Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 to 22333

What is the **space** complexity of `maxVal()`?

```
static int maxVal(int[] nums, int b, int e) {
  if (e - b == 1) {
    return nums[b];
  }
  int m = b + (e - b)/2;
  int lMax = maxVal(nums, b, m);
  int rMax = maxVal(nums, m, e);
  return Math.max(lMax, rMax);
}
```

$O(1)$	**(A)**
$O(\log N)$	**(B)**
$O(N)$	**(C)**
$O(N \log N)$	**(D)**

Poll Everywhere

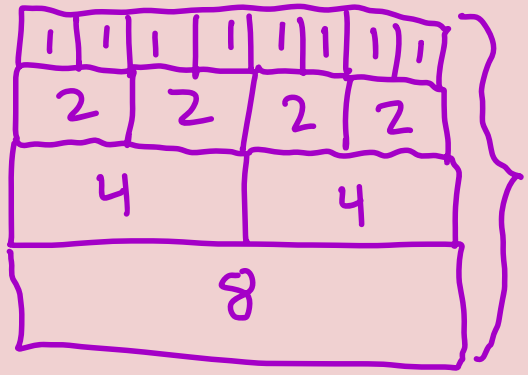
PollEv.com/2110fa25 text 2110fa25 to 22333

What is the **space** complexity of `maxVal()`?

Each call frame allocates O(1) memory

```
static int maxVal(int[] nums, int b, int e) {
  if (e - b == 1) {
    return nums[b];
  }
  int m = b + (e - b)/2;
  int lMax = maxVal(nums, b, m);
  int rMax = maxVal(nums, m, e);
  return Math.max(lMax, rMax);
}
```

"Call stack diagram" when N = nums.length = 8



Lecture 7: Sorting Algorithms

CS 2110

September 16, 2025

Today's Learning Outcomes

30. Compare and contrast the *insertion sort*, *merge sort*, and *quicksort* algorithms, discussing aspects such as time/space complexity and stability.

20. Describe the loop invariant of an iterative method involving an array and visualize it using a diagram.

26. Determine the asymptotic time and space complexity of a piece of code involving one or more loops and/or method calls.

29. Determine the number of recursive calls and the maximum depth of the call stack of a recursive method and use these to compute its time and space complexities.

The Importance of Sorting

Having sorted data can greatly improve code effiency

O(N) linear search vs. O(logN) binary search

Sorting is an important subroutine for:

- Statistics / data analysis (analyze medians, quantiles)
- Optimization (prioritize search results, navigation, rendering order in graphics applications)
- Designing "greedy" algorithms

Different Sorting Algorithms

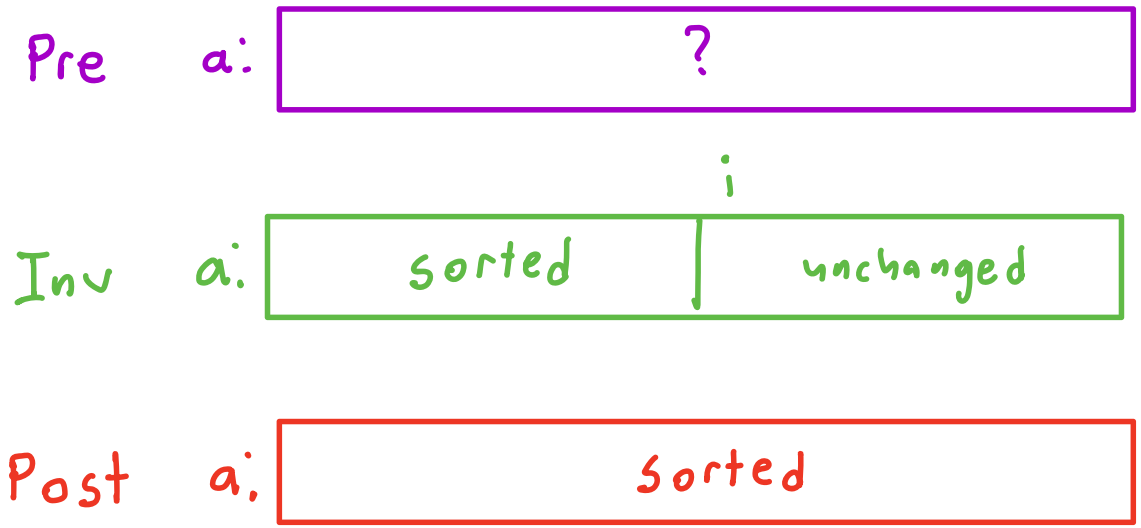
Different sorting procedures trade off different desirable properties:

