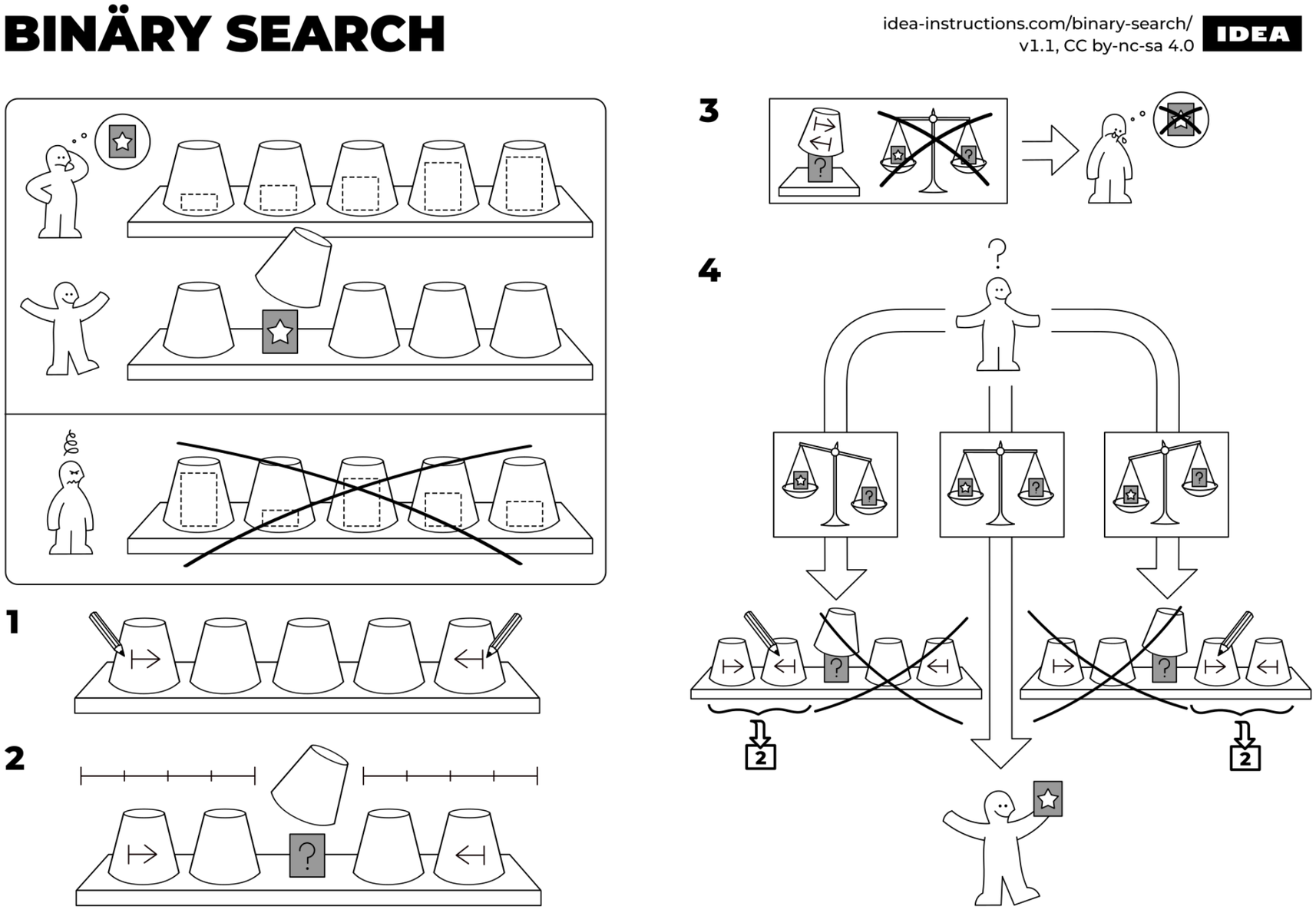
O(n) average

O(1) best

item we are looking for is at the beginning

What if the list is sorted? Can we use this to our advantage?

Yes: *binary search*.



What if the list is sorted? Can we use this to our advantage?

Yes: *binary search*.

