05

October 1-7, 2024

Sorting Algorithms, Binary Search

Announcements

- **Project 2 is due on October 11th at 11:59 PM**
- Lab 4 autograder and quiz are due on October 7th at 11:59 PM
- Lab 5 handwritten is due IN LAB by October 7th. DO NOT CALL SORT()!!!
- Lab 5 quiz is due on October 14th at 11:59 PM
  - No autograder portion (quiz is worth 15 points)

- **Midterm: Thursday, October 17th from 7:00 PM to 9:00 PM**
  - If you need an alternate time, or SSD accommodations, you must complete the alternate exam request form as soon as possible!

- **No labs until October 22nd!**

Agenda

- Midterm Topics
- Sorting Algorithms
- Binary Search
- Handwritten Problem

Midterm Topics

Midterm Topics to Review

- (Asymptotic) Complexity and Runtime Analysis, Math Foundations
- Recursion and the Master Theorem
- Container Data Structures and Array-Based Containers
- The Standard Template Library (STL)
- Stacks, Queues, and Priority Queue ADTs
- Ordered Arrays and Related Algorithms
- Set Operations (Union/Intersection) and Union-Find
- Elementary Sorts and Library-Implemented Sorts
- Quicksort and Mergesort
- Heaps and Heapsort
- Linear-Time Sorting Algorithms

Additional Reminders

- Review all your lecture and lab notes!
- Understand how to do all the lab coding exercises and projects!
- Check out [Piazza @1137](#) for additional (leetcode-style) FRQ practice
  - Midterm topics include sorting, binary search, union-find, stacks/queues, linked list, and array
  - Not required for success, simply a resource that's available
- Bring a pencil, eraser, MCard, and an 8.5"x11" note sheet (both sides OK)
- No calculators or other electronic devices

Types of Iterators

- Understand the differences between different types of iterators
- Further reading here: https://ajzhou.gitlab.io/eecs281/notes/chapter11/

Iterators in the STL can be placed into five different categories based on the operations they support. These five categories are:
- *Input iterators:* read only, forward moving (single pass only)
- *Output iterators:* write only, forward moving (single pass only)
- *Forward iterators:* forward moving (multiple passes allowed)
- *Bidirectional iterators:* forward and backward moving
- *Random access iterators:* provides random access

	Input	Output	Forward	Bidirect.	Random
Supports dereference ($*$) and read	✓		✓	✓	✓
Supports dereference ($*$) and write		✓	✓	✓	✓
Supports forward movement (++)	✓	✓	✓	✓	✓
Supports backward movement (−−)				✓	✓
Supports multiple passes			✓	✓	✓
Supports == and !=	✓		✓	✓	✓
Supports pointer arithmetic (+, −, etc.)					✓
Supports pointer comparison (<, >, etc.)					✓

Sorting Algorithms

Sorting Algorithms

- What to consider?
  - time complexity
  - space complexity
  - recursion
    - is it tail recursive?
    - if not, what is the stack depth?
  - stability
    - does it preserve the original order of elements that are equal?
  - adaptability
    - does it perform optimizations based on the order the elements are already in?

Overview of Sorts

- Elementary sorts:
  - bubble sort
  - selection sort
  - insertion sort
- More advanced sorts:
  - mergesort
  - quicksort
  - heapsort
  - counting sort
- The following website is a great resource for visualizing these sorts!
  - https://visualgo.net/en

Bubble Sort

Bubble Sort

- If you're ever asked a sorting question during an interview...

Bubble Sort

- Characteristics of bubble sort:

Best Case	Average Case	Worst Case	Memory	Stable?
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Yes

- Not used in practice because it is inefficient for large datasets.
- Algorithm: Repeatedly step through list to be sorted, comparing each pair of adjacent items and swapping if they are in the wrong order. Each time pass through a smaller section of the list until no more swaps are needed.
- Stable because swaps only happen for elements with different values.
- Adaptive bubble sort accomplishes best-case $\Omega(n)$ runtime: keep a boolean "swap" flag to indicate if any swaps were made during each pass of the array - if not, then the array is already sorted!

