Sort Algorithms

15-110 - Friday 10/09

Learning Objectives

- Recognize the general algorithm and trace code for three algorithms: **selection sort**, **insertion sort**, and **merge sort**

- Compute the **Big-O runtimes** of selection sort, insertion sort, and merge sort

Search Algorithms Benefit from Sorting

We use search algorithms a *lot* in computer science. Just think of how many times a day you use Google, or search for a file on your computer.

We've determined that search algorithms work better when the items they search over are **sorted**. Can we write an algorithm to sort items **efficiently**?

Note: Python already has built-in sorting functions (`sorted(lst)` is non-destructive, `lst.sort()` is destructive). This lecture is about a few different algorithmic approaches for sorting.

Many Ways of Sorting

There are a **ton** of algorithms that we can use to sort a list.

We'll use https://visualgo.net/bn/sorting to visualize some of these algorithms.

Today, we'll specifically discuss three different sorting algorithms: **selection sort**, **insertion sort**, and **merge sort**. All three do the same action (sorting) but use different algorithms to accomplish it.

Selection Sort

Selection Sort Sorts From Smallest to Largest

The core idea of selection sort is that you **sort from smallest to largest**.

1. Start with none of the list sorted
2. Repeat the following steps until the whole list is sorted:
   a) Search the unsorted part of the list to find the smallest element
   b) Swap the found element with the first unsorted element
   c) Increment the size of the 'sorted' part of the list by one

Note: for selection sort, swapping the element currently in the front position with the smallest element is faster than sliding all of the numbers down in the list.