$\log_2(n) = \frac{\log(n)}{\log(2)}$

```
def binary_search(a_list, item, lo, hi):
    """
    Return index of item in a sorted a_list or None if not found
    Args:
       a_list: A sequence of values sorted in ascending order
       item: A value to search for in a_list
       lo: Inclusive start index to search in a_list
       hi: Inclusive end index to search in a_list
    Returns:
       Index of item or None if not found
    """
    if lo > hi:
        return None
    else:
        middle = (lo + hi) // 2
        middle_elem = a_list[middle]
        if item == middle_elem:
          return middle
        elif item < middle_elem:
          return binary_search(a_list, item, lo, middle - 1)
        else:
          return binary_search(a_list, item, middle + 1, hi)
```

0 1 2 3 4 5 6 7

[2, 3, 7, 9, 10, 12, 18, 19]

find 9 middle = (0+7)//2 = 3

best case: O(1)

find 2

0 1 2

[2, 3, 7]

lo = 0 hi = 2

middle = (0+2)//2 = 1

0

[2]

lo = 0 hi = 0

middle = (0+0)//2 = 0

return 0

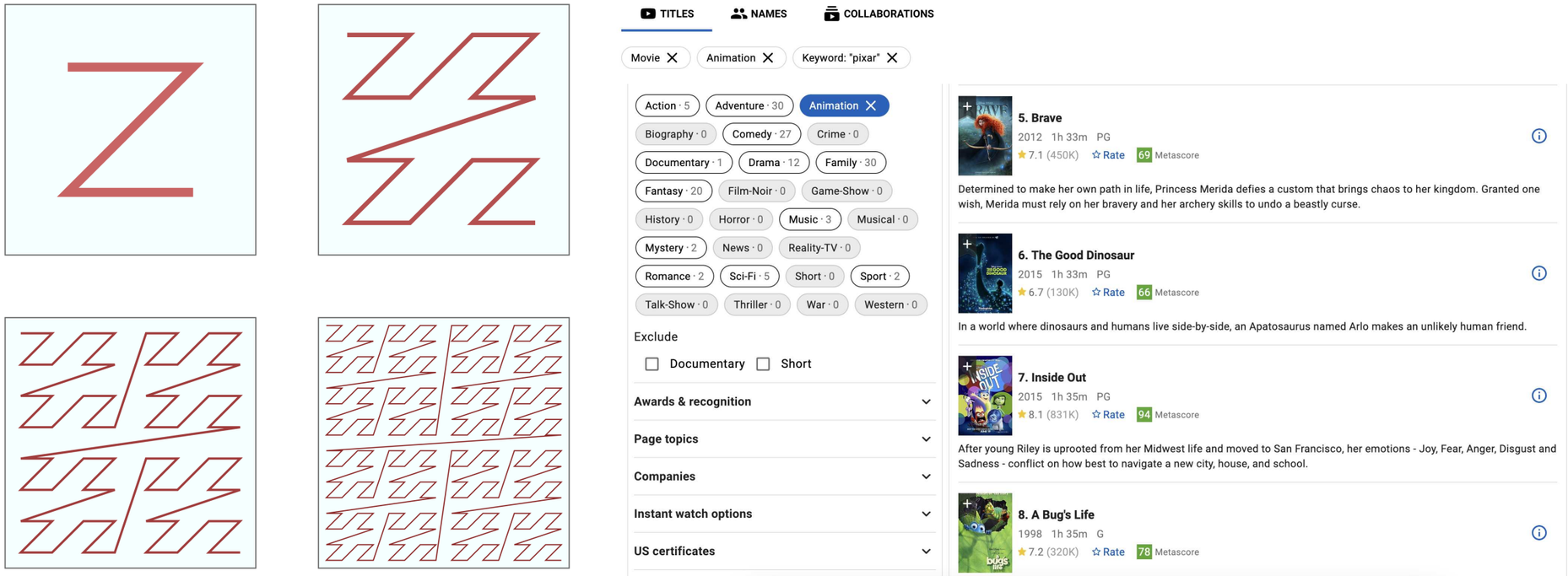
length n

how many times does the list get halved until we get to a sublist of length 1?

when is $\frac{n}{2^k} = 1$ → $k = \log_2 n$

O(logn) worst and average

How do we get sorted lists in the first place? And what are other applications of sorting?

