Homework 1: CS 652 Advanced Algorithms

Devin Koehl

Fall Semester 2018

Question 1:

1. $\frac{n^3}{1000} - 100 * n^2 + 50$

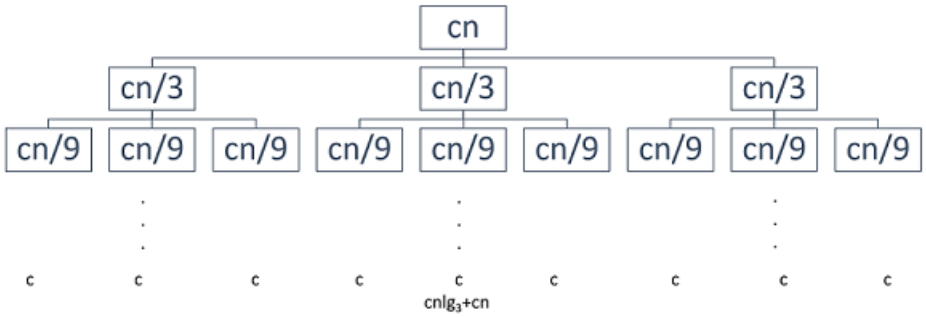
Answer: O(n^3). Big O of a function is the asmptotic upper bound. It is used to bound the worst-case running time of an algorithm.

2. $n^a + n^b (a > b, b > 0)$

Answer: O(n^a). If $a > b$ then this means that we must account for the worst case with a. For example, $n^3 + n^2$ would be O(n^3) but would accomodate n^2 as well because everything from n^2 would be in n^3.

Question 2:

Consider the following modification to the MergeSort algorithm: divide the input array into thirds (rather than halves), recursively sort each third, and finally combine the results using a three-way Merge subroutine. What is the running time of this algorithm as a function of the length n of the input array, ignoring the constant factors and lowest order terms?



Answer:

Invoking a three way split creates a recursion tree that resembles the above. For every split, we will have $\frac{n}{3^i}$ splits. Seeing this, we can form the recurrence relationship T(n) = 3T($\frac{n}{3} + n$). If we use the Master Method in the Cormen text (source: page 95) with a=3, b=3, and c=1 we have $n^{log_b^a}$. Filling in our values we have $n^{log_3^3}$ which yeilds 1. This is case II of the Master Method in the Cormen text which tells us that our time complexity will be O($nlogn$)

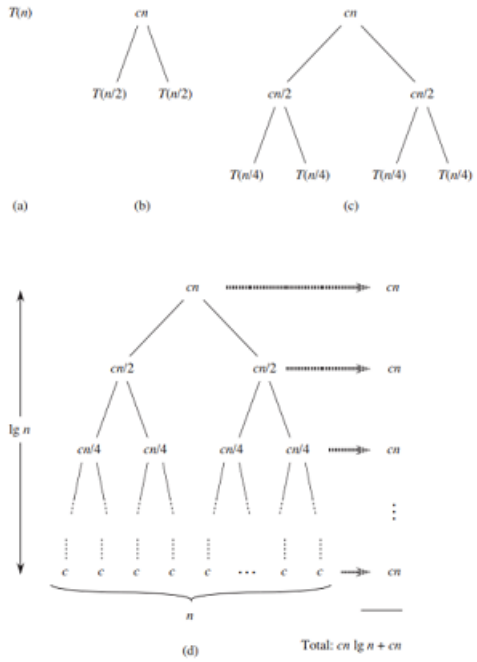
Question 3:

Suppose you are given k sorted arrays, each with n elements, and you want to combine them into a single array of kn elements. Our approach is to use the linear time Merge Subroutine O(n) repeatedly, first merging the first two arrays, then merging the result with the third array, then with the fourth array, and so on until you merge in the k-th and the final input array. What is the running time taken by this successive merging algorithm, as a function of k and n, ignoring constant factors and lower-order terms.

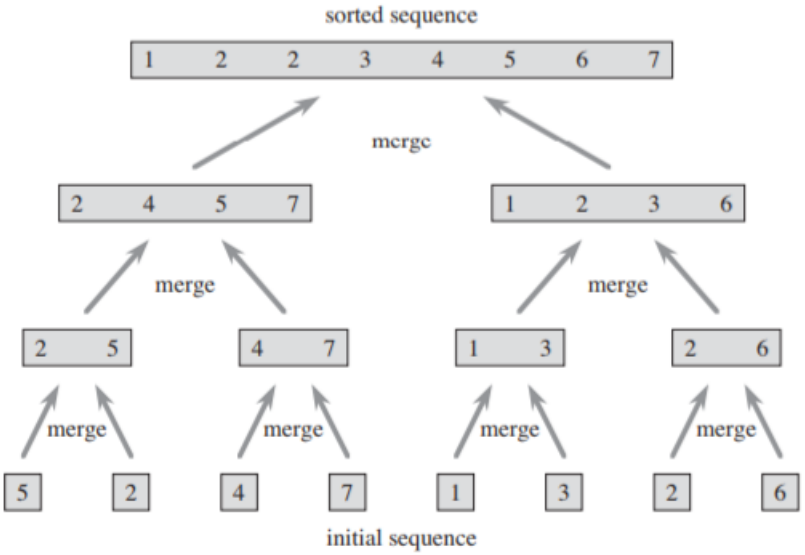
Answer: Let us suppose we have array A = [4 5 6 7 8] and array B = [3 4 5 6 8]. These are two k arrays of size n. If we invoke the merge routine which takes O(n) time, we will have array O(n) on the first array and O(n) on the second array. This will give us (n+n) or 2n. Now we will merge this result with a third array, 2n + n. Now merge this result with another array, 3n + n. Each time we merge these arrays together, we are mutliplying k*k. This yields our time complexity of O(k^2n).