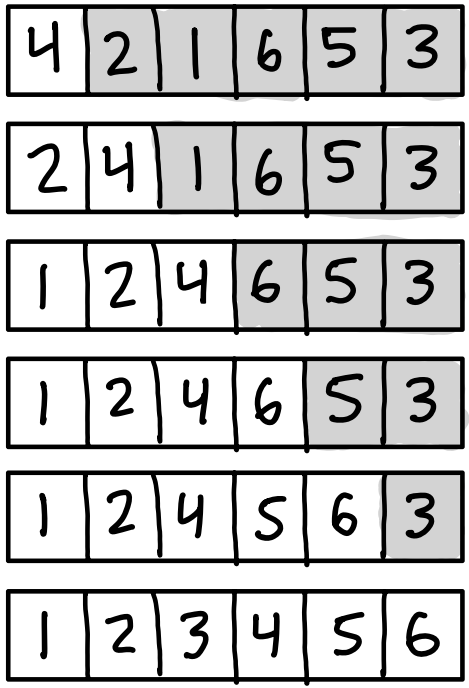
- Runtime (best-case, worst-case, expected case, adaptivity)
- Space complexity
- Memory locality, parallelizability
- Stability
- ⋮

Being familiar with a varied toolbox of sorting algorithms will help you choose the best for a particular scenario.

Insertion Sort

Big Idea: Build up a sorted range by inserting entries into their sorted positions one at a time.

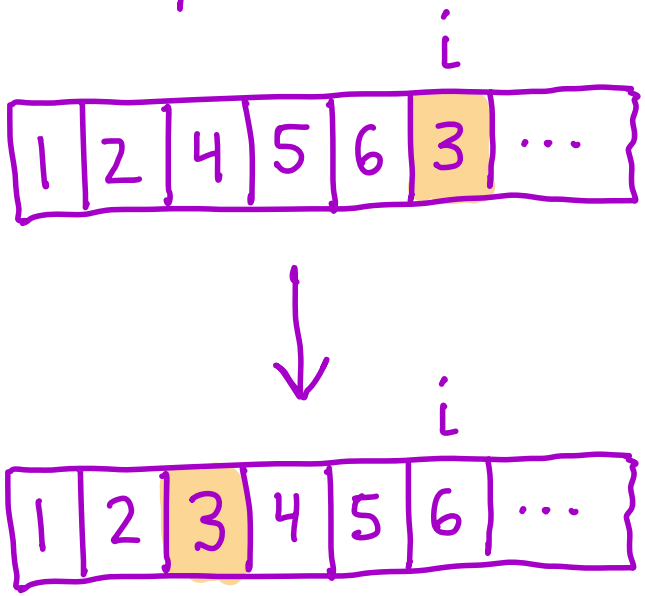




Coding Demo: Insertion Sort

```
/** Sorts `a` using the insertion sort algorithm. */
static void insertionSort(int[] a) {
  /* Loop invariant: a[..i) sorted, a[i..] unchanged. */
  for (int i = 0; i < a.length; i++) {
     insert(a,i);
  }}

/** Inserts `a[i]` into its sorted position in `a[..i)`
   * so `a[..i]` becomes sorted. Requires that
   * `0 <= i < a.length` and `a[..i)` is sorted. */
static void insert(int[] a, int i) { ... }
```

How can we achieve this behavior with a loop?



