Search Algorithms

Linear Search

Key idea: search linearly through array from front to back to find item

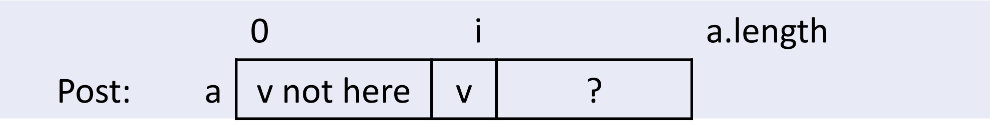
```
/** Returns: the smallest index i such that a[i] == v.
    Requires: v is in a. */
int linear_search(int[] a, int v) {
  int i = 0;

  while (a[i] != v) i++;
  return i;
}
```

Exercise

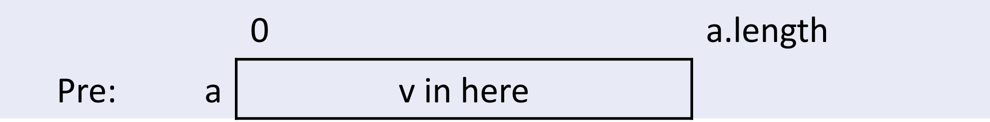
State the loop invariant.

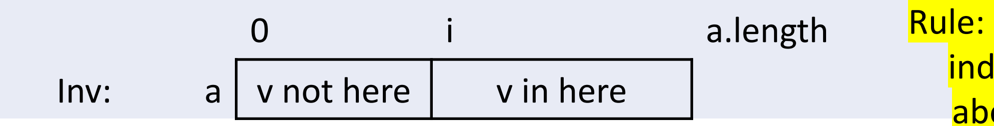
```
/** Returns: the smallest index i such that a[i] == v.
    Requires: v is in a. */
int linear_search(int[] a, int v) {
  int i = 0;
  // inv: TODO
  while (a[i] != v) i++;
  return i;
}
```

Discovering the loop invariant



```
/** Returns: the smallest index i such that a[i] == v.
    Requires: v is in a. */
```





in array diagrams

Rule: We never draw indices directly above a line in diagram! Always to left or right.

Post:
v not in a[0..i)
a[i] == v

Pre:
v in a[0..]

Inv:
v not in a[0..i)
v in a[i..] *in math*

Discovering the loop invariant

```
/** Returns: the smallest index i such that a[i] == v.
    Requires: v is in a. */
```

Discovering an invariant from the pre and post conditions requires creativity and practice.

Theorem. There is no algorithm that can do it for you.
Corollary: ChatGPT can’t replace human programmers yet!

Inv:
v not in
a[0..i)
v in a[i..]

Linear Search: with invariant

```
/** Returns: the smallest index i such that a[i] == v.
    Requires: v is in a. */
int linear_search(int[] a, int v) {
  int i = 0;
  // inv: v not in a[0..i), and v in a[i..]
  while (a[i] != v) i++;
  return i;
}
```

Linear Search: loop checklist

```
/** Returns: the smallest index
      i such that a[i] == v.
    Requires: v is in a. */
int linear_search(int[] a, int v) {
  int i = 0;
  // inv: v not in a[0..i), and v in a[i..]
  while (a[i] != v) i++;
  return i;
}
```

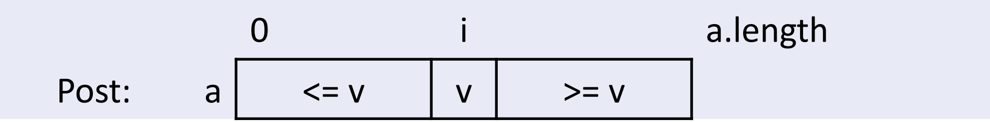
- ☐ Does it start right?
- ☐ Does it maintain the invariant?
- ☐ Does it end right?
- ☐ Does it make progress?

Binary Search

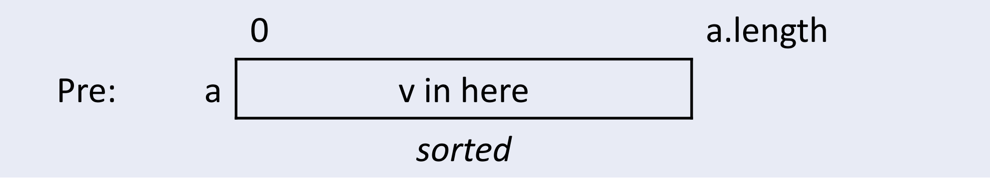
Key idea: maintain upper and lower bounds on where value could be.

