Middlebury

CSCI 201: Data Structures

Fall 2024

Lecture 5R: More Sorting

Goals for today:

- Implement the steps in `MergeSort` and analyze the runtime complexity.
- List the steps in `QuickSort` and analyze the runtime complexity.

divide and conquer.

IDEA

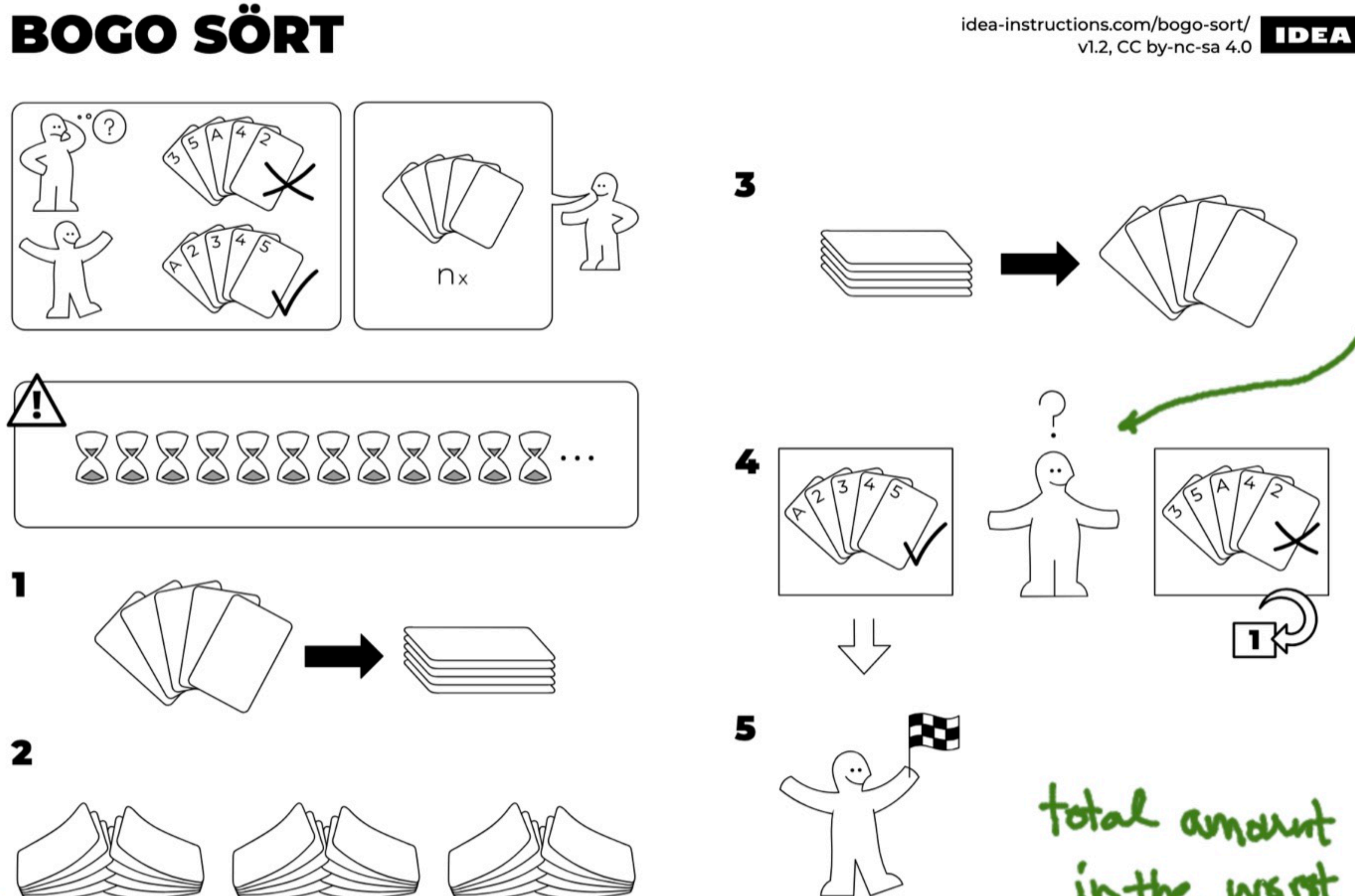
A series of nonverbal algorithm assembly instructions.

https://idea-instructions.com/

what algorithms have we seen so far?

selection
insertion
bucket
radix

BogoSort: one of the worst sorting algorithms.



shuffle randomly, ←
but let's assume
we go through all possible orderings of cards
permutations
n!

how much work is done to check if cards are sorted

≈ n comparisons

total amount of work in the worst case

$= n! \times n$

$\boxed{O(n \times n!)}$

Sorting Algorithm #5: `MergeSort`.