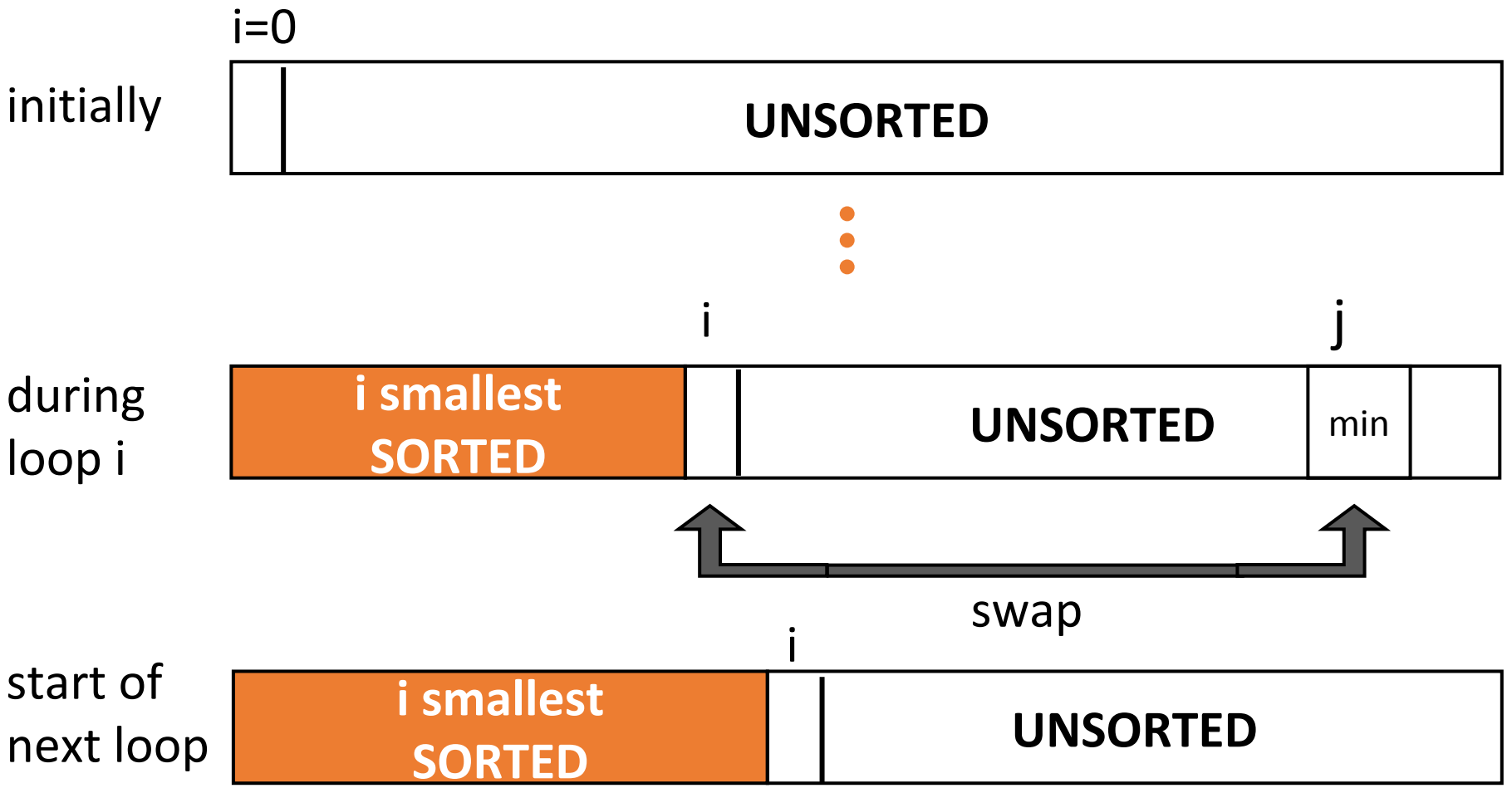
Sidebar: Swapping Elements in a List

We'll often need to **swap** elements in lists as we sort them. Let's implement swapping first.

To swap two elements, you need to create a **temporary** variable to hold one of them. This keeps the first element from getting overwritten.

```
def swap(lst, i, j):
    tmp = lst[i]
    lst[i] = lst[j]
    lst[j] = tmp
```

Selection Sort: Repeatedly select the next smallest and add it to sorted part



Selection Sort Code

```
def selectionSort(lst):
    # i is the index of the first unsorted element
    # everything before it is sorted
    for i in range(len(lst)-1):
        # find the smallest element
        j = i
        for index in range(i + 1, len(lst)):
            if lst[index] < lst[j]:
                j = index
        swap(lst, i, j)
    return lst

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
lst = selectionSort(lst)
print(lst)
```

Selection Sort – Efficiency Analysis

When we analyze the efficiency of sorting algorithms, we'll consider the number of **comparisons** and **swaps** that are performed.

We'll also talk about individual **passes** of the sorting algorithms. A pass is a single iteration of the outer loop (or putting a single element into its sorted location).

Selection Sort Code – Comparisons and Swaps

```
def selectionSort(lst):
    # i is the index of the first unsorted element
    # everything before it is sorted
    for i in range(len(lst)-1):
        # find the smallest element
        j = i
        for index in range(i + 1, len(lst)):
            if lst[index] < lst[j]:
                j = index
        swap(lst, i, j)
    return lst

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
lst = selectionSort(lst)
print(lst)
```

A single iteration of this is a **pass**

Comparison

Swap

Selection Sort – Comparisons

What's the **worst case input** for Selection Sort?

Answer: Any list, really. The list doesn't affect the actions taken.

How many comparisons does Selection Sort do in the worst case, if the input list has n elements?

Search for 1^{st} smallest: n-1 comparisons
Search for 2^{nd} smallest: n-2 comparisons
...
Search for 2^{nd}-to-last smallest: 1 comparison

Total comparisons:

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

Selection Sort – Swaps

What about swaps?

The algorithm does a single swap at the end of each pass, and there are n-1 passes, so there are n-1 swaps.

Overall, we do $n^2/2 - n/2 + n-1$ actions.

This is **$O(n^2)$**.