Bubble Sort

- Bubble sort in action:
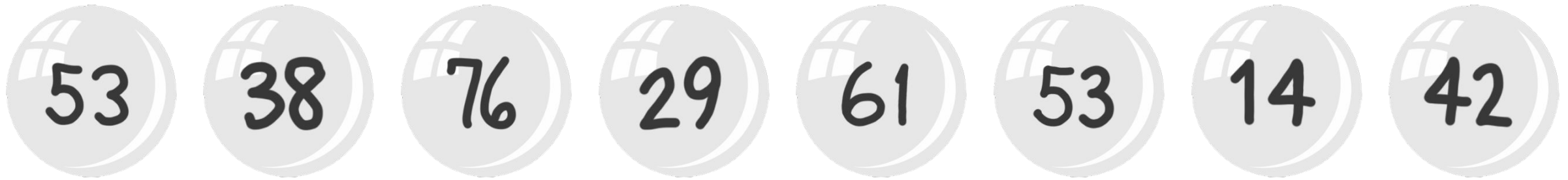
See full lab 5 slides for the complete animation

Selection Sort

Selection Sort

- Characteristics of selection sort:

Best Case	Average Case	Worst Case	Memory	Stable?
$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	No*

- Good when auxiliary memory is limited.
- Algorithm: divide input list into sub-list of sorted items and sub-list of unsorted items. Add the smallest non-selected item to the back of the sorted portion (usually by swapping by whatever was there). This increases the sorted portion's size. Repeat until the whole array is sorted.
  - in other words, find the smallest element in the list and swap with the first position, find the second smallest element and swap with second position...
- Additional memory can make the sort stable (break ties using original index)

Selection Sort

- Selection sort in action:

See full lab 5 slides for the complete animation

Insertion Sort

Insertion Sort

- Characteristics of insertion sort:

Best Case	Average Case	Worst Case	Memory	Stable?
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Yes

- Very efficient on "almost" sorted list!
- One of the fastest sorting algorithms on small input sizes.
- Algorithm: take each item and "insert" it into the correct location when only considering all elements before it (i.e. swap left until it is where it belongs).

Insertion Sort

- Insertion sort in action:
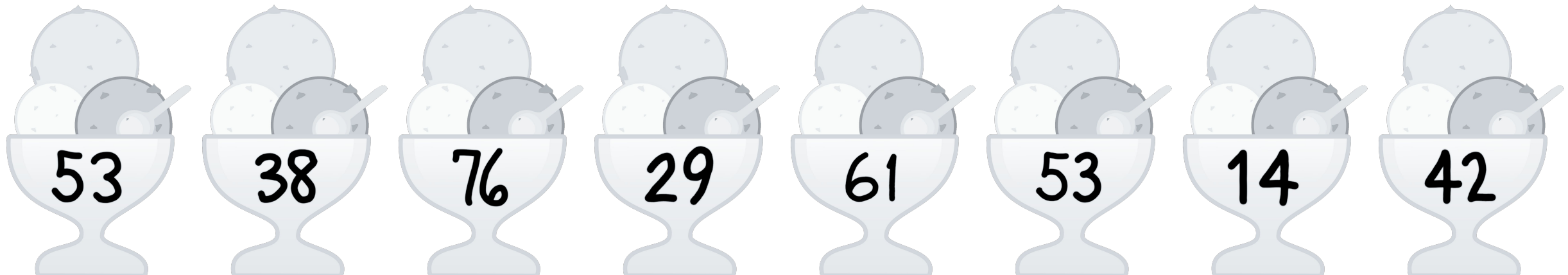
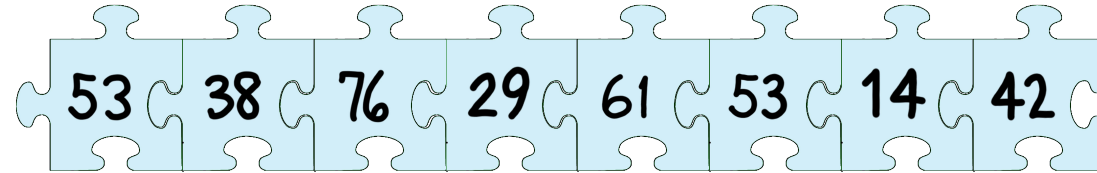**See full lab 5 slides for the complete animation**

Merge Sort (Top Down)

- Take a list, split it in half
  - Recursively split both halves
- Combine halves with merge function
- Base case occurs when halves have length 1

Merge Sort (Top Down)

- Merge sort in action:

See full lab 5 slides for the complete animation

Merge Sort Complexity Analysis

```
mergesort(int [] a, int left, int right) {
   if (right > left) {
      middle = left + (right - left)/2;
      mergesort(a, left, middle);
      mergesort(a, middle+1, right);
      merge(a, left, middle, right);
   }
   return;
}
```

$$T(n) \in \begin{cases} \Theta(n^{log_b a}), & \text{if } a > b^c \\ \Theta(n^c log_2 n), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