Insertion Sort (Worst-Case) Complexity

```
static void insertionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        insert(a,i);
    }}

static void insert(int[] a, int i) {
    int j = i;
    while (j > 0 && a[j - 1] > a[j]) {
        swap(a, j - 1, j); j--;
    }}
```

O(N) iterations, each does O(1) + complexity of insert() work

O(N)

runtime of insertionSort = $O(N^2)$

both methods use O(1) space, so space complexity = O(1)

O(i) = O(N) iterations, each does O(1) work

runtime of insert() = O(N)

Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 to 22333

If the array a is *already sorted*, what is the runtime of `insertionSort(a)` in terms of N = `a.length` ?

```
static void insertionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        insert(a,i);
    }}
static void insert(int[] a, int i) {
    int j = i;
    while (j > 0 && a[j - 1] > a[j]) {
        swap(a, j - 1, j); j--;
    }}
```

O(N)

O(1)

for sorted inputs, this immediately false

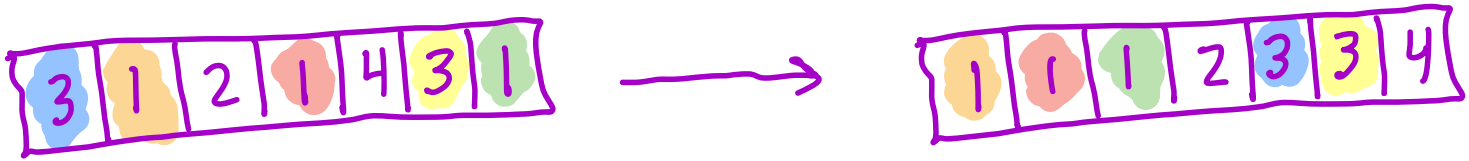
$O(1)$	**(A)**
$O(\log N)$	**(B)**
$O(N)$	**(C)**
$O(N^2)$	**(D)**

Adaptivity and Stability

A sorting algorithm is adaptive if it performs fewer operations when the input starts closer to sorted.

- Insertion Sort is an adaptive sorting algorithm.

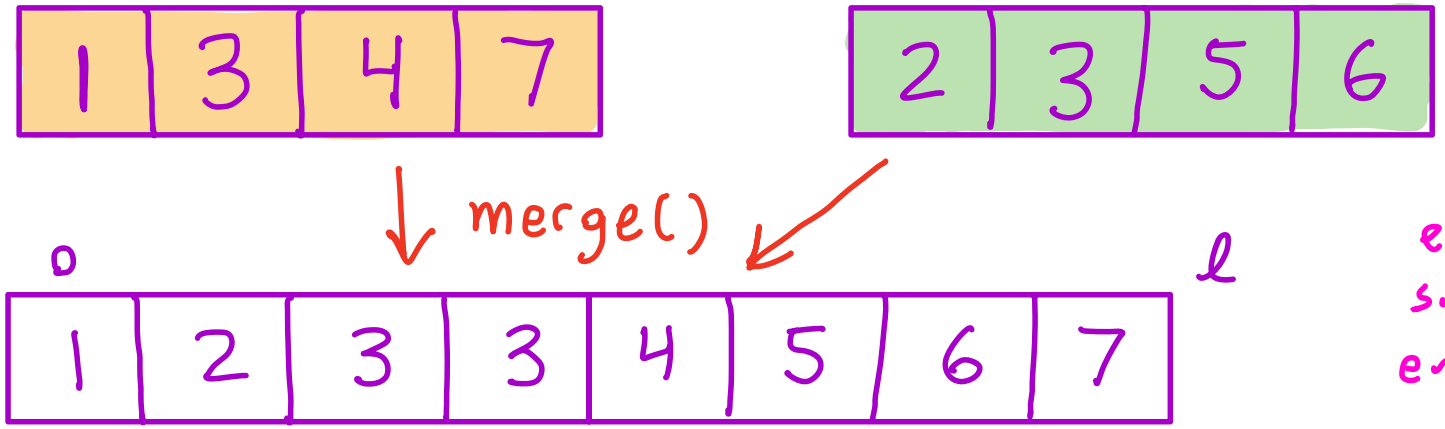
A sorting algorithm is stable if it preserves the relative order of equivalent elements.



- Insertion sort is stable because it never swaps equal elements in insert() method.