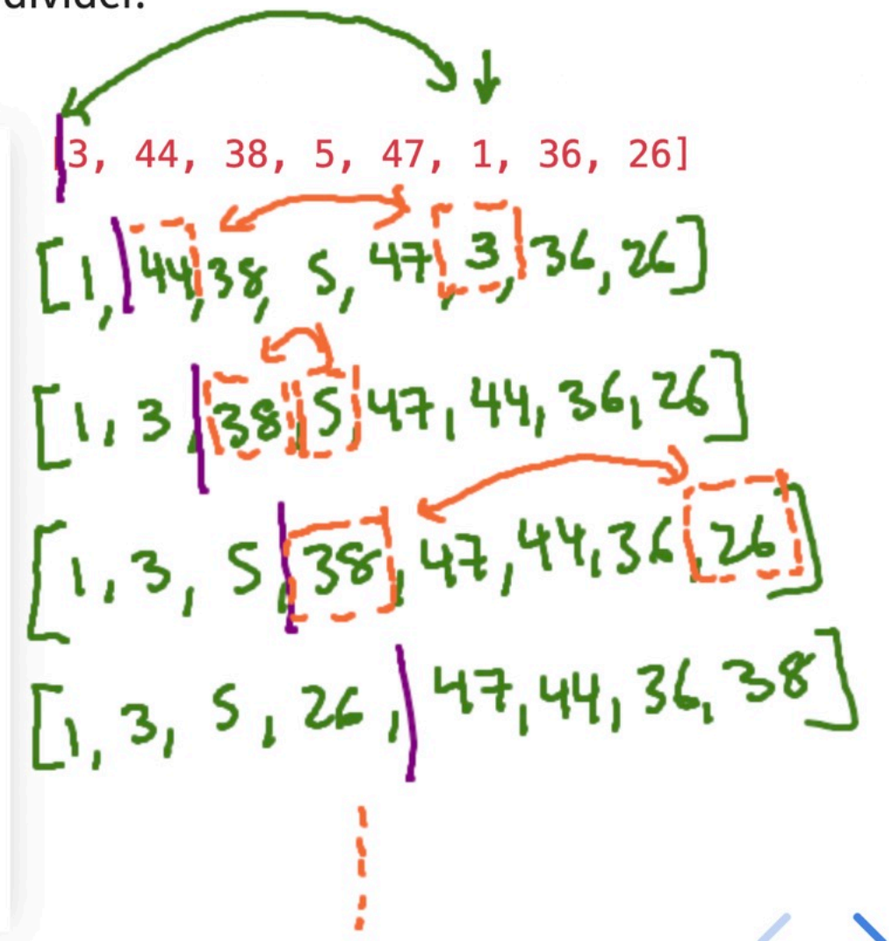


Sorting Algorithm #1 (Selection Sort):

Main idea: maintain sorted elements on the left (of some imaginary divider) and unsorted elements on the right.

1. Find the smallest element in unsorted part.
2. Swap this smallest element with the element to the right of this divider.
3. Move the divider to the right (by one) and go back to Step 1.

```python
def selection_sort(a_list):
    """
    Sort list in place using the selection sort algorithm

    Args:
        a_list : List to sort in place
    """
    # In each iteration, find the next smallest element in the list
    # and swap it into the appropriate place
    for i in range(len(a_list)):
        # Find the index of the smallest value from i onwards
        min_index = i
        min_value = a_list[i]

        for j in range(i+1, len(a_list)):
            if a_list[j] < min_value:
                min_index = j
                min_value = a_list[j]

        # Swap i and min_index
        a_list[i], a_list[min_index] = a_list[min_index], a_list[i]
```