T(n) = time to sort left half + time to sort right half + time to merge halves

=   T(n/2)   +   T(n/2)   +   n

Therefore, we have: T(n) = 2 * T(n/2) + n, T(1) = 1

Merge Sort Complexity Analysis

```
mergesort(int [] a, int left, int right) {
    if (right > left) {
        middle = left + (right - left)/2;
        mergesort(a, left, middle);
        mergesort(a, middle+1, right);
        merge(a, left, middle, right);
    }
    return;
}
```

By the Master Theorem, the complexity is Θ(*n* log *n*).

$$T(n) \in \begin{cases} \Theta(n^{log_b a}), & \text{if } a > b^c \\ \Theta(n^c log_2 n), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

T(n) = time to sort left half + time to sort right half + time to merge halves

    =   T(n/2)   +   T(n/2)   +   n

Therefore, we have: T(n) = 2 * T(n/2) + n, T(1) = 1

Merge Sort

- Characteristics of merge sort:

Best Case	Average Case	Worst Case	Memory	Stable?
Ω(n log n)	Θ(n log n)	O(n log n)	O(n)	Yes

- Comparison-based, divide-and-conquer algorithm.
- Bottom-up and top-down algorithms available.
- Algorithm: divide unsorted list into *n* sublists, each containing 1 element. Repeatedly merge sublists to produce new sublists until there is only one sublist (the final sorted list) left.
- Stable as long as merge is stable. Not adaptive.
- Memory is O(n) worst-case because most implementations merge into a separate container and then copy over to the original container.
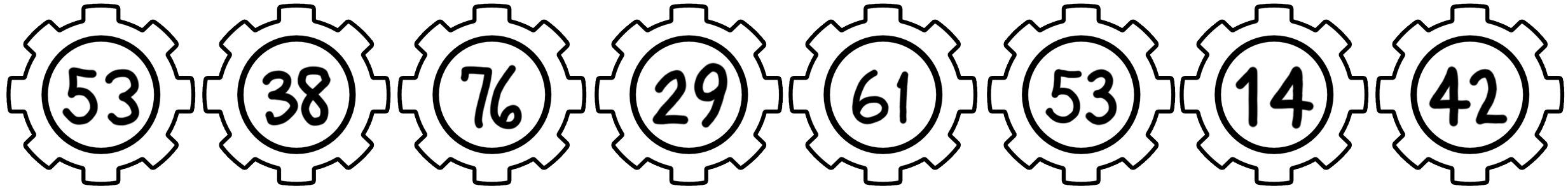
Quicksort

Quicksort

- Characteristics of quicksort:

Best Case	Average Case	Worst Case	Memory	Stable?
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$	No*

- Comparison-based sorting algorithm.
- Most implementations are unstable, except for complex in-place partitioning.
- Often faster in practice than other $n \log n$ sorting algorithms.
- Algorithm: pick a pivot from the list. Reorder list so that elements with values less than the pivot come before the pivot, while elements with values greater than or equal to the pivot come after it. After partitioning, the pivot is in the final position. Recursively sort the sub-list of elements with smaller values and the sub-list of elements with greater values.

Quicksort

- Quicksort in action (with last element as pivot):

See full lab 5 slides for the complete animation

Quicksort

- What causes the worst-case O(n^2) behavior?

- What is the ideal pivot choice?

- Why can't we do this? What can we do instead?

Quicksort

- What causes the worst-case O(n^2) behavior?

  When the smallest or largest element is chosen as the pivot every time (the more uneven the partition, the worse the performance).

- What is the ideal pivot choice?

- Why can't we do this? What can we do instead?

Quicksort

- What causes the worst-case O(n^2) behavior?

  When the smallest or largest element is chosen as the pivot every time (the more uneven the partition, the worse the performance).

- What is the ideal pivot choice?

  The median element (most evenly partitions the array - roughly equal number of elements to both left and right of pivot.

- Why can't we do this? What can we do instead?

Quicksort

- What causes the worst-case O(n^2) behavior?

  When the smallest or largest element is chosen as the pivot every time (the more uneven the partition, the worse the performance).

- What is the ideal pivot choice?

  The median element (most evenly partitions the array - roughly equal number of elements to both left and right of pivot.

- Why can't we do this? What can we do instead?

  The median is often hard to find. To solve this, we can use median sampling instead (e.g. take the median of a sample).