Merge Sort

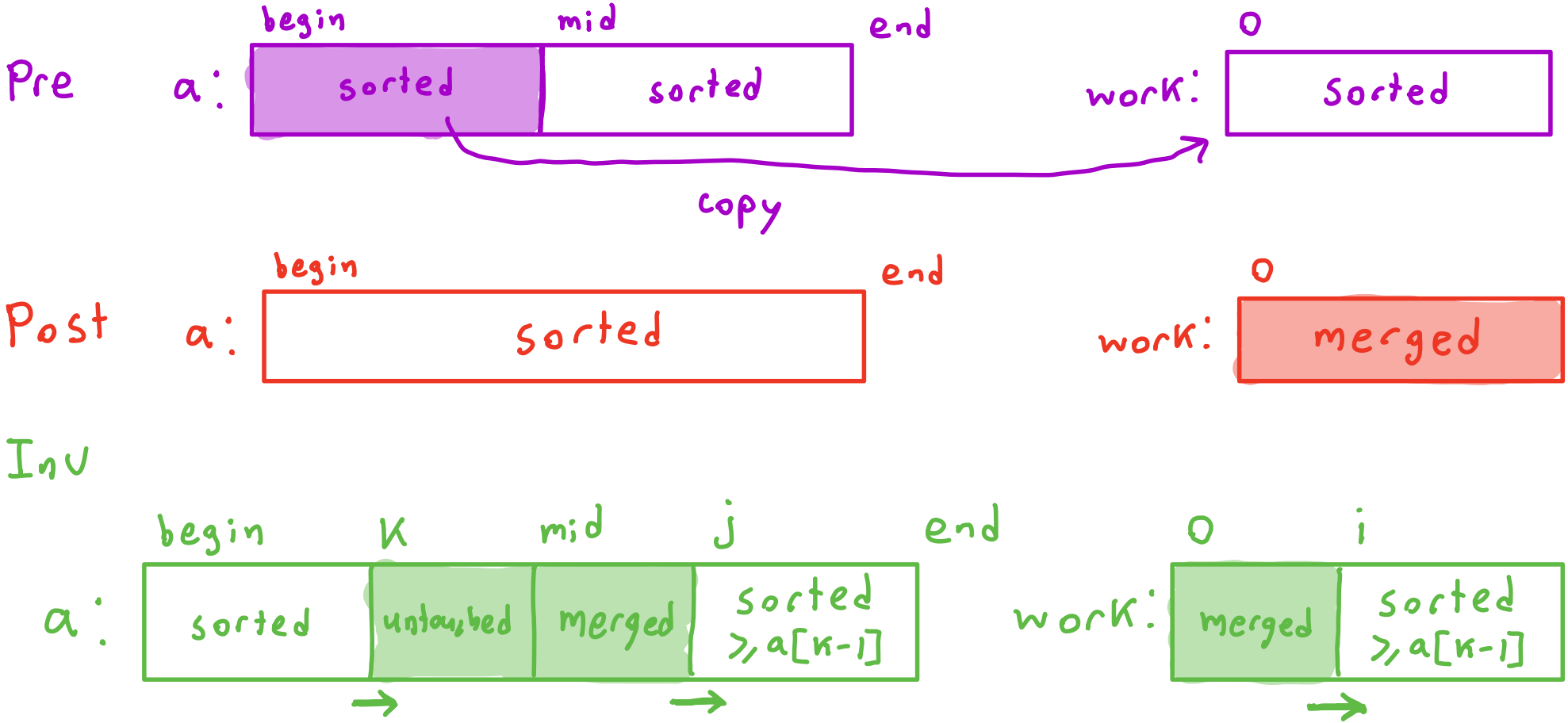
Big Idea: If we have two smaller sorted arrays, it's fairly easy to combine them into a single sorted array.