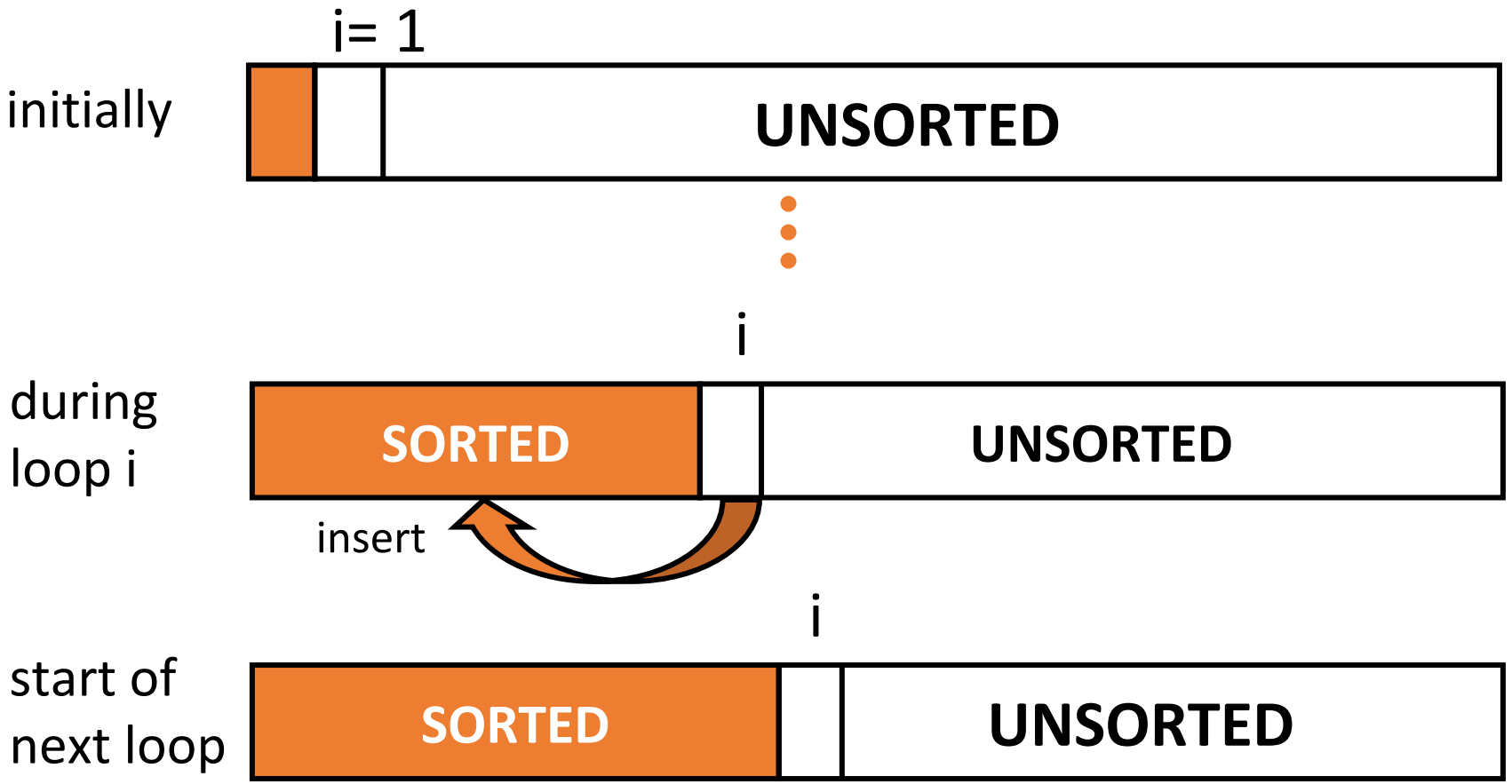
Insertion Sort

Insertion Sort Builds From the Front

The core idea of insertion sort is to **insert** each item from front to back into **a sorted list at the front**.

1. Start with only the first element of the list sorted
2. Repeat the following steps until the whole list is sorted:
   a) Compare the first unsorted element with the element directly to its left
   b) If the unsorted element is smaller, swap the two.
   c) Repeat a and b until the unsorted element is bigger.
   d) Increment the size of the 'sorted' part of the list by one

Insertion Sort : repeatedly insert the next element into the sorted part



Insertion Sort Code

```
def insertionSort(lst):
    # i is the index of the first unsorted element
    # everything before it is sorted
    for i in range(1, len(lst)):
        j = i
        # compare and swap until new item is in the correct place
        while j > 0 and lst[j-1] > lst[j:
            swap(lst, j, j-1)
            j = j - 1
    return lst

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
lst = insertionSort(lst)
print(lst)
```

Insertion Sort Code – Comparisons and Swaps

```
def insertionSort(lst):
    # i is the index of the first unsorted element
    # everything before it is sorted
    for i in range(1, len(lst)):
        j = i
        # compare and swap until new item is in the right place
        while j > 0 and lst[j-1] > lst[j]:
            swap(lst, j, j-1)
            j = j - 1
    return lst

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
lst = insertionSort(lst)
print(lst)
```

A single iteration of this is a **pass**

Comparison

Swap

Insertion Sort – Efficiency Analysis

What's the **worst case input** for Insertion Sort?