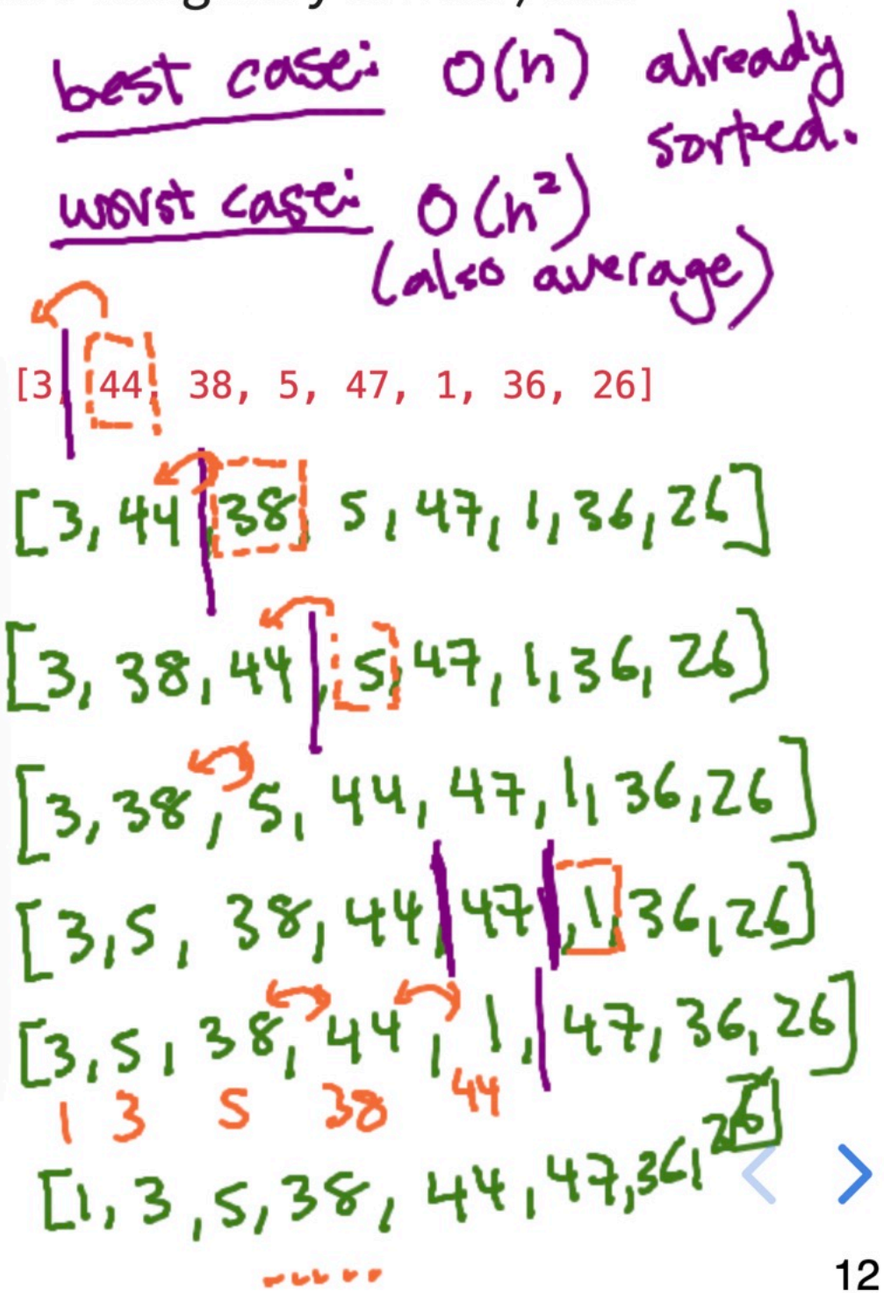


Sorting Algorithm #2 (Insertion Sort):

Main idea: maintain sorted elements on the left (of some imaginary divider) and unsorted elements on the right.

1. Look at first element in unsorted part (to the right of divider).
2. Iteratively swap this into the correct place in the sorted part.
3. Move the divider to the right (by one) and go back to Step 1.

```python
def insertion_sort(a_list):
    """
    Sort list in place using the insertion sort algorithm

    Args:
        a_list : List to sort in place
    """
    for i in range(1,len(a_list)):
        # Values at [0,i-1] are sorted already
        # Shift up all values in [0,i-1] greater than a_list[i]
        value = a_list[i]
        index = i

        while index > 0 and a_list[index-1] > value:
            a_list[index] = a_list[index-1]
            index -= 1
        # Now insert value (old a_list[i]) in its proper place
        a_list[index] = value
        # Now everything from 0...i is sorted
```



Runtime analysis of selection and insertion sort.

```python
def selection_sort(a_list):
    for i in range(len(a_list)):
        min_index = i
        min_value = a_list[i]

        for j in range(i+1, len(a_list)):
            if a_list[j] < min_value:
                min_index = j
                min_value = a_list[j]

        a_list[i], a_list[min_index] = a_list[min_index], a_list[i]
```

$\leq 1+2+3+\dots+(n-1)+n$

$i=0 \quad + \; n-1 \quad < \quad \frac{n(n+1)}{2}$

$i=1 \quad + \; n-2 \quad <$

$i=2 \quad + \; n-3 \quad < \quad O(n^2)$

$\vdots \quad + \quad \vdots$

$i=n-2 \quad + \; 1 \quad <$

$i=n-1 \quad + \; 0 \quad <$

best, worst, average.

```python
def insertion_sort(a_list):
    for i in range(1,len(a_list)):
        value = a_list[i]
        index = i

        while index > 0 and a_list[index-1] > value:
            a_list[index] = a_list[index-1]
            index -= 1
        a_list[index] = value
```

?

