

TUTORIAL 38

Ian Yong

CS2040S – Data Structures and Algorithms

28 February 2022 (Week 7)

FRUIT JUMBLE

(CS2040S 2022 MIDTERM)

Fruit Jumble

The first column in the table below contains an unsorted list of words. The last column contains a sorted list of words. Each intermediate column contains a partially sorted list.

Each intermediate column was constructed by beginning with the unsorted list at the left and running one of the sorting algorithms that we learned about in class, stopping at some point before it finishes. Each algorithm is executed exactly as described in the lecture notes. One column has been sorted using a fake sorting algorithm. **(Recursive algorithms recurse on the left half of the array before the right half. QuickSort uses the first element as the pivot and uses in-place 2-way partitioning.)**

Identify which column was (partially) sorted with which sorting algorithm.

Unsorted	A	B	C	D	E	F	Sorted
Elderberry	Apple	Banana	Apple	Banana	Apple	Banana	Apple
Orange	Banana	Elderberry	Cherry	Elderberry	Banana	Elderberry	Banana
Guava	Cherry	Guava	Banana	Guava	Elderberry	Guava	Cherry
Banana	Durian	Honeydew	Durian	Honeydew	Guava	Orange	Durian
Honeydew	Honeydew	Orange	Elderberry	Imbe	Honeydew	Honeydew	Elderberry
Mango	Mango	Mango	Mango	Jackfruit	Mango	Imbe	Fig
Imbe	Imbe	Imbe	Imbe	Kiwi	Imbe	Mango	Guava
Papaya	Papaya	Papaya	Papaya	Fig	Jackfruit	Papaya	Honeydew
Jackfruit	Jackfruit	Jackfruit	Jackfruit	Lychee	Kiwi	Jackfruit	Imbe
Kiwi	Kiwi	Kiwi	Kiwi	Durian	Fig	Kiwi	Jackfruit
Fig	Fig	Fig	Fig	Cherry	Lychee	Fig	Kiwi
Lychee	Lychee	Lychee	Lychee	Mango	Durian	Lychee	Lychee
Durian	Orange	Durian	Honeydew	Apple	Cherry	Durian	Mango
Cherry	Guava	Cherry	Guava	Nectarine	Nectarine	Cherry	Nectarine
Nectarine	Nectarine	Nectarine	Nectarine	Orange	Orange	Nectarine	Orange
Apple	Elderberry	Apple	Orange	Papaya	Papaya	Apple	Papaya
Unsorted	A	B	C	D	E	F	Sorted

Fruit Jumble

- Do not just execute each sorting algorithm, step-by-step, until it matches one of the columns
 - You will run out of time if you do that
 - Only do so if you've already narrowed down to a few possible search algorithms and you cannot make use of any other invariants
- Think in terms of invariants that are true at every step of the algorithm!

Sorting Algorithm	Description	Invariant	Is stable?
Bubble Sort	“Bubble” the largest element to the end of the array through repeated swapping of out-of-order adjacent pairs (inversions)	Largest k elements are at the end of the array (after k iterations)	Yes
Selection Sort	Select the minimum element and add it to the sorted region of the array by swapping. Repeat until all elements have been selected.	Smallest k elements are at the start of the array (after k iterations)	No
Insertion Sort	Select the first element in the unsorted region of the array and find where to place it in the sorted region. Repeat until all elements have been selected.	First k elements are sorted (after k iterations), last $n - k$ elements are untouched	Yes
Merge Sort	Halve the array, recursively sort, then merge	Each subarray is already sorted when merging	Yes
Quick Sort (with first element pivot)	Partition around the first element, then repeat on subarrays	All elements to the left/right of the pivot are smaller/larger	No

Unsorted	A	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Cherry	Cherry
Banana	Durian	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Orange	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Elderberry	Papaya
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Cherry	Cherry
Banana	Durian	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Orange	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Elderberry	Papaya
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Cherry	Cherry
Banana	Durian	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Orange	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Elderberry	Papaya
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array A is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Papaya) is not at the end of the array. Thus, sorting algorithm A is not Bubble Sort.

Unsorted	A	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Cherry	Cherry
Banana	Durian	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Orange	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Elderberry	Papaya
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Cherry	Cherry
Banana	Durian	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Orange	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Elderberry	Papaya
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

The smallest k elements are at the start of array A. Thus, sorting algorithm A is Selection Sort!

Unsorted	B	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Orange	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Durian	Mango
Cherry	Cherry	Nectarine
Nectarine	Nectarine	Orange
Apple	Apple	Papaya
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Orange	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Durian	Mango
Cherry	Cherry	Nectarine
Nectarine	Nectarine	Orange
Apple	Apple	Papaya
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Orange	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Durian	Mango
Cherry	Cherry	Nectarine
Nectarine	Nectarine	Orange
Apple	Apple	Papaya
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array B is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Papaya) is not at the end of the array. Thus, sorting algorithm B is not Bubble Sort.

Unsorted	B	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Orange	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Durian	Mango
Cherry	Cherry	Nectarine
Nectarine	Nectarine	Orange
Apple	Apple	Papaya
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
Elderberry Orange Guava Banana Honeydew	Banana Elderberry Guava Honeydew Orange	Apple Banana Cherry Durian Elderberry Fig Guava Honeydew Imbe Jackfruit Kiwi Kiwi Lychee Lychee Durian Cherry Nectarine Orange Papaya
Mango Imbe Papaya Jackfruit Kiwi Fig Lychee Durian Cherry Nectarine Apple	Mango Imbe Papaya Jackfruit Kiwi Kiwi Lychee Durian Cherry Nectarine Apple	
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the first 5 elements are sorted and the last 11 elements are untouched, sorting algorithm B is Insertion Sort!

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array C is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Papaya) is not at the end of the array. Thus, sorting algorithm C is not Bubble Sort.

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array C is different from the unsorted array, we know that **at least one** iteration of the sort was run. In addition, we observe that the smallest element (Apple) has moved from the bottom to the top of the array. For that to occur in Merge Sort, the merge step must have occurred between the first half & the second half of the array. This also means that the array should already be sorted if it is Merge Sort. Thus, sorting algorithm C is not Merge Sort.

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Elderberry	Apple	Apple
Orange	Cherry	Banana
Guava	Banana	Cherry
Banana	Durian	Durian
Honeydew	Elderberry	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Honeydew	Mango
Cherry	Guava	Nectarine
Nectarine	Nectarine	Orange
Apple	Orange	Papaya
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the first element in the unsorted array (Elderberry) appears in its sorted location in array C and the elements to the left/right of Elderberry are smaller/larger, sorting algorithm C is Quick Sort!

Unsorted	D	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Imbe	Elderberry
Mango	Jackfruit	Fig
Imbe	Kiwi	Guava
Papaya	Fig	Honeydew
Jackfruit	Lychee	Imbe
Kiwi	Durian	Jackfruit
Fig	Cherry	Kiwi
Lychee	Mango	Lychee
Durian	Apple	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging

Unsorted	D	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Imbe	Elderberry
Mango	Jackfruit	Fig
Imbe	Kiwi	Guava
Papaya	Fig	Honeydew
Jackfruit	Lychee	Imbe
Kiwi	Durian	Jackfruit
Fig	Cherry	Kiwi
Lychee	Mango	Lychee
Durian	Apple	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging

Unsorted	D	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Honeydew	Durian
Honeydew	Imbe	Elderberry
Mango	Jackfruit	Fig
Imbe	Kiwi	Guava
Papaya	Fig	Honeydew
Jackfruit	Lychee	Imbe
Kiwi	Durian	Jackfruit
Fig	Cherry	Kiwi
Lychee	Mango	Lychee
Durian	Apple	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging

Since the last 3 elements are sorted, sorting algorithm D is potentially Bubble Sort. **However, we cannot say for sure as array E has the same properties!** We can either try to execute Bubble Sort step-by-step (not recommended), or try to figure out if array E's identity can be determined to find out the identity of array D through elimination.

Also, there is not much point checking if sorting algorithm D is Merge Sort so we skip over.

Unsorted	E	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Elderberry	Cherry
Banana	Guava	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Jackfruit	Honeydew
Jackfruit	Kiwi	Imbe
Kiwi	Fig	Jackfruit
Fig	Lychee	Kiwi
Lychee	Durian	Lychee
Durian	Cherry	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging

Unsorted	E	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Elderberry	Cherry
Banana	Guava	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Jackfruit	Honeydew
Jackfruit	Kiwi	Imbe
Kiwi	Fig	Jackfruit
Fig	Lychee	Kiwi
Lychee	Durian	Lychee
Durian	Cherry	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging

Unsorted	E	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Elderberry	Cherry
Banana	Guava	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Jackfruit	Honeydew
Jackfruit	Kiwi	Imbe
Kiwi	Fig	Jackfruit
Fig	Lychee	Kiwi
Lychee	Durian	Lychee
Durian	Cherry	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Merge Sort	Each subarray is already sorted when merging

Since the last 3 elements are sorted, sorting algorithm E is potentially Bubble Sort. Seems like we're stuck here. We have no choice but to execute a few iterations of Bubble Sort. :(

Unsorted	D	E	Sorted
Elderberry	Banana	Apple	Apple
Orange	Elderberry	Banana	Banana
Guava	Guava	Elderberry	Cherry
Banana	Honeydew	Guava	Durian
Honeydew	Imbe	Honeydew	Elderberry
Mango	Jackfruit	Mango	Fig
Imbe	Kiwi	Imbe	Guava
Papaya	Fig	Jackfruit	Honeydew
Jackfruit	Lychee	Kiwi	Imbe
Kiwi	Durian	Fig	Jackfruit
Fig	Cherry	Lychee	Kiwi
Lychee	Mango	Durian	Lychee
Durian	Apple	Cherry	Mango
Cherry	Nectarine	Nectarine	Nectarine
Nectarine	Orange	Orange	Orange
Apple	Papaya	Papaya	Papaya
Unsorted	D	E	Sorted

But wait! It takes $n - 1$ swaps for the smallest element (Apple) to be swapped to the front in Bubble Sort. This means that sorting algorithm D must be Bubble Sort!

What about array E? I suspect that it might be the fake sorting algorithm, so let's take a look at array F first to see if it's Merge Sort...

Unsorted	F	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Orange	Durian
Honeydew	Honeydew	Elderberry
Mango	Imbe	Fig
Imbe	Mango	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Durian	Mango
Cherry	Cherry	Nectarine
Nectarine	Nectarine	Orange
Apple	Apple	Papaya
Unsorted	F	Sorted

Sorting Algorithm	Invariant
Merge Sort	Each subarray is already sorted when merging

Unsorted	F	Sorted
Elderberry	Banana	Apple
Orange	Elderberry	Banana
Guava	Guava	Cherry
Banana	Orange	Durian
Honeydew	Honeydew	Elderberry
Mango	Imbe	Fig
Imbe	Mango	Guava
Papaya	Papaya	Honeydew
Jackfruit	Jackfruit	Imbe
Kiwi	Kiwi	Jackfruit
Fig	Fig	Kiwi
Lychee	Lychee	Lychee
Durian	Durian	Mango
Cherry	Cherry	Nectarine
Nectarine	Nectarine	Orange
Apple	Apple	Papaya
Unsorted	F	Sorted

Sorting Algorithm	Invariant
Merge Sort	Each subarray is already sorted when merging

Unsorted	F	Sorted
Elderberry Orange Guava Banana	Banana Elderberry Guava Orange	Apple Banana Cherry Durian Elderberry Fig Guava Honeydew Imbe Jackfruit Kiwi Lychee Mango Nectarine Orange Papaya
Honeydew Mango Imbe Papaya	Honeydew Imbe Mango Papaya	
Jackfruit Kiwi Fig Lychee	Jackfruit Kiwi Fig Lychee	
Durian Cherry Nectarine Apple	Durian Cherry Nectarine Apple	
Unsorted	F	Sorted

Sorting Algorithm	Invariant
Merge Sort	Each subarray is already sorted when merging

Since each subarray (when split by powers of 2) is locally sorted, sorting algorithm F is Merge Sort!

Unsorted	E	Sorted
Elderberry	Apple	Apple
Orange	Banana	Banana
Guava	Elderberry	Cherry
Banana	Guava	Durian
Honeydew	Honeydew	Elderberry
Mango	Mango	Fig
Imbe	Imbe	Guava
Papaya	Jackfruit	Honeydew
Jackfruit	Kiwi	Imbe
Kiwi	Fig	Jackfruit
Fig	Lychee	Kiwi
Lychee	Durian	Lychee
Durian	Cherry	Mango
Cherry	Nectarine	Nectarine
Nectarine	Orange	Orange
Apple	Papaya	Papaya
Unsorted	E	Sorted

Sorting Algorithm

Invariant

By the process of elimination, sorting algorithm E is the fake sorting algorithm!

**ASYMPTOTICALLY
APPROACHING ANSWERS
(CS2040S 2022 MIDTERM)**

Asymptotically Approaching Answers

Choose the tightest possible bound for the following function:

$$T(n) = 24n^2 \log^2 n + 1.7 \log^3 n$$

- 1. $O(n)$
- 2. $O(n \log n)$
- 3. $O(n \log^2 n)$

- 4. $O(n^2)$
- 5. $O(n^3)$
- 6. $O(2^n)$

Asymptotically Approaching Answers

Choose the tightest possible bound for the following function:

$$T(n) = 24n^2 \log^2 n + 1.7 \log^3 n$$

- 1. $O(n)$
- 2. $O(n \log n)$
- 3. $O(n \log^2 n)$

- 4. $O(n^2)$
- 5. **$O(n^3)$**
- 6. $O(2^n)$

Asymptotically Approaching Answers

Choose the tightest possible bound for the following function:

$$\begin{aligned}T(n) &= 24n^2 \log^2 n + 1.7 \log^3 n \\&= O(n^2 \log^2 n) + O(\log^3 n) \\&= O(n^2 \log^2 n)\end{aligned}$$

The tightest possible bound out of the options given is $O(n^3)$.

- 1. $O(n)$
- 2. $O(n \log n)$
- 3. $O(n \log^2 n)$

- 4. $O(n^2)$
- 5. **$O(n^3)$**
- 6. $O(2^n)$

Asymptotically Approaching Answers

$\log_2 n = O(\log_3 n)$: True or False?

Asymptotically Approaching Answers

$\log_2 n = O(\log_3 n)$: **True** or False?

Asymptotically Approaching Answers

$\log_2 n = O(\log_3 n)$: **True** or False?

$$\log_2 n = \frac{\log_3 n}{\log_3 2} = O(\log_3 n)$$

$\frac{1}{\log_3 2}$ is a constant!

Asymptotically Approaching Answers

Suppose you have developed a new algorithm *NeuralDeepCatFinder*, and you have run some experiments to see how long it takes on different sized inputs. You observe the following, where N is the input size and time is measured in seconds:

N	time
10,000	0.2 seconds
20,000	1.2 seconds
40,000	3.9 seconds
80,000	16.1 seconds
160,000	63.8 seconds

Asymptotically Approaching Answers

Based on this experimental data, you conclude that your algorithm has what asymptotic complexity? (Choose the best answer. You may assume that you were testing the algorithm on the worst case inputs.)

1. $O(\sqrt{n})$

2. $O(n)$

3. $O(n \log n)$

4. $O(n^2)$

5. $O(n^3)$

6. $O(2^n)$

Asymptotically Approaching Answers

Based on this experimental data, you conclude that your algorithm has what asymptotic complexity? (Choose the best answer. You may assume that you were testing the algorithm on the worst case inputs.)

1. $O(\sqrt{n})$

2. $O(n)$

3. $O(n \log n)$

4. $O(n^2)$

5. $O(n^3)$

6. $O(2^n)$

Asymptotically Approaching Answers

Based on this experimental data, you conclude that your algorithm has what asymptotic complexity? (Choose the best answer. You may assume that you were testing the algorithm on the worst case inputs.)

When N doubles, the time roughly quadruples.

1. $O(\sqrt{n})$

2. $O(n)$

3. $O(n \log n)$

4. $O(n^2)$

5. $O(n^3)$

6. $O(2^n)$

MIDTERM SORTING

(CS2040S 2021 MIDTERM)

Midterm Sorting

Consider the following sorting routines. These sorting algorithms use standard $O(n \log n)$ MergeSort, as defined in class, as a blackbox to sort segments of the array: `MergeSort(int[] A, int low, int high)`. It sorts the array segment `A[low..high]` (inclusive of the endpoints) using MergeSort. It also uses a function `isSorted(int[] A)` which returns true when the array is sorted (and false otherwise). The `isSorted` routine takes $\Theta(n)$ time if array A is of size n .

Midterm Sorting

Assume we are using `midtermSort(A, k)` to sort an array A of unique elements where each item is in an array position at a distance $\leq k$ from its array position in the sorted array. For example, in the following array:

4	5	6	1	2	3	10	11	12	7	8	9
---	---	---	---	---	---	----	----	----	---	---	---

Notice that each element is within distance $k = 3$ of its final position. The value in $A[4] = 2$ belongs in position $A[1]$, and $4 - 1 \leq 3$. Assume $k \geq 2$.

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, jk, (j+2)k-1)
```

```
superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n-1)
```


Midterm Sorting

As a matter of notation, when we talk about the prefix of the array $A[0, x]$, we mean the set of array slots $A[0], A[1], \dots, A[x]$. For example, the prefix of the array $A[0, 3]$ in the example above contains $[4, 5, 6, 1]$. More generally, the segment of the array $A[x, y]$ includes all the array elements from $A[x]$ to $A[y]$, inclusive of endpoints. You may assume that MergeSort will work correctly on the elements within the specified range if the range specified is larger than the valid range for A .

Midterm Sorting

A. Which of the following is always true:

- I. After each iteration of the loop, the array prefix $A[0, (j+1)k-1]$ is sorted.
 - II. After each iteration of the loop, the array prefix $A[0, (j+2)k-1]$ is sorted.
-
- 1. Statement I.
 - 2. Statement II.
 - 3. Both Statements I and II.
 - 4. Neither statement is true.

[2 marks]

Midterm Sorting

A. Which of the following is always true:

- I. After each iteration of the loop, the array prefix $A[0, (j+1)k-1]$ is sorted.
 - II. After each iteration of the loop, the array prefix $A[0, (j+2)k-1]$ is sorted.
- 1. Statement I.
 - 2. Statement II.
 - 3. Both Statements I and II.
 - 4. Neither statement is true.

[2 marks]

Midterm Sorting

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)
```

A. Which of the following is always true:

- I. After each iteration of the loop, the array prefix $A[0, (j+1)k - 1]$ is sorted.
 - II. After each iteration of the loop, the array prefix $A[0, (j+2)k - 1]$ is sorted.
- 1. Statement I.
 - 2. Statement II.
 - 3. Both Statements I and II.
 - 4. Neither statement is true.

After the j -th iteration of the loop, we would have called MergeSort on $[0, (j+2)k - 1]$. Thus, the range must have been sorted since each element is within k distance of their sorted position.

[2 marks]

Midterm Sorting

B. Which of the following is always true:

- I. After each iteration of the loop, the array prefix $A[0, (j+1)k-1]$ contains the $(j+1)k-1$ smallest elements in the array.
- II. After each iteration of the loop, the array prefix $A[0, (j+2)k-1]$ contains the $(j+2)k-1$ smallest elements in the array.

- 1. Statement I.
- 2. Statement II.
- 3. Both Statements I and II.
- 4. Neither statement is true.

[2 marks]

Midterm Sorting

B. Which of the following is always true:

- I. After each iteration of the loop, the array prefix $A[0, (j+1)k-1]$ contains the $(j+1)k-1$ smallest elements in the array.
- II. After each iteration of the loop, the array prefix $A[0, (j+2)k-1]$ contains the $(j+2)k-1$ smallest elements in the array.

1. Statement I.

2. Statement II.

3. Both Statements I and II.

4. Neither statement is true.

[2 marks]

Midterm Sorting

B. Which of the following is always true:

- I. After each iteration of the loop, the array prefix $A[0, (j+1)k-1]$ contains the $(j+1)k-1$ smallest elements in the array.
- II. After each iteration of the loop, the array prefix $A[0, (j+2)k-1]$ contains the $(j+2)k-1$ smallest elements in the array.