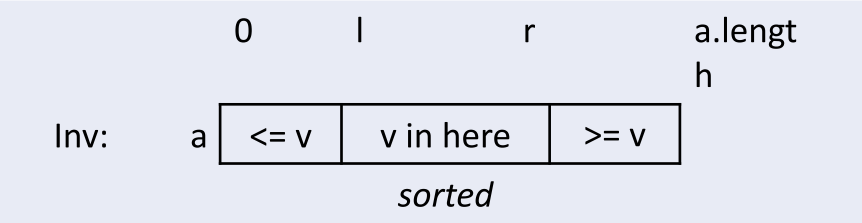
```
/** Returns: an index i such that a[i] == v.
    Requires: v is in a, and a is sorted in ascending order. */
int bin_search(int[] a, int v) {
  int l = 0;
  int r = a.length - 1;
  // inv: 0 <= l <= r < a.length, and v in a[l..r]
  while (l != r) {
    int m = (l + r) / 2;
    if (v <= a[m]) { r = m; }
    else { l = m + 1; }
  }
  return l;
}
```

Understanding the loop invariant



```
/** Returns: an index i such that a[i] == v.
    Requires: v is in a, and a is sorted in ascending order. */
```





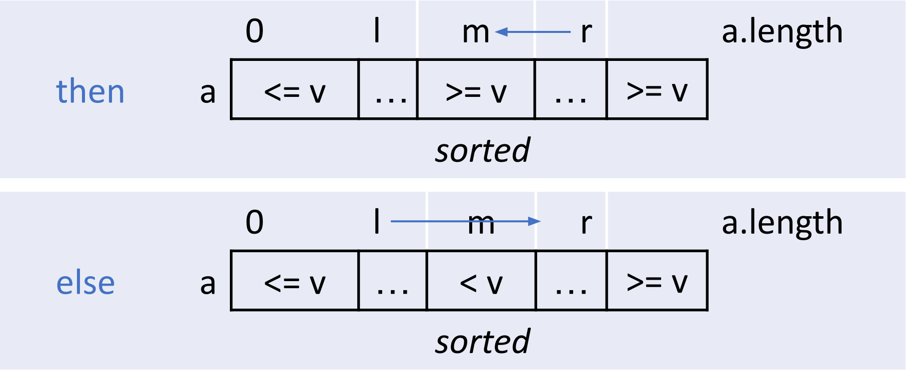
Nothing about the loop invariant requires halving the search space!

Efficiency is distinct from **correctness.**

Binary Search: loop checklist

```
/** Returns: an index i such that a[i] == v.
    Requires: v is in a, and a is sorted in ascending order.
*/
int bin_search(int[] a, int v) {
  int l = 0;
  int r = a.length - 1;
  // inv: 0 <= l <= r < a.length, and v in a[l..r]
  while (l != r) {
    int m = (l + r) / 2;
    if (v <= a[m]) { r = m; }
    else { l = m + 1; }
  }
  return l;
}
```

- ☐ Does it start right?
- ☐ Does it maintain the invariant?
- ☐ Does it end right?
- ☐ Does it make progress?



Loops (in)variants are your friends.