1. Divide input array into two subarrays.
2. Call `MergeSort` on each subarray.
3. Merge the result. → how? we'll implement this today.

here, n = 8

how many times do we need to divide an input array of length n until we get to arrays of length 1

(from Wikipedia)

Analyzing MergeSort (Part 1).

for n = 8, we had 3 "splits"

cs201-lecture05R

How many times would an input array of length 32 need to be divided (in half) before we get to arrays of size 1 (slido.com #32060084).

15

- 32
- 16
- 8
- 5
- 4
- 3
- 2

Send

$32 / 2 = 16$

$16 / 2 = 8$

$8 / 2 = 4$

$4 / 2 = 2$

$2 / 2 = 1$

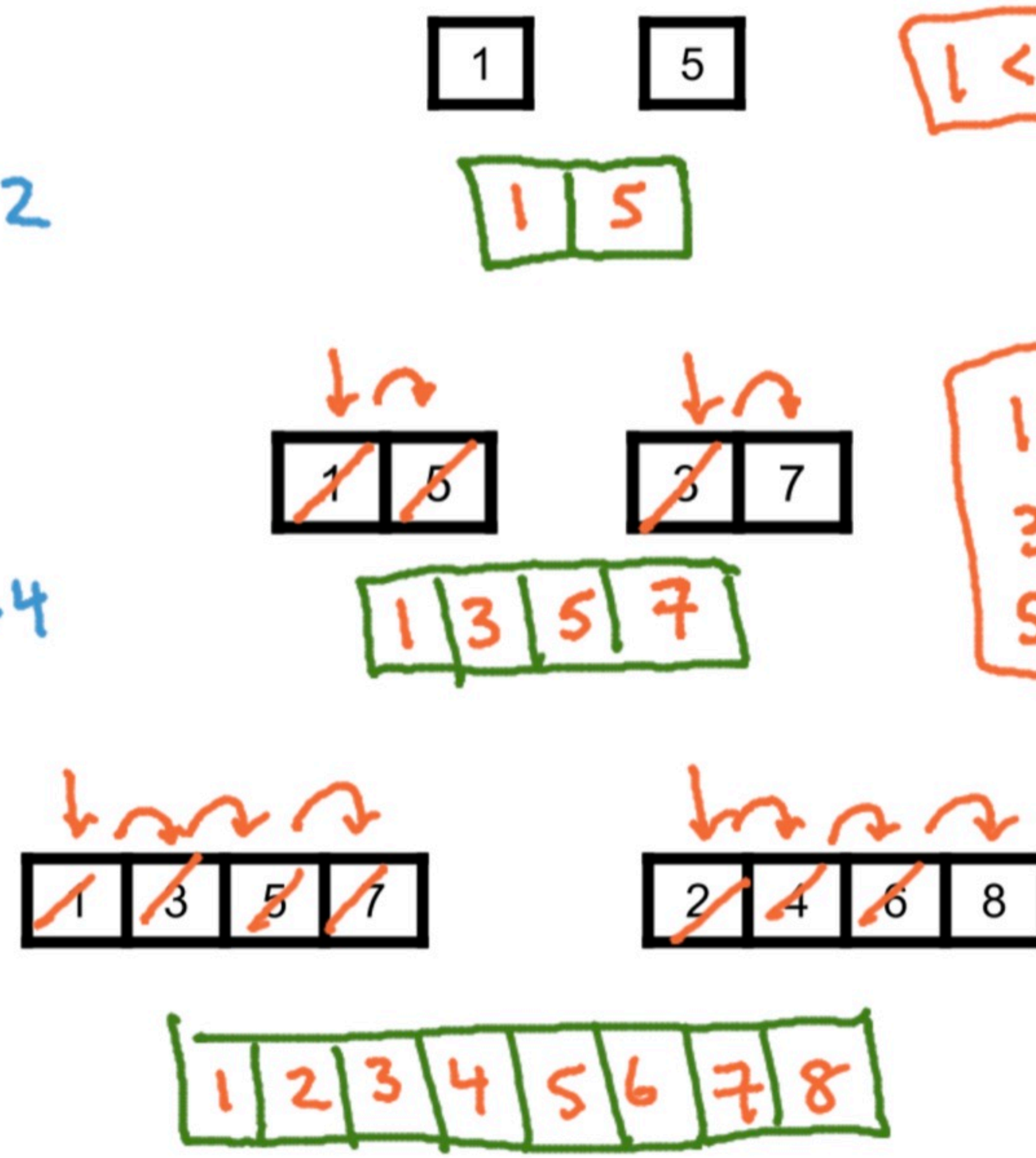
total: 5 /'s

in general: for array of length n

$$n \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdots \frac{1}{2} = 1 = \frac{n}{2^d} = 1 \rightarrow n = 2^d \quad \boxed{d = \log_2(n)}$$

Analyzing MergeSort (Part 2).

How many times do you need to ask "which leading element is smallest?" when merging these two subarrays?