1. Statement I.

3. Both Statements I and II.

2. Statement II.

4. Neither statement is true.

Because each item in the array is at a position $\leq k$ from its position in the sorted array, after each iteration of the loop, the elements in $A[0, (j+1)k-1]$ are the same as the sorted array. [2 marks]

Midterm Sorting

C. If an element $A[i]$ is initially in position i and ends in position $i_\ell \leq i$ (when the sorting algorithm completes), then at the end of every iteration of the loop it is never moved to a position $i_h > i$: True or False? [2 marks]

Midterm Sorting

C. If an element $A[i]$ is initially in position i and ends in position $i_\ell \leq i$ (when the sorting algorithm completes), then at the end of every iteration of the loop it is never moved to a position $i_h > i$: True or False? [2 marks]

Midterm Sorting

C. If an element $A[i]$ is initially in position i and ends in position $i_\ell \leq i$ (when the sorting algorithm completes), then at the end of every iteration of the loop it is never moved to a position $i_h > i$: **True** or False? [2 marks]

- Let us look at the first iteration that overlaps $A[i]$
 - $A[i]$ must be in the second half of the subset of the array.
 - Case 1: $A[i]$ ends up in the first half of the subset of the array that is being sorted
 - Then, its position is fixed after just 1 iteration and the statement is true
 - Case 2: $A[i]$ ends up in the second half of the subset of the array that is being sorted
 - Then, in order for $i_\ell \leq i$ and $i_h > i$, the number of elements smaller than $A[i]$ must be smaller in the second iteration than in the first iteration
 - This is impossible as the number of elements smaller than $A[i]$ can only increase in the second iteration
 - Thus, the statement is true

Midterm Sorting

If $A[i]$ is in the very first or very last subset, then this statement is trivially true because there is no i_h ($A[i]$ will only be moved once, not twice!)

C. If an element $A[i]$ is initially in position i and ends in position $i_\ell \leq i$ (when the sorting algorithm completes), then at the end of every iteration of the loop it is never moved to a position $i_h > i$: **True** or False? [2 marks]

- Let us look at the first iteration that overlaps $A[i]$
 - $A[i]$ must be in the second half of the subset of the array.
 - Case 1: $A[i]$ ends up in the first half of the subset of the array that is being sorted
 - Then, its position is fixed after just 1 iteration and the statement is true
 - Case 2: $A[i]$ ends up in the second half of the subset of the array that is being sorted
 - Then, in order for $i_\ell \leq i$ and $i_h > i$, the number of elements smaller than $A[i]$ must be smaller in the second iteration than in the first iteration
 - This is impossible as the number of elements smaller than $A[i]$ can only increase in the second iteration
 - Thus, the statement is true

Midterm Sorting

D. Assume we run `superMidtermSort(A)` to sort an array A of unique elements where each item is in an array position at at distance $\leq k$ from its array position in the sorted array. (In this case, note that k is not given to the algorithm.) What is the running time of the algorithm as a function of n and k ? Give the tightest bound possible. [3 marks]

- | | | |
|-----------------------|-------------------------|-----------------------|
| 1. $\Theta(k)$ | 4. $\Theta(k \log n)$ | 7. $\Theta(nk)$. |
| 2. $\Theta(k \log k)$ | 5. $\Theta(n \log^2 k)$ | 8. None of the above. |
| 3. $\Theta(n \log k)$ | 6. $\Theta(n^2)$ | |

Midterm Sorting

D. Assume we run `superMidtermSort(A)` to sort an array A of unique elements where each item is in an array position at at distance $\leq k$ from its array position in the sorted array. (In this case, note that k is not given to the algorithm.) What is the running time of the algorithm as a function of n and k ? Give the tightest bound possible. [3 marks]

1. $\Theta(k)$

4. $\Theta(k \log n)$

7. $\Theta(nk)$.

2. $\Theta(k \log k)$

5. $\Theta(n \log^2 k)$

8. None of the above.

3. $\Theta(n \log k)$

6. $\Theta(n^2)$

Midterm

D. Assume we run superMidtermSort on an array of n elements where each element is in an array position i such that $i \bmod k = 0$. In this case, note that k is not a function of n and k ? Give the time complexity of the algorithm as a function of n and k . [3 marks]

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)
```

```
superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, 0, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n - 1)
```

elements where each element is in an array position i such that $i \bmod k = 0$. In this case, note that k is not a function of n and k ? Give the time complexity of the algorithm as a function of n and k . [3 marks]

(nk) .

one of the above.

Midterm $\Theta\left(\frac{n}{k}\right)$

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)
```

D. Assume we run `superMidtermSort` on an array of n elements where each element is in an array position i such that $i \bmod k = 0$. In this case, note that k is not a function of n and k ? Give the time complexity of the algorithm as a function of n and k . [3 marks]

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

```
superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, 0, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n-1)
```

elements where each element is in an array position i such that $i \bmod k = 0$. In this case, note that k is not a function of n and k ? Give the time complexity of the algorithm as a function of n and k . [3 marks]

(nk) .

one of the above.

Midterm $O\left(\frac{n}{k}\right)$

```

midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)

```

D. Assume we run superMidtermSort on an array of n elements where each item is in an array position i such that $i \bmod k = r$. In this case, note that k is not a function of n and k ? Give the time complexity of superMidtermSort in terms of n and k .

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

```

superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, 0, n - 1, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n - 1)

```

$O(2k \log 2k)$
 $= O(k \log 2k)$
 $= O(k(\log 2 + \log k))$
 $= O(k \log 2 + k \log k)$
 $= O(k + k \log k)$
 $= O(k \log k)$
 ONE OF THE ABOVE.

Midterm $O\left(\frac{n}{k}\right)$

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run superMidtermSort on an array of size n where each item is in an array position i such that $i \bmod k = r$. In this case, note that k is not a function of n and k ? Give the complexity of superMidtermSort in terms of n and k .

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = 2^{2^j}

midtermSort(A, n, d)

if isSorted(A) return

j = j + 1

until d > n

MergeSort(A, 0, n-1)

$O(2k \log 2k)$

= $O(k \log 2k)$

= $O(k(\log 2 + \log k))$

= $O(k \log 2 + k \log k)$

= $O(k + k \log k)$

= $O(k \log k)$

ONE OF THE ABOVE.

elements where each
this
as a
rks]

Midterm $O(\frac{n}{k})$

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run superMidtermSort on an array of size n where each item is in an array position i such that i % k = r for some r. In this case, note that k is not a function of n and k? Give the complexity in terms of n and k.

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = $2^{\{2^j\}}$

$O(n \log d)$ midtermSort(A, n, d)

if isSorted(A) return

j = j + 1

until d > n

MergeSort(A, 0, n-1)

$O(2k \log 2k)$

$= O(k \log 2k)$

$= O(k(\log 2 + \log k))$

$= O(k \log 2 + k \log k)$

$= O(k + k \log k)$

$= O(k \log k)$

one of the above.

elements where each
this
as a
rks]

Midterm $O(\frac{n}{k})$

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run superMidtermSort on an array of size n where each item is in an array position i such that i mod k = r for some r. In this case, note that k is not a function of n and k? Give the time complexity of superMidtermSort in terms of n and k.

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = $2^{\{2^j\}}$

$O(n \log d)$ midtermSort(A, n, d)

if isSorted(A) return

j = j + 1 $O(n)$

until d > n

MergeSort(A, 0, n-1)

$O(2k \log 2k)$

$= O(k \log 2k)$

$= O(k(\log 2 + \log k))$

$= O(k \log 2 + k \log k)$

$= O(k + k \log k)$

$= O(k \log k)$

ONE OF THE ABOVE.

Midterm $O(\frac{n}{k})$

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run superMidtermSort on an array of size n where each item is in an array position i such that i mod k = r for some r. In this case, note that k is not a function of n and k? Give the time complexity of superMidtermSort in terms of n and k.

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = $2^{\{2^j\}}$

$O(n \log d)$ midtermSort(A, n, d)

if isSorted(A) return

j = j + 1 $O(n)$

until d > n $O(\log \log n)$

MergeSort(A, 0, n-1)

$O(2k \log 2k)$

= $O(k \log 2k)$

= $O(k(\log 2 + \log k))$

= $O(k \log 2 + k \log k)$

= $O(k + k \log k)$

= $O(k \log k)$

one of the above.

Midterm $O(\frac{n}{k})$

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run superMidtermSort on an array of size n where each item is in an array position i such that i mod k = r for some r. In this case, note that k is not a function of n and k? Give the time complexity of superMidtermSort in terms of n and k.

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = $2^{\{2^j\}}$

$O(n \log d)$ midtermSort(A, n, d)

if isSorted(A) return

j = j + 1

until d > n $O(\log \log n)$

MergeSort(A, 0, n-1)

$O(2k \log 2k)$

= $O(k \log 2k)$

= $O(k(\log 2 + \log k))$

= $O(k \log 2 + k \log k)$

= $O(k + k \log k)$

= $O(k \log k)$

one of the above.

However, d is actually bounded by k as once $d > k$, midtermSort(A, d) sorts the array and isSorted(A) returns true

Midterm $O(\frac{n}{k})$

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run superMidtermSort on an array of size n where each item is in an array position i such that i mod k = r for some r. In this case, note that k is not a function of n and k? Give the time complexity of superMidtermSort in terms of n and k.

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = 2^{2^j}

$O(n \log d)$ midtermSort(A, n, d)

if isSorted(A) return

j = j + 1 $O(n)$

$O(\log \log k)$ until d > n $O(\log \log n)$

MergeSort(A, 0, n-1)

$O(2k \log 2k)$

= $O(k \log 2k)$

= $O(k(\log 2 + \log k))$

= $O(k \log 2 + k \log k)$

= $O(k + k \log k)$

= $O(k \log k)$

one of the above.

However, d is actually bounded by k as once $d > k$, midtermSort(A, d) sorts the array and isSorted(A) returns true

midtermSort(int[] A, int k) Overall $O(n \log k)$

int n = A.length;

Midterm $O\left(\frac{n}{k}\right)$

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

D. Assume we run s

$$O(n \log 2^{2^0}) + O(n \log 2^{2^1}) + O(n \log 2^{2^2}) + \dots$$

$$+ O(n \log 2^{2^{\log \log k}}) = O(n(2^0 + 2^1 + 2^2 + \dots + 2^{\log \log k}))$$

$$= O\left(n \sum_{i=0}^{\log \log k} 2^i\right) = O\left(n \left(\frac{2^{\log \log k + 1} - 1}{2 - 1}\right)\right)$$

$$= O(n(2^{\log \log k} - 1)) = O(n \log k)$$

$$O(2k \log 2k)$$

$$= O(k \log 2k)$$

$$= O(k(\log 2 + \log k))$$

$$= O(k \log 2 + k \log k)$$

$$= O(k + k \log k)$$

$$= O(k \log k)$$

one of the above.

3. $\Theta(n \log k)$

$O(n \log d)$

midtermSort(A, n, d)

if isSorted(A) return

j = j + 1

$O(n)$

$O(\log \log k)$ until d > n $O(\log \log n)$

MergeSort(A, 0, n-1)

However, d is actually bounded

by k as once $d > k$,

midtermSort(A, d) sorts the array

and isSorted(A) returns true

Midterm Sorting

E. Assume we run `superMidtermSort(A)` to sort an array `A` with the possibility of repeated elements. Is the resulting sorting algorithm stable? [2 marks]

Midterm Sorting

E. Assume we run `superMidtermSort(A)` to sort an array `A` with the possibility of repeated elements. Is the resulting sorting algorithm stable? [2 marks]

True, because merge sort is stable

Midterm Sorting

F. Is the `superMidtermSort(A)` an in-place sorting algorithm? [2 marks]

Midterm Sorting

F. Is the `superMidtermSort(A)` an in-place sorting algorithm?

[2 marks]

False, because merge sort (as defined in class) is not in-place

Midterm Sorting

G. Assume instead we run `InsertionSort(A)` to sort the array A of unique elements where each item is in an array position at at distance $\leq k$ from its array position in the sorted array. What is the running time of the algorithm as a function of n and k ? Give the tightest bound possible. [3 marks]

- | | | |
|-----------------------|-------------------------|-----------------------|
| 1. $\Theta(k)$ | 4. $\Theta(k \log n)$ | 7. $\Theta(nk)$. |
| 2. $\Theta(k \log k)$ | 5. $\Theta(n \log^2 k)$ | 8. None of the above. |
| 3. $\Theta(n \log k)$ | 6. $\Theta(n^2)$ | |

Midterm Sorting

G. Assume instead we run `InsertionSort(A)` to sort the array A of unique elements where each item is in an array position at at distance $\leq k$ from its array position in the sorted array. What is the running time of the algorithm as a function of n and k ? Give the tightest bound possible. [3 marks]

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

4. $\Theta(k \log n)$

5. $\Theta(n \log^2 k)$

6. $\Theta(n^2)$

7. $\Theta(nk)$.

8. None of the above.

Midterm

G. Assume instead v
each item is in an arra
What is the running ti
possible.

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)
```

```
superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, n, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n - 1)
```

unique elements where
on in the sorted array.
ive the tightest bound
[3 marks]

(nk) .

one of the above.

Midterm $\Theta\left(\frac{n}{k}\right)$

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)
```

G. Assume instead v
each item is in an arra
What is the running ti
possible.

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

```
superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, n, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n-1)
```

nique elements where
on in the sorted array.
ive the tightest bound
[3 marks]

$\Theta(nk)$.

one of the above.

Midterm $O\left(\frac{n}{k}\right)$

G. Assume instead v
each item is in an arra
What is the running ti
possible.

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

```
midtermSort(int[] A, int k)
    int n = A.length;
    for (int j = 0; j <= n/k - 1; j++)
        MergeSort(A, j*k, (j+2)*k - 1)
```

```
superMidtermSort(int[] A)
    int d = 2
    int j = 0
    int n = A.length
    repeat
        d = 2^{2^j}
        midtermSort(A, n, d)
        if isSorted(A) return
        j = j + 1
    until d > n
    MergeSort(A, 0, n-1)
```

$O((2k)^2)$
 $= O(4k^2)$ where
 $= O(k^2)$ array.
ive the tightest bound
[3 marks]

(nk) .

one of the above.

Midterm $O\left(\frac{n}{k}\right)$

```
midtermSort(int[] A, int k) Overall  $O(nk)$ 
```

```
    int n = A.length;
```

```
    for (int j = 0; j <= n/k - 1; j++)
```

```
        MergeSort(A, j*k, (j+2)*k-1)
```

$O((2k)^2)$

$= O(4k^2)$ where

$= O(k^2)$ array.

Give the tightest bound

[3 marks]

G. Assume instead v
each item is in an arra
What is the running ti
possible.

```
superMidtermSort(int[] A)
```

```
    int d = 2
```

```
    int j = 0
```

```
    int n = A.length
```

```
    repeat
```

```
        d = 2^{2^j}
```

```
        midtermSort(A, n, d)
```

```
        if isSorted(A) return
```

```
        j = j + 1
```

```
    until d > n
```

```
    MergeSort(A, 0, n-1)
```

(nk) .

one of the above.

1. $\Theta(k)$

2. $\Theta(k \log k)$

3. $\Theta(n \log k)$

Midterm $O(\frac{n}{k})$

midtermSort(int[] A, int k) Overall $O(nk)$

int n = A.length;

for (int j = 0; j <= n/k - 1; j++)

MergeSort(A, j*k, (j+2)*k-1)

$O((2k)^2)$

$= O(4k^2)$

$= O(k^2)$

where
array.
give the tightest bound

[3 marks]

G. Assume instead v
each item is in an arra
What is the running ti
possible.

1. $\Theta(k)$
2. $\Theta(k \log k)$
3. $\Theta(n \log k)$

superMidtermSort(int[] A)

int d = 2

int j = 0

int n = A.length

repeat

d = $2^{\{2^j\}}$

$O(nd)$ midtermSort(A, n, d)

if isSorted(A) return

j = j + 1

until d > n

MergeSort(A, 0, n-1)

(nk) .

one of the above.

```
midtermSort(int[] A, int k) Overall  $O(nk)$ 
```

```
    int n = A.length;
```

Midterm $O\left(\frac{n}{k}\right)$

```
    for (int j = 0; j <= n/k - 1; j++)
```

```
        MergeSort(A, j*k, (j+2)*k-1)
```

$O((2k)^2)$

$= O(4k^2)$ where

$= O(k^2)$ array.

Give the tightest bound

[3 marks]

G. Assume instead v

$$O(2^{2^0}n) + O(2^{2^1}n) + O(2^{2^2}n) + \dots + O(2^{2^{\log \log k}}n)$$

$$= O\left(n(2^{2^0} + 2^{2^1} + 2^{2^2} + \dots + 2^{2^{\log \log k}})\right)$$

$$\leq O\left(n(2^1 + 2^2 + 2^3 + \dots + 2^{\log k})\right) = O\left(n \sum_{i=0}^{\log k} 2^i - 1\right)$$

$$= O\left(n\left(\frac{2^{\log k + 1} - 1}{2 - 1}\right)\right) = O(n(2k - 1)) = O(nk)$$

(nk) .

one of the above.

$, d)$
return

```
        j = j + 1
```

```
    until d > n
```

```
    MergeSort(A, 0, n-1)
```

HOW DO THEY WORK?

(CS2040S 2021 MIDTERM)

How Do They Work?

Your job is to solve a load balancing problem: you have p servers and T tasks to solve. Your boss suggests using a hash function: for each task t , compute $h(t.name)$ (the hash of the name of the task) and assign the task to the server with that number. (Note there is no hash table here, we are just using the hash function to determine a server.) If you assign tasks in this manner, and if the hash function satisfies the simple uniform hashing assumption, what is the expected number of tasks on each server?

1. p/T
2. T/p
3. T
4. $(p/T) \log p$
5. $(T/p) \log p$
6. $T + p$
7. None of the above.

How Do They Work?

Your job is to solve a load balancing problem: you have p servers and T tasks to solve. Your boss suggests using a hash function: for each task t , compute $h(t.name)$ (the hash of the name of the task) and assign the task to the server with that number. (Note there is no hash table here, we are just using the hash function to determine a server.) If you assign tasks in this manner, and if the hash function satisfies the simple uniform hashing assumption, what is the expected number of tasks on each server?

1. p/T
2. T/p
3. T
4. $(p/T) \log p$
5. $(T/p) \log p$
6. $T + p$
7. None of the above.

How Do They Work?

Your job is to solve a load balancing problem: you have p servers and T tasks to solve. Your boss suggests using a hash function: for each task t , compute $h(t.name)$ (the hash of the name of the task) and assign the task to the server with that number. (Note there is no hash table here, we are just using the hash function to determine a server.) If you assign tasks in this manner, and if the hash function satisfies the **simple uniform hashing assumption**, what is the expected number of tasks on each server?

1. p/T

2. T/p

3. T

4. $(p/T) \log p$

5. $(T/p) \log p$

6. $T + p$

7. None of the above.

Every key is equally
likely to map to every
bucket

How Do They Work?

Which of the following is not true of a properly balanced AVL tree?

1. The height of a leaf is 0.
2. The height of a node is always one less than the height of its parent.
3. The height of a tree is always less than the number of nodes in the tree.
4. If u and v have the same parent, then $|height(u) - height(v)| \leq 1$.
5. If u and v have the same grandparent, then $|height(u) - height(v)| \leq 2$.
6. All of the above are true.