Summary of Sorts

Summary

Sort	Best	Average	Worst	Memory	Stable?	Adaptive?
Bubble	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	No	No
Insertion	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Heap	$\Omega(n \log n)$ (distinct keys)	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$	No	No
Merge	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$	Yes (if merge is stable)	No
Quick	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$	No	No

Binary Search

Binary Search Interview Problems

- Binary search is a classic topic for interview problems.
- Let's look at a few problems that can be solved using binary search!
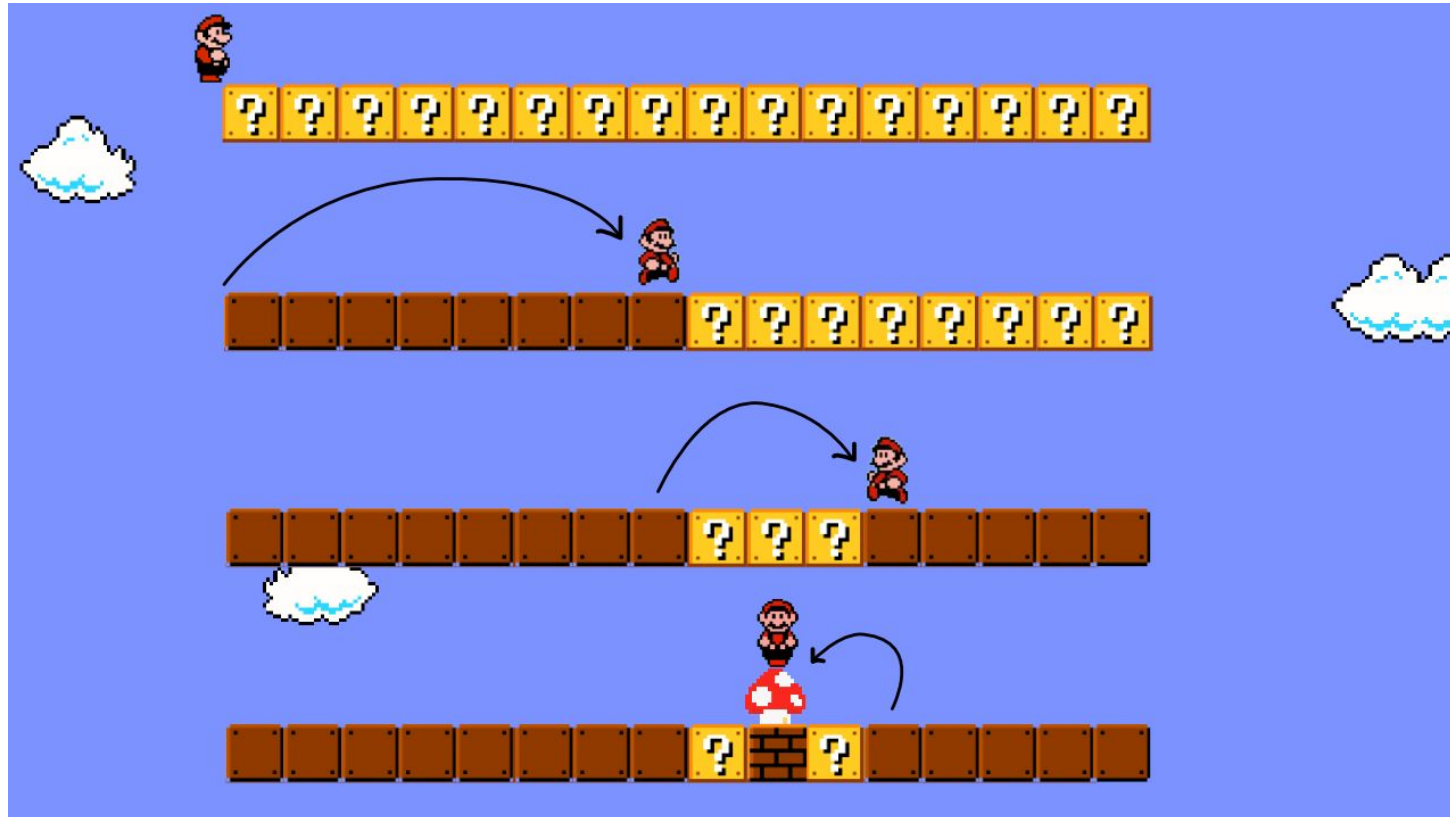
Binary Search Question #1

- You are given an array that is first increasing and then decreasing. Find the maximum value of this array.

```
INPUT: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}
OUTPUT: 500

INPUT: arr[] = {1, 3, 50, 10, 9, 7, 6}
OUTPUT: 50

INPUT: arr[] = {10, 20, 30, 40, 50}
OUTPUT: 50

INPUT: arr[] = {120, 100, 80, 20, 0}
OUTPUT: 120
```

Binary Search Question #1

- You are given an array that is first increasing and then decreasing. Find the maximum value of this array.

