CS311

Kazumi Slott

Sorting Algorithms

Your name: Ruben Cerda

Your programmer number: 7

Hours spent: 2

You may type or handwrite your answers. If it is not legible, you will not receive credit.

Submit a hardcopy at the beginning of lecture. Absolutely NO late submissions will be accepted (anything submitted after lecture started). If you have to skip class or cannot come to class on time, you have to email it to me before the class starts (printing deduction -5%). No need to submit it to Canvas.

If you implemented the following algorithms and understood what you coded, you should be able to answer these questions. For every question, give your explanation as well as your answer.

Quick sort

Concise explanation of algorithm:

It picks an element as a pivot and partitions the array to the picked pivot in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot. We then combine them back into a sorted array.

Give and explain Best/average Time complexity:

The best-case time complexity of quicksort is O(n*logn). We can produce this complexity if the pivot value is the middle element in the array, so we can partition all other elements to the pivot. The average case is O(n*logn). This complexity is when all elements are scattered around, meaning not property ascending or descending.

Give and explain Worst Time complexity:

The worst case occurs when the pivot element could be the greatest or smallest element causing the partition process to be unbalanced (unbalanced tree) causing a O(N^2).

Give and explain Space Complexity in terms of auxiliary space

Since we are not storing any temporary arrays to sort our broken down subarrays using recursion the space complexity in terms of auxiliary space will be O(1).

Give and explain Space Complexity in terms of runtime stack

The runtime stack will be O(logN). The size of subarrays will be half the size of the array we wish to sort.

Give and explain In-place?

Quick sort is In-place because we aren't making any temporary arrays to do any left or right subarray comparison, we only do the comparisons using the recursion that goes directly to the runtime stack.

Give and explain Stable?

Quick Sort is not stable because it swaps non-adjacent elements when comparing the left subarray to the right subarray.

Merge Sort

Algorithm:

Merge sort divides the array into halves each recursive call into we eventually hit our base case in which we have 1 element. Then they are combined in a sorted subarray each time it's popped from the run-time stack until it's eventually all subarrays merge back to the original size.

Best Time complexity:

The best case is that we are splitting the array into equal halves giving us a tree height of logN. Then we get to the merging process. It takes N comparisons to merge the left and right subarrays. We get the complexity O(n*logn).

Worst Time complexity

The worst case can still be sid O(N*logN) because it always divides the array into two halves even if it's not an equal half, and takes N comparisons to merge two halves of the left and right subarrays.

Space Complexity in terms of auxiliary space

Since merge sort requires temporary subarrays each recursive call we will need to make an auxiliary space of O(N) to be able to store the merged subarrays.

Space Complexity in terms of runtime stack

The space complexity of merge sort is O(logn) because it requires the runtime stack to grow to the height of a tree. The runtime stack pops each time the two halves are merged giving us a height of a tree.

In-place?

Merge sort is not in place since it requires temporary subarrays to be able to perform comparisons between both halves.

Stable?

It's stable because the order in which two elements will be preserved as long as the condition in which you are comparing is less than or equal to or the other way around greater than or equal to.

Radix Sort

Algorithm

Radix sort is an algorithm that looks at a rightmost digit or character depending on its value and moves our way to the leftmost digit or character. We solely look at the digit and insert it to a bucket or a character and find the base(ascii value) and subtract it to get the difference and insert it to a bucket array.

Best Time complexity

The best time complexity is O(n*m). We apply a number of digits/characters n times to the maximum number of digits/characters m times.

Worst Time complexity

This algorithm has the same complexity for all cases since there's no other way that an array of integers or strings will produce a bad implementation. The time it takes only varies depending on the array of elements and the length of each character/digit. So the complexity is still O(n*m).

Space Complexity for **our implementation**

The space complexity of this implementation is O(n) because we use a linked list to be able to manipulate pointers into merging the slots of each bucket array to a linked list that stores the original values n times.

Space complexity **using counting sort**

The space complexity of this implementation is O(n + m) where m depends on the number of possibilities for each digit and n is the size of the array.

In-place for our implementation?

Our implementation isn't in-place since each cycle of radix sort we are storing the current element into a bucket array with each bucket being a linked list based on the character. We then clear the original linked list and insert again based on the bucket order. So we are constantly taking space to get a final sorted linked list.

In-place using counting sort?

In counting sort, we use an auxiliary amount of space based on the size of the range of data. We place it into temporary storage by checking the rightmost digit/character to the leftmost digit/character for each pass. So counting sort is not in-place.