Answer: A list that is in **reverse sorted order**. We'll have to move every element all the way to the front.

You do: how many **comparisons** and **swaps** happen in insertion sort in the worst case?

Insertion Sort – Comparisons and Swaps

In the worst case: For every comparison, we will also make a swap.

Insert 2nd element: 1 comparison & swap

Insert 3rd element: 2 comparisons & swaps

...

Insert last element: n-1 comparisons & swaps

Total actions:

$2*(1 + 2 + ... + (n-1)) = 2 * (n * (n-1) / 2) = n^2 - n$

$= \mathbf{O(n^2)}$

Sidebar: Insertion Sort Best Case

Why do we care about insertion sort? While its worst case is just as bad as Selection Sort, its best case is much better!

The best case for insertion sort is **an already-sorted list**. On this input, the algorithm does 1 comparison and no swaps on each pass.

The best-case time for insertion sort runs in **linear time**.

Merge Sort

Improve Efficiency with a Drastic Change

If we want to do better than $O(n^2)$, we need to make a drastic change in our algorithms.

One common strategy is to use **Divide and Conquer**:

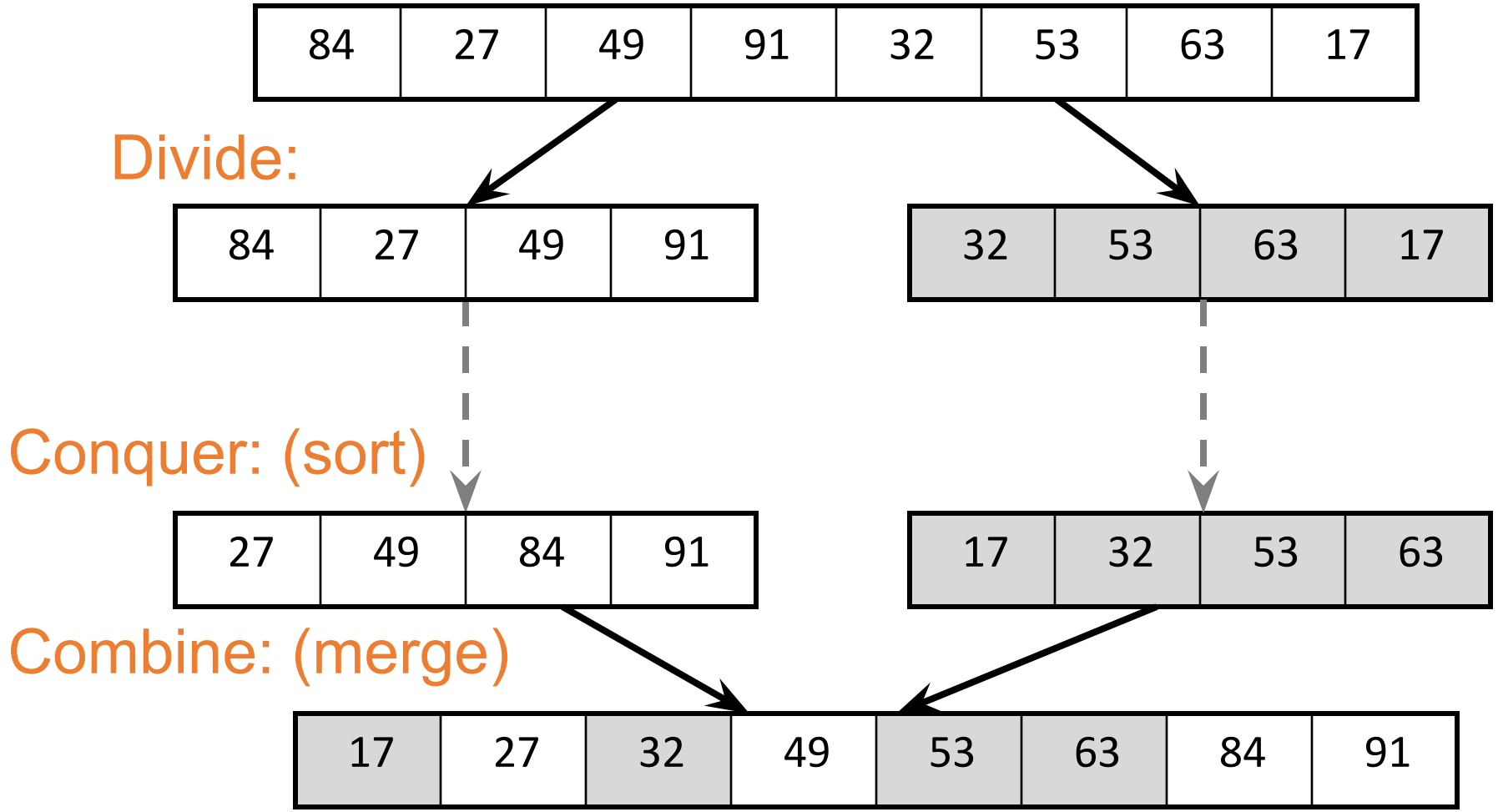
1. **Divide** the problem into "simpler" versions of itself (usually in two halves).
2. **Conquer** each problem using the same process (usually recursively).
3. **Combine** the results of the "simpler" versions to form your final solution.