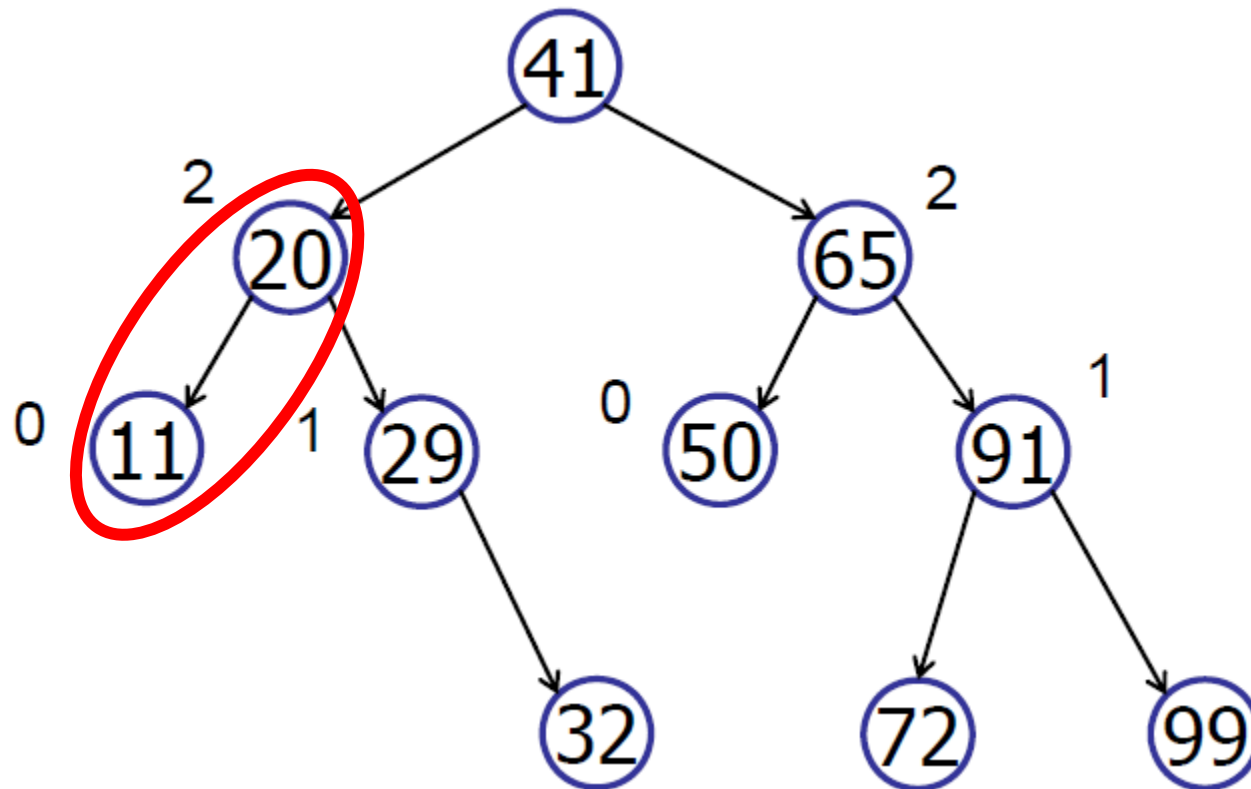
How Do They Work?

Which of the following is not true of a properly balanced AVL tree?

1. The height of a leaf is 0.
2. **The height of a node is always one less than the height of its parent.**
3. The height of a tree is always less than the number of nodes in the tree.
4. If u and v have the same parent, then $|height(u) - height(v)| \leq 1$.
5. If u and v have the same grandparent, then $|height(u) - height(v)| \leq 2$.
6. All of the above are true.

How Do They Work?

- Counter-example:



How Do They Work?

Assume that the following array has just been partitioned using the standard in-place (2-way) partitioning:

4	17	23	3	5	19	17	4	25	45	29	28
---	----	----	---	---	----	----	---	----	----	----	----

Which of the following values could have been the value of the pivot for this partition operation? (If a value appears more than once, it is a possible pivot if any of the indices containing that value have been chosen as the pivot.)

1. 5
2. 17
3. 23
4. 25
5. 29
6. None of the above.

How Do They Work?

Assume that the following array has just been partitioned using the standard in-place (2-way) partitioning:

4	17	23	3	5	19	17	4	25	45	29	28
---	----	----	---	---	----	----	---	----	----	----	----

Which of the following values could have been the value of the pivot for this partition operation? (If a value appears more than once, it is a possible pivot if any of the indices containing that value have been chosen as the pivot.)

1. 5
2. 17
3. 23
4. **25**
5. 29
6. None of the above.

How Do They Work?

Assume that the following array has just been partitioned using the standard in-place (2-way) partitioning:

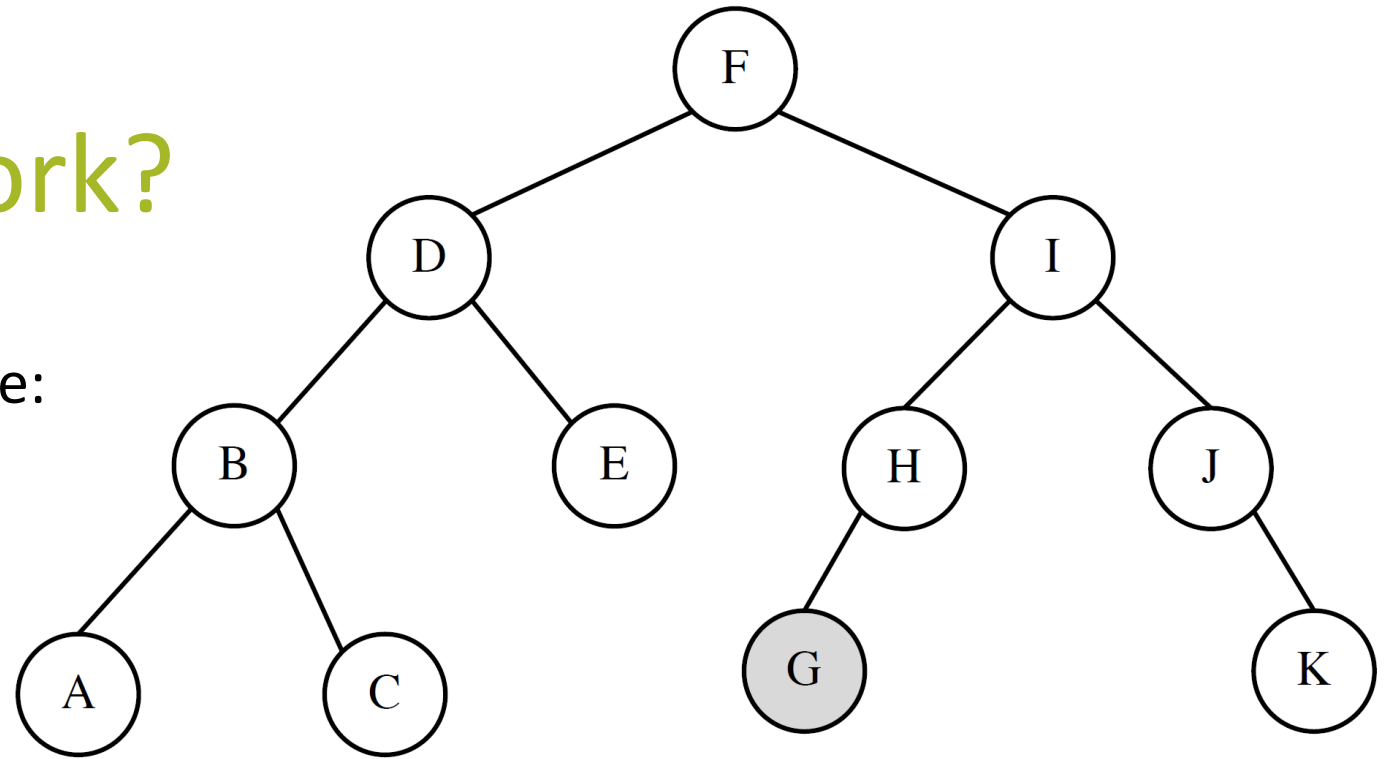
4	17	23	3	5	19	17	4	25	45	29	28
---	----	----	---	---	----	----	---	----	----	----	----

Which of the following values could have been the value of the pivot for this partition operation? (If a value appears more than once, it is a possible pivot if any of the indices containing that value have been chosen as the pivot.)

1. 5
2. 17
3. 23
4. **25**
5. 29
6. None of the above.
- Everything to the left of 25 is smaller than 25 while everything to the right is larger

How Do They Work?

Consider the following AVL tree:



The previous operation inserted the node G , which then triggered a double-rotation. Which node was the grandparent of G (i.e., the parent of the parent of G) immediately after G was inserted and before the rotations were performed?

1. D

3. F

5. I

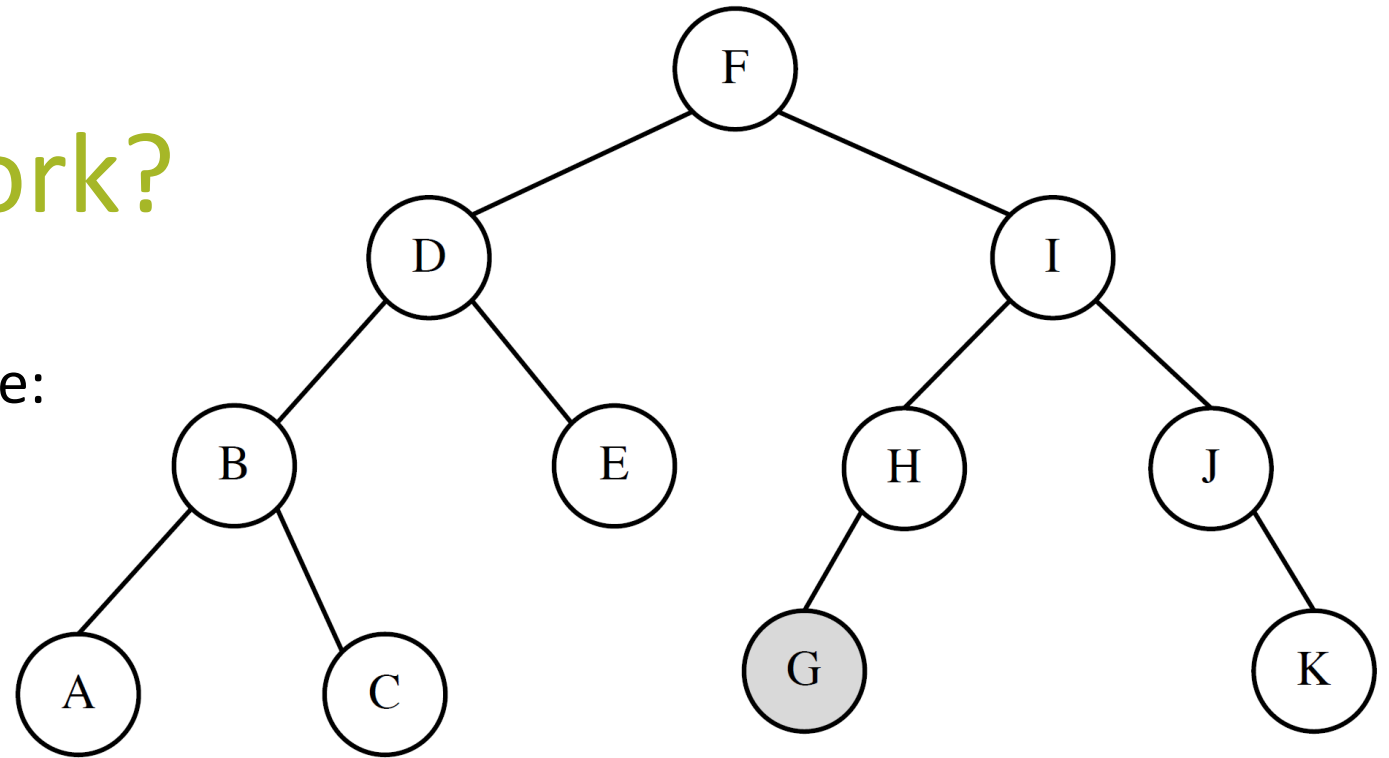
2. E

4. H

6. J

How Do They Work?

Consider the following AVL tree:



The previous operation inserted the node G , which then triggered a double-rotation. Which node was the grandparent of G (i.e., the parent of the parent of G) immediately after G was inserted and before the rotations were performed?

1. D

3. F

5. I

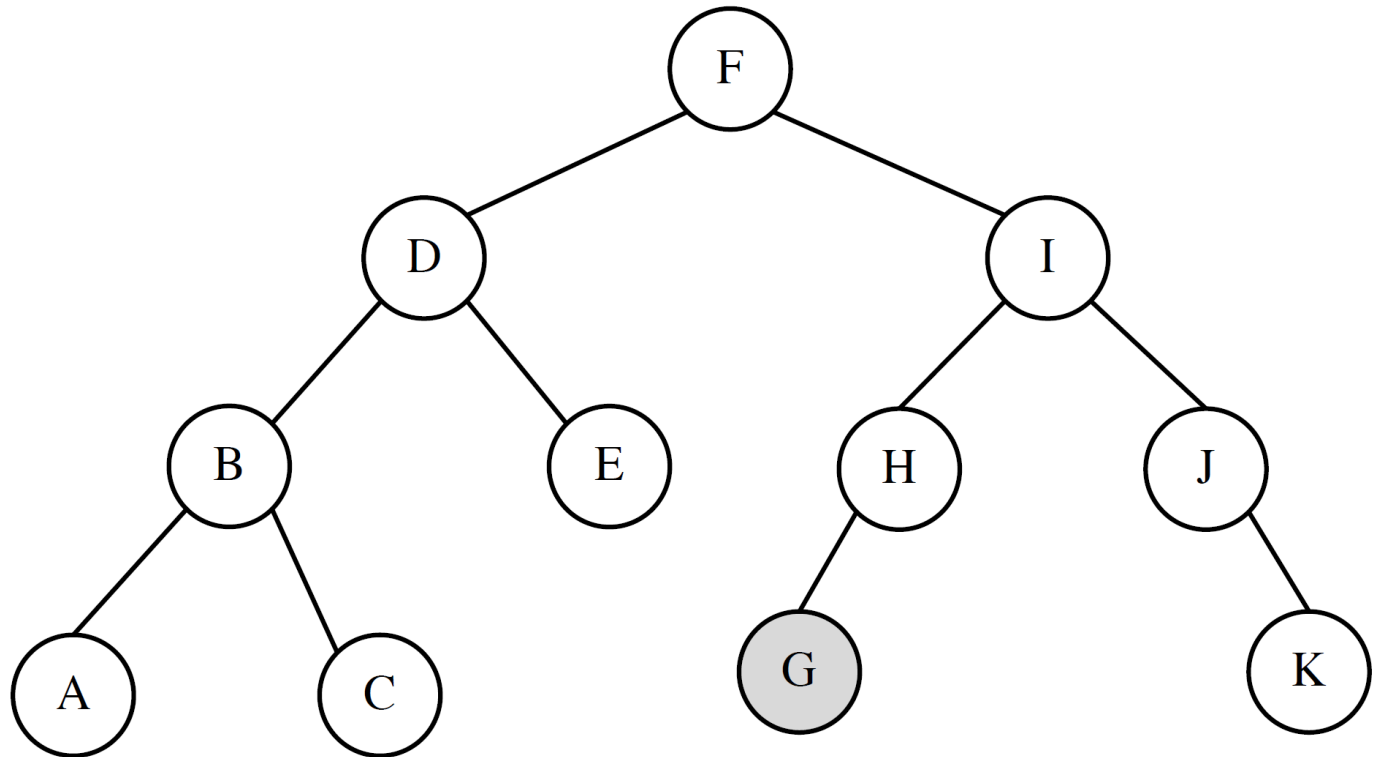
2. E

4. H

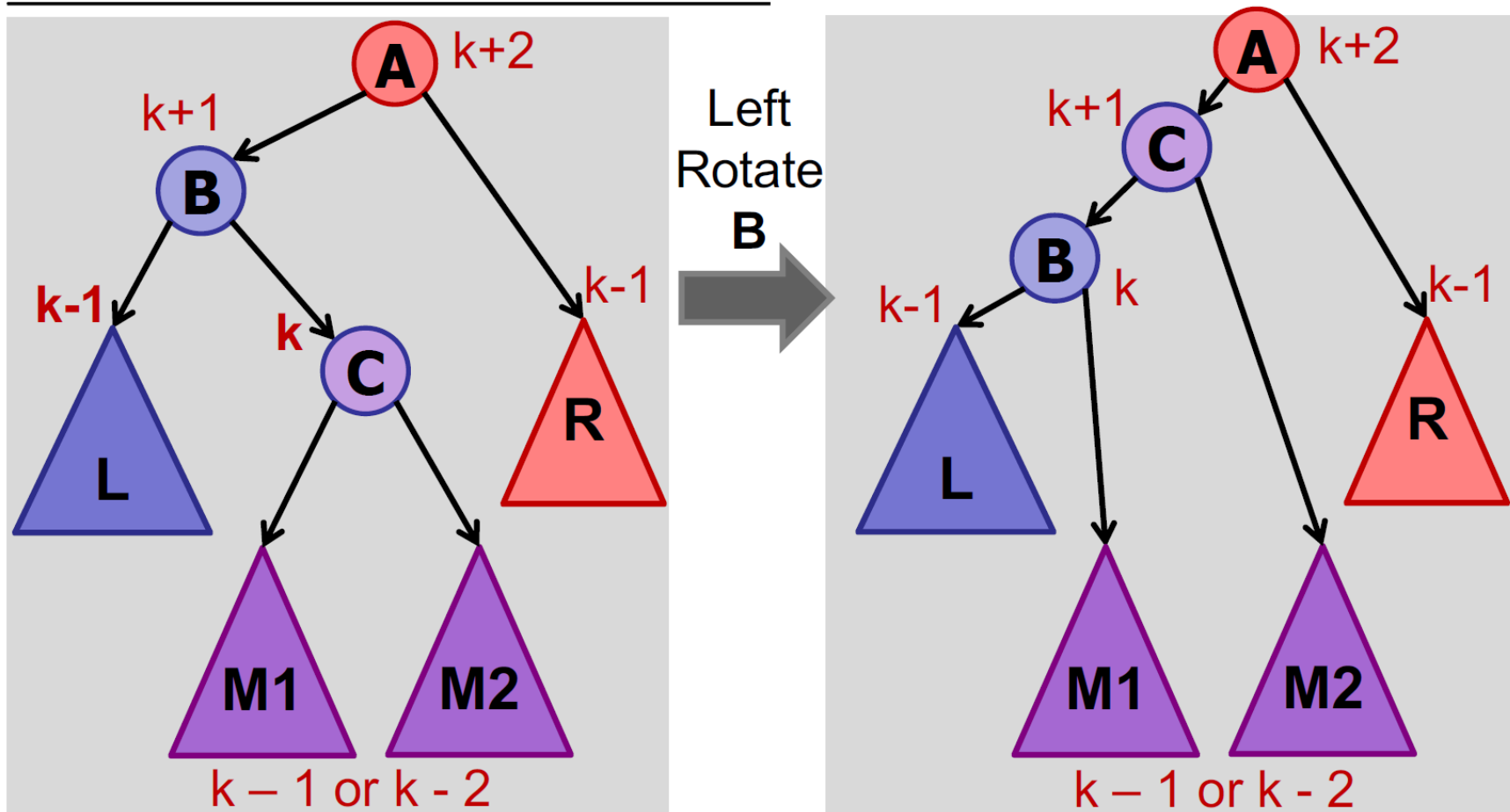
6. J

How Do They Work?

