Sorting

1332 Recitation: Week of June 16th

Announcements

- Exam 1 regrade window will be open until this Friday, June 19th at 11:59PM EDT
- Homework 3 regrade window will be open until this Wednesday, June 17th at 11:59PM EDT
- Homework 6 over sorting algorithms have been released, and it's due June 22nd at 11:55PM EDT.
- 2-4 trees removal (more complicated examples) video coming up!
- Today's recitation is brought to you by the CS 1332 Visualization Tool
  - csvistool.com
- If you need help with sorting, check out the Slides_Supplement folder on Canvas

Properties for Sorting Algorithms

- In-place vs. Out-of-place
  - In-place: does not need additional arrays or other data structures to temporarily store data
  - Out-of-place: does require additional data structures (more than O(1))
  - This is an indicator of a sorting algorithm's space efficiency
- Stability
  - When a sorting algorithm maintains the relative ordering of duplicate items
  - A useful property for sorting items by more than one thing
    - Example: sorting students by last name, then by first name to break ties would only work if the second sort by first name is stable (we would get names listed by first name then last name)

Properties for Sorting Algorithms

- Adaptability
  - An adaptive sorting algorithm can make optimizations if a set of items is sorted or almost sorted, doing less work overall
  - A good question to ask: does this algorithm treat an unsorted array and an already sorted array the same way?
    - If the answer is no, then the algorithm is probably adaptive
    - This implies that the best-case runtime will be different from the worst-case runtime, but the other way around does not necessarily hold true (e.g. QuickSort)

Bubble Sort

- Works by repeatedly swapping adjacent, out-of-order elements through the array, moving (bubbling) the largest element to its proper location
  - Keep track of last swapped index to avoid excessive iterations
  - You could also bubble the smallest element backwards
    - Moving front - back sequentially would be cocktail shaker sort
- Best case: an already sorted array
  - No swaps and the algorithm will terminate after 1 iteration
- Worst case: an array in reverse order
  - There will be (n+(n-1)+(n-2)+...+1) = O(n^2) comparisons

Best	Average	Worst	Place?	Stable?	Adaptive?
O(n)	O(n^2)	O(n^2)	In	Yes	Yes

Selection Sort

- Works by ‘selecting’ the smallest/largest element not known to be in its correct position, and placing it in its correct position (they can choose to use either)
- One iteration of selection sort looks like:
  - Locate the smallest/largest element that has not already been selected
  - Swap this element directly into its correct position, adjacent to the other selected elements
  - Repeat until the entire array is sorted
- After the ith iteration, either first i elements (or last i elements) will be in their correct location

Best	Average	Worst	Place?	Stable?	Adaptive?
O(n^2)	O(n^2)	O(n^2)	In	No	No

Insertion Sort

- Works by dividing the array into 2 parts, a sorted part and an unsorted part, and “inserting” the elements from the unsorted part one by one into their correct locations
- Algorithm
  - One iteration of insertion sort works as follows:
    - Assume the first j elements are sorted
    - Bubble the (j+1)th element backwards to its sorted location (and immediately stop comparing once a swap is not needed)
    - Repeat until the entire array is sorted
  - After the ith iteration, the first i+1 elements of the array will be sorted, but not necessarily in their final locations; we call this *relatively sorted*
- Example

Insertion Sort - Efficiency

- Best case: already sorted array
  - No bubbling; one clean pass
- Worst case: an array in reverse order
  - There will be O(n+(n-1)+(n-2)+...+1) = O(n^2) comparisons

Best	Average	Worst	Place?	Stable?	Adaptive?
O(n)	O(n^2)	O(n^2)	In	Yes	Yes

Merge Sort

- Premise
  - Recursive sorting algorithm; a staple of the divide-and-conquer algorithm paradigm
  - Split up our array in half repeatedly, until we have sub-arrays of size 1, which are considered sorted
  - Then merge them back together in pairs until the entire array is sorted