Merge Sort Delegates, Then Merges

The core idea of the Merge Sort algorithm is that you **sort by merging**.

1. If there are less than two elements,
   return a **copy** of the list (it's already sorted)
2. Otherwise...
   1. Delegate sorting the front half of the list (recursion!)
   2. Delegate sorting the back half of the list (recursion!)
   3. **Merge** the two sorted halves into a **new** sorted list.

Merge Sort Process



Merge Sort Code

```
def mergeSort(lst):
    # base case: 0-1 elements are sorted.
    if len(lst) < 2:
        return lst
    # divide
    mid = len(lst) // 2
    half1 = lst[:mid]
    half2 = lst[mid:]
    # conquer by sorting
    half1 = mergeSort(half1)
    half2 = mergeSort(half2)
    # combine sorted halves
    return merge(half1, half2)
```

Merge By Checking the Front of the Lists

How do we merge two sorted lists?

1. Create a new empty 'result' list
2. Keep track of two pointers to the two lists, each starting at the first element
3. Repeat the following until we've added all the elements of one of the lists:
   a) Compare the pointed-to elements in each of the two lists
   b) Copy the smaller element to the end of the result list
   c) Move the pointer from the smaller element to the next one in the list
4. Move the rest of the unfinished list to the end of the result list

Merge Code

```
def merge(half1, half2):
    result = [ ]
    i = 0
    j = 0
    while i < len(half1) and j < len(half2):
        # only compare first two- guaranteed to be smallest due to sorting
        if half1[i] < half2[j]:
            result.append(half1[i])
            i = i + 1
        else:
            result.append(half2[j])
            j = j + 1
    # add remaining elements (only one of the halves still has values)
    result = result + half1[i:] + half2[j:]
    return result
```

Merge Sort – Efficiency Analysis

Merge Sort doesn't have swaps. Instead, we'll consider the number of **comparisons** and **copies** that are performed.

What's the **worst case input**? Any list, really; it doesn't matter.

Merge Sort Code

```
def mergeSort(lst):
    if len(lst) < 2:
        return lst
    mid = len(lst) // 2
    half1 = lst[:mid]
    half2 = lst[mid:]
    half1 = mergeSort(hafl1)
    half2 = mergeSort(half2)
    return merge(half1, half2)

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
lst = mergeSort(lst)
print(lst)
```