- Try to rotate nodes to determine if it works
- A = I, B = D, C = F



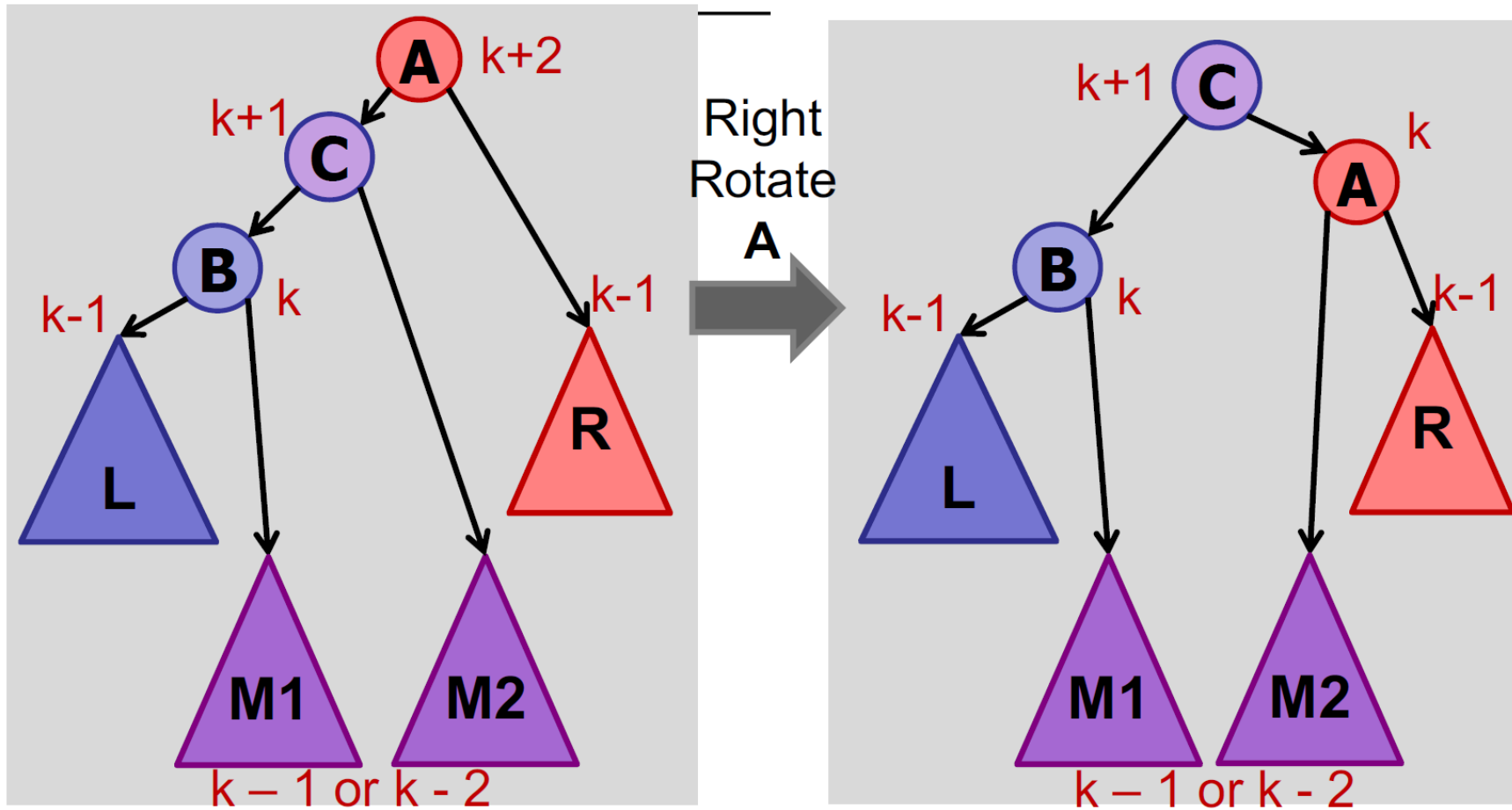
Tree Rotations



Left-rotate B

After left-rotate B: **A** and **C** still out of balance.

Tree Rotations



After right-rotate **A**: all in balance.

How Do They Work?

You have been asked to design an application that can store key/value pairs, supporting insert, delete, and search (by key). For which of the following specifications would you choose a hash table instead of an AVL tree?

- I. An application where a slow worst-case operation may cause catastrophic disaster.
- II. An application where achieving the maximum average speed is important.
- III. An application that can support queries of the form, “find me an entry with a key at least as big as x ”

- | | | |
|-------------|--------------|-----------------------|
| 1. Only I. | 3. Only III. | 5. II and III. |
| 2. Only II. | 4. I and II. | 6. None of the above. |

How Do They Work?

You have been asked to design an application that can store key/value pairs, supporting insert, delete, and search (by key). For which of the following specifications would you choose a hash table instead of an AVL tree?

- I. An application where a slow worst-case operation may cause catastrophic disaster.
- II. An application where achieving the maximum average speed is important.
- III. An application that can support queries of the form, “find me an entry with a key at least as big as x ”

- | | | |
|--------------------|--------------|-----------------------|
| 1. Only I. | 3. Only III. | 5. II and III. |
| 2. Only II. | 4. I and II. | 6. None of the above. |

How Do They Work?

- An application where a slow worst-case operation may cause catastrophic disaster.
 - In the worst case, a hash table runs in $O(n)$ time vs $O(\log n)$ for AVL trees
- An application where achieving the maximum average speed is important.
 - On average, a hash table runs in $O(1)$ time vs $O(\log n)$ for AVL trees
- An application that can support queries of the form, “find me an entry with a key at least as big as x ”
 - Hash tables do not support range queries, only equality queries

How Do They Work?

Imagine you have a weird hash function h : with probability $\frac{1}{2}$, it maps a key uniformly at random to one of the array slots in the range $[0, \frac{m}{4} - 1]$; with probability $\frac{1}{2}$, it maps a key uniformly at random to one of the array slots in the range $[\frac{m}{4}, m - 1]$. (Note: this assumption is a replacement for the simple uniform hashing assumption.)

Assume chaining is used to resolve collisions, and that n items have been previously inserted into the hash table. If a new item x is inserted, what is the expected length of the linked list in the bucket $h(x)$ where x is inserted?

- | | | |
|-----------------|-----------------|-----------------------|
| 1. (n/m) | 4. $(1/2)(m/n)$ | 7. $(3/4)(m/n)$ |
| 2. (m/n) | 5. $(4/3)(n/m)$ | 8. None of the above. |
| 3. $(1/2)(n/m)$ | 6. $(3/4)(n/m)$ | |

How Do They Work?

Imagine you have a weird hash function h : with probability $\frac{1}{2}$, it maps a key uniformly at random to one of the array slots in the range $[0, \frac{m}{4} - 1]$; with probability $\frac{1}{2}$, it maps a key uniformly at random to one of the array slots in the range $[\frac{m}{4}, m - 1]$. (Note: this assumption is a replacement for the simple uniform hashing assumption.)

Assume chaining is used to resolve collisions, and that n items have been previously inserted into the hash table. If a new item x is inserted, what is the expected length of the linked list in the bucket $h(x)$ where x is inserted?

- | | | |
|-----------------|-----------------|-----------------------|
| 1. (n/m) | 4. $(1/2)(m/n)$ | 7. $(3/4)(m/n)$ |
| 2. (m/n) | 5. $(4/3)(n/m)$ | 8. None of the above. |
| 3. $(1/2)(n/m)$ | 6. $(3/4)(n/m)$ | |

How Do They Work?

- Each item has a $\frac{1}{2}$ probability of being mapped uniformly at random to $[0, \frac{m}{4} - 1]$ and $[\frac{m}{4}, m - 1]$
 - The average length of the linked list in buckets in the range $[0, \frac{m}{4} - 1]$ is $\frac{\frac{n}{2}}{\frac{m}{4}}$
 - The average length of the linked list in buckets in the range $[\frac{m}{4}, m - 1]$ is $\frac{\frac{n}{2}}{\frac{3m}{4}}$
- When x is inserted, it has a $\frac{1}{2}$ probability of falling into a bucket in the range $[0, \frac{m}{4} - 1]$ as well as $[\frac{m}{4}, m - 1]$
 - The expected length of the linked list is thus $\frac{1}{2} \times \frac{\frac{n}{2}}{\frac{m}{4}} + \frac{1}{2} \times \frac{\frac{n}{2}}{\frac{3m}{4}} = \frac{4}{3} \times \frac{n}{m}$

SUPER VIRUS FIGHTER

(CS2040S 2020 MIDTERM)

Super Virus Fighter

To pass the time while we are staying home due to pandemic viruses, you decide to design a brand new game: Super Virus Fighter. The game is a massive multiplayer game in which players from around the world play the heroes of our world: the virus fighters! Throughout the game, when a player accomplishes a particular difficult feat, they may temporarily acquire certain special skills, e.g., immunity, cure, quarantine, sleep, etc. And if they ever acquire at least two (2) of these special skills, then (for a time) they become Super Virus Fighter.

Super Virus Fighter

To build this game, you are going to need a data structure. Your data structure needs to store the identities of the players, along with their current special skills. For the purpose of this question, we will identify players by their names (i.e., a string), and you can assume that all names are unique. There are exactly k special skills, and we will refer to the skills by (integer) number, e.g., $1, 2, \dots, k$.

The data structure should support two types of search operations. First, it should allow you to search for a player by name, returning their current set of skills. Second, it should support searching for a player with at least two of the special skills, i.e., it can find a Super Virus Fighter.

Super Virus Fighter

Your data structure should support the following operations:

- `insert(String name)`: adds a player to the data structure with the specified name.
- `addSkill(String name, int skill)`: updates the player with the specified name to have the specified skill.
- `deleteSkill(String name, int skill)`: updates the player with the specified name to not have the specified skill.
- `search(String name)`: returns the set of skills for the specified player.
- `searchSuper()`: returns the name of a player that has at least two skills (or `null`, if none have more than one skill).

Super Virus Fighter

Give a solution that involves a single data structure, which is augmented. Each operation should be implemented as efficiently as possible. (Less efficient solutions, if correct, will get some limited partial credit.) **You may assume that you can compare two strings in $O(1)$ time (so the length of a name does not matter).**

When describing your solution, you do not need to reproduce/restate an algorithm covered in class. You only need to describe how it needs to be changed to solve your problem. When asked for an algorithm, you may give pseudocode or describe the algorithm in words. (Java code is not needed.) You are encouraged to draw pictures to illustrate how your algorithm works.

Super Virus Fighter

To solve the problem, which existing data structure will you start with? (E.g., array, linked list, AVL tree, hash table, etc.).

Super Virus Fighter

To solve the problem, which existing data structure will you start with? (E.g., array, linked list, AVL tree, hash table, etc.).

- AVL tree
 - To allow us to insert and search as efficiently as possible
- Note that while a hash table allows for $O(1)$ insertion and searching by key, searching for a Super Virus Fighter would be $O(n)$ time
 - An $O(n)$ solution for `searchSuper()` is so much slower than $O(\log n)$ that it overwhelms any other benefits

Super Virus Fighter

Describe and explain what you will do. What is stored in each cell/entry/node/bucket of your data structure? What additional information will you store in each cell/entry/node/bucket of your structure?

Super Virus Fighter

Describe and explain what you will do. What is stored in each cell/entry/node/bucket of your data structure? What additional information will you store in each cell/entry/node/bucket of your structure?

- A string to store the name of the player
 - This will be used as the key to order the tree
- A (resizable) hash table containing the special skills the player has
- An int to store the number of special skills the player has
- An int to store the total number of Super Virus Fighters in the subtree rooted at the node

Super Virus Fighter

How will you insert a new player? (Be sure to include how to maintain the additional information listed above.)

Super Virus Fighter

How will you insert a new player? (Be sure to include how to maintain the additional information listed above.)

- Insert a node with the specified name, an empty skill set, and a skill count of 0 into the AVL tree
 - Since the inserted node is a leaf, set the Super Virus Fighter count to 0
- Perform rotations if necessary
 - When a rotation occurs, the Super Virus Fighter counts of all nodes involved must be updated
 - This can be done in $O(1)$ time for all the possible AVL tree rotation cases.

Super Virus Fighter

How will you implement `addSkill` and `deleteSkill`? (Be sure to include how to maintain the additional information listed above.)

Super Virus Fighter

How will you implement `addSkill` and `deleteSkill`? (Be sure to include how to maintain the additional information listed above.)

- Search for the node with the same name as the player using the standard AVL tree search mechanism
- Check whether the player has the specified skill by looking up the hash table
- Add/remove the skill from the hash table as necessary
- Update the skill count as necessary

Super Virus Fighter

How will you implement `addSkill` and `deleteSkill`? (Be sure to include how to maintain the additional information listed above.)

- If the skill count was updated from 1 -> 2, increment the node's Super Virus Fighter count by 1, then walk up the tree to the root, incrementing the Super Virus Fighter count of each node on the path
- If the skill count was updated from 2 -> 1, decrement the node's Super Virus Fighter count by 1, then walk up the tree to the root, decrementing the Super Virus Fighter count of each node on the path

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of `insert`?

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of `insert`?

- $O(\log N)$
 - Searching for place to insert in AVL tree – $O(\log N)$
 - Instantiation of node to insert – $O(1)$
- If an array of size k were used instead of the resizable hash table, node instantiation would take $O(k)$ time, giving us a worst case running time of $O(k + \log N)$ for `insert`

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of `addSkill`, if the player being updated for has s skills?

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of `addSkill`, if the player being updated for has s skills?

- $O(\log N)$
 - Searching for player in AVL tree – $O(\log N)$
 - Updating skill – $O(1)$
 - Updating Super Virus Fighter count when walking up tree – $O(\log N)$

Super Virus Fighter

How will you implement search?

Super Virus Fighter

How will you implement search?

- Search for the node with the same name as the player using the standard AVL tree search mechanism
- Enumerate the keys in the hash table to return an array of skills that the player has

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of search, if the user being searched for has s skills?

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of search, if the user being searched for has s skills?

- $O(k + \log N)$
 - Searching for player in AVL tree – $O(\log N)$
 - Enumerating skills – $O(k)$

Super Virus Fighter

How will you implement searchSuper?

Super Virus Fighter

How will you implement searchSuper?

- Walk down the tree, following a path where the Super Virus Fighter count is at least 1
 - If the Super Virus Fighter count of either child node is at least 1, recurse on either child
 - Else, recurse on the child with a Super Virus Fighter count of at least 1
- When a player with a skill count of at least 2 is found, return that player
- If the root has a Super Virus Fighter count of 0, then there are no Super Virus Fighters

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of `searchSuper`, if the user being searched for has s skills?

Super Virus Fighter

Let N be the total number of items and k the total number of skills. What is the asymptotic worst-case running time of `searchSuper`, if the player being searched for has s skills?

- $O(\log N)$
 - Searching for Super Virus Fighter player in AVL tree – $O(\log N)$
 - Our search is bounded by the height of the tree

INVARIANTS

(CS2040S 2020 MIDTERM)

Invariants

Recall the following (pseudo)code for InsertionSort (where $swap(a, i, j)$ swaps the items at index i and j in array a):

```
void InsertionSort(int[] array)
    int size = array.length;
    for (int i=1; i<size; i++)
        for (int j=i; j>0; j--)
            if (array[j-1] > array[j])
                swap(array, j-1, j);
            else break;
```

Invariants

Which of the following is a good loop invariant for the outer loop in InsertionSort (where the loop invariant is evaluated at the end of the loop). (Choose one.)

- A. For all k such that $k < i$: $A[k] \leq A[k + 1]$.
- B. For all k such that $j < k < i$: $A[k] \leq A[i]$.
- C. The subarray $A[0..i]$ contains the $i + 1$ smallest elements in the array.
- D. The subarray $A[0..i - 1]$ contains the i smallest elements in the array.
- E. None of the above.

Invariants

Which of the following is a good loop invariant for the outer loop in InsertionSort (where the loop invariant is evaluated at the end of the loop). (Choose one.)

- A. **For all k such that $k < i$: $A[k] \leq A[k + 1]$.**
- B. For all k such that $j < k < i$: $A[k] \leq A[i]$.
- C. The subarray $A[0..i]$ contains the $i + 1$ smallest elements in the array.
- D. The subarray $A[0..i - 1]$ contains the i smallest elements in the array.
- E. None of the above.

Invariants

Which of the following are invariants for an AVL tree (evaluated at the end of every operation)? (Select all that apply.) Assume height is defined as in class, where a leaf has height 0.

- A. If node u and v are siblings, then $|height(u) - height(v)| < 2$.
- B. If node u is the parent of node v , then $|height(u) - height(v)| < 2$.
- C. If node u is the parent of node v , then $|height(u) - height(v)| > 0$.
- D. If node u has height h , then the number of nodes in the subtree rooted at u is at most 2^h .
- E. If node u has height h , then the number of nodes in the subtree rooted at u is at least 2^{h-2} .

Invariants

Which of the following are invariants for an AVL tree (evaluated at the end of every operation)? (Select all that apply.) Assume height is defined as in class, where a leaf has height 0.

- A. If node u and v are siblings, then $|\text{height}(u) - \text{height}(v)| < 2$.
- B. If node u is the parent of node v , then $|\text{height}(u) - \text{height}(v)| < 2$.
- C. If node u is the parent of node v , then $|\text{height}(u) - \text{height}(v)| > 0$.
- D. If node u has height h , then the number of nodes in the subtree rooted at u is at most 2^h .
- E. If node u has height h , then the number of nodes in the subtree rooted at u is at least 2^{h-2} .

Invariants

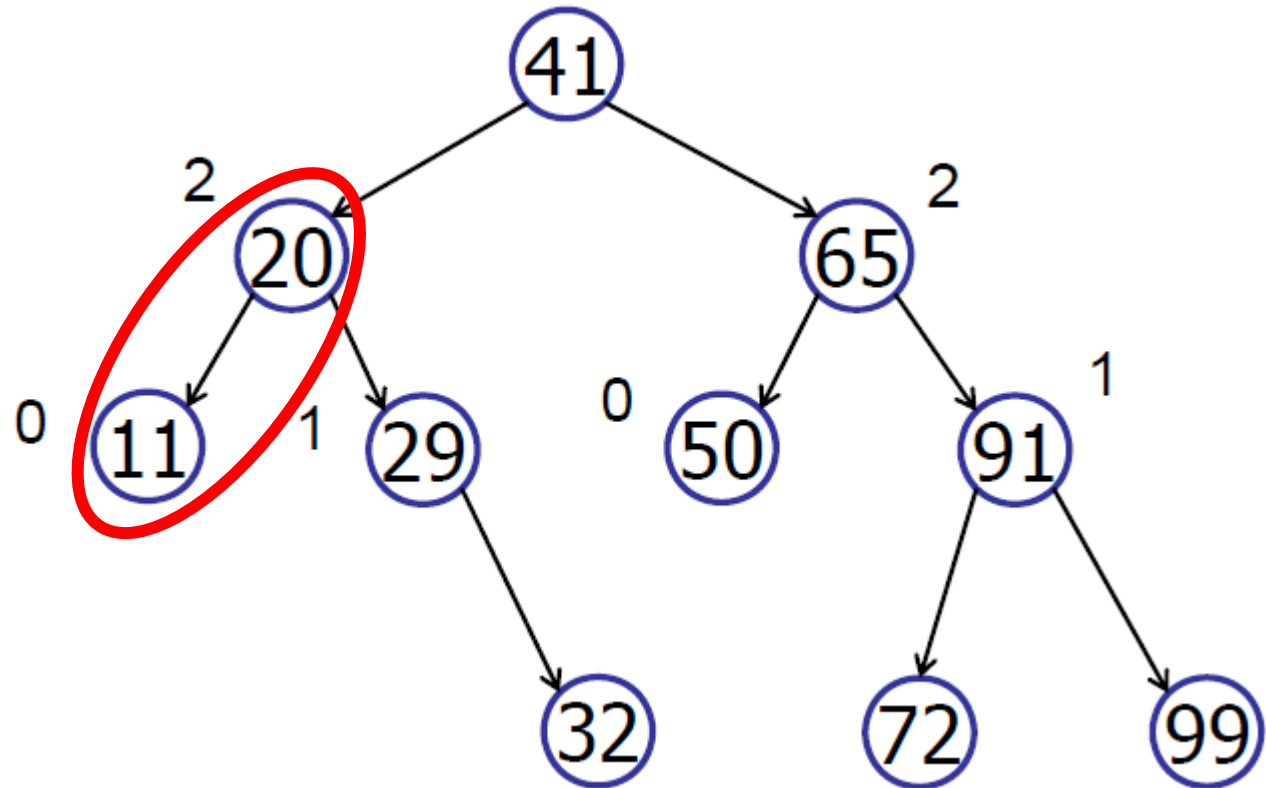
If node u and v are siblings, then $|height(u) - height(v)| < 2$.