- Solution:
  - perform a binary search
  - if middle element is greater than both of its adjacent elements, it is the maximum
  - if middle is greater than element to its right and smaller than element to its left, then max element lies to the left of mid (we are on decreasing side of the array)
  - if middle element is smaller than element to its right and greater than element to its left, then max element lies to the right of mid (we are on increasing side)

Binary Search Question #2

- You are given a sorted array that is rotated at a certain position. Return the minimum element.

```
INPUT: arr[] = {5, 6, 1, 2, 3, 4}
OUTPUT: 1

INPUT: arr[] = {1, 2, 3, 4}
OUTPUT: 1

INPUT: arr[] = {2, 1}
OUTPUT: 1

INPUT: arr[] = {2, 3, 4, 5, 6, 1}
OUTPUT: 1
```

Binary Search Question #2

- You are given a sorted array that is rotated at a certain position. Return the minimum element.
  - Solution:
    - perform a binary search
    - the minimum element is the only element whose previous element is greater than it
    - check the middle element - if the element to its left is greater than it, mid is the min
    - if the element to its left is less than or equal, it is not the minimum; the minimum lies either in its left or right half
    - if the middle element is smaller than the last element, the minimum lies in the left half; else, it lies in the right half
    - When you have narrowed the minimum down to two elements, you can compare their values to find the min

Example:

{5, 6, 1, 2, 3, 4}

Mid element is 2. The element to the left is not greater than it. Hence, it is not the minimum.

Since 2 is smaller than 4 (the last element), it means the rotation point lies before mid. Therefore, we check the left half.

Binary Search Question #3

- You are given a sorted array where every number occurs twice except for one number. Find this number that only appears once.

```
INPUT: arr[] = {1, 1, 6, 6, 9, 9, 22, 22, 26, 30, 30, 40, 40}
OUTPUT: 26

INPUT: arr[] = {1, 2, 2, 3, 3, 4, 4}
OUTPUT: 1

INPUT: arr[] = {1, 1, 2, 2, 3, 3, 4}
OUTPUT: 4

INPUT: arr[] = {281}
OUTPUT: 281
```

Binary Search Question #3

- You are given a sorted array where every number occurs twice except for one number. Find this number that only appears once.
- Key observation: if there were no odd element out, the first occurence of each number would be found at an even index, and the second occurrence would be found at an odd index.
  - This is true until we reach the odd one out! Once we reach this point, the opposite will be true (the first occurrence would be found at an odd index, the second at an even index). We will do a binary search to find the index of the odd one out.
  - If mid is even, check if a[mid] == a[mid + 1]
  - if mid is odd, check if a[mid] == a[mid - 1]
    - if these conditions hold, the odd one out must be found to the right, else it must be found to the left!

Bonus Interview Brain Teaser

Brain Teaser (Exercise for Home)

- There are 2810 light bulbs lined up in a row in a long room. Each bulb has its own switch, and all switches are switched OFF at the beginning. The bulbs are numbered consecutively from 1 to 2810.

  There are 2810 people lined up outside the entry door. Person 1 enters the room, flips the switch on every bulb, and exits. After person 1 leaves the room, person 2 enters and flips the switch on every second bulb (turning off bulbs 2, 4, 6...). After person 2 leaves, person 3 enters and flips the switch of every third bulb (changing the state on bulbs 3, 6, 9...). This continues until all 2810 people have passed through the room, where the 2810th person only flips the switch of the 2810th bulb.

  After all 2810 people have exited the room, how many light bulbs are turned ON?

  Suppose you have the ability to remove bulbs from the room in reverse order, first removing bulb 2810, then 2809, etc. What is the minimum number of bulbs you would have to remove for the number of bulbs that are turned ON to change?

Handwritten Problem

Handwritten Problem

- Given a vector with *n* elements with values of either 0, 1, or 2, devise an O(n) algorithm to sort this vector. You must do this in a **single pass** of the vector. You may **NOT** copy items, create arrays or strings, or do any other memory allocation. Make sure your algorithm works for all cases.
- You may use `std::swap` to swap two items in the vector.

```
BEFORE: arr[] = {2, 1, 0, 0, 2, 1, 2, 2, 0, 1, 1, 1, 0}
AFTER: arr[] = {0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2}

// sort a vector of 0s, 1s, and 2s in linear time
void sort012(vector<int>& nums);
```

Good Luck on the Exam!