$i=0 \quad + \; 0 \quad <$

$i=1 \quad + \; 1 \quad <$

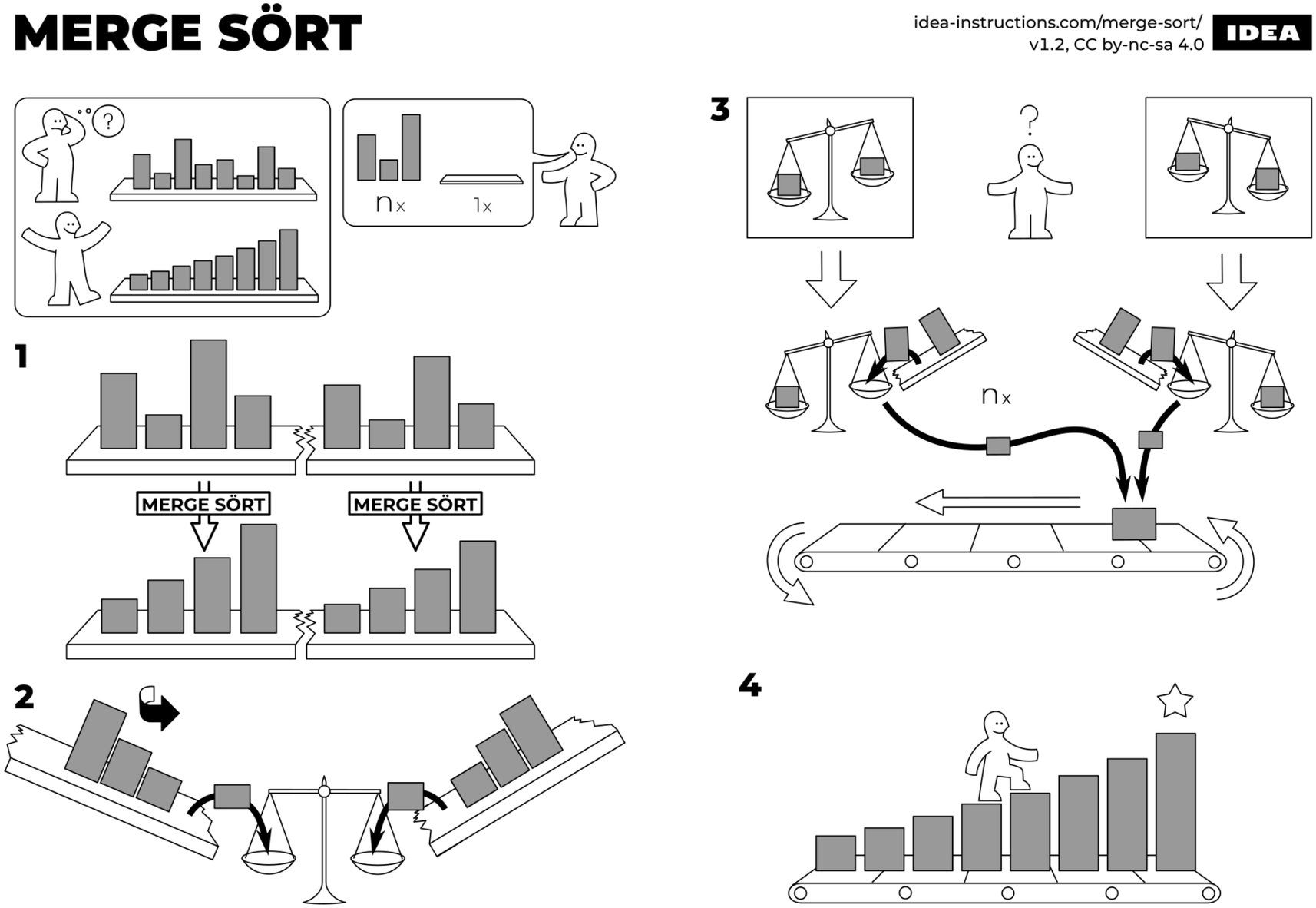
$\vdots$

$i=n-1 \quad + \; n-1$

$O(n^2)$ worst, average

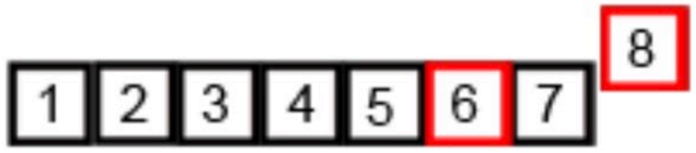
$O(n)$ best (sorted)

Sorting Algorithm #3: `merge_sort`.



Sorting Algorithm #3: `merge_sort`.

