Algorithms & Data Structures I Week 18 Lecture Note

Notebook:	Algorithms & Data Structures I		
Created:	2020-10-21 4:14 PM	**Updated:**	2021-01-23 6:03 PM
Author:	SUKHJIT MANN		
URL:	https://www.coursera.org/learn/uol-algorithms-and-data-structures-1/lecture/3baSV/9-...		

Cornell Notes	**Topic:** Sorting Data III, Part 2	Course: BSc Computer Science Class: CM1035 Algorithms & Data Structures I [Lecture] Date: January 23, 2021

Essential Question:

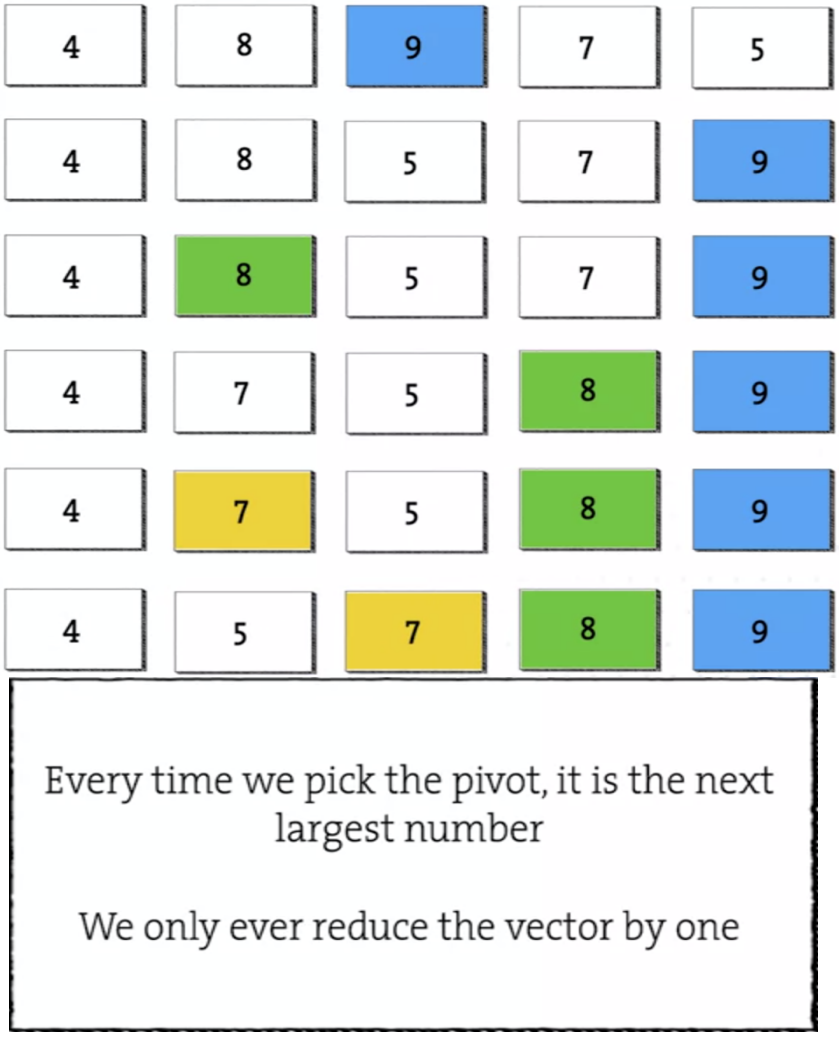
What are divide and conquer algorithms?

Questions/Cues:

- What is the worst-case time complexity for Quicksort?
- What is the average case and the average case time complexity?
- What is the worst-case space complexity of bubble and insertion sort?

