Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What is the **space** complexity of this `maxVal()` implementation in terms of N = e – b?

```
1  static int maxVal(int[] nums, int b, int e) {
2    if (e - b == 1) {
3      return nums[b];
4    }
5    int m = b + (e - b)/2;
6    int lMax = maxVal(nums, b, m);
7    int rMax = maxVal(nums, m, e);
8    return Math.max(lMax, rMax);
9  }
```

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

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What is the **space** complexity of this `maxVal()` implementation in terms of $N = e - b$?

Each call frame allocates O(1) memory

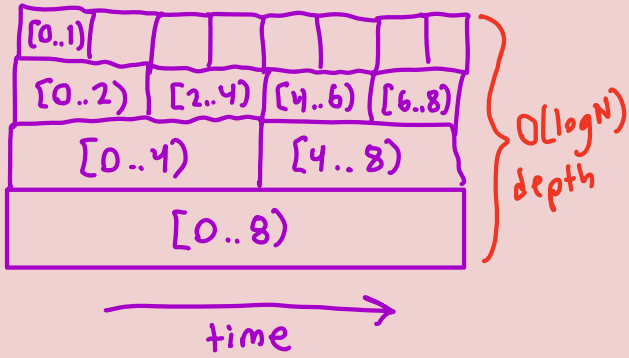
```
1  static int maxVal(int[] nums, int b, int e) {
2    if (e - b == 1) {
3      return nums[b];
4    }
5    int m = b + (e - b)/2;
6    int lMax = maxVal(nums, b, m);
7    int rMax = maxVal(nums, m, e);
8    return Math.max(lMax, rMax);
9  }
```

each call does O(1) nonrecursive work

2N-1 = O(N) calls

so O(N) time complexity

"Call Stack Diagram" when N=8



Lecture 7: Sorting Algorithms

CS 2110

September 15, 2026

Today's Learning Outcomes

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

18. Identify the loop invariant of an iterative method involving an array and visualize it using an array diagram.

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

27. 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

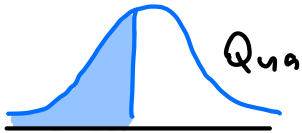
Having sorted data can greatly improve code efficiency.

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

Sorting is an important subroutine for many other computations:

Statistics / Data Analysis

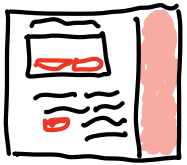
Order Statistics



Quantiles

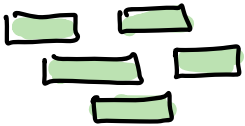
Optimization

Ranking Pages in Search Indices

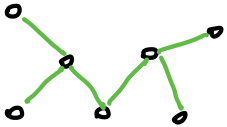


Efficient Graphics Rendering

Designing Greedy Algorithms



Scheduling



Connectivity

Different Sorting Algorithms

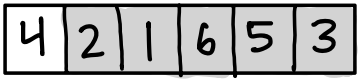
Different sorting procedures trade off different desirable properties:

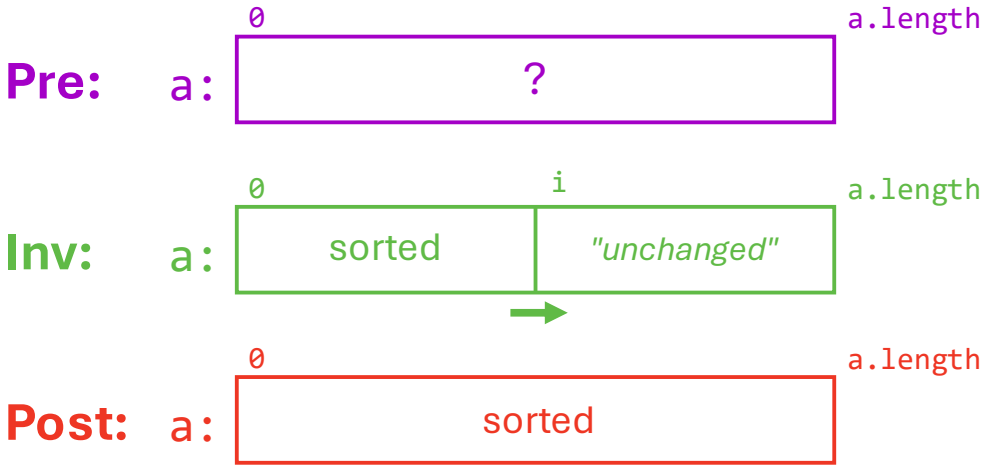
- Runtime (best-case, worst-case, expected case, adaptivity)
- Space complexity
- Memory locality, parallelizability
- Stability
- ⋮

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

Insertion Sort

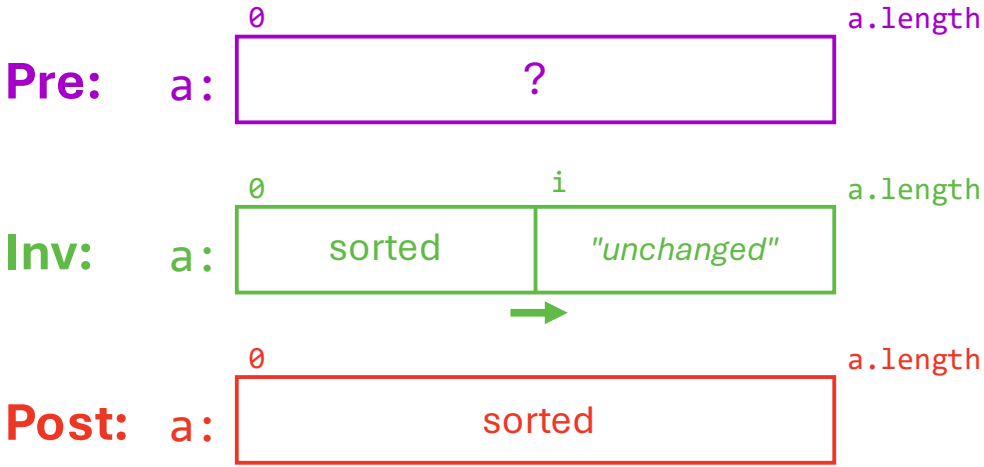
Big Idea: Build up a sorted range by inserting entries into their sorted positions one at a time.

