Middlebury

CSCI 201: Data Structures

Fall 2025

Lecture 5R: More Sorting

Goals for today:

- Continue our analysis of the runtime of Binary Search
- Implement the steps in `MergeSort` and analyze the runtime complexity.
- List the steps in `QuickSort` and analyze the runtime complexity.

Searching for an element in an array

This is the algorithm *binary search*.

n	# of calls (k)
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4
16	5

```java
public static int binarySearch(int value, int[] sortedArray) {
  return binarySearch(value, sortedArray, 0, sortedArray.length - 1);
}

private static int binarySearch(int value, int[] sortedArray, int left, int right) {
 if (right < left) {
   return -1;
 }

 int mid = left + (right - left) / 2;
 if (sortedArray[mid] == value) {
    return mid;
 } else if (sortedArray[mid] < value) {
    return binarySearch(value, sortedArray, mid + 1, right); // search right half
 } else {
    return binarySearch(value, sortedArray, left, mid - 1); // search left half
 }
}
```

ignore this call

$2^{k-1} = n \Rightarrow \log_2 n = k-1 \Rightarrow k = \log_2 n + 1$

$O(\log_2 n) = O(\log n)$

Another way to write *binary search*, returning a boolean

```java
public static boolean binarySearch(int value, int[] sortedArray) {
 if (sortedArray.length == 0) {
   return false;
 }

 int mid = sortedArray.length / 2;
 if (sortedArray[mid] == value) {
    return true;
 } else if (sortedArray[mid] < value) {
    return binarySearch(value, Arrays.copyOfRange(sortedArray, mid + 1, sortedArray.length)); // search right half
 } else {
    return binarySearch(value, Arrays.copyOfRange(sortedArray, 0, mid)); // search left half
 }
}
```

Another way to write *binary search*, returning a boolean

```java
public static boolean binarySearch(int value, int[] sortedArray) {
 if (sortedArray.length == 0) {
   return false;
 }

 int mid = sortedArray.length / 2;
 if (sortedArray[mid] == value) {
    return true;
 } else if (sortedArray[mid] < value) {
    return binarySearch(value, Arrays.copyOfRange(sortedArray, mid + 1, sortedArray.length)); // search right half
 } else {
    return binarySearch(value, Arrays.copyOfRange(sortedArray, 0, mid)); // search left half
 }
}
```

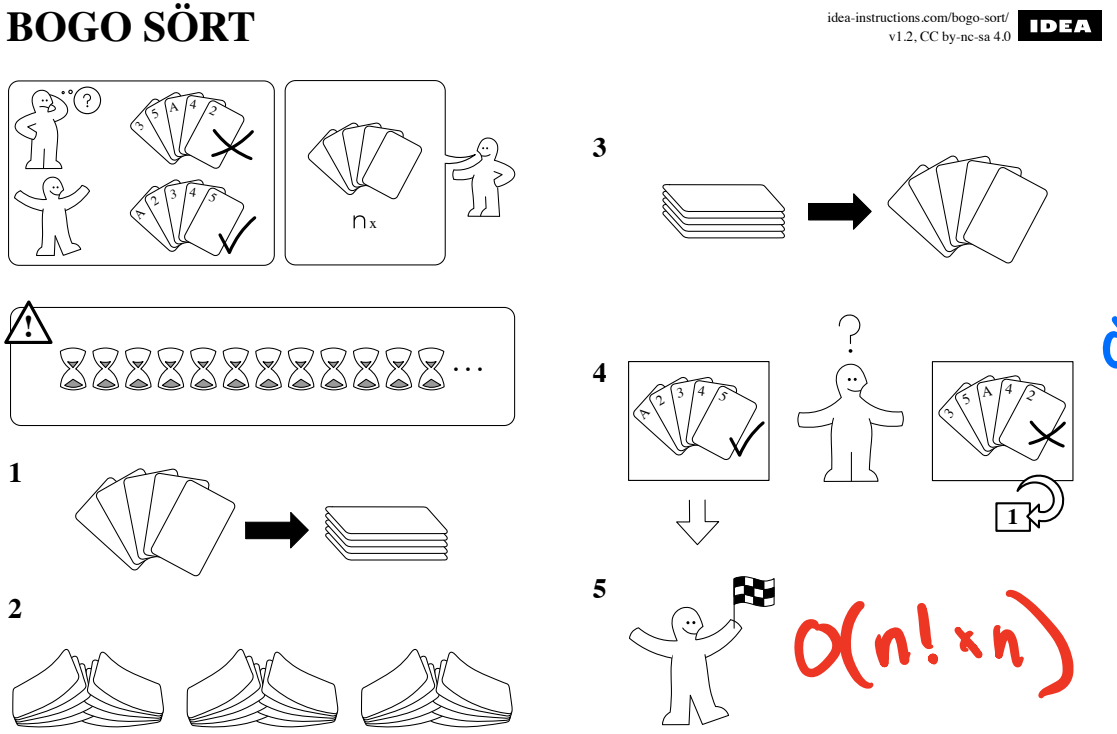
temp array of length $\frac{n}{2}-1$ copy everything to it

Discuss with your group: what's good about this approach? What could be improved?

pros: no helper
no keeping track of indices } cleaner

cons: returns a boolean – no index
copy makes this O(n)

BogoSort: one of the worst sorting algorithms.



shuffle cards
↓
consider all permutations
n!