- Trivially true by the definition of balance
 - An AVL tree is a height balanced binary search tree
 - A binary search tree is height balanced if every node in the tree is height balanced
 - A node v is height balanced if $|v.left.height - v.right.height| \leq 1$

Invariants

If node u is the parent of node v , then $|height(u) - height(v)| < 2$.

- Counter-example:



Invariants

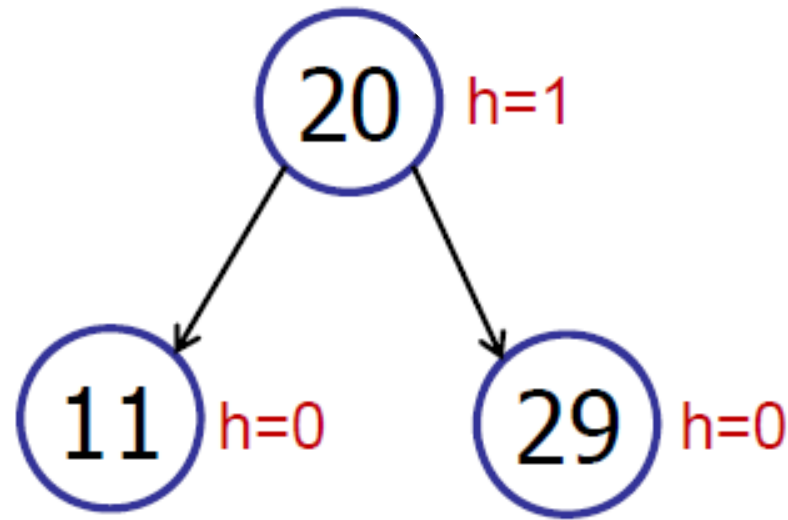
If node u is the parent of node v , then $|height(u) - height(v)| > 0$.

- Trivially true by the definition of height
 - $height = \max(left.height, right.height) + 1$

Invariants

If node u has height h , then the number of nodes in the subtree rooted at u is at most 2^h .

- Counter-example:
 - $2^1 = 2 < 3$



Invariants

If node u has height h , then the number of nodes in the subtree rooted at u is at least 2^{h-2} .

- Counter-example:
 - A Fibonacci tree of order n has $F_{n+2} - 1$ nodes, where F_n is the n th Fibonacci number
 - Fibonacci tree of order 11
 - 233 nodes
 - Root node has height 10 $\Rightarrow 2^{10-2} = 2^8 = 256$

Invariants

Which of the following are invariants for an (a, b) -tree (i.e., properties that are true at the end of every operation)? (Select all that apply.) Assume height is as defined as in an AVL tree, where a leaf has height 0. Define $\deg(u)$ to be the number of children that node u has.

- A. If node u and node v are siblings, then $|\deg(u) - \deg(v)| \leq b$.
- B. If node u and node v are siblings, then $|\deg(u) - \deg(v)| \geq a$.
- C. If node u and node v are siblings, then $|\text{height}(u) - \text{height}(v)| < 1$.
- D. If node u and node v are siblings, then $|\text{height}(u) - \text{height}(v)| < 2$.
- E. If node u has height h , then the subtree rooted at u contains at least a^h nodes.
- F. If node u has height h , then the subtree rooted at u contains at least h^a nodes.

Invariants

Which of the following are invariants for an (a, b) -tree (i.e., properties that are true at the end of every operation)? (Select all that apply.) Assume height is as defined as in an AVL tree, where a leaf has height 0. Define $\deg(u)$ to be the number of children that node u has.

- A. If node u and node v are siblings, then $|\deg(u) - \deg(v)| \leq b$.**
- B. If node u and node v are siblings, then $|\deg(u) - \deg(v)| \geq a$.**
- C. If node u and node v are siblings, then $|\text{height}(u) - \text{height}(v)| < 1$.**
- D. If node u and node v are siblings, then $|\text{height}(u) - \text{height}(v)| < 2$.**
- E. If node u has height h , then the subtree rooted at u contains at least a^h nodes.**
- F. If node u has height h , then the subtree rooted at u contains at least h^a nodes.**

Invariants

If node u and node v are siblings, then $|\deg(u) - \deg(v)| \leq b$.

- Trivially true by the definition of (a, b) -trees
 - a refers to the minimum number of children an internal node (i.e. non-root, non-leaf) can have
 - b refers to the maximum number of children an internal node (i.e. non-root, non-leaf) can have

Invariants

If node u and node v are siblings, then $|\deg(u) - \deg(v)| \geq a$.

- Counter-example
 - When $\deg(u) = \deg(v)$, $|\deg(u) - \deg(v)| = 0$ for any (a, b) -tree

Invariants

If node u and node v are siblings, then $|\text{height}(u) - \text{height}(v)| < 1$.

- Trivially true by the definition of (a, b) -trees
 - All leaf nodes of an (a, b) -tree must be at the same depth (from root)
 - Thus, if node u and node v are siblings, they must have the same height

Invariants

If node u and node v are siblings, then $|\text{height}(u) - \text{height}(v)| < 2$.

- Trivially true by the definition of (a, b) -trees
 - All leaf nodes of an (a, b) -tree must be at the same depth (from root)
 - Thus, if node u and node v are siblings, they must have the same height

Invariants

If node u has height h , then the subtree rooted at u contains at least a^h nodes.

- Trivially true by the definition of (a, b) -trees
 - a refers to the minimum number of children an internal node (i.e. non-root, non-leaf) can have
 - All leaf nodes of an (a, b) -tree must be at the same depth (from root)
 - Complete tree!

Invariants

If node u has height h , then the subtree rooted at u contains at least h^a nodes.

- Counter-example:
 - Node with height $h = 3$ where $a = 5$
 - Total number of nodes:
 - $5^0 + 5^1 + 5^2 + 5^3 = 1 + 5 + 25 + 125 = 156$
 - $h^a = 3^5 = 243$

**WHICH IS BEST?
(CS2040S 2020 MIDTERM)**

Which is Best?

We have seen several different data structures that support a similar set of operations, e.g., insertion, deletion, search, successor, predecessor, etc. For each of the following questions, choose one of the following three options to indicate which data structure is fastest. Assume each contains the same number of keys. (Note that for a randomized algorithm, by “worst-case” we mean the worst case cost for the worst possible random choices.)

- A. AVL Tree
- B. Hash Table (with chaining, load factor 1)

Which is Best?

Which data structure has the fastest worst-case insertion time?

Which is Best?

Which data structure has the fastest worst-case insertion time?

- AVL Tree
 - The worst case insertion time of any tree is $O(h)$ where h is the height of the tree
 - Since AVL trees are balanced binary search trees, $h = O(\log n)$, giving us a worst case insertion time of $O(h) = O(\log n)$
- **Hash Table (with chaining, load factor 1)**
 - $O(1)$ worst case insertion time regardless of the number of keys in the bucket
 - Just insert the key at the front of the linked list!

Which is Best?

Which data structure has the fastest expected insertion time?

Which is Best?

Which data structure has the fastest expected insertion time?

- AVL Tree
 - The expected insertion time of any tree is $O(h)$ where h is the height of the tree
 - Since AVL trees are balanced binary search trees, $h = O(\log n)$, giving us an expected insertion time of $O(h) = O(\log n)$
- **Hash Table (with chaining, load factor 1)**
 - $O(1)$ expected insertion time

Which is Best?

Which data structure has the fastest worst-case search time?

Which is Best?

Which data structure has the fastest worst-case search time?

- **AVL Tree**

- The worst case search time of any tree is $O(h)$ where h is the height of the tree
 - This occurs when the key being searched for is at the deepest leaf node
- Since AVL trees are balanced binary search trees, $h = O(\log n)$, giving us a worst case search time of $O(h) = O(\log n)$

- Hash Table (with chaining, load factor 1)

- The worst case search time occurs when all n keys map to the same bucket, and the key that is being searched for is at the end of the linked list, giving us a worst case search time of $O(n)$

Which is Best?

Which data structure has the fastest expected search time?

Which is Best?

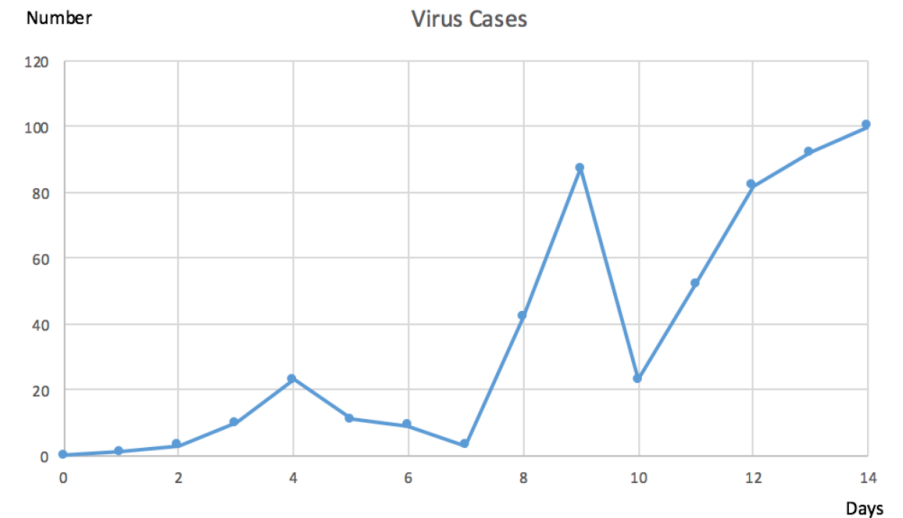
Which data structure has the fastest expected search time?

- AVL Tree
 - The expected search time of any tree is $O(h)$ where h is the height of the tree
 - Since AVL trees are balanced binary search trees, $h = O(\log n)$, giving us an expected search time of $O(h) = O(\log n)$
- **Hash Table (with chaining, load factor 1)**
 - The expected search time is $1 + \text{load factor} = 1 + 1 = O(1)$

**WHAT GOES UP MUST COME
DOWN?
(CS2040S 2020 MIDTERM)**

What Goes Up Must Come Down?

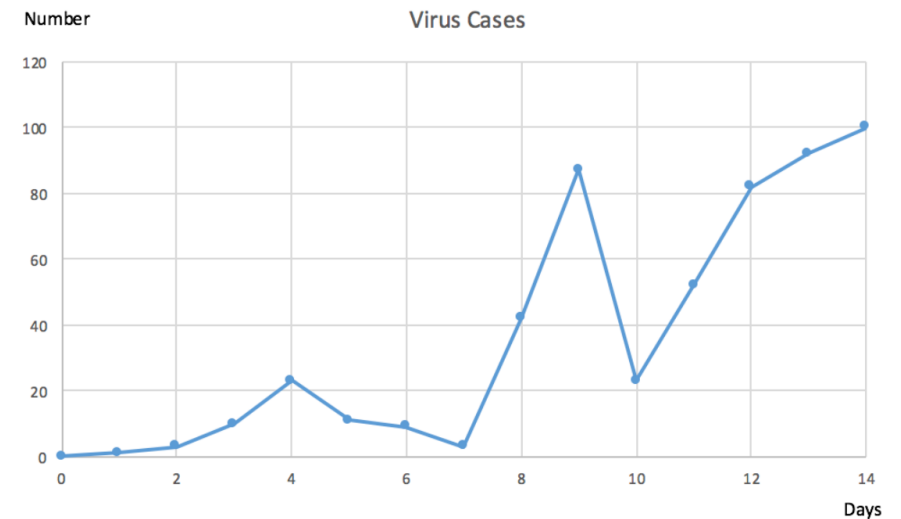
The goal of your game Super Virus Hunters is, of course, to decrease the number of sick people in your simulated world. Therefore, as the game runs, you keep careful track of the number of currently sick patients. But the game is hard! Sometimes the levels go up, sometimes they go down. Whenever the number of sick patients drops below a previously seen level, that is good! We call that occurrence a recovery period. For example, consider the following data from two weeks of the game:



`dataArray = [0, 1, 3, 10, 23, 11, 9, 3, 42, 87, 23, 52, 82, 92, 100]`

What Goes Up Must Come Down?

There is one recover period from day 3 to day 6 (where it starts at 10 and ends at 9); another recover period is from day 4 to day 7 (from 23 to 3), and yet another recovery period from day 3 to day 7 (from 10 to 3). In fact, the latter is the longest recover period in this two week interval. (Notice that a recovery period can continue even as the levels drop; it does not stop as soon as it drops below its previous level.) In case you are still confused, only the start and end days matter. You can ignore what happens in between.



dataArray = [0, 1, 3, 10, 23, 11, 9, 3, 42, 87, 23, 52, 82, 92, 100]

What Goes Up Must Come Down?

As the game gets harder, the length of the recovery periods seems to be increasing. Your goal in this problem is to find the longest recovery period in your data.

More specifically, you are given an array `data[1..n]` where each `data[i]` is the number of sick patients on day `i` of the game. We want to find a pair of indices `(low, high)` where `data[low] > data[high]` that maximizes `(high-low)`. We will do this in three steps, below.

For the purpose of this problem, you may not use a hash table. You may only compare the values in the table.

For each of the following parts, give the most efficient algorithm you can.

What Goes Up Must Come Down?

First, give an algorithm for computing a prefix max-array M . That is, given an array of integers A , compute $M[j] = \max_{i=1}^j A[i]$. For example, if $A = [5, 2, 7, 1]$, then $M = [5, 5, 7, 7]$.

What Goes Up Must Come Down?

First, give an algorithm for computing a prefix max-array M . That is, given an array of integers A , compute $M[j] = \max_{i=1}^j A[i]$. For example, if $A = [5, 2, 7, 1]$, then $M = [5, 5, 7, 7]$.

- Perform a linear scan of the array while keeping track of the largest element seen thus far
- For each element, overwrite it with the largest element

```
computePrefixMax(A)
    M = new array of size A.length
    max = 0;
    for j = 0 to A.length-1
        if (A[j] > max) then max = A[j]
        M[j] = max
    return M
```

What Goes Up Must Come Down?

Next, assume you are given a prefix max-array as described in the previous part. Give the pseudocode for the function `findFirst(M, key)` that finds the smallest index j in M where $M[j] > \text{key}$.

What Goes Up Must Come Down?

Next, assume you are given a prefix max-array as described in the previous part. Give the pseudocode for the function `findFirst(M, key)` that finds the smallest index j in M where $M[j] > \text{key}$.

- Observe that the prefix max-array is monotonically increasing
- We can find j by binary searching on M

```
findFirst(M, key)
    low = 0
    high = M.length-1
    if M[high] < key then return -1
    while (low < high)
        mid = (low + high)/2
        if (M[mid] <= key) then low = mid+1
        else if (M[mid] > key) then high = mid
    return low
```

What Goes Up Must Come Down?

Now give an algorithm for computing the maximum length recovery period, i.e., given data, find begin and end such that $\text{data}[\text{begin}] > \text{data}[\text{end}]$ where $(\text{end} - \text{begin})$ is maximum. You can use the algorithms from the previous parts as black-box functions, even if you were not able to write the pseudocode.

What Goes Up Must Come Down?

Now give an algorithm for computing the maximum length recovery period, i.e., given data, find begin and end such that $\text{data}[\text{begin}] > \text{data}[\text{end}]$ where $(\text{end} - \text{begin})$ is maximum. You can use the algorithms from the previous parts as black-box functions, even if you were not able to write the pseudocode.

- First, compute the prefix max array
- Then, for each element i in the array, find the smallest index j where $M[j] > M[i]$
- If such a $j < i$ exists, then we have found a recovery period!
- Keep track of the longest recovery period we have found and return it at the end

```
computeMaxRecoveryPeriod(data)
    M = computePrefixMax(data)
    max = 0;
    begin = -1;
    end = -1;
    for j = 0 to A.length-1
        k = findFirst(M, data[j])
        if (k >= 0) and (k < j) then
            if (j - k) > max then
                begin = k
                end = j
                max = end - begin
    return (begin, end)
```

What Goes Up Must Come Down?

Explain how algorithm works (in words) and why it is correct. (You might want to give an example execution.)

- As explained on the previous slide.
- Model solution on the next slide.

What Goes Up Must Come Down?

The basic idea is to iterate through all the possible endpoints of a recovery interval, and find the earliest point that could be the beginning of a recovery interval. If we are looking at time j , where the value is $\text{data}[j]$, we want to find the smallest index in data that is larger than $\text{data}[j]$; that will maximize the length of the recovery interval. If $M[t] > j$, we know that some point in the interval $[1, t]$ must be larger than j , and the converse is true as well. So all we need to do is find the smallest t where $M[t] > j$.

And that is exactly what the `findFirst` routine does. Luckily, M is a sorted array, so we can use binary search. The `findFirst` routine is just an implementation of binary search where it recurses left if the midpoint is $> key$ and right if the midpoint is $\leq key$. Throughout it maintains the invariant that there is at least one element in the range $> key$. It also maintains the invariant that in every iteration, every value at an index less than `low` is $\leq key$. Every iteration reduces the range between `low` and `high`, and when it completes, `low` = `high`. This ensures that it finds the smallest index that is larger than the specified key.

What Goes Up Must Come Down?

What is the asymptotic worst-case running time of the complete algorithm for finding the maximum recovery period?

What Goes Up Must Come Down?

What is the asymptotic worst-case running time of the complete algorithm for finding the maximum recovery period?

- First, compute the prefix max array – $O(n)$
- Then, for each element i in the array, find the smallest index j where $M[j] > M[i] - O(n) \times O(\log n) = O(n \log n)$
- Overall time complexity of $O(n \log n)$

SORTING JUMBLE

(CS2040S 2020 MIDTERM)

Sorting Jumble

The first column in the table below contains an unsorted list of words. The last column contains a sorted list of words. Each intermediate column contains a partially sorted list.

Each intermediate column was constructed by beginning with the unsorted list at the left and running one of the sorting algorithms that we learned about in class, stopping at some point before it finishes. Each algorithm is executed exactly as described in the lecture notes. One column has been sorted using a sorting algorithm that you have not seen in class. **(Recursive algorithms recurse on the left half of the array before the right half.)**

Identify which column was (partially) sorted with which sorting algorithm.

Unsorted	A	B	C	D	E	F	Sorted
Mary	Eddie	Eddie	Alice	Alice	Alice	Eddie	Alice
Harry	Fred	Gina	Bob	Bob	Bob	Gina	Bob
Patty	Gina	Harry	Carol	Carol	Carol	Harry	Carol
Eddie	Harry	Kelly	Eddie	Dave	Dave	Fred	Dave
Gina	Ina	Mary	Gina	Eddie	Eddie	Alice	Eddie
Kelly	Kelly	Patty	Kelly	Fred	Fred	Ina	Fred
Ina	Mary	Ina	Ina	Gina	Ina	Bob	Gina
Fred	Patty	Fred	Fred	Kelly	Harry	Kelly	Harry
Alice	Alice	Alice	Dave	Mary	Gina	Carol	Ina
Noah	Bob	Noah	John	Noah	Kelly	John	John
Bob	Linda	Bob	Harry	Harry	John	Dave	Kelly
Linda	Noah	Linda	Linda	Linda	Ophelia	Linda	Linda
Carol	Carol	Carol	Mary	Patty	Patty	Mary	Mary
John	Dave	John	Noah	John	Noah	Noah	Noah
Dave	John	Dave	Patty	Ina	Mary	Ophelia	Ophelia
Ophelia	Ophelia	Ophelia	Ophelia	Ophelia	Linda	Patty	Patty
Unsorted	A	B	C	D	E	F	Sorted

Sorting Jumble

- Do not just execute each sorting algorithm, step-by-step, until it matches one of the columns
 - You will run out of time if you do that
 - Only do so if you've already narrowed down to a few possible search algorithms and you cannot make use of any other invariants
- Think in terms of invariants that are true at every step of the algorithm!