Copy

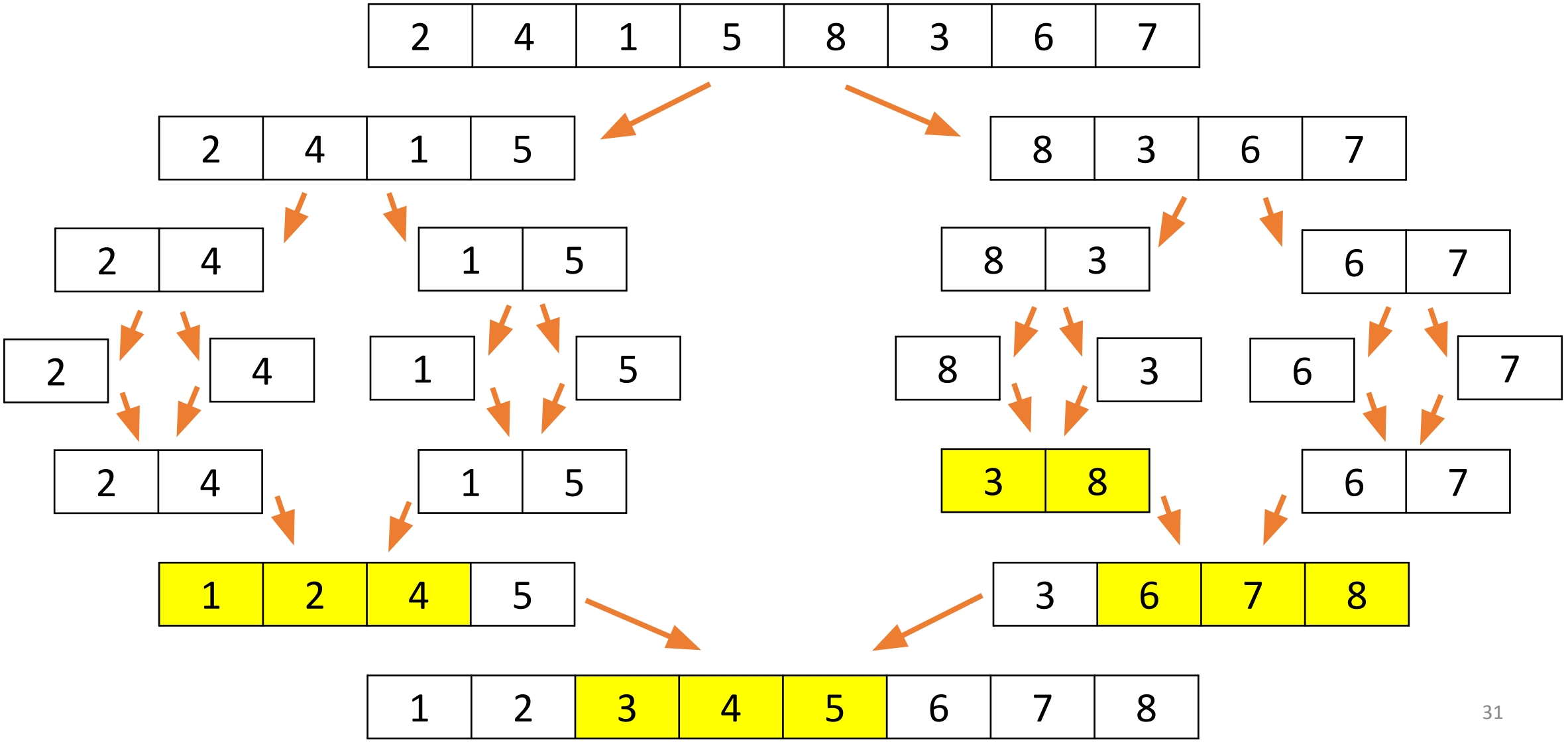
```
def merge(half1, half2):
    result = [ ]
    i = 0
    j = 0
    while i < len(half1) and j < len(half2):
        if half1[i] < half2[j]:
            result.append(half1[i])
            i = i + 1
        else:
            result.append(half2[j])
            j = j + 1
    result = result + half1[i:] + half2[j:]
    return result
```