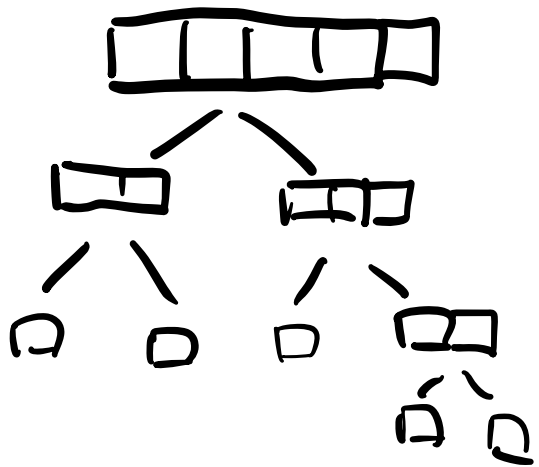
check if sorted
O(n)

O(n! × n)

Sorting Algorithm #5: `MergeSort`.

1. Divide input array into two subarrays.
2. Call `MergeSort` on each subarray.
3. Merge the result.

base case: length 1



Analyzing `MergeSort` (Part 1).

How many times would an array of length 32 need to be divided in half before the subarrays are all of size 1?

Answer on sli.do - find the link on the course schedule or use #3205097!

$\frac{32}{2} = 16$ $\quad \frac{16}{2} = 8$ $\quad \frac{8}{2} = 4$ $\quad \frac{4}{2} = 2$ $\quad \frac{2}{2} = 1$

5 splits!

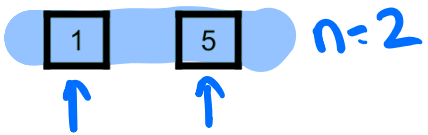
$n \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdots \cdot \frac{1}{2} = 1 \Rightarrow \frac{n}{2^d} = 1 \Rightarrow n = 2^d \Rightarrow d = \log_2 n$

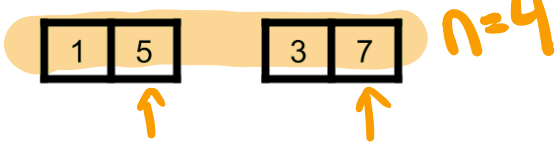
d = # of splits

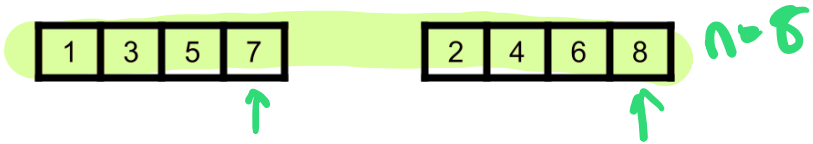
Analyzing `MergeSort` (Part 2).

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

Comparisons	Count
1 < 5	1
1 < 3 5 < 3 5 < 7	3
1 < 2 3 < 2 3 < 4 5 < 4 5 < 6 7 < 6 7 < 8	7