Sorting Algorithm	Description	Invariant	Is stable?
Bubble Sort	“Bubble” the largest element to the end of the array through repeated swapping of out-of-order adjacent pairs (inversions)	Largest k elements are at the end of the array (after k iterations)	Yes
Selection Sort	Select the minimum element and add it to the sorted region of the array by swapping. Repeat until all elements have been selected.	Smallest k elements are at the start of the array (after k iterations)	No
Insertion Sort	Select the first element in the unsorted region of the array and find where to place it in the sorted region. Repeat until all elements have been selected.	First k elements are sorted (after k iterations), last $n - k$ elements are untouched	Yes
Merge Sort	Halve the array, recursively sort, then merge	Each subarray is already sorted when merging	Yes
Quick Sort (with first element pivot)	Partition around the first element, then repeat on subarrays	All elements to the left/right of the pivot are smaller/larger	No

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array A is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Patty) is not at the end of the array. Thus, sorting algorithm A is not Bubble Sort.

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array A is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the smallest element (Alice) is not at the start of the array. Thus, sorting algorithm A is not Selection Sort.

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since only the last element (Ophelia) is untouched, at least $n - 1$ iterations of the sort was run. However, the first $n - 1$ elements of the array A are not sorted. Thus, sorting algorithm A is not Insertion Sort.

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	A	Sorted
Mary	Eddie	Alice
Harry	Fred	Bob
Patty	Gina	Carol
Eddie	Harry	Dave
Gina	Ina	Eddie
Kelly	Kelly	Fred
Ina	Mary	Gina
Fred	Patty	Harry
Alice	Alice	Ina
Noah	Bob	John
Bob	Linda	Kelly
Linda	Noah	Linda
Carol	Carol	Mary
John	Dave	Noah
Dave	John	Ophelia
Ophelia	Ophelia	Patty
Unsorted	A	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Merge Sort	Each subarray is already sorted when merging
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since each subarray (when split by powers of 2) is locally sorted, sorting algorithm A is Merge Sort!

Unsorted	B	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Kelly	Dave
Gina	Mary	Eddie
Kelly	Patty	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Alice	Ina
Noah	Noah	John
Bob	Bob	Kelly
Linda	Linda	Linda
Carol	Carol	Mary
John	John	Noah
Dave	Dave	Ophelia
Ophelia	Ophelia	Patty
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Kelly	Dave
Gina	Mary	Eddie
Kelly	Patty	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Alice	Ina
Noah	Noah	John
Bob	Bob	Kelly
Linda	Linda	Linda
Carol	Carol	Mary
John	John	Noah
Dave	Dave	Ophelia
Ophelia	Ophelia	Patty
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Kelly	Dave
Gina	Mary	Eddie
Kelly	Patty	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Alice	Ina
Noah	Noah	John
Bob	Bob	Kelly
Linda	Linda	Linda
Carol	Carol	Mary
John	John	Noah
Dave	Dave	Ophelia
Ophelia	Ophelia	Patty
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array B is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Patty) is not at the end of the array. Thus, sorting algorithm B is not Bubble Sort.

Unsorted	B	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Kelly	Dave
Gina	Mary	Eddie
Kelly	Patty	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Alice	Ina
Noah	Noah	John
Bob	Bob	Kelly
Linda	Linda	Linda
Carol	Carol	Mary
John	John	Noah
Dave	Dave	Ophelia
Ophelia	Ophelia	Patty
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Kelly	Dave
Gina	Mary	Eddie
Kelly	Patty	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Alice	Ina
Noah	Noah	John
Bob	Bob	Kelly
Linda	Linda	Linda
Carol	Carol	Mary
John	John	Noah
Dave	Dave	Ophelia
Ophelia	Ophelia	Patty
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array B is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the smallest element (Alice) is not at the start of the array. Thus, sorting algorithm B is not Selection Sort.

Unsorted	B	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Kelly	Dave
Gina	Mary	Eddie
Kelly	Patty	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Alice	Ina
Noah	Noah	John
Bob	Bob	Kelly
Linda	Linda	Linda
Carol	Carol	Mary
John	John	Noah
Dave	Dave	Ophelia
Ophelia	Ophelia	Patty
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	B	Sorted
<div>Mary</div> <div>Harry</div> <div>Patty</div> <div>Eddie</div> <div>Gina</div> <div>Kelly</div> <div>Ina</div> <div>Fred</div> <div>Alice</div> <div>Noah</div> <div>Bob</div> <div>Linda</div> <div>Carol</div> <div>John</div> <div>Dave</div> <div>Ophelia</div>	<div>Eddie</div> <div>Gina</div> <div>Harry</div> <div>Kelly</div> <div>Mary</div> <div>Patty</div> <div>Ina</div> <div>Fred</div> <div>Alice</div> <div>Noah</div> <div>Bob</div> <div>Linda</div> <div>Carol</div> <div>John</div> <div>Dave</div> <div>Ophelia</div>	<div>Alice</div> <div>Bob</div> <div>Carol</div> <div>Dave</div> <div>Eddie</div> <div>Fred</div> <div>Gina</div> <div>Harry</div> <div>Ina</div> <div>John</div> <div>Kelly</div> <div>Linda</div> <div>Mary</div> <div>Noah</div> <div>Ophelia</div> <div>Patty</div>
Unsorted	B	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Insertion Sort	First k elements are sorted (after k iterations), last $n - k$ elements are untouched
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the first 6 elements are sorted and the last 10 elements are untouched, sorting algorithm B is Insertion Sort!

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array C is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Patty) is not at the end of the array. Thus, sorting algorithm C is not Bubble Sort.

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

The first 3 elements of array C are sorted, but not the 4th element. If sorting algorithm C is Selection Sort, it must have been run for **no more than three iterations**.

Regardless of whether 1, 2, or 3 iterations were run, Mary should be in Alice's original position in the array (due to swapping). However, this is not the case. Thus, sorting algorithm C is not Selection Sort.

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the first element in the unsorted array (Mary) appears in its sorted location in array C and the elements to the left/right of Mary are smaller/larger, sorting algorithm C is possibly Quick Sort. **However, we cannot say for sure as array F has the same properties!** We can either try to execute Quick Sort step-by-step (not recommended), or try to figure out if array F's identity can be determined to find out the identity of array C through elimination.

Unsorted	D	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Gina	Gina
Fred	Kelly	Harry
Alice	Mary	Ina
Noah	Noah	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Patty	Mary
John	John	Noah
Dave	Ina	Ophelia
Ophelia	Ophelia	Patty
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	D	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Gina	Gina
Fred	Kelly	Harry
Alice	Mary	Ina
Noah	Noah	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Patty	Mary
John	John	Noah
Dave	Ina	Ophelia
Ophelia	Ophelia	Patty
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	D	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Gina	Gina
Fred	Kelly	Harry
Alice	Mary	Ina
Noah	Noah	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Patty	Mary
John	John	Noah
Dave	Ina	Ophelia
Ophelia	Ophelia	Patty
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array D is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Patty) is not at the end of the array. Thus, sorting algorithm D is not Bubble Sort.

Unsorted	D	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Gina	Gina
Fred	Kelly	Harry
Alice	Mary	Ina
Noah	Noah	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Patty	Mary
John	John	Noah
Dave	Ina	Ophelia
Ophelia	Ophelia	Patty
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	D	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Gina	Gina
Fred	Kelly	Harry
Alice	Mary	Ina
Noah	Noah	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Patty	Mary
John	John	Noah
Dave	Ina	Ophelia
Ophelia	Ophelia	Patty
Unsorted	D	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Selection Sort	Smallest k elements are at the start of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

The smallest k elements are at the start of array D. However, this is also the case for array E. As such, we have no choice but to execute Selection Sort. D:

Unsorted	D	E	Sorted
Mary	Alice	Alice	Alice
Harry	Bob	Bob	Bob
Patty	Carol	Carol	Carol
Eddie	Dave	Dave	Dave
Gina	Eddie	Eddie	Eddie
Kelly	Fred	Fred	Fred
Ina	Gina	Ina	Gina
Fred	Kelly	Harry	Harry
Alice	Mary	Gina	Ina
Noah	Noah	Kelly	John
Bob	Harry	John	Kelly
Linda	Linda	Ophelia	Linda
Carol	Patty	Patty	Mary
John	John	Noah	Noah
Dave	Ina	Mary	Ophelia
Ophelia	Ophelia	Linda	Patty
Unsorted	D	E	Sorted

Selection sort

- Select the minimum element
- Add it to the sorted region of the array by swapping
- Repeat until all elements have been selected

Unsorted	D	E	Sorted
Mary	Alice	Alice	Alice
Harry	Bob	Bob	Bob
Patty	Carol	Carol	Carol
Eddie	Dave	Dave	Dave
Gina	Eddie	Eddie	Eddie
Kelly	Fred	Fred	Fred
Ina	Gina	Ina	Gina
Fred	Kelly	Harry	Harry
Alice	Mary	Gina	Ina
Noah	Noah	Kelly	John
Bob	Harry	John	Kelly
Linda	Linda	Ophelia	Linda
Carol	Patty	Patty	Mary
John	John	Noah	Noah
Dave	Ina	Mary	Ophelia
Ophelia	Ophelia	Linda	Patty
Unsorted	D	E	Sorted

Selection sort

- Select the minimum element
- Add it to the sorted region of the array by swapping
- Repeat until all elements have been selected

Unsorted	D	E	Sorted
Mary	Alice	Alice	Alice
Harry	Bob	Bob	Bob
Patty	Carol	Carol	Carol
Eddie	Dave	Dave	Dave
Gina	Eddie	Eddie	Eddie
Kelly	Fred	Fred	Fred
Ina	Gina	Ina	Gina
Fred	Kelly	Harry	Harry
Alice	Mary	Gina	Ina
Noah	Noah	Kelly	John
Bob	Harry	John	Kelly
Linda	Linda	Ophelia	Linda
Carol	Patty	Patty	Mary
John	John	Noah	Noah
Dave	Ina	Mary	Ophelia
Ophelia	Ophelia	Linda	Patty
Unsorted	D	E	Sorted

Selection sort

- Select the minimum element
- Add it to the sorted region of the array by swapping
- Repeat until all elements have been selected

Unsorted	D	E	Sorted
Mary	Alice	Alice	Alice
Harry	Bob	Bob	Bob
Patty	Carol	Carol	Carol
Eddie	Dave	Dave	Dave
Gina	Eddie	Eddie	Eddie
Kelly	Fred	Fred	Fred
Ina	Gina	Ina	Gina
Fred	Kelly	Harry	Harry
Alice	Mary	Gina	Ina
Noah	Noah	Kelly	John
Bob	Harry	John	Kelly
Linda	Linda	Ophelia	Linda
Carol	Patty	Patty	Mary
John	John	Noah	Noah
Dave	Ina	Mary	Ophelia
Ophelia	Ophelia	Linda	Patty
Unsorted	D	E	Sorted

Selection sort

- Select the minimum element
- Add it to the sorted region of the array by swapping
- Repeat until all elements have been selected

When executing Selection Sort, the position of Mary in array E is wrong. Thus, sorting algorithm D is Selection Sort!

Unsorted	E	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Ina	Gina
Fred	Harry	Harry
Alice	Gina	Ina
Noah	Kelly	John
Bob	John	Kelly
Linda	Ophelia	Linda
Carol	Patty	Mary
John	Noah	Noah
Dave	Mary	Ophelia
Ophelia	Linda	Patty
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	E	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Ina	Gina
Fred	Harry	Harry
Alice	Gina	Ina
Noah	Kelly	John
Bob	John	Kelly
Linda	Ophelia	Linda
Carol	Patty	Mary
John	Noah	Noah
Dave	Mary	Ophelia
Ophelia	Linda	Patty
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	E	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Ina	Gina
Fred	Harry	Harry
Alice	Gina	Ina
Noah	Kelly	John
Bob	John	Kelly
Linda	Ophelia	Linda
Carol	Patty	Mary
John	Noah	Noah
Dave	Mary	Ophelia
Ophelia	Linda	Patty
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the ordering of elements in array E is different from the unsorted array, we know that **at least one** iteration of the sort was run. However, the largest element (Patty) is not at the end of the array. Thus, sorting algorithm E is not Bubble Sort.

Unsorted	E	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Ina	Gina
Fred	Harry	Harry
Alice	Gina	Ina
Noah	Kelly	John
Bob	John	Kelly
Linda	Ophelia	Linda
Carol	Patty	Mary
John	Noah	Noah
Dave	Mary	Ophelia
Ophelia	Linda	Patty
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	E	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Ina	Gina
Fred	Harry	Harry
Alice	Gina	Ina
Noah	Kelly	John
Bob	John	Kelly
Linda	Ophelia	Linda
Carol	Patty	Mary
John	Noah	Noah
Dave	Mary	Ophelia
Ophelia	Linda	Patty
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the position of the pivot element Mary is incorrect, sorting algorithm E cannot be Quick Sort.

Unsorted	E	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Dave	Dave
Gina	Eddie	Eddie
Kelly	Fred	Fred
Ina	Ina	Gina
Fred	Harry	Harry
Alice	Gina	Ina
Noah	Kelly	John
Bob	John	Kelly
Linda	Ophelia	Linda
Carol	Patty	Mary
John	Noah	Noah
Dave	Mary	Ophelia
Ophelia	Linda	Patty
Unsorted	E	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

By elimination, sorting algorithm E is none of the 5 sorting algorithms!

Unsorted	F	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Fred	Dave
Gina	Alice	Eddie
Kelly	Ina	Fred
Ina	Bob	Gina
Fred	Kelly	Harry
Alice	Carol	Ina
Noah	John	John
Bob	Dave	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Ophelia	Ophelia
Ophelia	Patty	Patty
Unsorted	F	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	F	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Fred	Dave
Gina	Alice	Eddie
Kelly	Ina	Fred
Ina	Bob	Gina
Fred	Kelly	Harry
Alice	Carol	Ina
Noah	John	John
Bob	Dave	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Ophelia	Ophelia
Ophelia	Patty	Patty
Unsorted	F	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Unsorted	F	Sorted
Mary	Eddie	Alice
Harry	Gina	Bob
Patty	Harry	Carol
Eddie	Fred	Dave
Gina	Alice	Eddie
Kelly	Ina	Fred
Ina	Bob	Gina
Fred	Kelly	Harry
Alice	Carol	Ina
Noah	John	John
Bob	Dave	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Ophelia	Ophelia
Ophelia	Patty	Patty
Unsorted	F	Sorted

Sorting Algorithm	Invariant
Bubble Sort	Largest k elements are at the end of the array (after k iterations)
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

Since the largest k elements are at the end of array F (and no other arrays have Patty at the end), sorting algorithm F is Bubble Sort!

Unsorted	C	Sorted
Mary	Alice	Alice
Harry	Bob	Bob
Patty	Carol	Carol
Eddie	Eddie	Dave
Gina	Gina	Eddie
Kelly	Kelly	Fred
Ina	Ina	Gina
Fred	Fred	Harry
Alice	Dave	Ina
Noah	John	John
Bob	Harry	Kelly
Linda	Linda	Linda
Carol	Mary	Mary
John	Noah	Noah
Dave	Patty	Ophelia
Ophelia	Ophelia	Patty
Unsorted	C	Sorted

Sorting Algorithm	Invariant
Quick Sort (with first element pivot)	All elements to the left/right of the pivot are smaller/larger

By the process of elimination, sorting algorithm C is Quick Sort!

ASYMPTOTIC ANALYSIS

(CS2040S 2021 MIDTERM)

Asymptotic Analysis

Choose the tightest possible bound for the following function:

$$T(n) = 24n^{0.5} \log^2 n + 1.7 \log^3 n$$

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n \log^2 n)$
6. $O(n^2)$
7. None of the above.

Asymptotic Analysis

Choose the tightest possible bound for the following function:

$$T(n) = 24n^{0.5} \log^2 n + 1.7 \log^3 n$$

- 1. $O(1)$
- 2. $O(\log n)$
- 3. **$O(n)$**
- 4. $O(n \log n)$
- 5. $O(n \log^2 n)$
- 6. $O(n^2)$
- 7. **None of the above.**

Asymptotic Analysis

Choose the tightest possible bound for the following function:

$$\begin{aligned}T(n) &= 24n^{0.5} \log^2 n + 1.7 \log^3 n \\&= O(n^{0.5} \log^2 n) + O(\log^3 n) \\&= O(n^{0.5} \log^2 n)\end{aligned}$$

- 1. $O(1)$
- 2. $O(\log n)$
- 3. **$O(n)$**
- 4. $O(n \log n)$

- 5. $O(n \log^2 n)$
- 6. $O(n^2)$
- 7. **None of the above.**

The intent of the question was to find the time complexity, hence the expected answer was “None of the above”. However, if interpreted as the tightest possible bound out of the options given, it’s $O(n)$.

Algorithm Analysis

$3^n = O(2^n)$: True or False?

Algorithm Analysis

$3^n = O(2^n)$: True or **False**?

Algorithm Analysis

$3^n = O(2^n)$: True or **False**?

- Bases matter in exponents!

Algorithm Analysis

$2^{\log_3 n} = O(n)$: True or False?

Algorithm Analysis

$2^{\log_3 n} = O(n)$: **True** or False?

Algorithm Analysis

$2^{\log_3 n} = O(n)$: **True** or False?

$$\begin{aligned} 2^{\log_3 n} &= 2^{\frac{\log_2 n}{\log_2 3}} \\ &= \left(2^{\log_2 n}\right)^{\frac{1}{\log_2 3}} \\ &\approx n^{0.631} \\ &\leq n \\ &= O(n) \end{aligned}$$

Algorithm Analysis

Suppose that $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$. Then which of the following statements is/are true?

- I. It is possible that $h(n) = \Theta(g(n))$.
 - II. It is always the case that $h(n) = O(g(n))$.
 - III. It is always the case that $h(n) = \Omega(g(n))$.
- 1. Statement I.
 - 2. Statement I and II.
 - 3. Statement I and III.
 - 4. Statement I and II and III.
 - 5. Statement II.
 - 6. Statement II and III.
 - 7. None of the statements are true.

Algorithm Analysis

Suppose that $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$. Then which of the following statements is/are true?

- I. It is possible that $h(n) = \Theta(g(n))$.
 - II. It is always the case that $h(n) = O(g(n))$.
 - III. It is always the case that $h(n) = \Omega(g(n))$.
- | | |
|--------------------------------|-------------------------------------|
| 1. Statement I. | 5. Statement II. |
| 2. Statement I and II. | 6. Statement II and III. |
| 3. Statement I and III. | 7. None of the statements are true. |
| 4. Statement I and II and III. | |

Algorithm Analysis

- Big O (O) Notation
 - Upper bound
- Big Omega (Ω) Notation
 - Lower bound
- Big Theta (Θ) Notation
 - Both upper and lower bound
- Constants don't matter
- Minor terms don't matter

Intuitively...

Algorithm Analysis

$$f(n) \leq g(n)$$

$$f(n) \geq h(n)$$

Suppose that $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$. Then which of the following statements is/are true?

I. It is possible that $h(n) = \Theta(g(n))$. Possible if $f(n) = g(n)$ and $g(n) = h(n)$

II. It is always the case that $h(n) = O(g(n))$. $h(n) \leq f(n) \leq g(n)$

III. It is always the case that $h(n) = \Omega(g(n))$. Opposite of statement II

1. Statement I.

5. Statement II.

2. **Statement I and II.**

6. Statement II and III.

3. Statement I and III.

7. None of the statements are true.

4. Statement I and II and III.

ALGORITHM ANALYSIS

(CS2040S 2020 MIDTERM)

Algorithm Analysis

For each of the following, choose the best (tightest) asymptotic function from among the given options. Some of the following may appear more than once, and some may appear not at all. Write the letter indicating the proper bound in the answer box.

A. $\Theta(1)$	B. $\Theta(\log n)$	C. $\Theta(n)$	D. $\Theta(n \log n)$
E. $\Theta(n^2)$	F. $\Theta(n^3)$	G. $\Theta(2^n)$	H. None of the above.