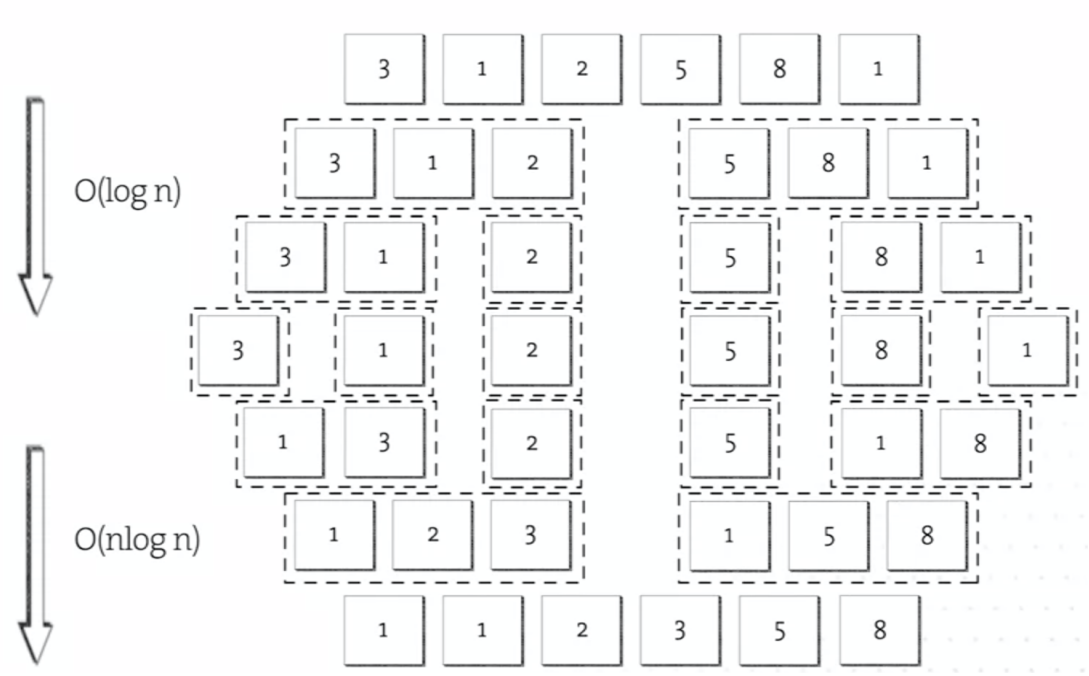
Notes

- The worst-case time complexity for Quicksort is O(n^2) like bubble and insertion sort. This was showcase in the example where the largest value in the vector was the pivot. This meant that the vector was not divided into two smaller vectors, but only one
  - Now each time we look at a new pivot, it's the next largest value in the vector, we will only ever have one smaller vector. This means that we won't be dividing the initial vector into two smaller vectors, but only essentially decreasing the size of the initial vector



- So if we start with n elements in a vector, then we compare n - 1 elements with the pivot, and we are left with a vector of length n - 1 after moving the pivot to the right. In the next step, we now compare n - 2 elements with the new pivot, and then we are left with n - 2 elements. This pattern continues until we are left with a vector of just one element. Therefore, the number of operations scales as the sum of all numbers from 1 to n - 1, so it belongs to the class O(n^2)
- The worst-case input to Quicksort is rare among possible inputs. In order to see the worst-case behaviour, we really need the situation that every choice of pivot stores the next largest value in the vector. Whereas a typically input to the QS algorithm will result in a vector being approximately halved for most, if not all choices of pivot
- As previously stated, for a vector of length n, we have halve it O(log n) times until we get down to vectors of just one element. For each time the vector is halved, we need to compare the elements' values with the value of the pivot and this will take O(n) time. So, the typical time complexity for implementing Quicksort is O(n log n), since for each approximate halving we have O(n) comparisons to make
    - In this context, typical means if we have the set of all possible vectors and we picked one at random, with high probability, it'll have a particular property or set of properties and with low probability, we'll get the worst-case behavior. This is then referred to as the average case and the average case time complexity is the time complexity for a typical instance of a problem.
- Mergesort has a worst-case time complexity of O(n log n), and mergesort has the same average case time complexity as Quicksort which is O(n log n)
- With mergesort, the smaller vectors are created from the initial vector independently of the values stored in the vector. Then in the procedure of merging two vectors of length at most n/2, we'll require O(n) operations. This is also independent of the values stored in the vector, and thus the MS algorithm performs the same for all possible inputs

- In MS, we keep halving the vector until we have nothing but single element vectors. This procedure of halving will take O(log n) operations. However, the merge procedure involves comparing elements of one smaller vector with the elements of another and this procedure will take O(n) operations. Therefore, the worst-case time complexity will be O(n log n)



- Another way to compare QS and MS is in the extra space required to implement them. In the worst-case, mergesort needs O(n) extra space to store all of the elements in the input vector of length n. This is because we create two new vectors of length at most n/2 that then get merged. For this, we need extra space in our RAM model implementation to store the values of the input
- Another way of phrasing this is that we need to store at one point n single element vectors, which'll then be subsequently merged
- For QS, we don't need to create new vectors like MS, everything can be done inside the original vector by storing the values of variables that indicate the left and right indices of the sub-vector in which we're interested. However, Quicksort uses recursion and recall in recursion we call functions within functions and to manage these function calls we use a call stack, which'll clearly take up some space
    - In each function call, we just need to store the indices of the left and right elements, and we can assume that is done using constant space in the RAM model. Finally, in the last call, when these elements are equal, then we return the relevant value in the input vector. So, the size of the stack will depend on the maximum number of recursive function calls, which itself depends on the number of times we can divide the input vector. In the worst-case, this'll be O(n) and so the number of elements in the call stack will be O(n). However, we can improve the space complexity of QS to do better than O(n) with optimization.
        - One such optimization uses something called tail recursion, where for certain function calls, we don't need to push a new stack frame to the call stack. Utilizing these optimizations, it's possible to have a worst-case space complexity of Quicksort being O(log n)
- Remember in Bubblesort, we only needed to keep track of the number of the pass of the algorithm and the indices of the two elements we were comparing, as well as the need to store an extra variable for each swap. In addition to this, we could have kept count of the number of swaps being applied. But each of these numbers requires a constant amount of space in the RAM model, and there is a constant number of variables being stored. Therefore, the worst-case space complexity of bubblesort is constant or O(1)
- For insertion sort, we have the same worst-case space complexity, since we only need to keep track of the indices being compared and the values being shifted. This all occupies a constant amount of memory in the RAM model

Algorithm	Worst-case Time Complexity	Worst-case Space Complexity
Bubble Sort	$O(n^2)$	O(1)
Insertion Sort	$O(n^2)$	O(1)
Quicksort	$O(n^2)$*	O(log n)
Merge Sort	O(nlog n)	O(n)

*O(nlog n) on average

Summary

In this week, we learned about worst-case time complexity for both Quicksort and Mergesort, the averge case time complexity and space complexity for all the sorting algorithms seen so far in the course.