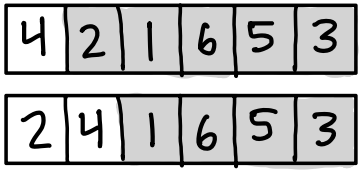




Insertion Sort

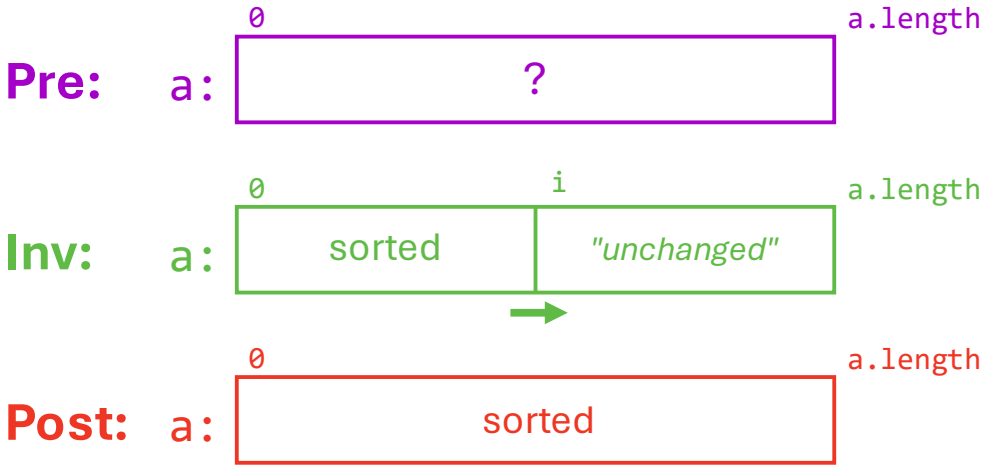
Big Idea: Build up a sorted range by inserting entries into their sorted positions one at a time.

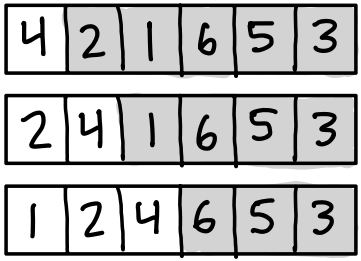




Insertion Sort

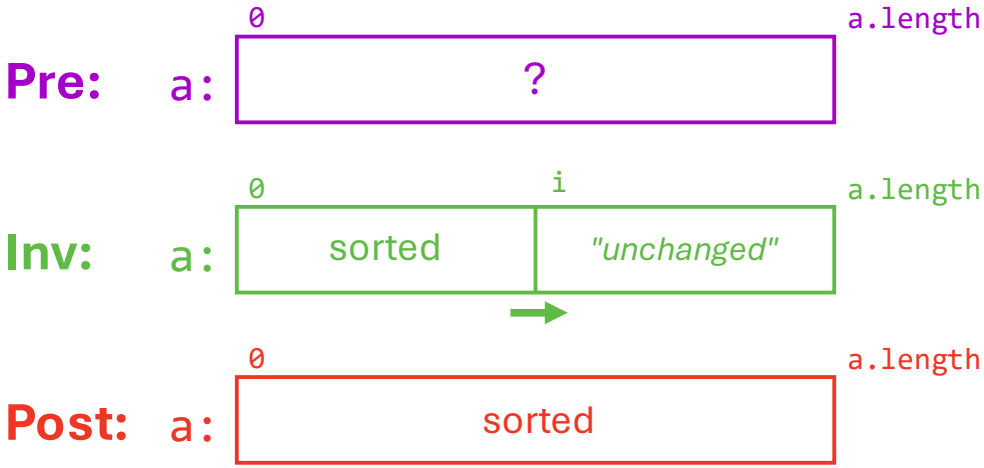
Big Idea: Build up a sorted range by inserting entries into their sorted positions one at a time.

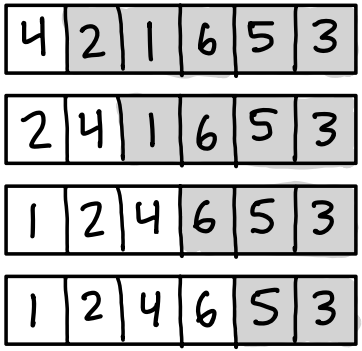




Insertion Sort

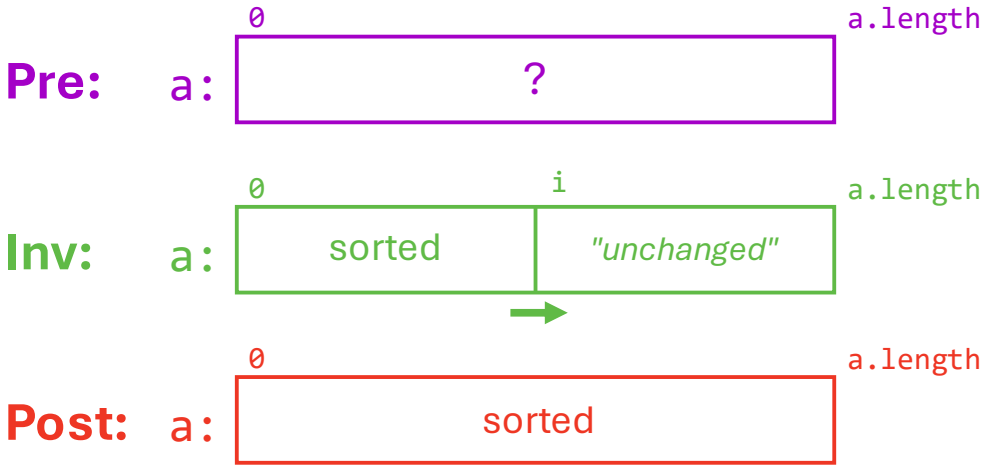
Big Idea: Build up a sorted range by inserting entries into their sorted positions one at a time.