1. Divide input list into two sublists.
2. Call `merge_sort` on each sublist.
3. Merge the result.



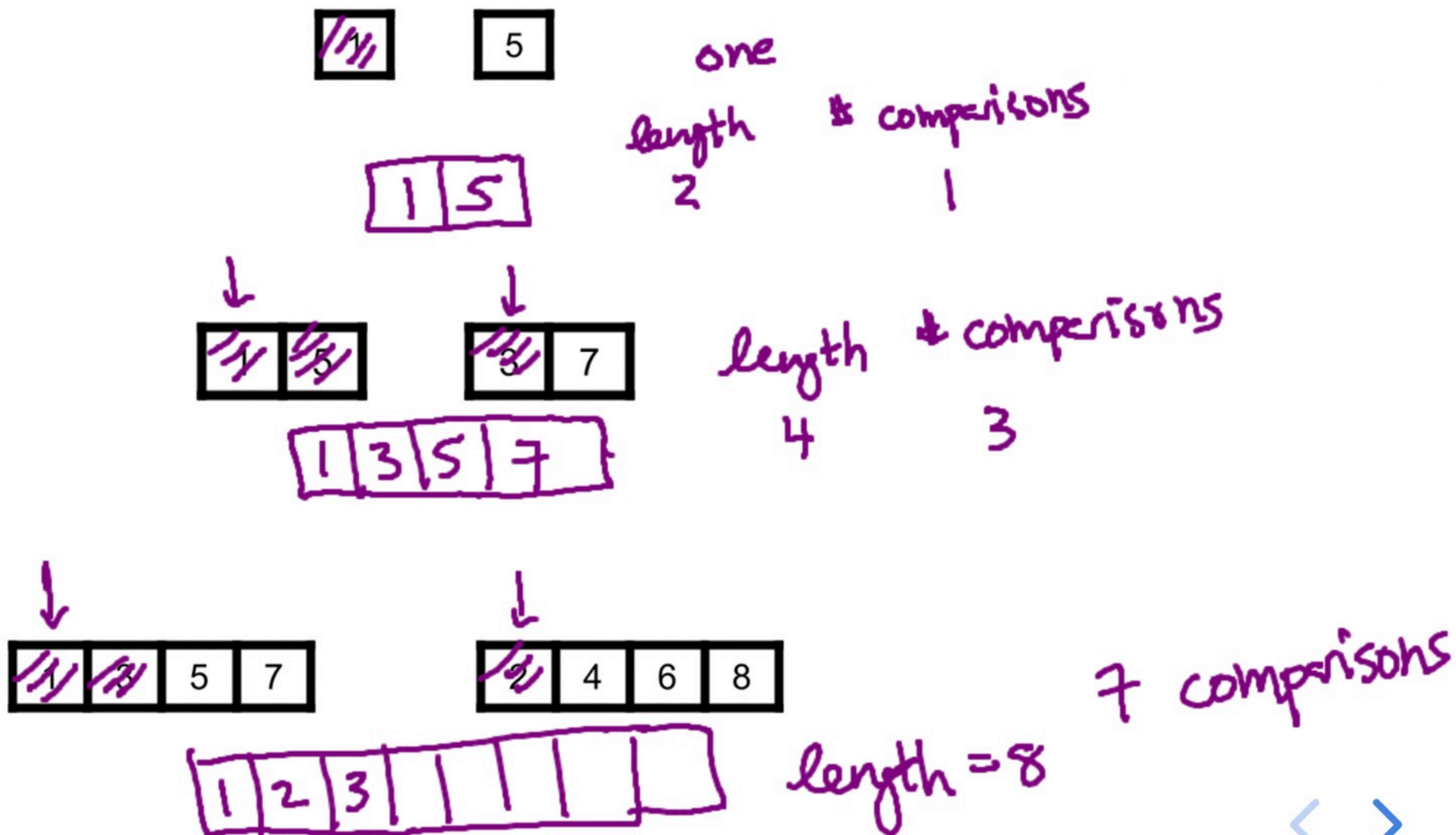
How many times do we need to halve a list of length n until we get a sublist of length 1?

(from Wikipedia)

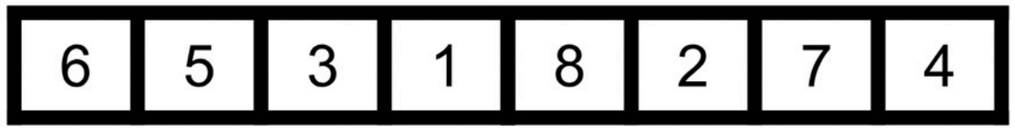
$\frac{n}{2^k} = 1$ $\quad k = \log_2 n$

Analyzing the merge in `merge_sort`.

How many times do you need to ask: *"which leading element is smallest?"*
(when merging two sublists)



Analyzing `merge_sort`: adding up the number of < from each level.



comparisons to merge at each level is n-1 ≈ n

#levels = $\log_2 n$

6, 5, 3, 1 8, 2, 7, 4

6, 5 3, 1 8, 2 7, 4

6 5 3 1 8 2 7 4

5,6 1,3 2,8 4,7

1,3,5,6 2,4,7,8

1, 2, 3, 4, 5, 6, 7, 8

$\frac{n}{4} \times 4 = n$

$\frac{n}{2} \times 2 = n$

$n = 8$

O(n log n)