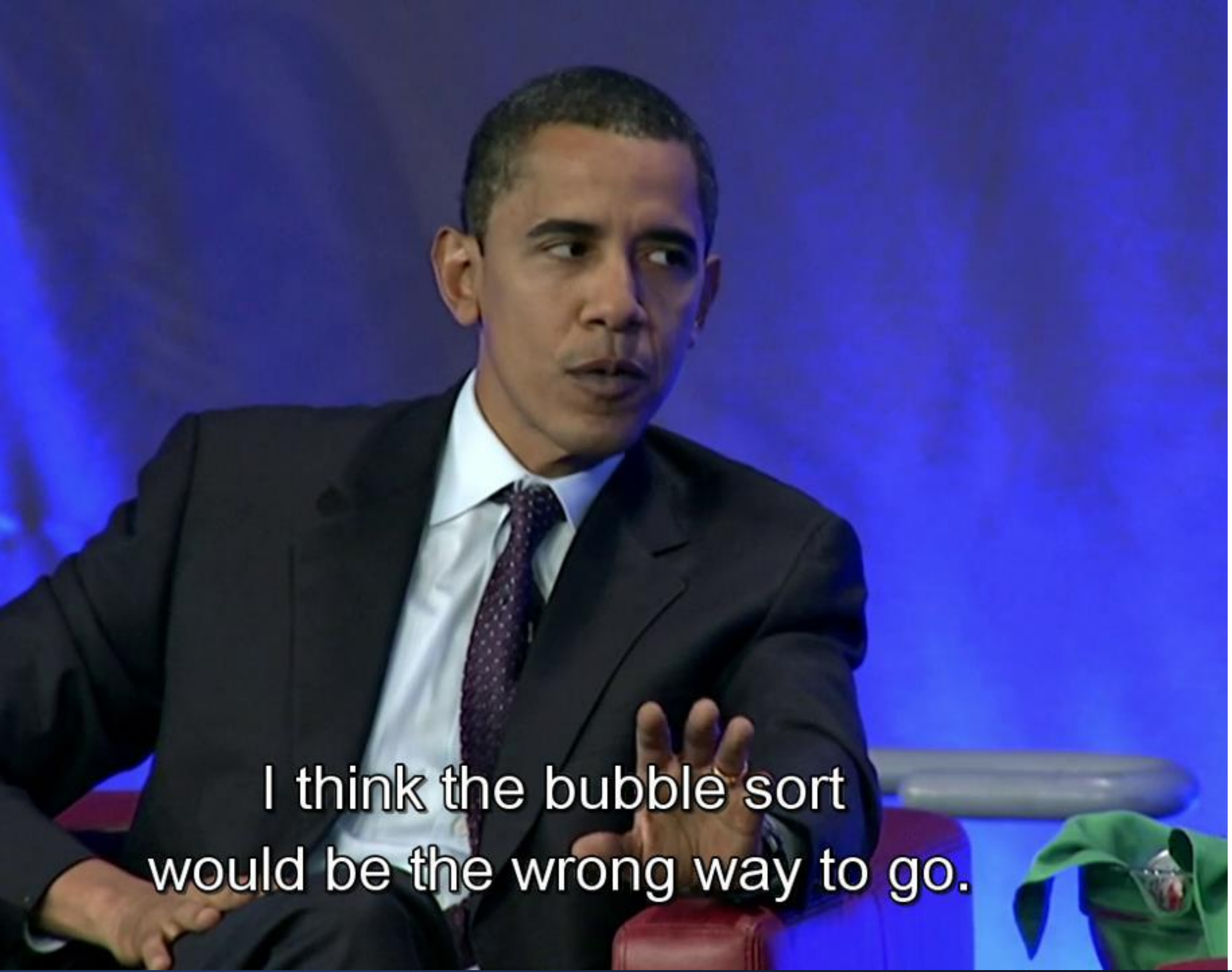
CS 2110 Lecture 12?

Sorting

- Selection sort
- Insertion sort
- Merge sort
- Quicksort

Why sort things?

- Makes looking things up faster
  - Binary search
- Compute robust statistics
  - Median, quantiles
  - Top-10 lists
- Prioritize/optimize
  - Search results
  - Drawing order



Why multiple algorithms?

- Tradeoffs: no one “best” algorithm
    - Speed
    - Memory
    - Expected vs. worst case
    - Stability
    - R/W locality
- You will be responsible for choosing appropriate methods

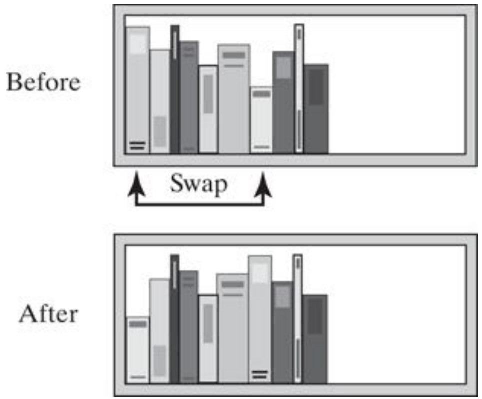
Setting: arrays

- Why arrays?
  - Data to be sorted is often in an array (or ArrayList)
  - Arrays are familiar
  - Good opportunity to visualize loop invariants with array diagrams
- Implications
  - Fast to read/write arbitrary locations, iterate in reverse
  - Swaps are cheap
  - Insertions are expensive
- Most algorithms generalize to linked structures