$\boxed{1 < 5}$ 1 comparison

comparisons is n - 1

n = 2

let's drop -1

$\boxed{\begin{array}{l} 1 < 3 \\ 3 < 5 \\ 5 < 7 \end{array}}$ 3 comparisons

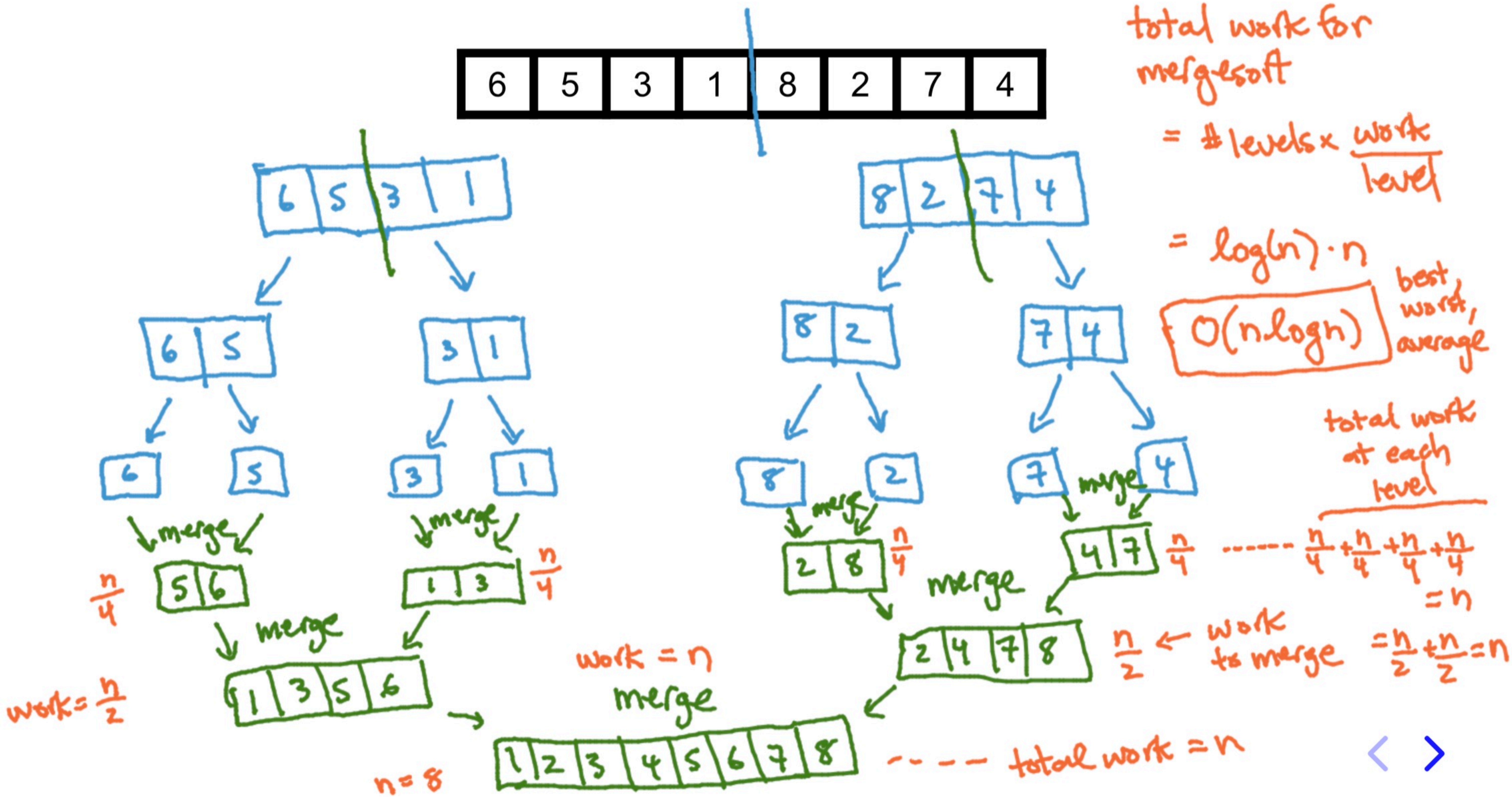
n = 4

comparisons ≃ n (work done during the merge)

$\boxed{\begin{array}{ll} 1 < 2 & 4 < 5 \\ 2 < 3 & 5 < 6 \\ 3 < 4 & 6 < 7 \\ & 7 < 8 \end{array}}$ 7 comparisons