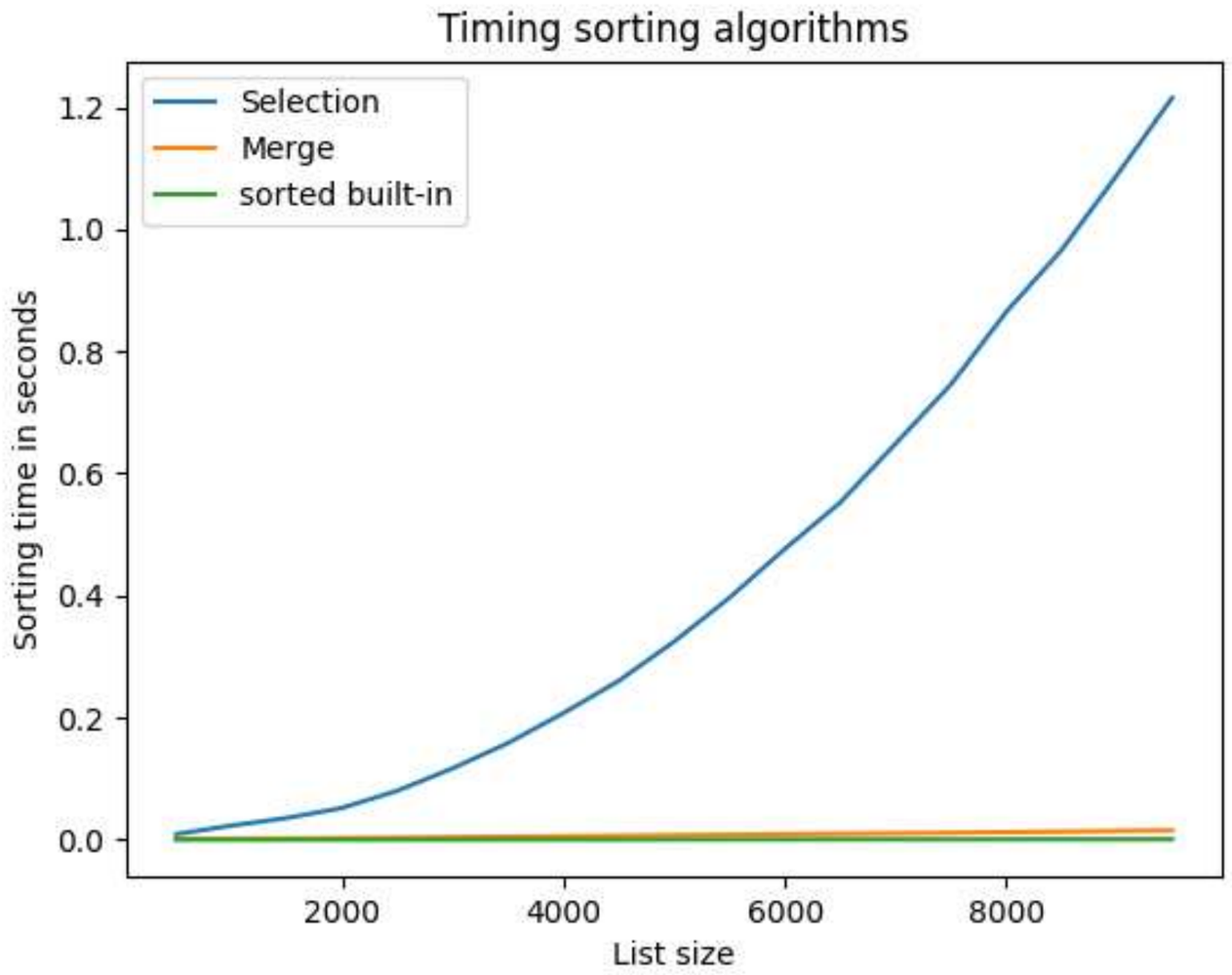
best, worst, average.

Possible implementation of `merge_sort`.