Selection sort

Analogy: bookshelf

- Find the shortest remaining (unsorted) book
- Move it just after all the already sorted (and shorter) books

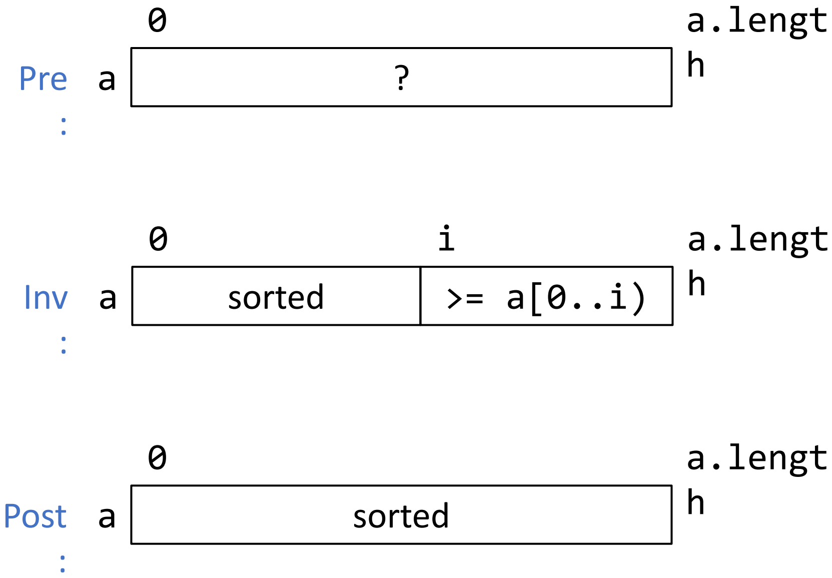


- How to "move" it?
  - Push subsequent books out of the way
    - Difficult; analogous to insertion
  - Trade places with book in desired position
    - Easy; analogous to swapping

Selection sort example

i=0	**3**	**1**	**4**	**1**	**5**
i=1					
i=2					
i=3					
i=4					

Selection sort invariant



Pre: a

0 ? a.length

Inv: a

0 sorted i `>= a[0..i)` a.length

Post: a

0 sorted a.length

Selection sort code

```java
// Invariant: a[0..i) is sorted, a[i..] >= a[0..i)
int i = 0;
while (i < a.length - 1) {
    // Find index of smallest element in a[i..]
    int jSmallest = i;
    for (int j = i + 1; j < a.length; ++j) {
        if (a[j] < a[jSmallest]) {
            jSmallest = j;
        }
    }
    // Swap smallest element to extend sorted portion
    swap(a, i, jSmallest);
    i += 1;
}
```

- Time complexity analysis (N = `a.length`)
  - i=0: N-1 comparisons
  - i=1: N-2 comparisons
  - i=2: N-3 comparisons
  - …
  - i=N-2: 1 comparison

  - Total comparisons: 1 + 2 + … + (N-1)

$$O(N^2)$$

Algorithm properties

Algorithm	Best case time complexity	Worst case time complexity	Space complexity
Selection sort			

Insertion sort

Analogy: a hand of playing cards

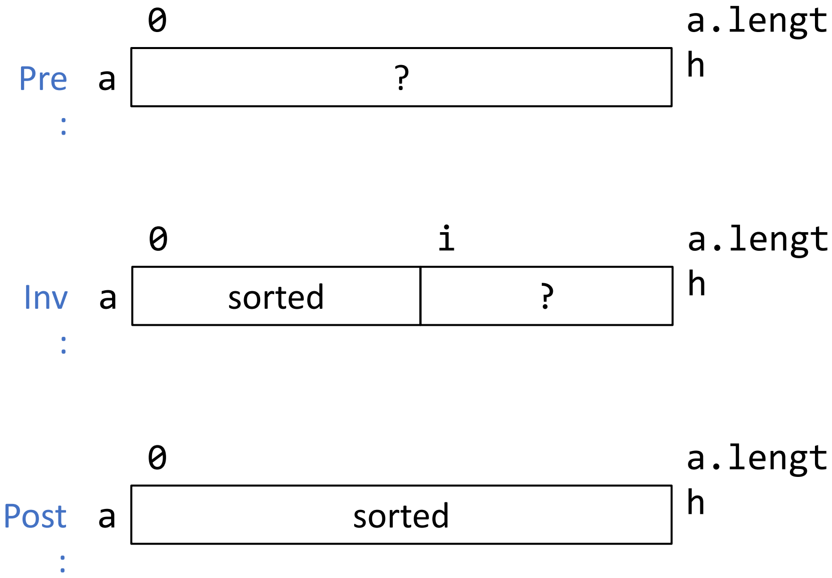
- Left hand holds cards that have already been sorted
- Take next card from right hand, insert it where it belongs in left hand