Comparison

Copy

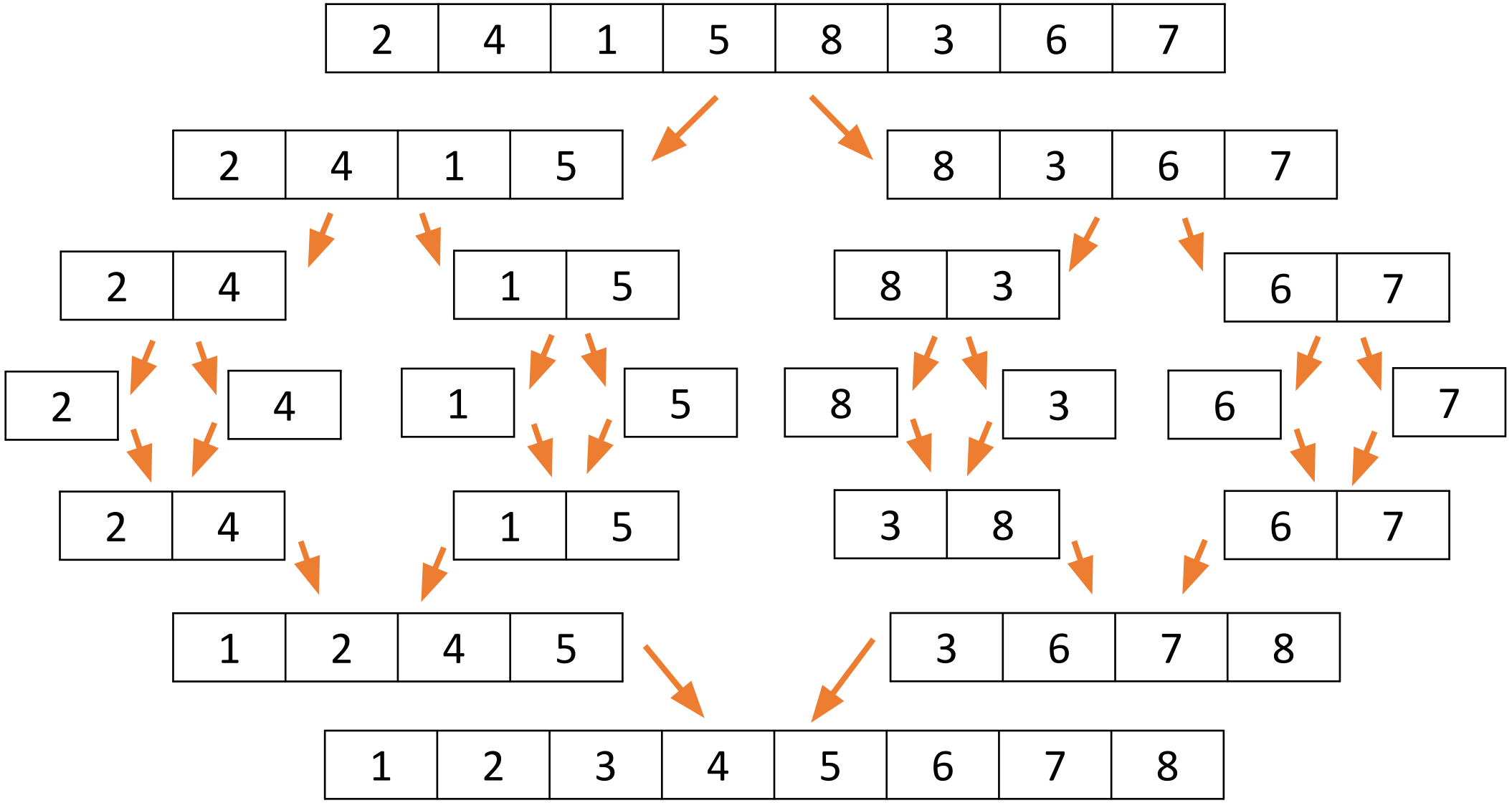
Copy

Merge Sort Call Breakdown



Merge Sort Call Breakdown

n copies in each split-pass
n copies + n comparisons in each merge-pass



Split Pass 1
Split Pass 2
Split Pass 3
Merge Pass 1
Merge Pass 2
Merge Pass 3

Merge Sort Efficiency

How many split-passes and merge-passes occur?

Every time a pass occurs, we cut the number of elements being sorted in **half**. The number of passes is the **number of times we can divide the list in half**.