n = 8

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

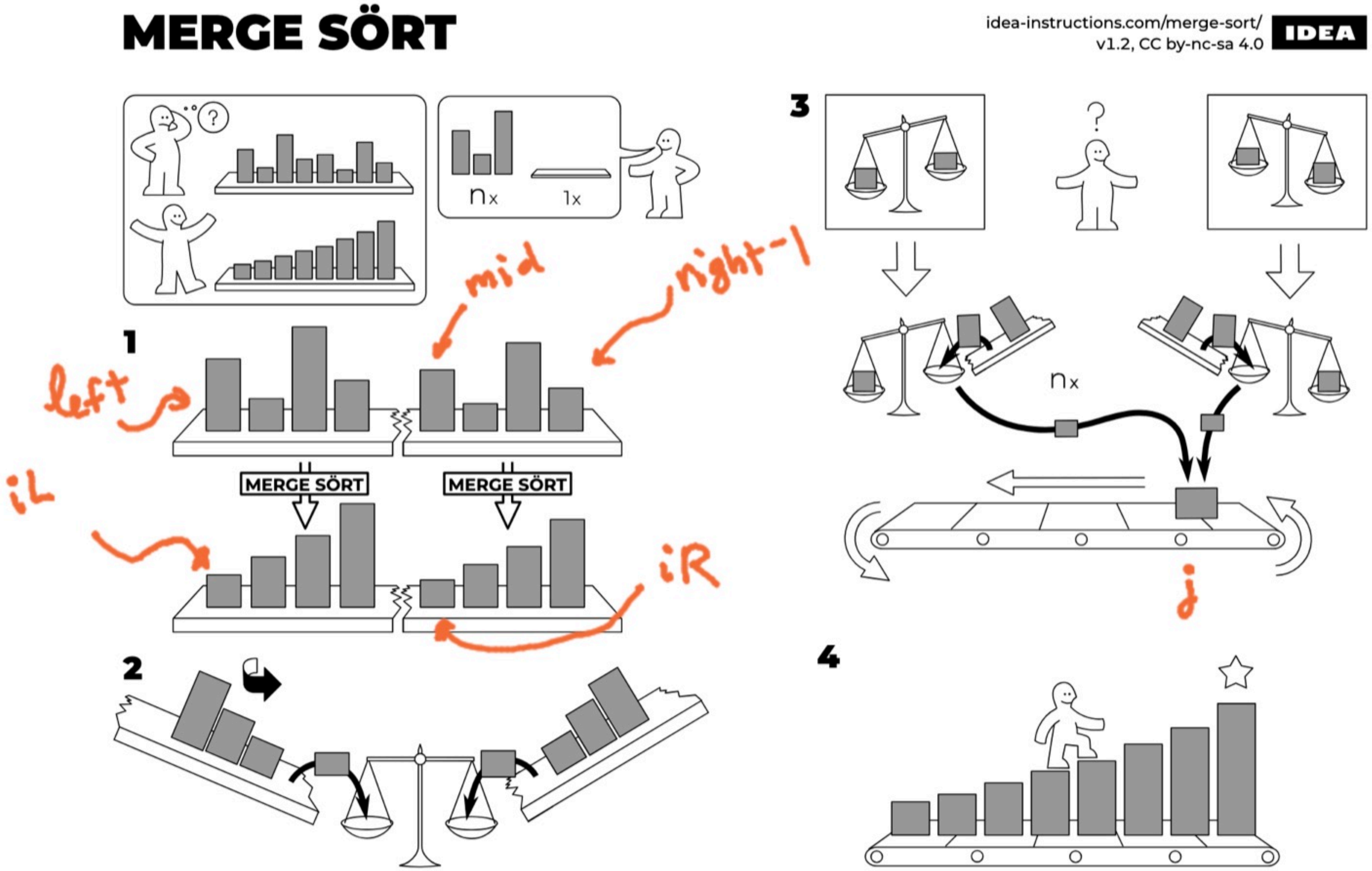


Possible implementation of MergeSort.

```java
/**
 * Sorts an array of items using merge-sort.
 */
public static void sort(int[] items) {
  mergesortHelper(items, 0, items.length);
}

/**
 * Runs merge-sort on a portion of the items, starting
 * at the "left" item, up to (but not including) the "right" item.
 *
 * @param items - entire array to be sorted.
 * @param left - left index of the subarray to be sorted.
 * @param right - right index of the subarray to be sorted.
 */
private static void mergesortHelper(int[] items, int left, int right) {
  int n = right - left;
  if (n <= 1) return;

  // recursively call merge sort on left and right subarrays
  int mid = left + n / 2;
  mergesortHelper(items, left, mid);
  mergesortHelper(items, mid, right);

  // merge the two subarrays
  merge(items, left, mid, right);
}
```

Excercise: complete the `merge` step in `MergeSorter.java`.