Stable?

It sorts by sorting one digit at a time. This ensures that numbers that appear before other numbers in the input array will maintain that same order in the final sorted array.

Selection Sort

Algorithm

Selection sort algorithm sorts an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

Best Time complexity

One loop to select an element of Array one by one = O(N). Another loop to compare that element with every other Array element = O(N). Therefore overall complexity = O(N) * O(N) = O(N*N) = O(N^2)

Worst Time complexity

We still need to go through the array n times using the minimum element and another loop that will go n times to compare the minimum elements to the other array elements. So we still get O(N^2)

Space Complexity

We get O(1) as the only extra memory used is for temporary variables while swapping two values in Array.

In-place?

Selection sort is considered in-place because it runs on the same storage or the same run-time stack. It's all done by iterating through loops.

Stable?

It's not stable since we could be swapping elements that aren't adjacent and not preserving the order to accomplish stable sorting

Bubble Sort

Using the bubble sort algorithm in sort.h, count the number of comparisons for the worst case.

Recall how we counted for selection sort in lecture. Check page 3 of my CS211 lecture notes “Lec Notes on Complexity Analysis w/o bubbleSort”

Give your answer here. Show your work:

Algorithm

Bubble sort works by repeatedly swapping the adjacent elements, and in each iteration the biggest number bubbles up to the very end;hence , the name bubble sort.

Best Time complexity:

The best time complexity is when we don't have to swap no adjacent elements so we only iterate the array once and get the time complexity O(N).

Worst Time complexity

The worst case is when we have to swap all adjacent elements that are before the most recent bubbled up element. So, the worst case occurs when an array is reverse sorted giving a time complexity of O(N^2).

Space Complexity

Bubble sort uses only a constant amount of extra space for variables, and doesn't require any additional storage to the runtime stack giving a complexity O(1).

In-place?

Bubble sort is considered in-place because it runs on the same storage or the same run-time stack. It's all done by iterating through loops.

Stable?

Yes, bubble sort is stable since we only swap adjacent elements keeping the order.

Insertion Sort

Count the number of comparisons for the worst case.

Give your answer here. Show your work:

Algorithm:

Insertion sort works by assuming the first element is in the sorted portion of the array, the rest can be considered unsorted. We then compare the last element in the sorted portion to the unsorted portion. If we need to swap then it also compares any previous elements to the sorted portion of the array but as long as we don't go past the first element. If the element were taking in from the unsorted to the sorted doesn't need to swap it's considered in its right place since all elements in this portion are already in a sorted manner.

Best Time complexity

The best time complexity is when the array is already sorted and we don't need to do any comparison from the sorted portion to the unsorted portion of the array. This gives us O(N).

Worst Time complexity

The worst case is when we have to do comparison to every element in the sorted portion of the array. This can be if the array is in descending order. Giving us a time complexity of O(N^2).

Space Complexity

The space complexity is O(1) because we're not using any additional storage for the run-time stack only for temporary variables.

In-place?

Insertion sort is considered in-place because it runs on the same storage or the same run-time stack. It's all done by iterating using loops.

Stable?

Insertion sort is stable because we only swap adjacent elements keeping the order of the elements.

Heap Sort

Algorithm

Heap sort keeps an array look like a tree. It follows that the property that parent nodes are either greater than or equal to or less than or equal to each of its children based on a max/min heap. I'll talk about max heap for this implementation. We then can use this property to make a max heap and in which we are exchanging the largest element to last and decrementing the size at the same time. We then fix any violation to max heap for the element we swapped.; thus, giving us a sorted array at the end for every iteration we do until the size is 0.

Worst Time complexity

The worst case for heap sort might happen when all elements in the array need to be shifted down to the children nodes when we remove an element. Shifting up will be the height of a tree. Still giving us O(NlogN). LogN is the size of the tree and N is the number of steps that include swapping elements at root and the last. Discard the last element from heap by decrementing heap size, and possibility of New root may violate max.

Space Complexity in terms of auxiliary space

The space complexity is O(1) because we don't need any additional storage to sort. It's all done by iterating the array in a tree like manner.

In-place?

Heap sort is in-place since we don't use any additional storage to be able to sort. Array elements are overwritten using temporary variables(swap function).

Stable?

Heap sort is not stable because operations can change the relative order of the array not giving us an order to the elements.

Don't forget to fill out