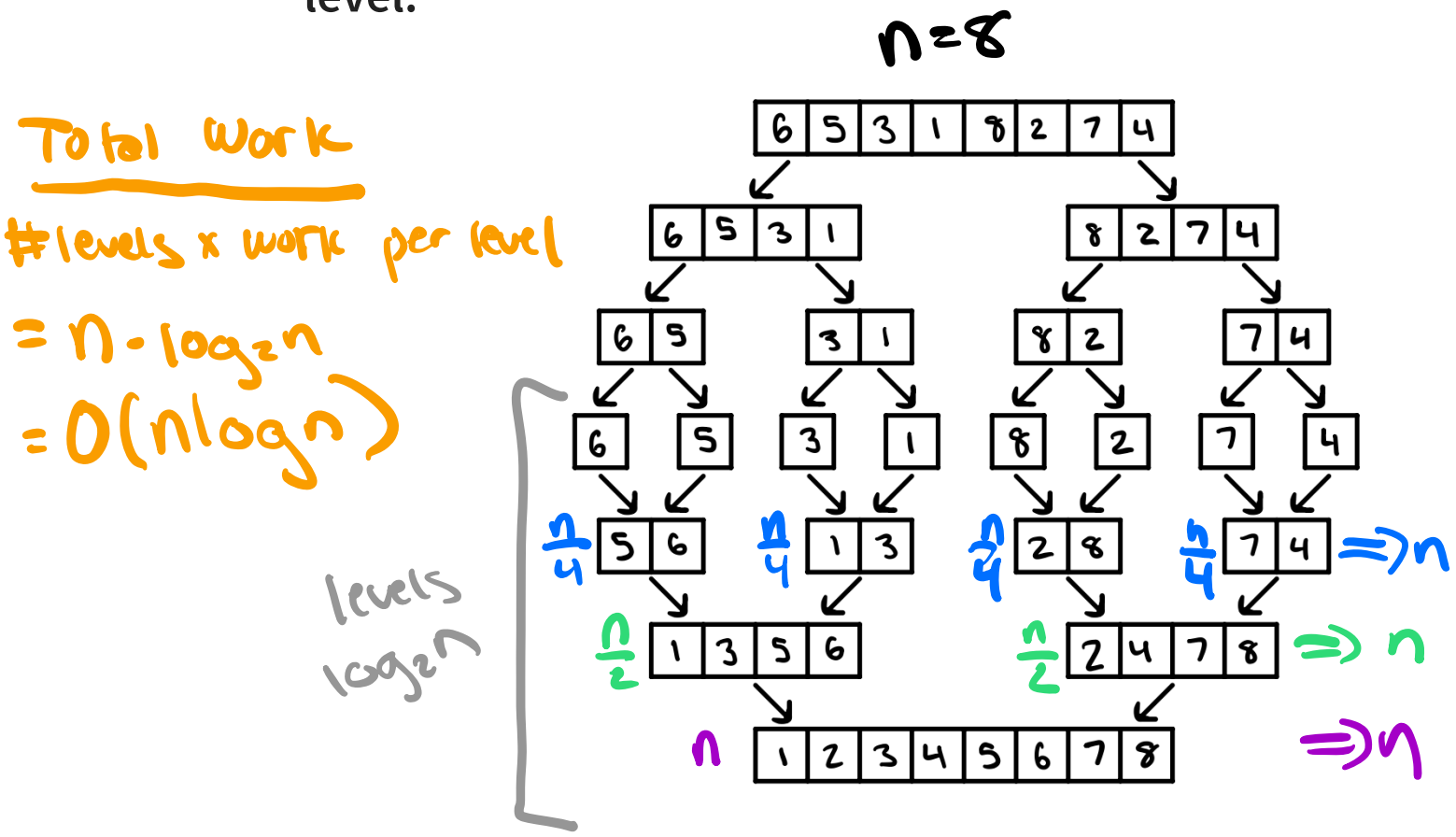






≈ n comparisons to merge

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);
}
```

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

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

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

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

Note: the operation is similar to what you need to do for the `merge_sorted_lists` programming exam problem, but you'll have to do more work here to keep track of indices and insert everything into the original array!

Possible implementation of `merge`.

```
1  private static void merge(int[] items, int left, int mid, int right) {
2    // get the min of (items[indexLeft], items[indexRight]) until a subarray is empty
3    int indexLeft = left;
4    int indexRight = mid;
5    int[] merged = new int[right - left];
6    int j = 0;
7    while (indexLeft < mid && indexRight < right) {
8      if (items[indexLeft] < items[indexRight]) {
9        merged[j] = items[indexLeft];
10       indexLeft++;
11     } else {
12       merged[j] = items[indexRight];
13       indexRight++;
14     }
15     j++;
16   }
17
18   // (1) retrieve any remaining items from the left subarray
19   while (indexLeft < mid) {
20     merged[j++] = items[indexLeft++];
21   }
22
23   // (2) retrieve any remaining items from the right subarray
24   while (indexRight < right) {
25     merged[j++] = items[indexRight++];
26   }
27
28   // (3) place the sorted items (in merged) back in the array (in items)
29   for (int i = left; i < right; i++) {
30     items[i] = merged[i - left];
31   }
32 }
```

merged[j] = items[indexLeft];
j++;
indexLeft++;

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.

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.

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

```
1  private static int partition(
2      int[] data, int left, int right) {
3    while (true) {
4      while (left < right && data[left] < data[right]) {
5        right--; // move right "pointer" toward left
6      }
7      if (left < right) {
8        swap(data, left++, right);
9      } else {
10       return left;
11     }
12     while (left < right && data[left] < data[right]) {
13       left++; // move left "pointer" toward right
14     }
15     if (left < right) {
16       swap(data, left, right--);
17     } else {
18       return right;
19     }
20   }
21 }
```

Analyzing `QuickSort`

In the average/ideal case,

- How deep is our "tree" of recursive calls? $\log_2 n$
- How much work do we do to partition the array at each level? n

average : $O(n \log n)$

worst : $O(n^2)$

See you Tuesday -- enjoy Fall Break!

- Start prepping for the midterm! See the study guide and complete all programming exam problems.
  - Available on canvas - please do not share publicly!
- Finish up homework 4 (due today at 5PM)
- Get started on homework 5 (**due two weeks from today**)
- Reminder that Smith (go/smith) has office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).