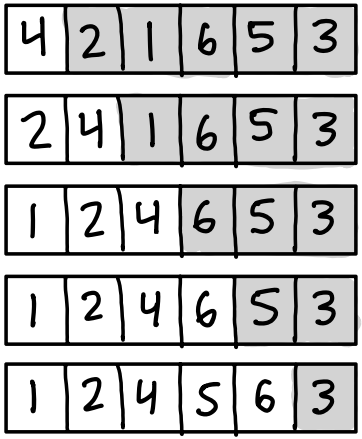




Insertion Sort

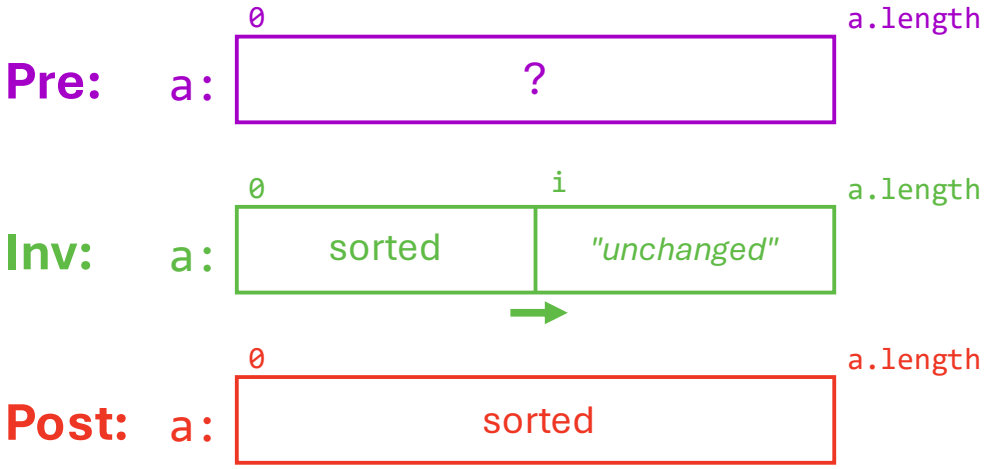
Big Idea: Build up a sorted range by inserting entries into their sorted positions one at a time.

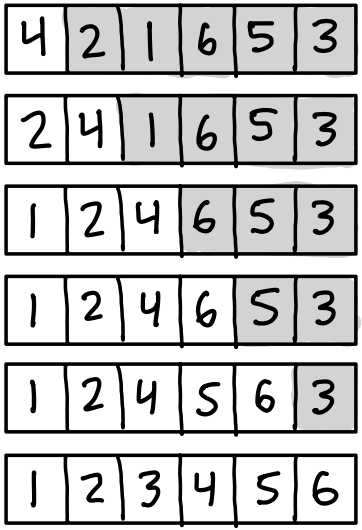




Insertion Sort

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

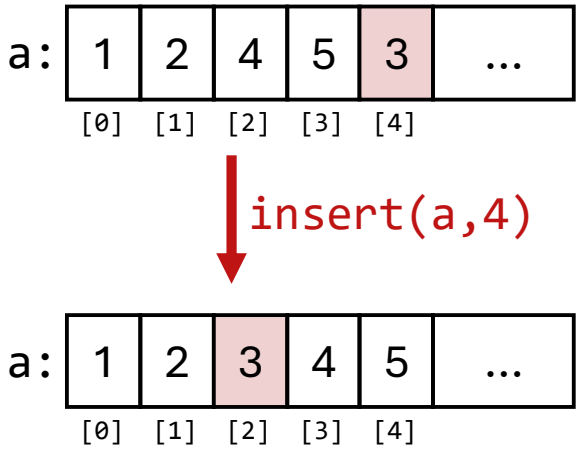




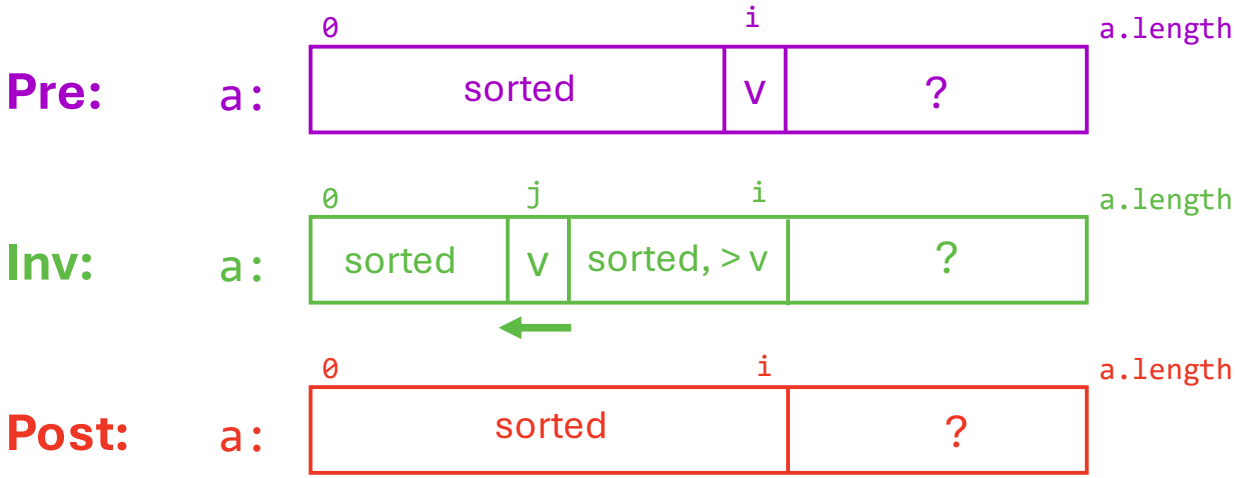
Exercise: The `insert()` helper

Take some time to write the "loopy" insert() implementation.

```
static void insertionSort(int[] a) {
  /* Loop Inv: 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
 *  `0 <= i < a.length` and `a[..i)` is sorted. */
static void insert(int[] a, int i) { ... }
```