Study the existing code and see the `TODO` comments for what steps are missing.



Possible implementation of `merge`.

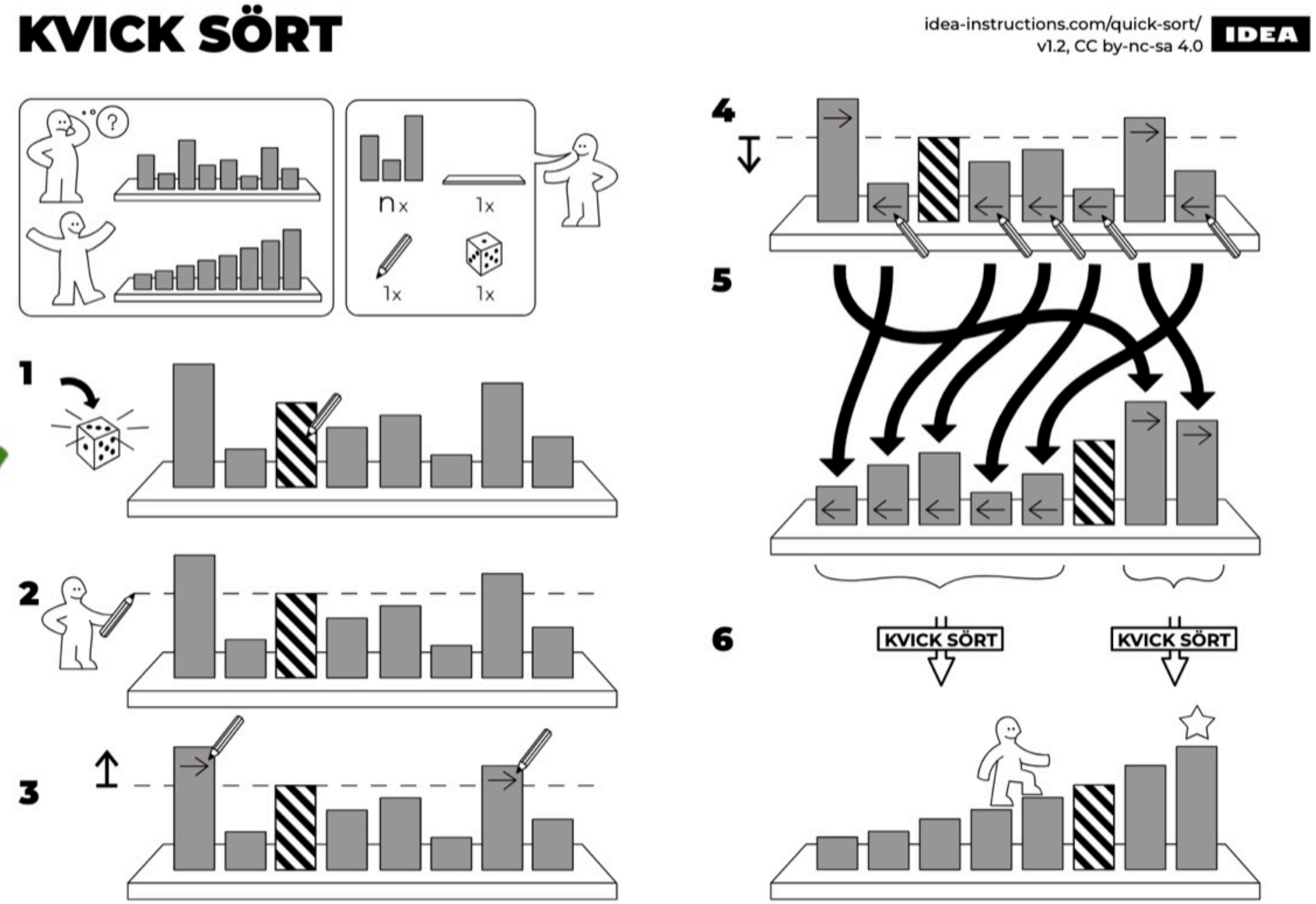
```java
private static void merge(int[] items, int left, int mid, int right) {
  // retrieve the minimum of (items[iL], items[iR]) until one subarray is empty
  int iL = left;
  int iR = mid;
  int[] merged = new int[right - left];
  int j = 0;
  while (iL < mid && iR < right) {
    if (items[iL] < items[iR]) {
      merged[j] = items[iL];
      iL++;
    } else {
      merged[j] = items[iR];
      iR++;
    }
    j++;
  }

  // TODO 1: retrieve any remaining items from the left subarray
  while (iL < mid) {
    merged[j++] = items[iL++];
  }

  // TODO 2: retrieve any remaining items from the right subarray
  while (iR < right) {
    merged[j++] = items[iR++];
  }

  // TODO 3: place the sorted items (in merged) back in the array (in items)
  for (int i = left; i < right; i++) {
    items[i] = merged[i - left];
  }
}
```