Merge Sort - Algorithm

- Split
  - Split and copy array into 2 sub-arrays
    - For odd-length arrays, the right subarray should be larger
  - mergeSort(leftArray)
  - mergeSort(rightArray)
    - Base-case is an array of length 1 -- that's already sorted!
- Merge
  - Merge back into the *original array*, not a new one
  - Compare the smallest elements of both sub-arrays that have not already been merged
  - Merge the smaller of the two into the original array by adding to the next available index
    - If there are equal smallest unmerged elements, merge the one from the left to preserve stability
  - Repeat until one of the sub-arrays become empty.
  - Then merge all elements from the remaining subarray

Efficiencies

- Best case and Worst case are the same
  - Merge Sort is non-adaptive
- How much does one split and one merge cost? O(n) + O(n) -> O(2n) -> O(n)
- How many splits/merges do we have to do?
  - How many times can we split n in two halves? O(log n)
- O(n) * O(log n) -> O(n log n)

Best	Average	Worst	Place?	Stable?	Adaptive?
O(nlogn)	O(nlogn)	O(nlogn)	Out	Yes	No

Least Significant Digit (LSD) Radix Sort

- The only non-comparison sort you'll learn in 1332
  - i.e., the items have to inherent values (Comparable is not enough); e.g. ints, chars, Strings
- Premise
  - Sort numbers according to place values (least significant digit first 13**2** to most significant digit **1**32
- Algorithm
  - Create 19 buckets (10 if there are no negatives)
  - Initial passthrough to determine largest magnitude (Integer.MIN_VALUE is an edge case). **k** is the number of digits in the largest magnitude element
  - For i = 0 to k iterations:
    - Put each element into its bucket based on the current place value
      - digit = *num* / (10^i) % 10 + 9 (remove the +9 if there are no negatives)
    - Remove elements from buckets (-9 to +9) in FIFO order, putting back into array left to right
- (please) please PLEASE use a running divisor to get 10^i (do NOT use Math.pow)

Efficiencies/Properties of LSD Radix

Best	Average	Worst	Place?	Stable?	Adaptive?
O(kn)	O(kn)	O(kn)	Out	Yes	No

Common Mistakes in LSD Radix

- Don’t use all buckets in 0^{th} bucket as a terminating condition
  - [102, 307, 201, 202, 504] as a counterexample
- Math.pow() is inefficient and banned (PLEASE DON’T DO THIS)
  - To find 10^i efficiently, use a “running divisor” -- start base = 1, then do base *= 10 at the end of each iteration
  - Also don’t write your own pow() function
- Make sure you use the correct formula to extract the digit and that you have 19 buckets, not 10
  - digit = num / (10^i) % 10 + 9

QuickSelect

Goal of one iteration: [<p | p | >p]

- Goal: Find the k^{th} smallest element in the array
  - How fast can we find the median?
- We don't have time to cover QuickSort :c but it's the exact same algorithm and it's what's on your homework
  - QuickSort recurses on both subarrays; QuickSelect only recurses on one
- Partitioning Algorithm (see Saikrishna slides)
  - Randomly select pivot index: this element swaps with the front of the subarray
  - Scan from (i = lower-bound) to (upper-bound) looking for something greater than pivot
  - Scan from (j = upper-bound) to (lower-bound) looking for something smaller than the pivot
    - If i < j, arr[i] > pivot, arr[j] < pivot, then swap arr[i] with arr[j], then continue scanning
  - If i > j at ANY point, stop scanning and swap pivot with arr[j]
    - if (j == k - 1), then we found it!
    - else if (j > k - 1), then recurse on left side
    - else, recurse on right side

Efficiencies of QuickSelect

Best	Average	Worst
$O(n)$	$O(n)$	$O(n^2)$

Efficiencies/Properties of QuickSort

- Similar idea to MergeSort
  - Partitioning costs O(n), we do O(log n) partitions -> O(n) * O(log n) -> O(n log n)
- Worst-case of both QuickSelect and QuickSort:
  - What if we pick the min/max at every point? “Divide”-and-conquer
  - Basically becomes Selection Sort

Best	Average	Worst	Place?	Stable?	Adaptive?
O(nlogn)	O(nlogn)	$O(n^2)$	in	No	No