- How to "insert"
  - Push all bigger cards out of the way
  - Swap with cards to left until in position

Insertion sort example

`i=0`	**3**	**1**	**4**	**1**	**5**
`i=1`					
`i=2`					
`i=3`					
`i=4`					

Insertion sort invariant



Pre: a

0 ? a.length

Inv: a

0 sorted i ? a.length

Post: a

0 sorted a.length

Insertion sort code

```
// Invariant: a[0..i) is sorted
int i = 0;
while (i < a.length) {
    // Slide a[i] to its sorted position in a[0..i]
    // Invariant: a[j] < a[j+1..i]
    int j = i;
    while (j > 0 && a[j - 1] > a[j]) {
        swap(a, j - 1, j);
        j -= 1;
    }
    i += 1;
}
```

- Time complexity analysis (N = `a.length`)
  - i=1: 1 comparison
  - i=2: < 2 comparisons
  - i=3: < 3 comparisons
  - …
  - i=N-1: < N-1 comparisons
  - Total comparisons (worst-case): 1 + 2 + … + (N-1)

$$O(N^2)$$

Poll: Complexity if array is already sorted?

- How many comparisons does Insertion Sort evaluate if the array is already sorted?

  (*N* is the number of elements in the array)

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

Insertion sort extras

- What if array is already sorted?
  - Each "insert" requires only 1 comparison
  - Overall complexity (best case) is $\Omega(N)$
- Fast in practice for *small N*
  - Often used as a "base case" in implementations of other algs

- What if there are duplicates?
  - E.g. sorting Students by last name
  - **Stable**: relative order of equal elements is preserved

  - Insertion sort is stable because elements only move right-to-left and stop when they hit a duplicate
  - Selection sort is *not* stable because long-range swaps can change order

Algorithm properties

Algorithm	Best case time complexity	Worst case time complexity	Space complexity	Stability
Selection sort				Unstable
Insertion sort				Stable

Merge sort

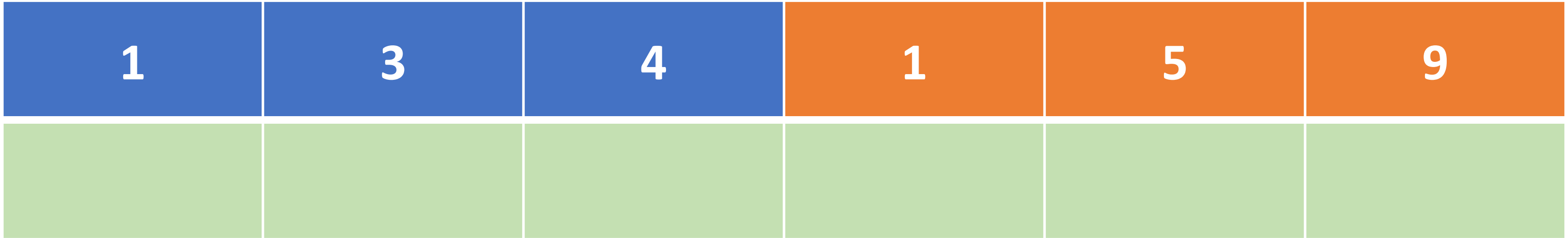
Merging sorted subarrays

- Given two sorted sequences, how hard is it to merge them?
  - Easy! Repeatedly take the smaller of what’s left of the two sequences
  - Complexity: $O(N)$ – easier than sorting (but requires $O(N)$ scratch space)
- What if, when tasked to sort, you outsourced the job to *two* assistants, who each sorted half of the list
  - Their jobs are easier (maybe much easier), since their lists are smaller
  - Your job is easier, since you only have to merge
- What if your assistants outsourced their tasks…?