`insert()` Array Diagrams



Coding Demo: Insertion Sort

Insertion Sort (Worst-Case) Complexity

```
1  static void insertionSort(int[] a) {
2    for (int i = 0; i < a.length; i++) {
3      insert(a,i);
4    }
5  }
6
7  static void insert(int[] a, int i) {
8    int j = i;
9    while (j > 0 && a[j - 1] > a[j]) {
10     swap(a, j - 1, j); j--;
11   }
12 }
```

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

$$\sum_{i=1}^{N} O(i) = O(N^2)$$

O(i) iterations each does O(1) work

O(i)

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

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

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

```
1  static void insertionSort(int[] a) {
2    for (int i = 0; i < a.length; i++) {
3      insert(a,i);
4    }
5  }
6
7  static void insert(int[] a, int i) {
8    int j = i;
9    while (j > 0 && a[j - 1] > a[j]) {
10     swap(a, j - 1, j); j--;
11   }
12 }
```

O(N)

O(1)

immediately false for sorted array

- $O(1)$ **(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.

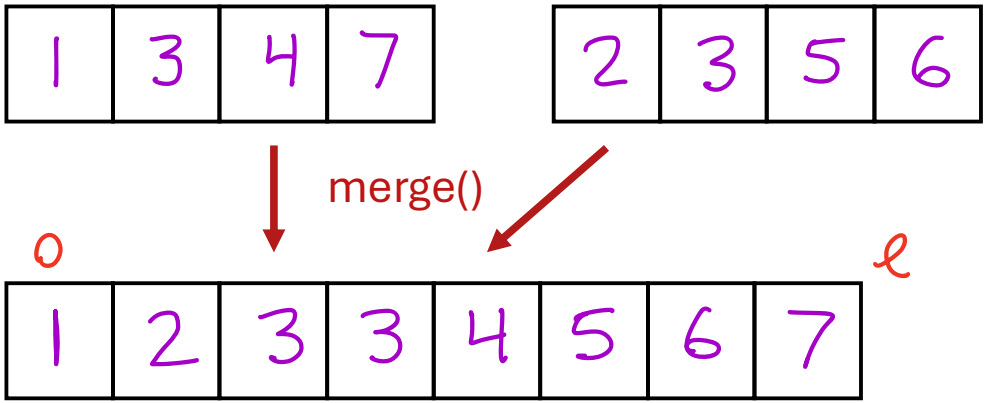
A sorting algorithm is **stable** if it preserves the relative order of equivalent elements.

- Useful when sorting by multiple attributes (A2)

- 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 larger sorted array.

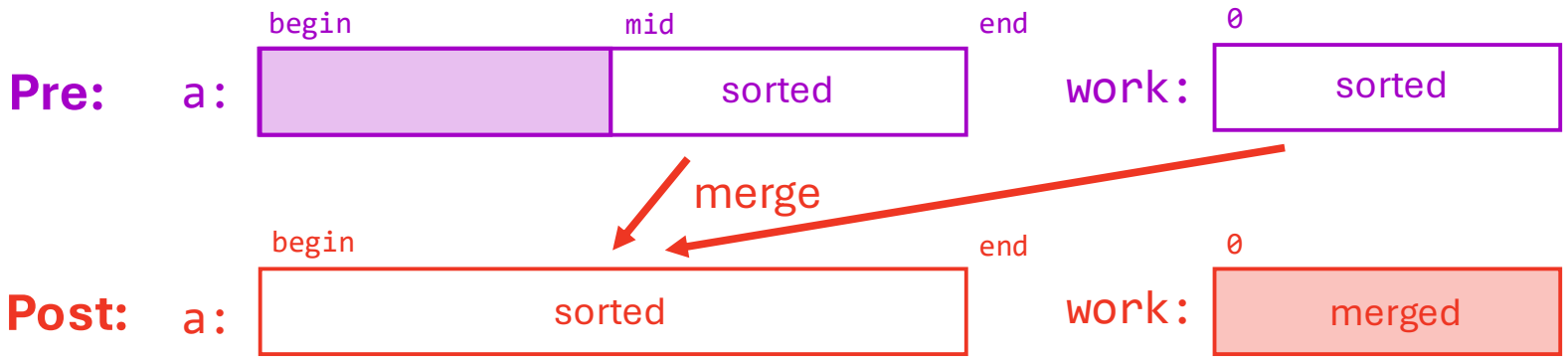


*We only need to look at one entry from each small array to fill one entry of the big array.