* Only need to look at one entry from each small array to fill entry of big array

O(ℓ) operation

The merge() Invariant



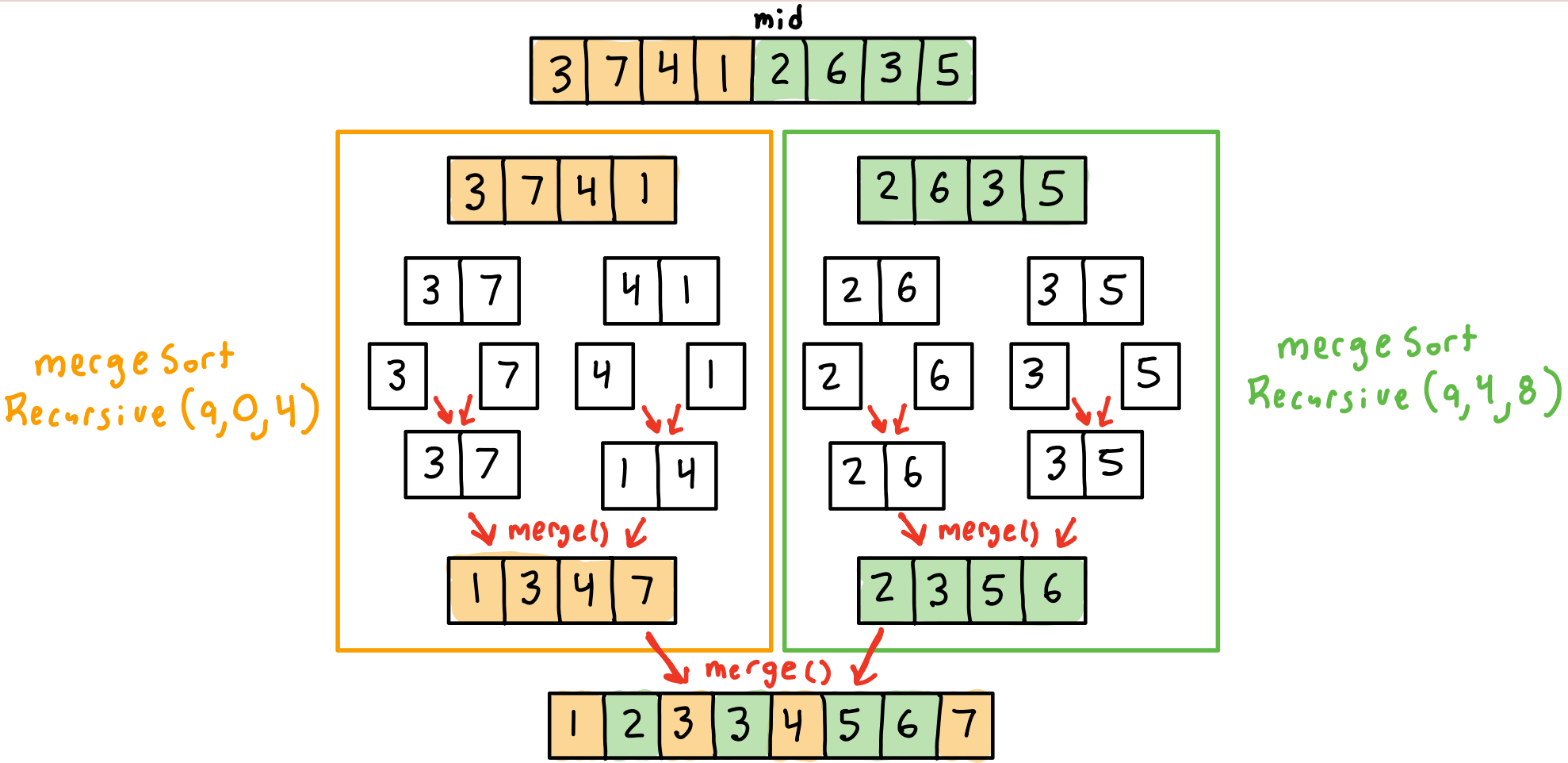
Coding Demo: The `merge()` Method

Merge Sort

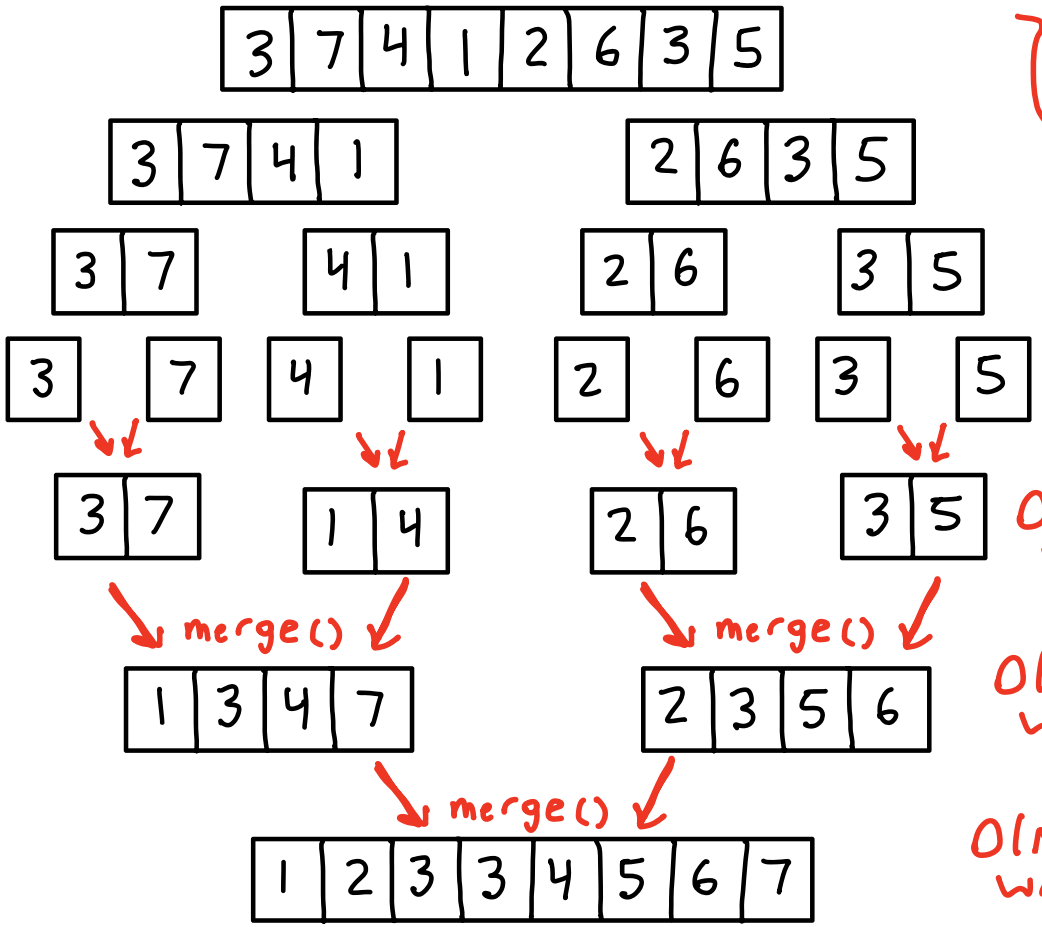
```
/** Sorts the entries of `a` using the merge sort algorithm. */
static void mergeSort(int[] a) {
  mergeSortRecursive(a, 0, a.length);
}

/** Recursively sorts `a[begin..end)` using the merge sort algorithm. */
static void mergeSortRecursive(int[] a, int begin, int end) {
    if (end - begin <= 1) { return; }      // base case
    int mid = begin + (end - begin) / 2;
    mergeSortRecursive(a, begin, mid);     // sort left half
    mergeSortRecursive(a, mid, end);       // sort right half
    merge(a, begin, mid, end);             // merge
}
```

divide + conquer algorithm

Visualizing Merge Sort



Merge Sort Runtime Analysis



O(logN) recursive depth

Non-recursive work in each call dominated by merge()

O(N) total # call frames

linear in length

O(N) merge() work at this level

O(N) merge() work at this level

O(N) merge() work at this level

O(NlogN) runtime

Merge Sort Space Complexity

Merge Sort has O(logN) recursive depth

The space complexity of each call is dominated by merge(), which allocates an O(N) work array.

Insight: At most one merge() happens at a time, so we can reuse one shared work array (that is passed as argument to recursive calls)

This guarantees **O(N)** space complexity.

Merge Sort: Other Considerations

Merge Sort is stable (we prioritized copying from work during merge()) but not adaptive.

O(N log N) runtime is best possible, so Merge Sort is default stable sort in many languages (including Java).

Merge Sort parallelizes well and is good for really large datasets since only small ranges of data are accessed at a time.

