• Questions

• Mergesort
  • divide & conquer
  • merge function

```python
def merge(left, right):
    left_cursor, right_cursor = 0, 0
    merged = []
    while left_cursor < len(left) and right_cursor < len(right):
        # Sort each one and place into the result
        if left[left_cursor] <= right[right_cursor]:
            merged.append(left[left_cursor])
            left_cursor += 1
        else:
            merged.append(right[right_cursor])
            right_cursor += 1

    for left_cursor in range(left_cursor, len(left)):
        merged.append(left[left_cursor])

    for right_cursor in range(right_cursor, len(right)):
        merged.append(right[right_cursor])

    return merged
```

• recursively sort left and right halves, then merge

```python
def merge_sort(arr):
    mid = len(arr) // 2
    # Perform merge_sort recursively on both halves
    left, right = merge_sort(arr[:mid]), merge_sort(arr[mid:]),

    # Merge each side together
    return merge(left, right)
```
base case

```python
if len(arr) <= 1:
    return arr
```

analysis

- merge operation is $O(n)$

- how many merges?
  - Number of times $n$ can be divided by 2 before base case—$\log_2(n)$
  - Gives us $O(n\log_2(n))$, which we will compare to

- diagram

```
merge_sort([70, 68, 93, 9, 63, 30]), left = [68, 70, 93], right = [9, 30, 63]
```

```
merge_sort([70, 68, 93]), left = [70], right = [68, 93]
```

```
merge_sort([68, 93]), left = [68], right = [93]
```

```
merge_sort([68]), base case
```

```
merge_sort([93]), base case
```

```
merge_sort([9, 63, 30]), left = [9], right = [30, 63]
```

```
merge_sort([9]), base case
```

```
merge_sort([63, 30]), left = [63], right = [30]
```

```
merge_sort([63]), base case
```

```
merge_sort([30]), base case
```

- compare timing

- Scenarios
  - in-place vs not
• requires \( O(1) \) extra space, usually modifies original array

• insertion sort is in-place, merge sort (as we implemented it) is not

▼ stability

• equal elements remain in the same relative order before and after sorting

▼ essential if we want to sort on one attribute and then another

• list of people, sort by age then by marital status

• both merge sort and insertion sort are stable, selection sort is not

▼ streaming data

• insertion sort is great for sorting data as it comes in \( O(n) \) to insert a single element), merge sort we have to run the whole sort again

▼ The ideal sorting algorithm would have the following properties:

• Stable: Equal keys aren’t reordered.

• Operates in place, requiring \( O(1) \) extra space.

• Worst-case \( O(n \cdot \lg(n)) \) key comparisons.

• Worst-case \( O(n) \) swaps.

• Adaptive: Speeds up to \( O(n) \) when data is nearly sorted or when there are few unique keys.

• There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

▼ Visualizations

• [https://www.toptal.com/developers/sorting-algorithms](https://www.toptal.com/developers/sorting-algorithms)

• [https://www.youtube.com/user/AlgoRythmics](https://www.youtube.com/user/AlgoRythmics)
• hand back quizzes (median 31), reflections due last day of class, 2nd peer evaluation due last day of class