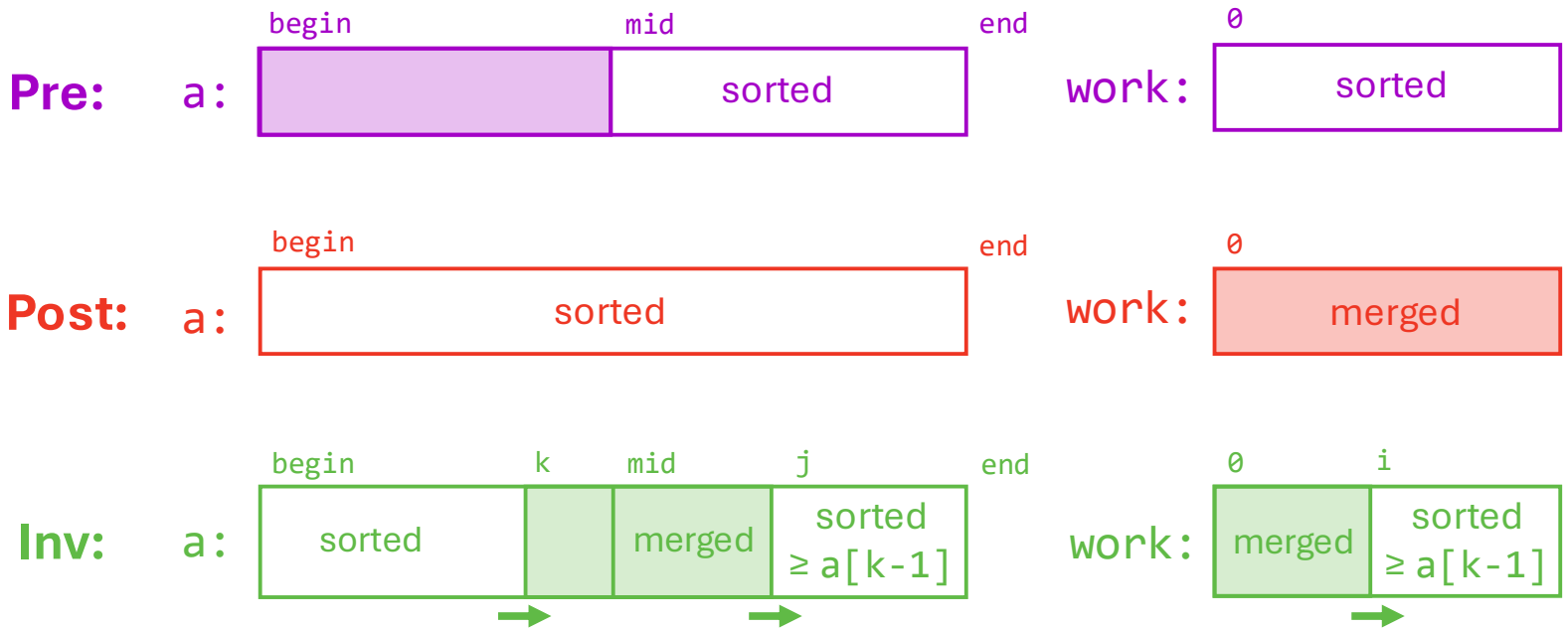
Overall runtime is $O(\ell)$.

The `merge()` Invariant

The `merge()` Invariant



The `merge()` Invariant



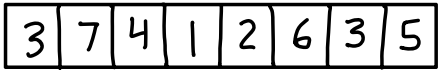
Coding Demo: The `merge()` Method

Merge Sort

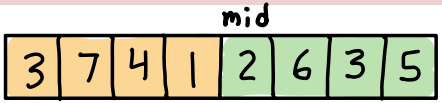
```
static void mergeSort(int[] a) {
  mergeSortRecursive(a, 0, a.length);
}
/**  Recursively sorts `a[begin..end)` using the merge sort algorithm. */
static void mergeSortRecursive(int[] a, int begin, int end) {
  if (end - begin <= 1) { return; }          // base case
  int mid = begin + (end - begin) / 2;
  mergeSortRecursive(a, begin, mid);         // sort left half
  mergeSortRecursive(a, mid, end);           // sort right half

  merge(a, begin, mid, end);                 // merge
}
```

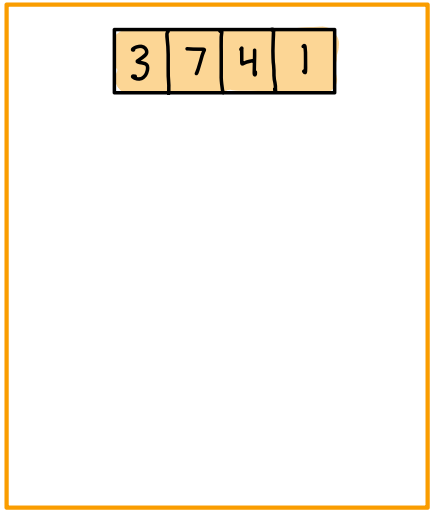
"divide and conquer" algorithm

Visualizing Merge Sort



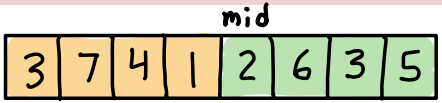
Visualizing Merge Sort

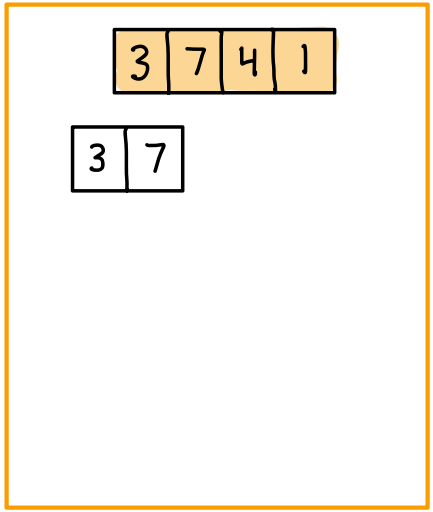




mergeSort
Recursive (a,0,4)

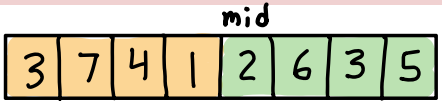
Visualizing Merge Sort

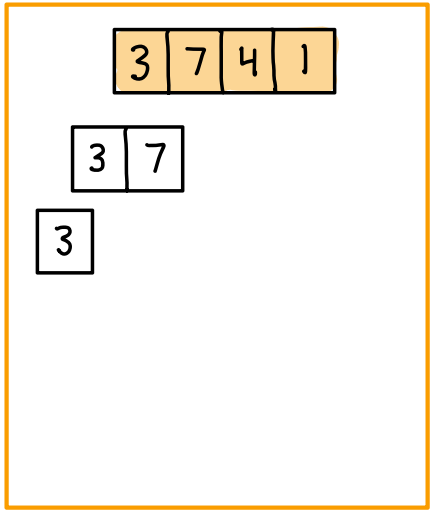




mergeSort Recursive (a, 0, 4)