Divide and conquer

- Divide task into *multiple* smaller subtasks, then assemble results into solution
- Natural fit for recursion

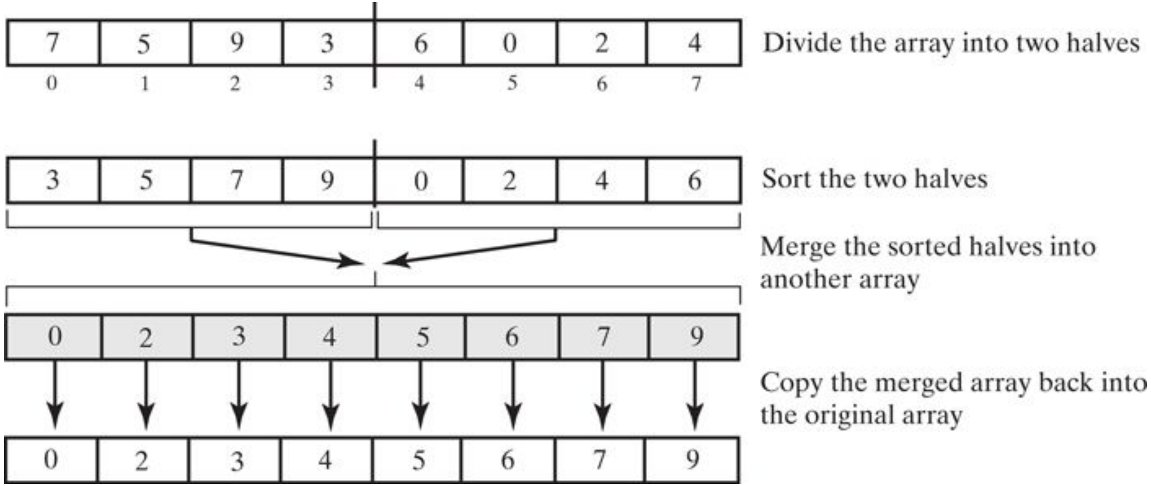
Merge example



1	3	4	1	5	9

Merge sort (high level)

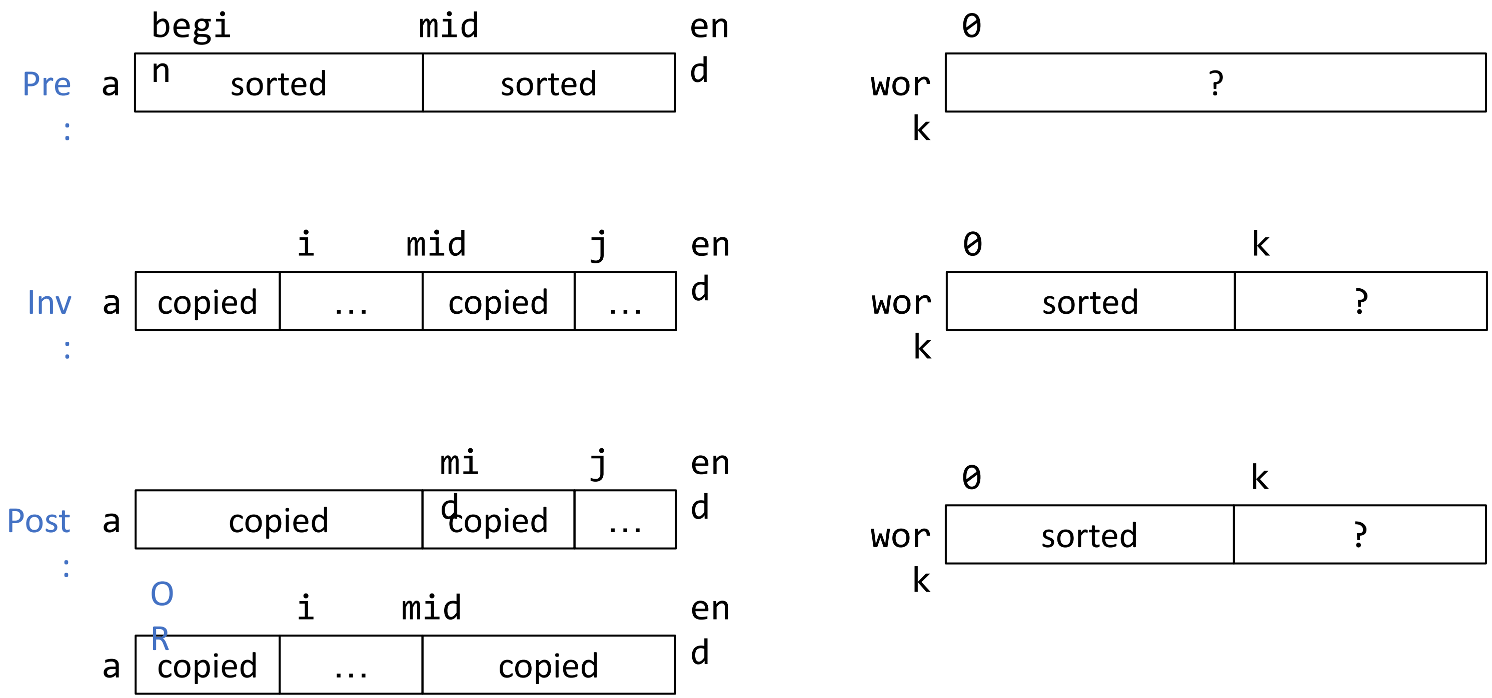
1. Sort left half of array (using merge sort)
2. Sort right half of array (using merge sort)
3. Merge left and right subarrays