Poll Everywhere

PollEv.com/2110fa25 text 2110fa25 to 22333

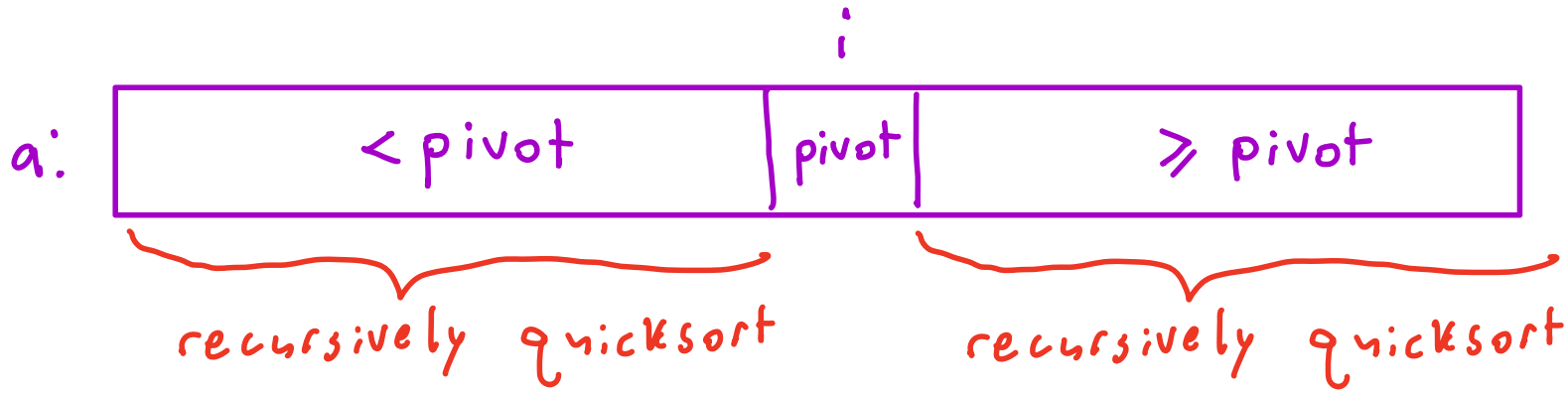
Motivating Quicksort:

After sorting the following array, at which index will 17 be?

17	6	2	25	13	8	31	24	14	62	3	51
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Quicksort

Big Idea: Partition array about some pivot value then recurse on left/right ranges

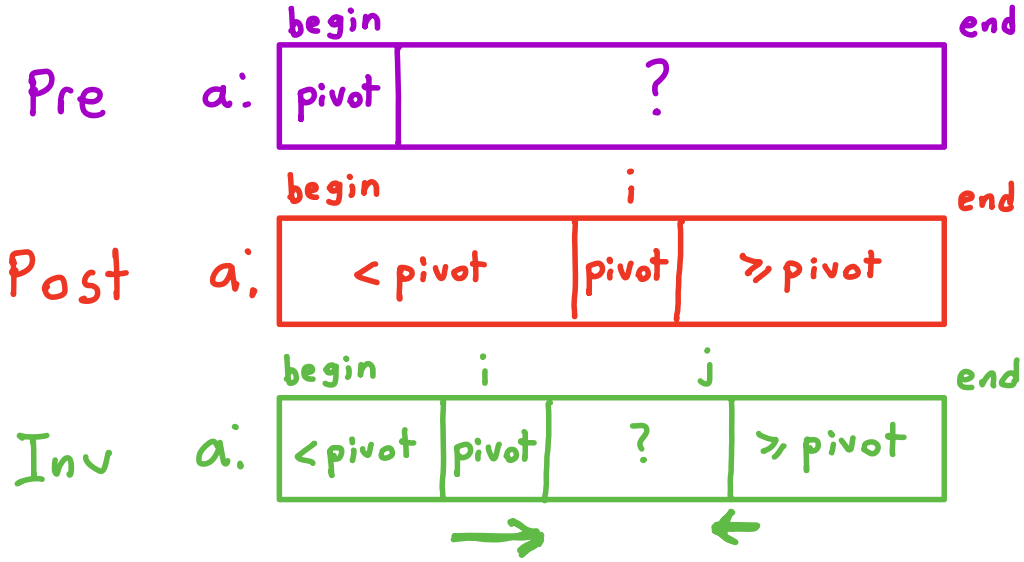


Finding partition is loop invariant problem (see Discussion 3)