Algorithm Analysis

$$T(n) = \left(\frac{\sqrt{n}}{17}\right) \left(\frac{\sqrt{n}}{4}\right) + \frac{n \log n}{1000}$$

Algorithm Analysis

$$\begin{aligned} T(n) &= \left(\frac{\sqrt{n}}{17}\right) \left(\frac{\sqrt{n}}{4}\right) + \frac{n \log n}{1000} \\ &= \frac{n}{68} + \frac{n \log n}{1000} \\ &= O(n) + O(n \log n) \\ &= O(n \log n) \end{aligned}$$

Algorithm Analysis

$$T(n) = (2^n)(2^n)$$

Algorithm Analysis

$$\begin{aligned}T(n) &= (2^n)(2^n) \\&= 2^{2n} \\&= O(2^{2n})\end{aligned}$$

Algorithm Analysis

$$\begin{aligned}T(n) &= (2^n)(2^n) \\&= 2^{2n} \\&= O(2^{2n})\end{aligned}$$

Note: $O(2^{2n}) \neq O(2^n)$

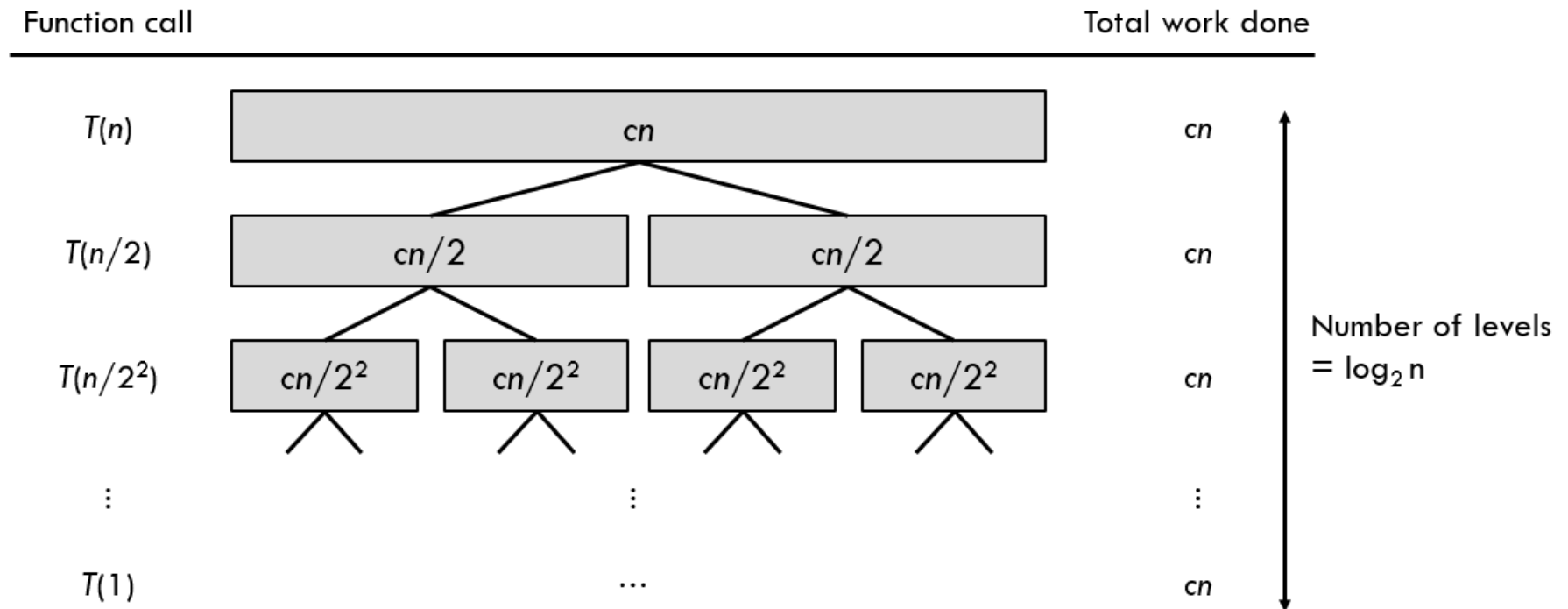
The constant in the exponent matters! $O(2^{2n}) = O((2^n)^2)$

Algorithm Analysis

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$T(1) = 1$$

Algorithm Analysis

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$T(1) = 1$$



Algorithm Analysis

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$T(1) = 1$$

- Given that
 - the total amount of work done in each level sums up to cn , and that
 - the height of the tree is $h = \log_2 n$,
- we can then calculate the total amount of work done across all levels by multiplying the total amount of work done by the height of the tree.

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

Algorithm Analysis

```
public static int loopy(int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < i; j++) {  
            System.out.println("Hello.");  
        }  
    }  
}
```

Algorithm Analysis

- During the i -th iteration of the outer for loop, the inner for loop runs for i iterations. If we add up all the iterations, we get:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \\ &= \sum_{i=0}^{n-1} i \\ &= 0 + 1 + 2 + 3 + \cdots + (n-1) \\ &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

Algorithm Analysis

```
public static int recursiveloopy(int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            System.out.println("Hello.");  
        }  
    }  
  
    if (n <= 2) {  
        return 1;  
    } else if (n % 2 == 0) {  
        return recursiveloopy(n + 1);  
    } else {  
        return recursiveloopy(n - 2);  
    }  
}
```

Algorithm Analysis

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        System.out.println("Hello.");  
    }  
}
```

Algorithm Analysis

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        System.out.println("Hello.");  
    }  
}
```

- During the i -th iteration of the outer for loop, the inner for loop runs for n iterations. If we add up all the iterations, we get:

$$\begin{aligned} T(n) &= n(n) \\ &= n^2 \\ &= O(n^2) \end{aligned}$$

Algorithm Analysis

```
if (n <= 2) {  
    return 1;  
} else if (n % 2 == 0) {  
    return recursiveloop(n + 1);  
} else {  
    return recursiveloop(n - 2);  
}
```

Algorithm Analysis

```
if (n <= 2) {  
    return 1;  
} else if (n % 2 == 0) {  
    return recursivelooopy(n + 1);  
} else {  
    return recursivelooopy(n - 2);  
}
```

- recursivelooopy is called $O(n)$ times
- Overall, we get $O(n) \times O(n^2) = O(n^3)$

ALGORITHM ANALYSIS

(CS2020 2017 QUIZ)

Algorithm Analysis

For each of the following, choose the best (tightest) asymptotic function T from among the given options. Some of the following may appear more than once, and some may appear not at all. **Please write the letter in the blank space beside the question.**

A. $\Theta(1)$	B. $\Theta(\log n)$	C. $\Theta(n)$	D. $\Theta(n \log n)$
E. $\Theta(n^2)$	F. $\Theta(n^3)$	G. $\Theta(2^n)$	H. None of the above.

Algorithm Analysis

$$T(n) = \left(\frac{n^2}{17}\right) \left(\frac{\sqrt{n}}{4}\right) + \frac{n^3}{n-7} + n^2 \log n$$

Algorithm Analysis

$$\begin{aligned} T(n) &= \left(\frac{n^2}{17} \right) \left(\frac{\sqrt{n}}{4} \right) + \frac{n^3}{n-7} + n^2 \log n \\ &= \frac{n^{\frac{5}{2}}}{68} + \frac{n^3}{n-7} + n^2 \log n \\ &= O\left(n^{\frac{5}{2}}\right) + O(n^2) + O(n^2 \log n) \\ &= O\left(n^{\frac{5}{2}}\right) \end{aligned}$$

Algorithm Analysis

The running time of the following code, as a function of n :

```
public static int loopy(int n) {  
    int j = 1;  
    int n2 = n;  
    for (int i = 0; i < n; i++) {  
        n2 *= 5.0/7.0;  
        for (int k = 0; k < n2; k++) {  
            System.out.println("Hello.");  
        }  
    }  
    return j;  
}
```

Algorithm Analysis

The running time of the following code, as a function of n :

```
public static int loopy(int n) {  
    int j = 1;  
    int n2 = n;  
    for (int i = 0; i < n; i++) {  
        n2 *= 5.0/7.0;  
        for (int k = 0; k < n2; k++) {  
            System.out.println("Hello.");  
        }  
    }  
    return j;  
}
```

Outer loop clearly runs in $O(n)$ time

Algorithm Analysis

The running time of the following code, as a function of n :

```
public static int loopy(int n) {  
    int j = 1;  
    int n2 = n;  
    for (int i = 0; i < n; i++) {  
        n2 *= 5.0/7.0;  
        for (int k = 0; k < n2; k++) {  
            System.out.println("Hello.");  
        }  
    }  
    return j;  
}
```

Clearly runs in $O(\log n)$ since $n2$ is being reduced by some factor at each step. At first glance, this seems like a nested loop. However, observe that $n2$ is never reset in the loop! As such, it should be treated independently.

Algorithm Analysis

The running time of the following code, as a function of n :

```
public static int loopy(int n) {  
    int j = 1;  
    int n2 = n;  
    for (int i = 0; i < n; i++) {  
        n2 *= 5.0/7.0;  
        for (int k = 0; k < n2; k++) {  
            System.out.println("Hello.");  
        }  
    }  
    return j;  
}
```

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

Algorithm Analysis

$T(n)$ is the running time of a divide-and-conquer algorithm that divides the input of size n into $n/10$ equal-sized parts and recurses on all of them. It uses $O(n)$ work in dividing/recombining all the parts (and there is no other cost, i.e., no other work done). The base case for the recursion is when the input is less than size 20, which costs $O(1)$.

Algorithm Analysis

Not 10 equal-sized parts!

$T(n)$ is the running time of a divide-and-conquer algorithm that divides the input of size n into $n/10$ equal-sized parts and recurses on all of them. It uses $O(n)$ work in dividing/recombining all the parts (and there is no other cost, i.e., no other work done). The base case for the recursion is when the input is less than size 20, which costs $O(1)$.

- $T(n) = \frac{n}{10} T(10) + O(n), n \geq 20$
- $T(n) = O(1), n < 20$
- Thus, $T(n) = \frac{n}{10} T(10) + O(n) = \frac{n}{10} O(1) + O(n) = O(n) + O(n) = O(n)$

Algorithm Analysis

The running time of the following code, as a function of n :

```
public static int recursiveloopy(int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            System.out.println("Hello.");  
        }  
    }  
  
    if (n <= 2) {  
        return 1;  
    } else if (n % 2 == 0) {  
        return recursiveloopy(n + 1);  
    } else {  
        return recursiveloopy(n - 2);  
    }  
}
```

Algorithm Analysis

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        System.out.println("Hello.");  
    }  
}
```

Algorithm Analysis

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        System.out.println("Hello.");  
    }  
}
```

- During the i -th iteration of the outer for loop, the inner for loop runs for n iterations. If we add up all the iterations, we get:

$$\begin{aligned} T(n) &= n(n) \\ &= n^2 \\ &= O(n^2) \end{aligned}$$

Algorithm Analysis

```
if (n <= 2) {  
    return 1;  
} else if (n % 2 == 0) {  
    return recursiveLoop(n + 1);  
} else {  
    return recursiveLoop(n - 2);  
}
```


Algorithm Analysis

```
if (n <= 2) {  
    return 1;  
} else if (n % 2 == 0) {  
    return recursivelooopy(n + 1);  
} else {  
    return recursivelooopy(n - 2);  
}
```

- recursivelooopy is called $O(n)$ times
- Overall, we get $O(n) \times O(n^2) = O(n^3)$

Algorithm Analysis

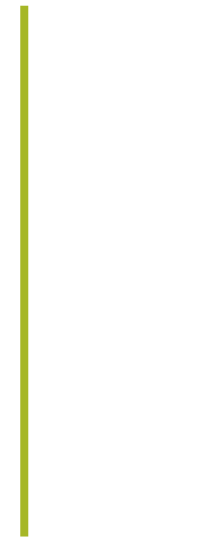
Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Stack

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

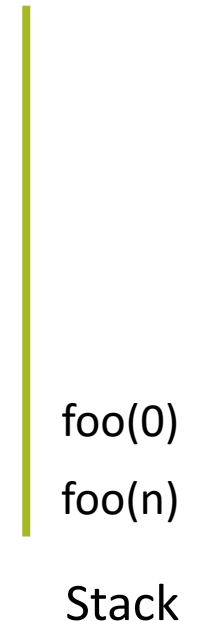
```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

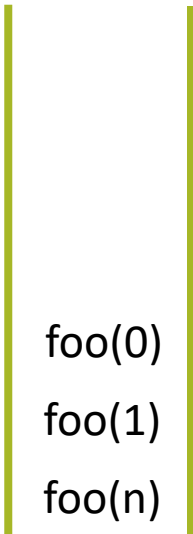
```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



foo(0)
foo(1)
foo(n)

Stack

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

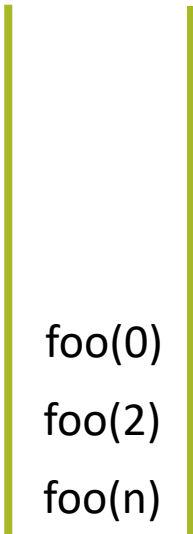
```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



foo(0)
foo(2)
foo(n)

Stack

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

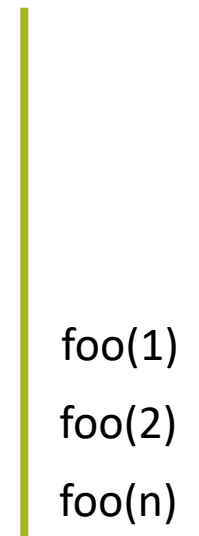
```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



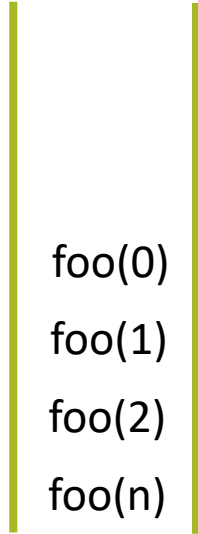
foo(1)
foo(2)
foo(n)

Stack

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



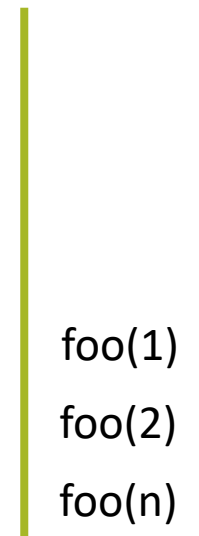
foo(0)
foo(1)
foo(2)
foo(n)

Stack

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



foo(1)
foo(2)
foo(n)

Stack

Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

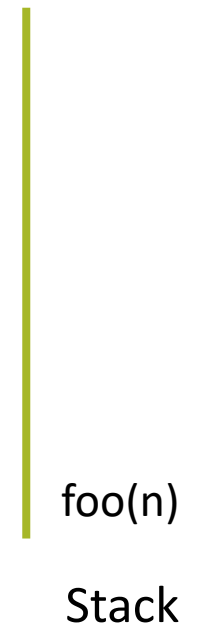
```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



Algorithm Analysis

Let $T(n)$ be the maximum stack depth of the following function, in terms of n .

```
double foo(int n) {  
    int i;  
    double sum;  
    if (n == 0) return 1.0;  
    else {  
        sum = 0.0;  
        for (i = 0; i < n; i++)  
            sum += foo(i);  
        return sum;  
    }  
}
```



So on, so forth... By tracing the code, we can see that the maximum stack depth is $O(n)$

HOW DO THEY WORK?

(CS2040S 2020 MIDTERM)

How Do They Work?

The maximum number of rotations necessary to rebalance an AVL tree containing n elements during the insertion of a new item is:

- A. 0
- B. 1
- C. 2
- D. 3
- E. $\Theta(\log n)$
- F. $\Theta(n)$

How Do They Work?

The maximum number of rotations necessary to rebalance an AVL tree containing n elements during the insertion of a new item is:

- A. 0
- B. 1
- C. **2**
- D. 3
- E. $\Theta(\log n)$
- F. $\Theta(n)$

Rotations

Summary:

If v is out of balance and left heavy:

1. $v.left$ is balanced: $right\text{-}rotate(v)$
2. $v.left$ is left-heavy: $right\text{-}rotate(v)$
3. $v.left$ is right-heavy: $left\text{-}rotate(v.left)$
 $right\text{-}rotate(v)$

If v is out of balance and right heavy:

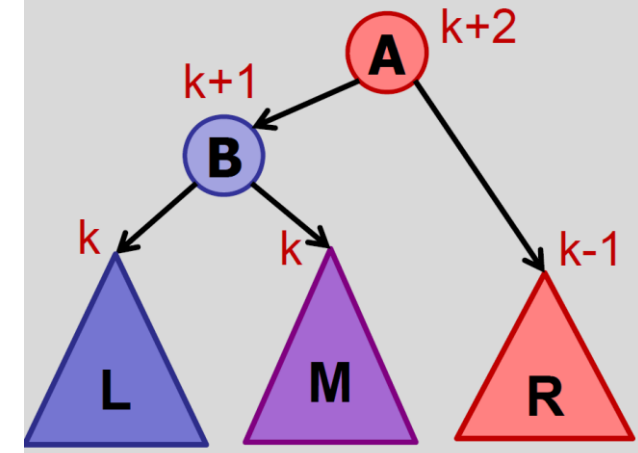
Symmetric three cases....

AVL Tree Insertion

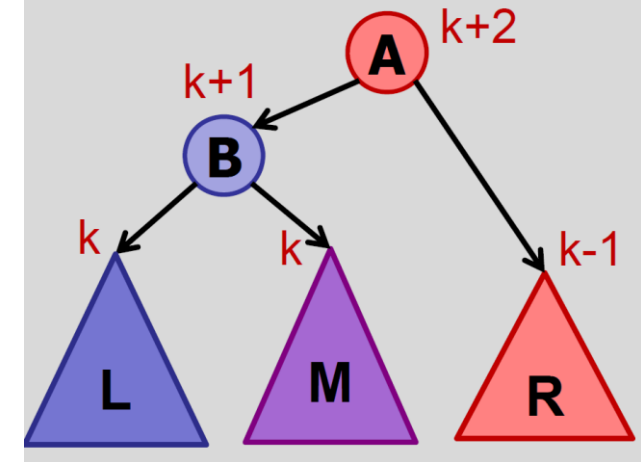
- Only need to perform 2 rotations (when case 3 occurs) in the worst case upon inserting an element into an AVL tree
 - Scenario 1: AVL tree is still balanced after insertion
 - No rotations needed!
 - Scenario 2: AVL tree is unbalanced after insertion
 - Cases 2 & 3: Only 1 rotation needed for Case 2 and 2 rotations for Case 3
 - This is because the height of the sub-tree is reduced by 1 in both cases
 - $1 \text{ (from inserting the node)} - 1 \text{ (from rotations)} = 0$
 - This means that the height of the sub-tree which we rotate remains unchanged after insertion, so no additional rotations needed
 - What about Case 1?

AVL Tree Insertion

- Let's take a look at Case 1
- The inserted node cannot have been inserted in the subtree L
 - If this were the case, the root node of subtree L would have been either height k or $k - 1$ before insertion. The height of node B would then have been $k + 1$ before insertion since the height of subtree M is k . This means that node A would have been unbalanced before insertion which violates our AVL tree invariant. Contradiction!
- The inserted node cannot have been inserted in the subtree M
 - Same reasoning as above, but swap L and M around.



AVL Tree Insertion



- The inserted node cannot have been inserted in the subtree R
 - If this were the case, the root node of subtree R would have been either height $k - 1$ or $k - 2$ before insertion. Either way, this means that node A would have been unbalanced before insertion which violates our AVL tree invariant. Contradiction!
- As such, we have shown that Case 1 can never occur after an insertion
- Thus, the maximum number of rotations necessary to rebalance any AVL tree is 2

How Do They Work?

The maximum number of rotations necessary to rebalance an AVL tree containing n elements during the deletion of a new item is:

- A. 0
- B. 1
- C. 2
- D. 3
- E. $\Theta(\log n)$
- F. $\Theta(n)$

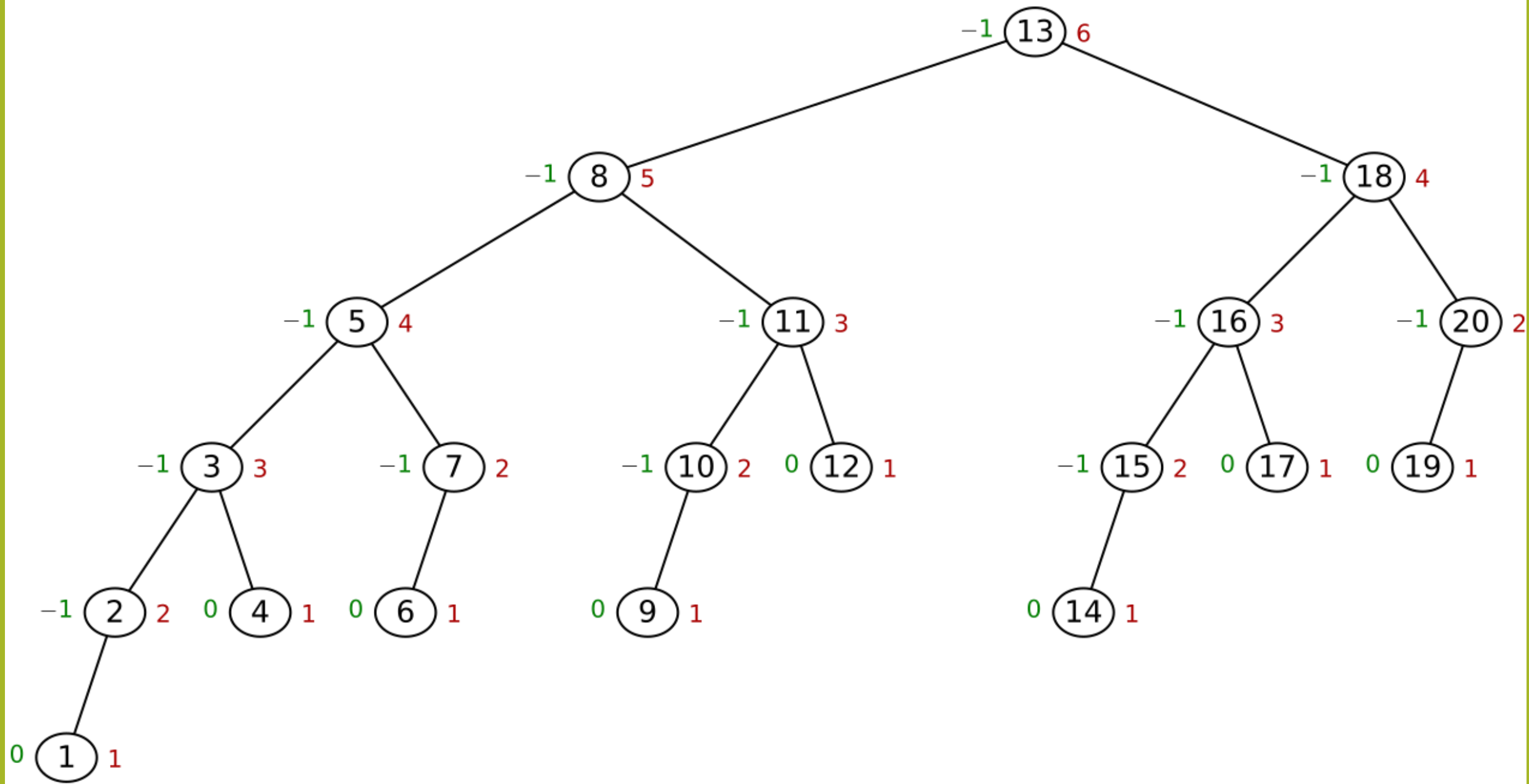
How Do They Work?

The maximum number of rotations necessary to rebalance an AVL tree containing n elements during the deletion of a new item is:

- A. 0
- B. 1
- C. 2
- D. 3
- E. $\Theta(\log n)$
- F. $\Theta(n)$

How Do They Work?

- The maximum number of rotations occurs when we have an AVL tree which is maximally lopsided
 - Otherwise known as a Fibonacci tree
- When the node containing 19 is removed, the node containing 18 needs to be rebalanced, along with every level above it
- Red indicates the height
- Green indicates the balance factor



How Do They Work?

Assume a standard implementation of binary search that stops as soon as it finds the item being queried, as in the following (pseudo)code:

```
int search(int key, int[] A, int low, int high)
    if (low > high) return NOT_FOUND;
    mid = (low+high)/2;
    if (key == A[mid]) return mid;
    else if (key < A[mid]) return search(key, A, low, mid-1);
    else if (key > A[mid]) return search(key, A, mid+1, high);
```

How Do They Work?

We know (from class) that if you are searching for a key x in a sorted array of size n , and if the key appears in the array, then it will be found in $\Theta(\log n)$ time (in the worst case).

What if the key x appears in the (sorted) array $m \geq 1$ times? As a function of n and m , what is the worst-case running time of binary search for key x ?

A. $\Theta(\log(m))$

B. $\Theta\left(\log\left(\frac{n}{m}\right)\right)$

C. $\Theta\left(\frac{\log(n)}{\log(m)}\right)$

D. $\Theta\left(\frac{\log(m)}{\log(n)}\right)$

E. $\Theta\left(\frac{n}{m}\right)$

F. $\Theta(n)$

G. None of the above

How Do They Work?

We know (from class) that if you are searching for a key x in a sorted array of size n , and if the key appears in the array, then it will be found in $\Theta(\log n)$ time (in the worst case).

What if the key x appears in the (sorted) array $m \geq 1$ times? As a function of n and m , what is the worst-case running time of binary search for key x ?

A. $\Theta(\log(m))$

B. $\Theta\left(\log\left(\frac{n}{m}\right)\right)$

C. $\Theta\left(\frac{\log(n)}{\log(m)}\right)$

D. $\Theta\left(\frac{\log(m)}{\log(n)}\right)$

E. $\Theta\left(\frac{n}{m}\right)$

F. $\Theta(n)$

G. None of the above

How Do They Work?

- Intuitively, when you increase the number of keys to be found by a factor of m , you're decreasing the search space by m **when n is constant**
 - For example, if $m = 2$, you get to skip $\log_2 2 = 1$ level of halving
 - If $m = 4$, you get to skip $\log_2 4 = 2$ levels of halving
 - If $m = 8$, you get to skip $\log_2 8 = 3$ levels of halving
- This is because the keys are grouped together in a single contiguous block since they have the same value
- Hence, the time complexity is $\Theta\left(\log\left(\frac{n}{m}\right)\right)$

How Do They Work?

Assume that this array was just partitioned by a QuickSort partitioning algorithm:

[19, 7, 8, 1, 16, 25, 62, 47, 80]

Of the following options, which is a possible pivot?

A. 19

B. 8

C. 16

D. 25

E. 47

F. None of the above

How Do They Work?

Assume that this array was just partitioned by a QuickSort partitioning algorithm:

[19, 7, 8, 1, 16, 25, 62, 47, 80]

Of the following options, which is a possible pivot?

A. 19

B. 8

C. 16

D. 25

E. 47

F. None of the above

How Do They Work?

Assume that this array was just partitioned by a QuickSort partitioning algorithm:

[19, 7, 8, 1, 16, 25, 62, 47, 80]

Of the following options, which is a possible pivot?

A. 19

B. 8

C. 16

D. 25

E. 47

F. None of the above

Everything to the left of
25 is smaller than 25
while everything to the
right is larger

How Do They Work?

Assume that comparing two strings of length k_1 and k_2 takes $\min(k_1, k_2)$ time. The worst-case running time for inserting a string of length L into an AVL tree of size n where all the keys in the tree have length L is:

- A.* $O(1)$
- B.* $O(L)$
- C.* $O(\log n)$
- D.* $O(L \log n)$
- E.* $O(\log n + L)$
- F.* $O(nL)$

How Do They Work?

Assume that comparing two strings of length k_1 and k_2 takes $\min(k_1, k_2)$ time. The worst-case running time for inserting a string of length L into an AVL tree of size n where all the keys in the tree have length L is:

- A.* $O(1)$
- B.* $O(L)$
- C.* $O(\log n)$
- D.* **$O(L \log n)$**
- E.* $O(\log n + L)$
- F.* $O(nL)$

How Do They Work?

- Inserting into an AVL tree takes $O(\log n)$ time
- During insertion, we need to perform a string comparison at each node
 - Since all keys in the AVL tree are of length L , each string comparison takes $O(L)$ time
- By multiplying both time complexities, we get an overall runtime of $O(L \log n)$

EXPLORING PLANET NINE, PART 2

(CS2020 2017 QUIZ 2)

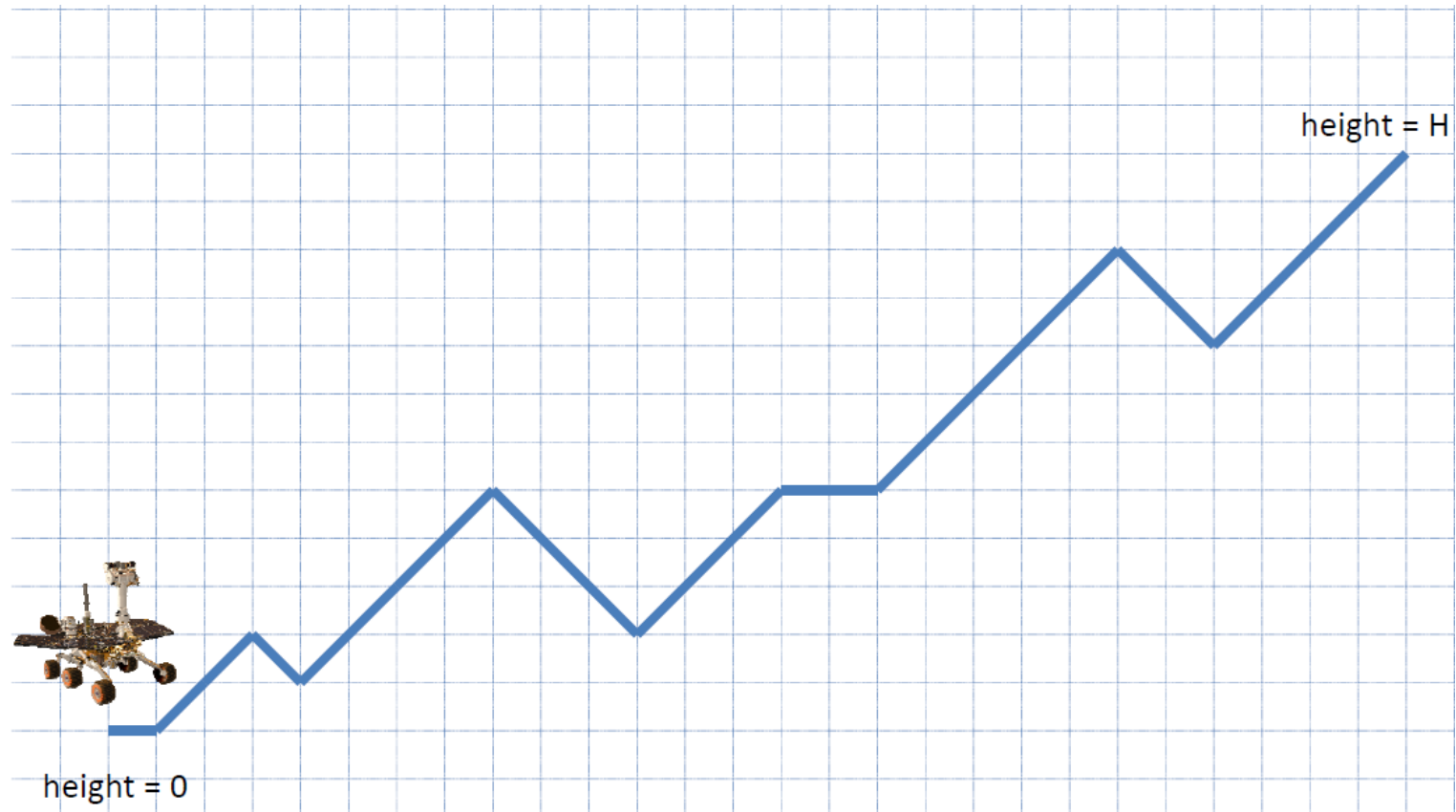
Exploring Planet Nine, Part 2

Our trusty rover has now arrived on Planet 9, and is beginning its exploration. The rover is currently at location “kilometre 0” at the base of a big mountain range. There is a route that continues for $n - 1$ kilometres up through the extra-terrestrial mountains up to a peak at height H .

From our reconnaissance mission, we have created a map of the elevations along this path. That is, we have an array $A[0..n - 1]$ where $A[i]$ is the altitude at kilometre i . All altitudes are integers. $A[0] = 0$ is the base of the mountain and $A[n - 1] = H$ is the top of the mountain. In between, it goes up and down and up and down, the way mountains do. (That is, the mountain is not strictly uphill.) Luckily, there are no cliffs. The changes in altitude are pretty gradual. Each $|A[i] - A[i + 1]| \leq 1$.

Exploring Planet Nine, Part 2

In this example, you see that $A[0] = A[1] = 0$, $A[2] = 1$, $A[3] = 2$, $A[4] = 1$, etc.



Exploring Planet Nine, Part 2

Given the array of the altitudes A , and a target height h , your job is to devise an algorithm to find one outpost i such that $A[i] = h$. In the example above, if the target height were 3, then it might return either kilometres 6, 10, or 12.

Describe your algorithm in at most two sentences.

Exploring Planet Nine, Part 2

Given the array of the altitudes A , and a target height h , your job is to devise an algorithm to find one outpost i such that $A[i] = h$. In the example above, if the target height were 3, then it might return either kilometres 6, 10, or 12.

Describe your algorithm in at most two sentences.

- Binary search while maintaining the invariant that the target is within the current range of heights

Exploring Planet Nine, Part 2

Give pseudocode specifying your algorithm in detail.

Exploring Planet Nine, Part 2

Give pseudocode specifying your algorithm in detail.

```
searchTarget(A, begin, end, target)
    // Base case
    if (begin==end) then return begin

    // Invariant:
    // A[begin] <= target <= A[end]

    int mid = begin + ((end-begin)/2)
    if (A[mid] == target) return mid;
    else if (target < A[mid])
        return searchTarget(A, begin, mid, target)
    else // if (target > A[mid])
        return searchTarget(A, mid+1, end, target)

searchTarget(A, target)
    searchTarget(A, 0, A.size()-1, target)
```

Just a typical
binary search!

Exploring Planet Nine, Part 2

Explain why your algorithm works.

Exploring Planet Nine, Part 2

Explain why your algorithm works.

Throughout the execution, the algorithm maintains the invariant that at all times, $A[begin] \leq target \leq A[end]$. Since the altitudes change only by unit steps of size 1, this ensures (by continuity) that there is some point t between $begin$ and end where $A[t] == target$. This invariant is maintained at each step of the binary search by comparing $A[mid]$ to the target and recursing on one of the two sides. The running time is $O(\log n)$.

HASH IT!

(CS2020 2017 QUIZ 2)

Hash It!

Suppose the following keys are inserted into a hash table in the following order:

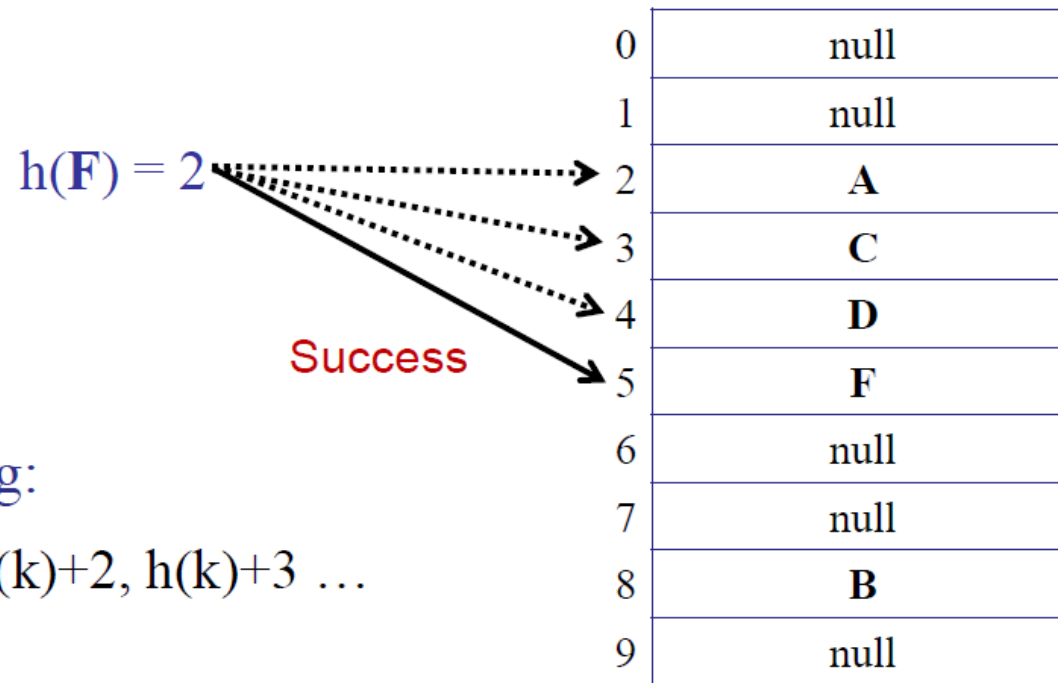
A B C D E F G

The keys are inserted using open addressing with linear probing. Indicate where each key is placed in the resulting array (drawn below with seven slots). Assume that the array size is fixed and does not double or shrink.

Open Addressing

On collision:

Probe a sequence of buckets until you find an empty one.



Linear Probing:

- $h(k)+1, h(k)+2, h(k)+3 \dots$

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	
4	
5	
6	

Hash It!

key	hash(key)
→ A	3
B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	
4	
5	
6	

Hash It!

key	hash(key)
→ A	3
B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
→ 3	
4	
5	
6	

Hash It!

key	hash(key)
→ A	3
B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
6	

Hash It!

key	hash(key)
A	3
→ B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
6	

Hash It!

key	hash(key)
A	3
→ B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
→ 6	

Hash It!

key	hash(key)
A	3
→ B	6
C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
→ C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
→ C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
→ 6	B

Hash It!

key	hash(key)
A	3
B	6
→ C	6
D	4
E	3
F	4
G	5

0	
1	
2	
3	A
4	
5	
→ 6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
→ C	6
D	4
E	3
F	4
G	5

→ 0	
1	
2	
3	A
4	
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
→ C	6
D	4
E	3
F	4
G	5

0	C
1	
2	
3	A
4	
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
→ D	4
E	3
F	4
G	5

0	C
1	
2	
3	A
4	
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
→ D	4
E	3
F	4
G	5

0	C
1	
2	
3	A
→ 4	
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
→ D	4
E	3
F	4
G	5

0	C
1	
2	
3	A
4	D
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
3	A
4	D
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
→ 3	A
4	D
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
→ 3	A
4	D
5	
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
3	A
→ 4	D
5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
3	A
→ 4	D
5	
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
3	A
4	D
→ 5	
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
→ E	3
F	4
G	5

0	C
1	
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
→ 4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
→ 4	D
5	E
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
4	D
→ 5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
4	D
→ 5	E
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
4	D
5	E
→ 6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	
2	
3	A
4	D
5	E
→ 6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

→ 0	C
1	
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

→ 0	C
1	
2	
3	A
4	D
5	E
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
→ 1	
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
→ F	4
G	5

0	C
1	F
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
2	
3	A
4	D
→ 5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
2	
3	A
4	D
→ 5	E
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
2	
3	A
4	D
5	E
→ 6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
2	
3	A
4	D
5	E
→ 6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

→ 0	C
1	F
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

→ 0	C
1	F
2	
3	A
4	D
5	E
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
→ 1	F
2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
→ 1	F
2	
3	A
4	D
5	E
6	B

Collision! Try
next bucket

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
→ 2	
3	A
4	D
5	E
6	B

Hash It!

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
→ G	5

0	C
1	F
2	G
3	A
4	D
5	E
6	B

ANY QUESTIONS?