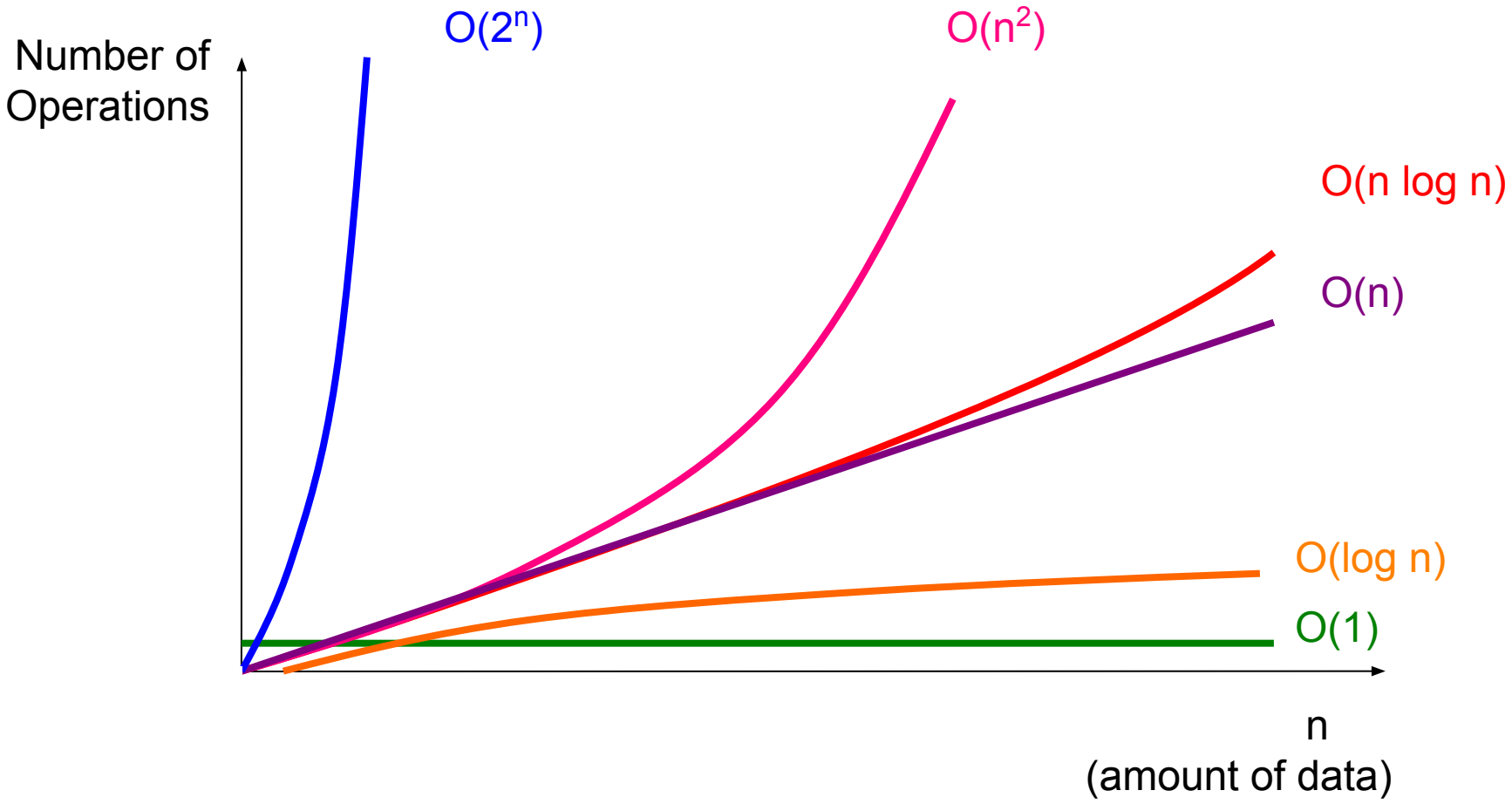
That means there are $\log_2 n$ split-passes, and $\log_2 n$ merge-passes.

Overall work: n log n + 2 * (n log n) = 3 * (n log n) = **O(n log n)**

Merge vs. Insertion Sort

n	insertion sort n * (n-1) / 2	merge sort n $\log_2 n$	Ratio
8	28	24	0.85
16	120	64	0.53
32	496	160	0.3
2^{10}	523,776	10,240	0.02
2^{20}	549,755,289,600	20,971,520	0.00004

Comparing Big-O Functions



Sidebar: General Sorting Efficiency

In general, the best we can do for sorting efficiency is O(n log n). In fact, this is the efficiency of the built-in Python sort!

You can't reduce the time to O(n) unless you put certain restrictions on the values being sorted.

Sorting takes more time than searching most of the time.

Learning Objectives

- Recognize the general algorithm and trace code for three algorithms: **selection sort**, **insertion sort**, and **merge sort**

- Compute the **Big-O runtimes** of selection sort, insertion sort, and merge sort

- **Feedback: https://bit.ly/110-feedback**