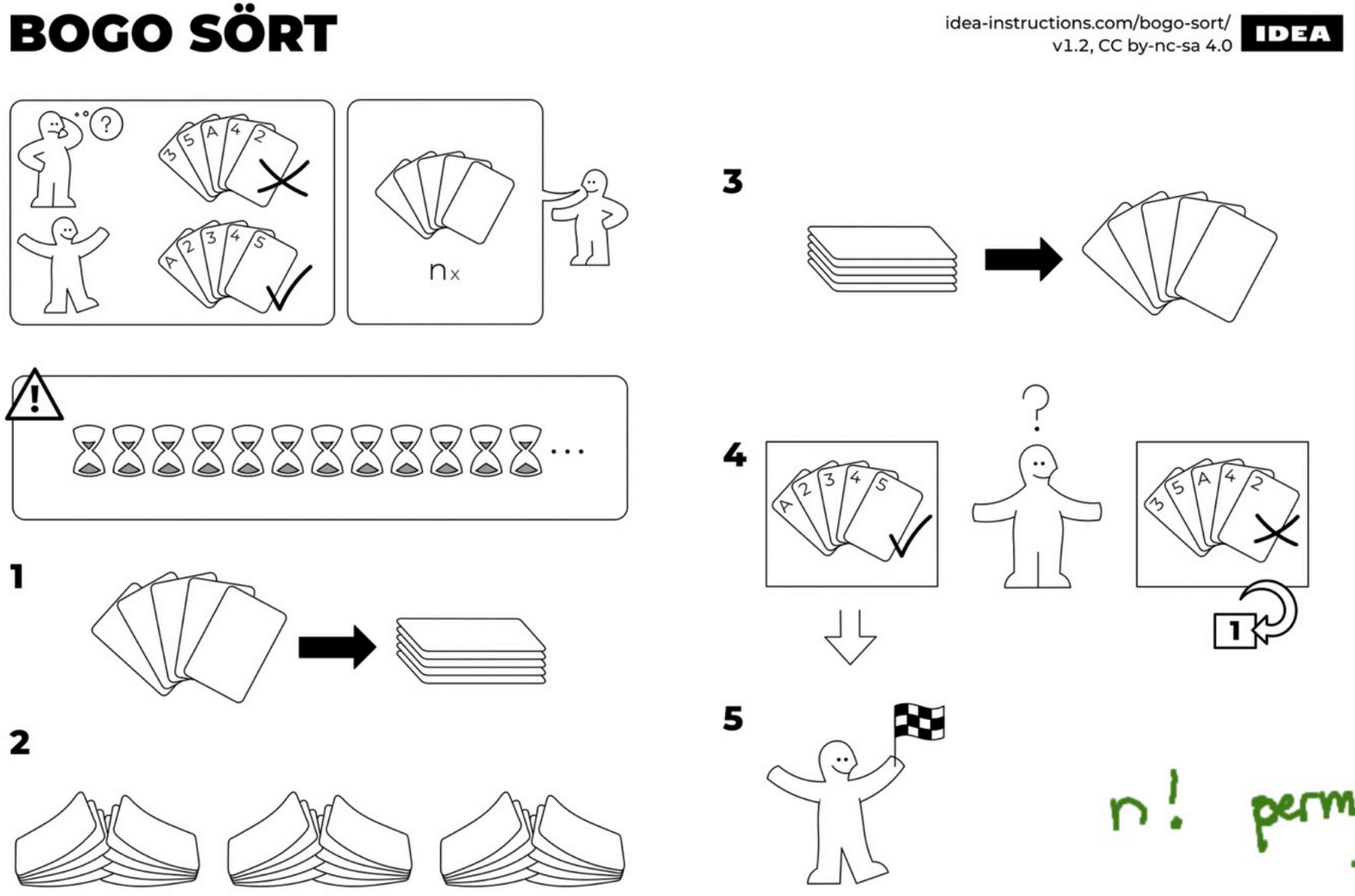
```python
def merge_sort(a_list):
    """
    Sort list using the merge sort
    algorithm returning a new sorted list.

    Args:
        a_list: List to sort

    Returns:
        New sorted list
    """

    if len(a_list) <= 1:
        # Base case: List with single
        # value is already sorted
        return a_list
    else:
        # Recursive case: Split list in half,
        # sort each half then merge
        # the resulting lists
        mid_index = len(a_list) // 2
        left  = merge_sort(a_list[:mid_index])
        right = merge_sort(a_list[mid_index:])

        merged = merge(left, right)
        return merged
```

```python
def merge(list1, list2):
    """
    Return a sorted list produced from merging
    two sorted lists

    Args:
        list1, list2: Sorted lists to merge

    Returns:
        Sorted, merged, list
    """
    result = []
    index1 = 0
    index2 = 0

    # Iterate each of the lists an item at a time
    while index1 < len(list1) and index2 < len(list2):
        if list1[index1] < list2[index2]:
            # If the current item in list1 is smaller,
            # copy and advance current item in list1
            result.append(list1[index1])
            index1 += 1
        else:
            # Otherwise, do the same in list2
            result.append(list2[index2])
            index2 += 1

    # Append any remaining elements in list1 or list2.
    # Only one of these lists should have any remaining
    # elements, the other slicing operation
    # will produce an empty list
    result += list1[index1:]
    result += list2[index2:]

    return result
```

See `sorting.py` (linked in reading) for a performance comparison.



`bogo_sort`: one of the worst sorting algorithms.



n! permutations of items

n to check if a list is sorted

O(n×n!)

Summary and Reminders

- **Linear Search:** best is $O(1)$, worst is $O(n)$, average is $O(n)$.
- **Binary Search:** best is $O(1)$, worst is $O(\log n)$, average is $O(\log n)$.
- **Selection Sort:** best is $O(n^2)$, worst is $O(n^2)$, average is $O(n^2)$.
- **Insertion Sort:** best is $O(n)$, worst is $O(n^2)$, average is $O(n^2)$.
- **Merge Sort:** best is $O(n \log n)$, worst is $O(n \log n)$, average is $O(n \log n)$.
- Programming Assignment 7 initial due date on Thursday.
- Quiz 8 this Friday includes retakes from Quizzes 4 - 7 + new Quiz 8 topics + Midterm 1 retakes of **Questions 6 and Question 7**:
  - Midterm 1 Question 6: "Finding errors"
  - Midterm 1 Question 7: "Writing functions with sequences"
- Only Quiz 8 cheat sheet (linked on calendar) will be allowed for Quiz 8 + Midterm 1 retakes.