Merge sort code

- Demo

Merge invariant

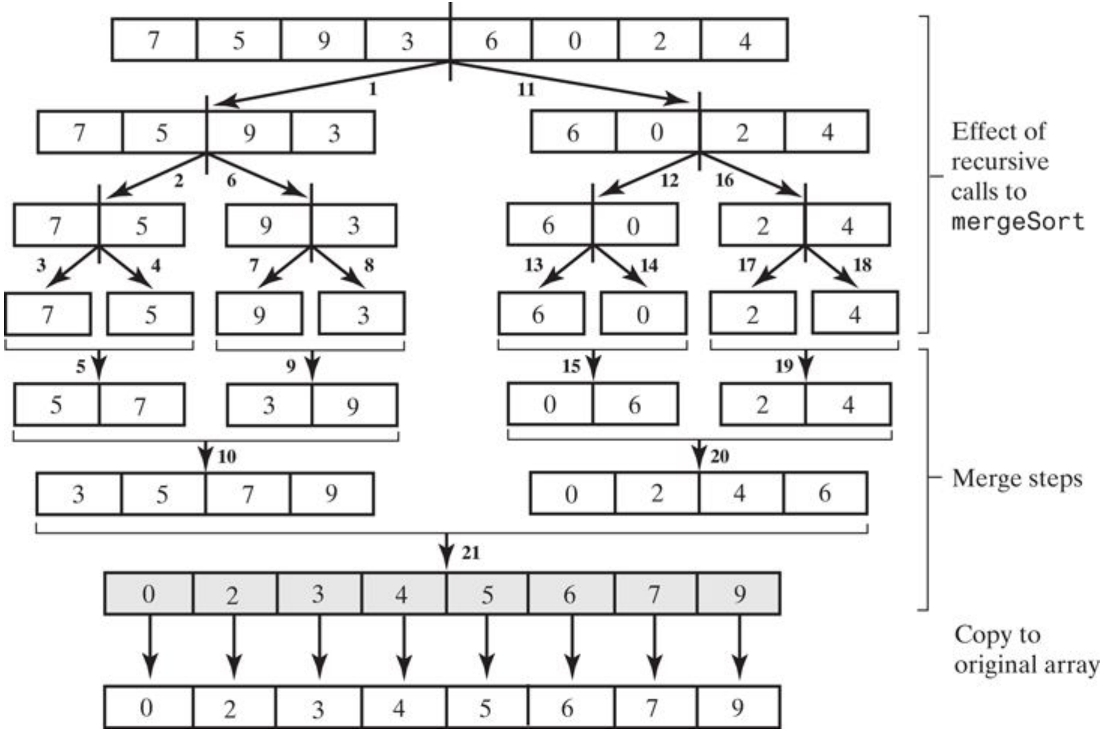


Pre:
a: begin | sorted | mid | sorted | end
work: 0 | ?