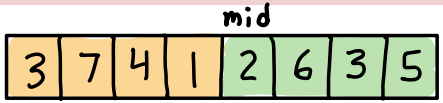
Visualizing Merge Sort

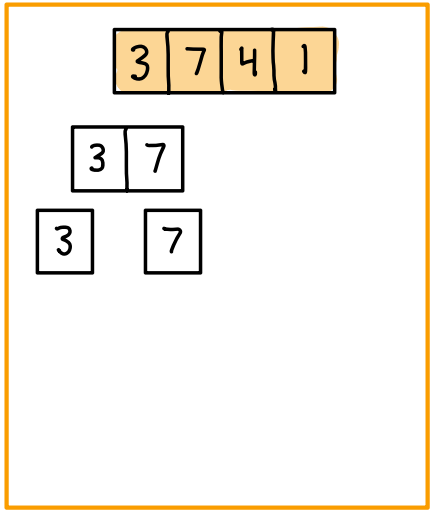




mergeSort Recursive (a, 0, 4)

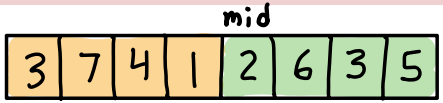
Visualizing Merge Sort

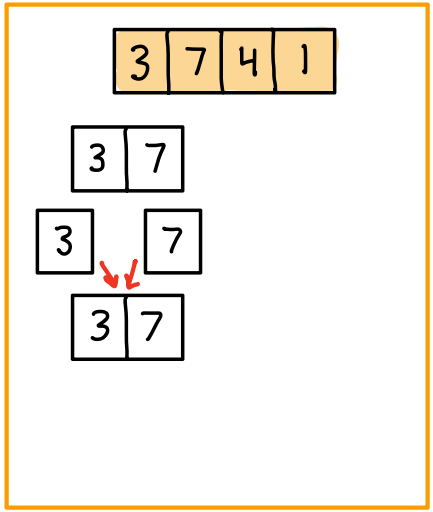




mergeSort Recursive (a, 0, 4)

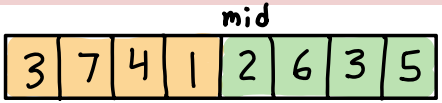
Visualizing Merge Sort



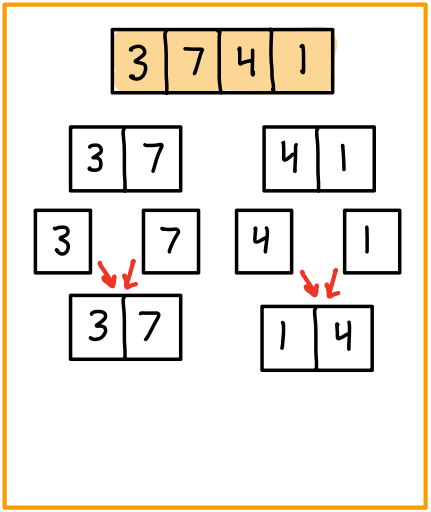


mergeSort Recursive (a, 0, 4)

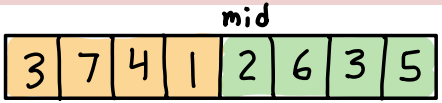
Visualizing Merge Sort



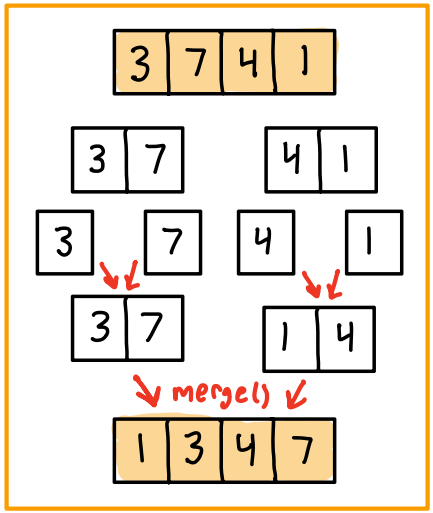
mergeSort Recursive (a,0,4)