For Now: pivot = leftmost entry of range

Coding Demo: `quicksort()`

partition():

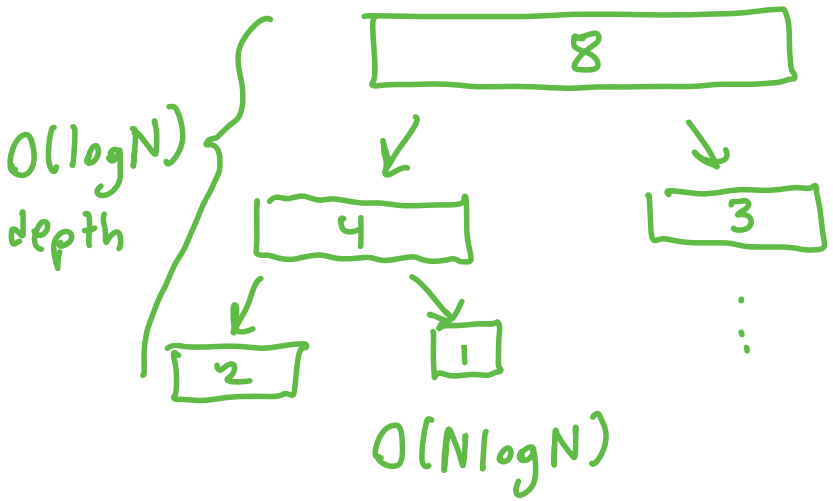


Quicksort Complexity Analysis

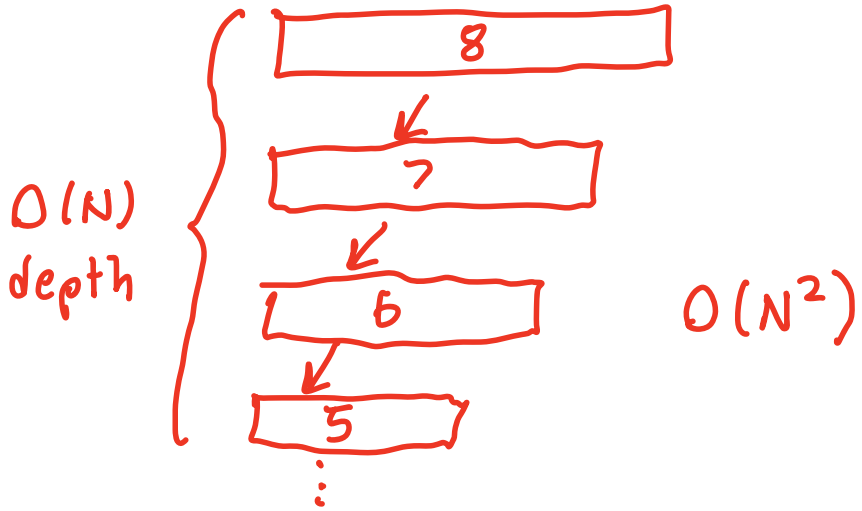
Each call dominated by partition: O(length) time
O(1) space

How many calls?

Ideally: pivot is median



Worst-case: pivot is smallest or largest element



Quicksort: Other Considerations

Being more clever when choosing pivot leads to better performance (see lec notes for ideas)

- Expected (typical) runtime is $O(N \log N)$, often outperforming merge sort in practice

- Not adaptive and not stable

- Default unstable sort in many languages (including Java)

Sorting Summary

No obvious "best approach", many trade-offs

Algorithm	Worst-Case Time Complexity	Expected Time Complexity	Best-Case Time Complexity	Space Complexity	Stable?	Adaptive?
Insertion Sort	$O(N^2)$	$O(N^2)$	$O(N)$	O(1)	Yes	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Yes	No
Quicksort	$O(N^2)$	$O(N \log N)$	$O(N \log N)$	$O(\log N)$*	No	No