Inv:
a: copied | i … | mid copied | j … | end
work: 0 sorted | k ?

Post:
a: copied | mid copied | j … | end
OR
a: copied | i … | mid copied | end
work: 0 sorted | k ?

Analysis


7 5 9 3 6 0 2 4
1
11
7 5 9 3
6 0 2 4
2
6
12
16
7 5
9 3
6 0
2 4
3
4
7
8
13
14
17
18
7
5
9
3
6
0
2
4
5
9
15
19
5 7
3 9
0 6
2 4
10
20
3 5 7 9
0 2 4 6
21
0 2 3 4 5 6 7 9
0 2 3 4 5 6 7 9
Effect of recursive calls to mergeSort
Merge steps
Copy to original array

Algorithm properties

Algorithm	Best case time complexity	Worst case time complexity	Space complexity	Stability
Selection sort				Unstable
Insertion sort				Stable
Merge sort				Stable

Merge sort in practice

- Usually the go-to stable sort (default in many language libraries)
- Since merging is always left-to-right, can be performed on data that does not fit in RAM

Quicksort

Quicksort on one slide

```
sort(a) = [ sort(a[a<p]), p, sort(a[a>=p]) ]
```

1. Partition array about a “pivot”
2. Sort the subarray of values less than the pivot
3. Sort the subarray of values greater than the pivot

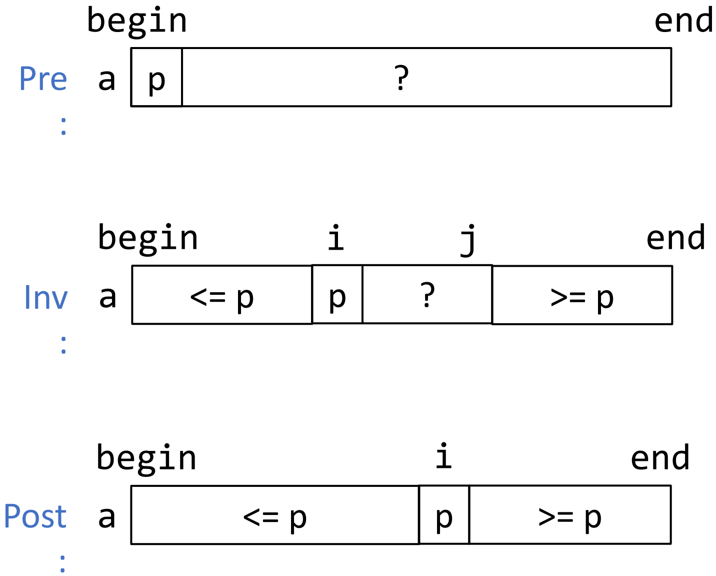
Sort via repeated partitioning

- How efficient is partitioning?
- How many times will you need to partition?

Partition example

3	1	4	1	5	9

Partition invariant



begin end

Pre: a | p | ? |

begin i j end

Inv: a | <= p | p | ? | >= p |

begin i end

Post: a | <= p | p | >= p |

Quicksort code

- Demo

Analysis

Best case

- Pivot is median value
- Each subarray is less than half the size of the original
- Depth of recursion: $O(\log N)$
  Cost of partitioning one level: $O(N)$
- Overall complexity: $\Omega(N \log N)$

Worst case

- Pivot is smallest (or largest) value
- One subarray is only 1 element shorter than original array
- Dept of recursion: $O(N)$
  Cost of partitioning one level: $O(N)$
- Overall complexity: $O(N^2)$

Choice of pivot

- Using first value is a bad choice!
  - In practice, many arrays are partially sorted
- Computing true median is not cost-effective
- Common heuristic: `med3(a[begin], a[mid], a[end-1])`

- Consequences of a bad pivot can be severe!
  - “Complexity attacks” to deny service

Algorithm properties

Algorithm	Best case time complexity	Worst case time complexity	Space complexity	Stability
Selection sort				Unstable
Insertion sort				Stable
Merge sort				Stable
Quicksort				Unstable

* Naïve implementation requires $O(N)$ worst-case space, but can use tail recursion to reduce to $O(\log N)$.

Quicksort in practice

- Despite poor worst-case complexity, Quicksort is often the fastest sort in practice (default unstable sort in many language libraries)
- Often augmented to detect and avoid worst-case behavior (e.g. fall back to heap sort)