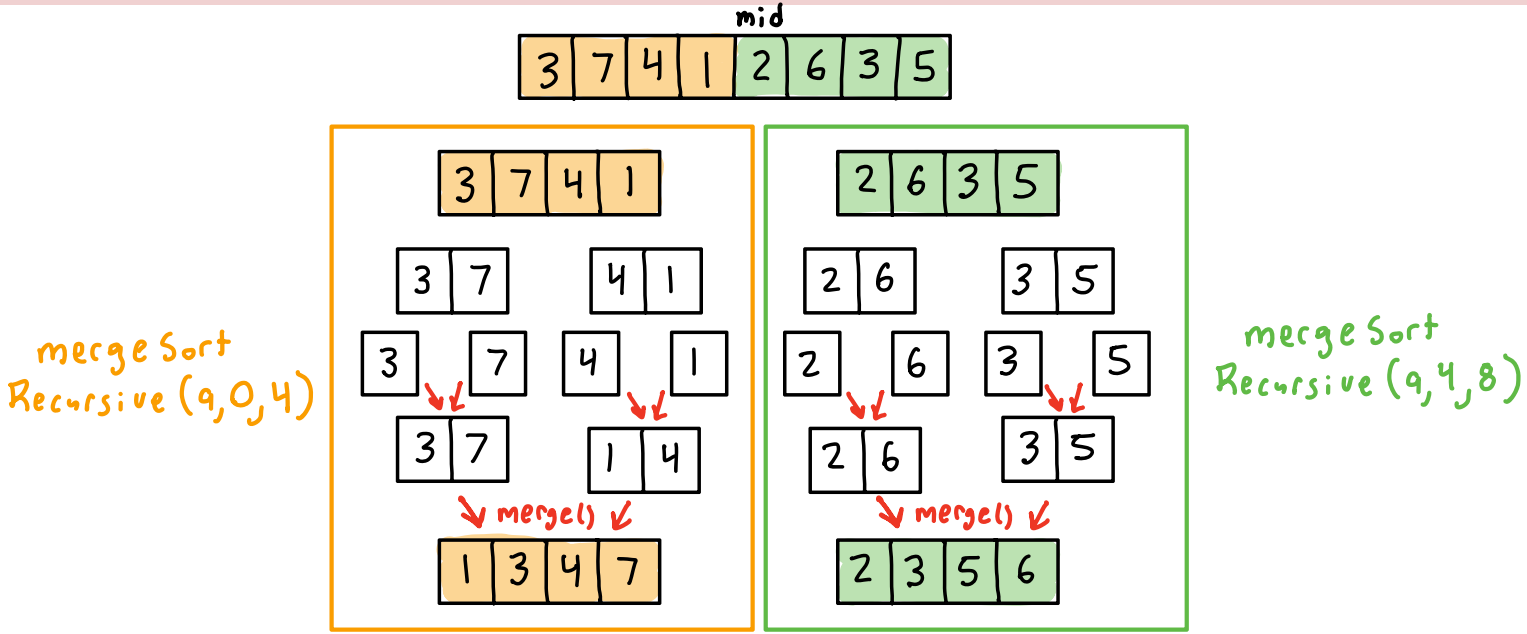
Visualizing Merge Sort



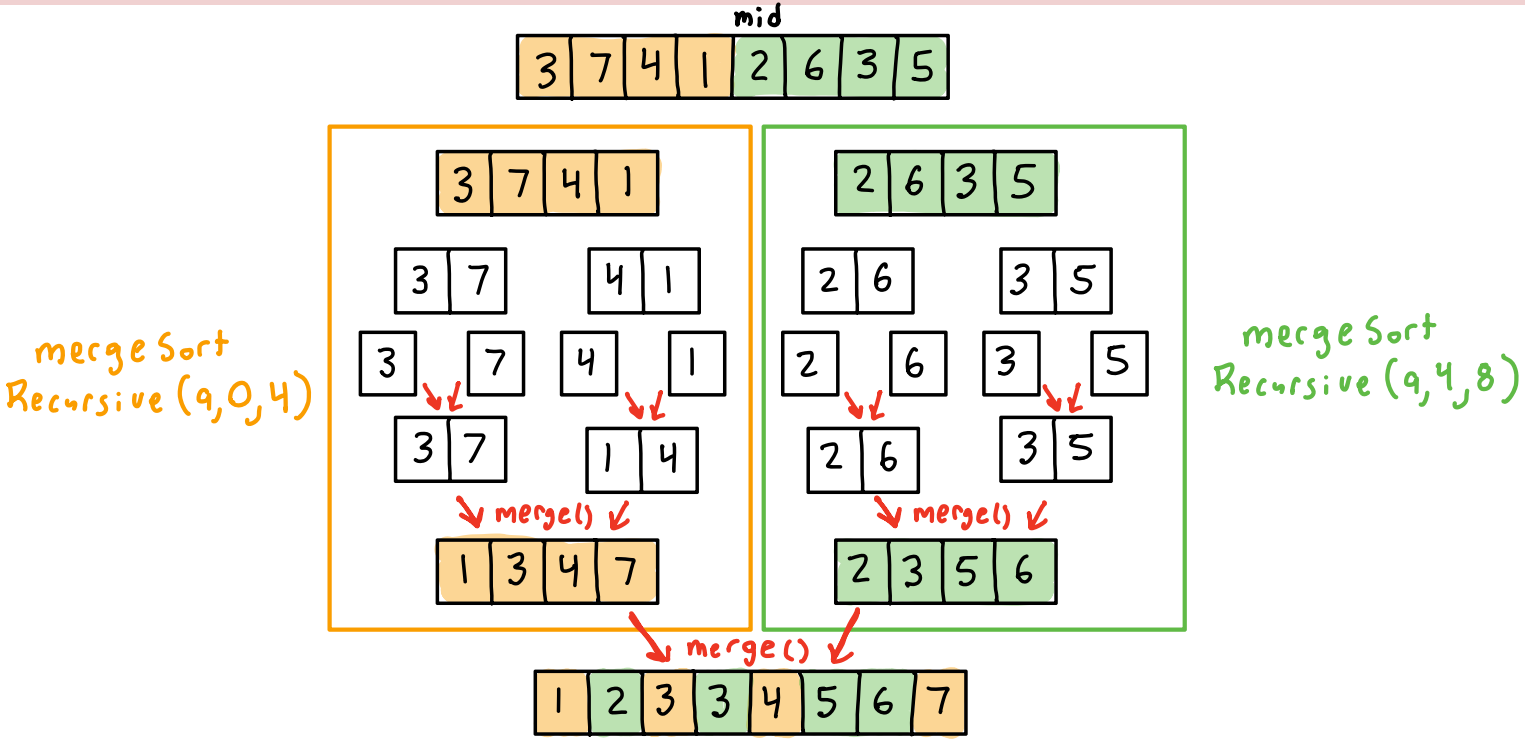
mergeSort Recursive (a,0,4)