not in-place

Sorting algorithm #6: QuickSort.

1. Pick a pivot.
2. Partition items so than any item < pivot is in left subarray and any item > pivot is in the right subarray.
3. Call QuickSort on each subarray.

here, random pivot (we'll use a different choice)



Sorting algorithm #6: QuickSort.

1. Pick a pivot. let's pick the LAST item in the array is pivot
2. Partition items so than any item < pivot is in left subarray and any item > pivot is in the right subarray.
3. Call QuickSort on each subarray.

```java
1  public static void sort(int[] items) {
2    quicksortHelper(items, 0, items.length - 1);
3  }
4
5  private static void quicksortHelper(int[] items,
6                                      int left, int right) {
7    if (left >= right) return;
8
9    // pick a pivot and partition items to the left/right
10   int p = partition(items, left, right);
11
12   // call quicksort on left and right subarrays
13   quicksortHelper(items, left, p - 1);
14   quicksortHelper(items, p + 1, right);
15 }
```

left, right, i, j

8	5	1	3	6	2	7	$\boxed{4}$

is items[i] < pivot? → is 8<4? X

no swap, i++

8 5 1 3 6 2 7 $\boxed{4}$ 5<4? X

no swap, i++

8 5 1 3 6 2 7 $\boxed{4}$ 1<4? ✓

swap items[i] with items[j], i++, j++

1 5 8 3 6 2 7 $\boxed{4}$ 3<4? ✓

swap, i++, j++

1 3 8 5 6 2 7 $\boxed{4}$ 6<4? X

no swap, i++

1 3 8 5 6 2 7 $\boxed{4}$ 2<4? ✓

i++, j++
swap

1 3 2 5 6 8 7 $\boxed{4}$ 7<4? X

no swap, i++ — done

last step: swap the item at j with pivot

1 3 2 $\boxed{4}$ 6 8 7 5

j → returned to line 10.

Have a nice Midterm Recess (see you on Tuesday)!

- Homework 4 extended to Tuesday 10/15 at 11:59pm.
    1. Implement Radix Sort.
    2. Derive number of `=` for `DIYList` `add` method with a different growth technique.
- Reminder that Noah (go/noah) and Smith (go/smith) have office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).
- Submit exit ticket 5R today.