Visualizing Merge Sort



Visualizing Merge Sort



Poll Everywhere

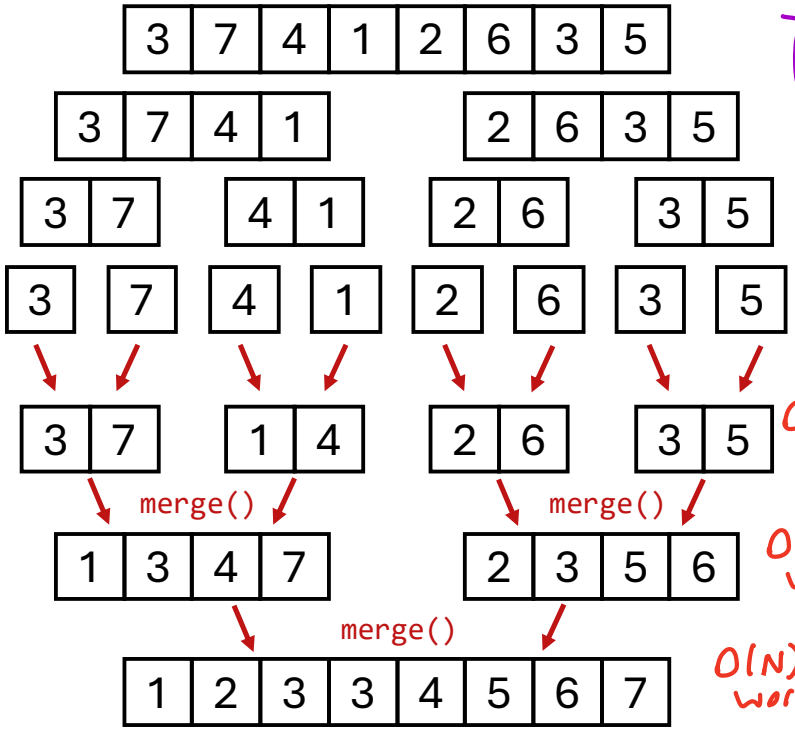
PollEv.com/javabear text **javabear** to 22333

What is the recursive depth of `mergeSort()` in terms of N = `a.length` ?

```
1  static void mergeSort(int[] a) {
2    mergeSortRecursive(a, 0, a.length);
3  }
4
5  static void mergeSortRecursive(int[] a, int begin, int end) {
6    if (end - begin <= 1) {
7      return; // base case
8    }
9    int mid = begin + (end - begin) / 2;
10   mergeSortRecursive(a, begin, mid);
11   mergeSortRecursive(a, mid, end);
12   merge(a, begin, mid, end);
13 }
```

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

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 $\boxed{O(N)}$ space complexity.

Merge Sort: Other Considerations

Merge sort is **stable** (since we prioritize copying from `work` in `merge()`).

Merge sort is **not adaptive**.

$O(N \log N)$ is the best possible worst-case asymptotic runtime for a `swap()`-based sorting algorithm, so merge sort has the optimal runtime.

Merge sort is the default stable sort in many languages (including Java).

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

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

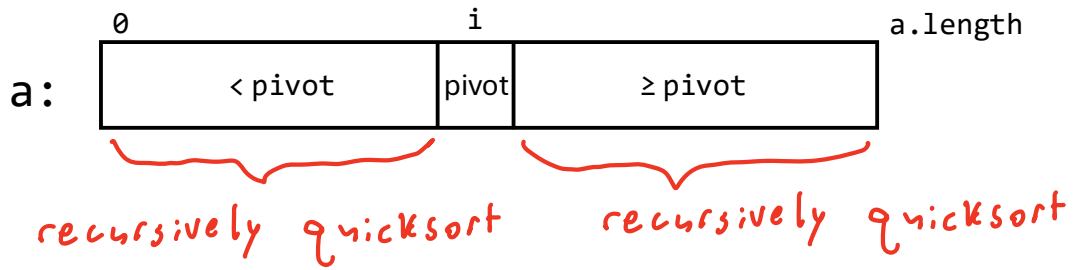
Motivating Quicksort:

At which index will 17 be when the following array is sorted?

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: After we partition an array about some pivot value, we're left with two smaller sorting problems that we handle recursively.



Computing the partition using a loop invariant problem (see Discussion 3).

For now: `pivot` = leftmost entry of range

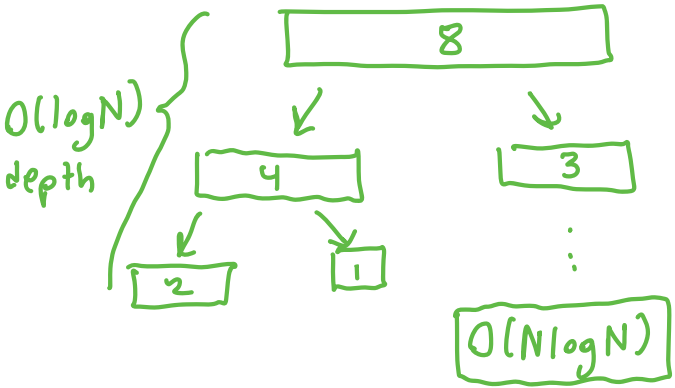
Coding Demo: `quicksort()`

Quicksort Complexity Analysis

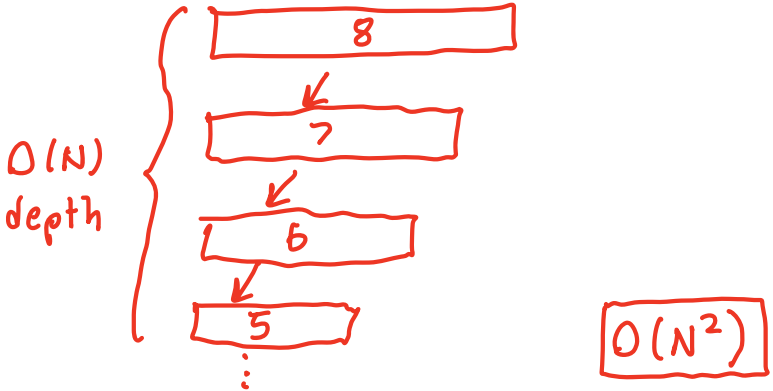
Each call is dominated by the `partition()`, which requires O(length) time and O(1) space.

How many calls are made?

Ideally: pivot is median



Worst-case: pivot is smallest or largest element



Quicksort: Other Considerations

Quicksort is **not adaptive** and **not stable**.

Choosing the pivot in a more sophisticated way leads to good performance.

Quicksort has an **expected** (typical) $O(N \log N)$ runtime, often outperforming merge sort in practice.

Quicksort is the default unstable sort in many languages (including Java).

Sorting Summary

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

No obvious "best" choice, it's all about the trade-offs!