Question 4:
Consider again the problem of merging k sorted length-n arrays into a single sorted length-kn array. Consider the algorithm that first divides the k arrays into $\frac{k}{2}$ pairs of arrays, and use the merge subroutine to combine each pair, resulting in $\frac{k}{2}$ sorted length-2n arrays. The algorithm repeats this step until there is only one length-kn sorted array. What is the running time taken by this successive merging algorithm, as a function of k and n, ignoring constant factors and lower-order terms.

Answer:



Let us again examine the recursion tree that occurs during merge sort above (source: Cormen text page 38) Let us also examine the process of merge sort (source: Cormen text, page 35):

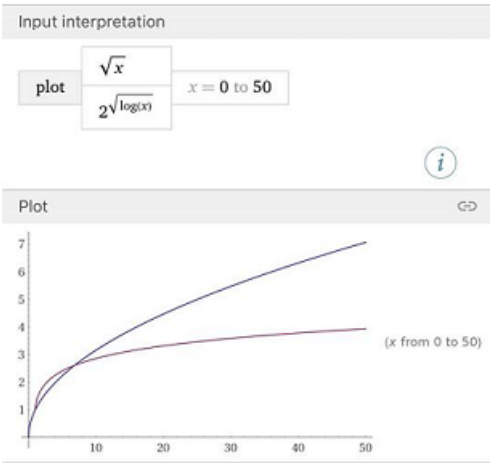


There is a step of the merge sort process that puts our two split arrays back together into one sorted array. The key word in the analysis of this problem is "already sorted". This means invoking the divide and conquer approach to this problem would be similar to a simple merge sort with divide and conquer approach routine. We would be in the step right before the final merge since both our arrays are already sorted. Our merge routine would take O(kn) time. Then our running time would be similar to the divide and conquer approach, yet this time we factor in k sine we had a single array of size kn. The overall running time would be O(klogkn)

Question 5:
Arrange the following functions in the order of increasing growth rate, with g(n) following f(n) in your list if and only if f(n) = O(g(n)).

1. $\sqrt{n}$
2. 10^n
3. $n^{1.5}$
4. $2^{\sqrt{log_2 n}}$
5. $n^{\frac{5}{3}}$

Answer: 4,1,3,5,2. Explanation: Upon inspection, it is immediately clear that 10^n will increase the fatest. $n^{\frac{5}{3}}$ is greater than $n^{1.5}$ because $\frac{5}{3}$ is 1.7. The two that required plotting were $2^{\sqrt{log_2 n}}$ and $\sqrt{n}$. At first glance, I would assume $\sqrt{n}$ would be smaller, then I graphed this in Wolphram Alpha.



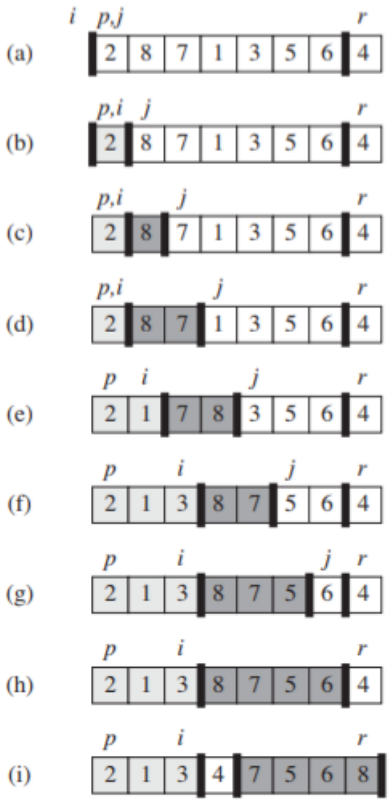
For small values of n, the square root increases slower than the log. Yet, as we increase n over time it is clear that the log is the winner.

Question 6:

Recall the partition subroutine employed by the Quicksort algorithm. You are told that the following array has just been partitioned around some pivot element: 3,1,2,4,5,8,7,6,9. Which of the elements could have been the pivot element? (List all that apply; there could be more than one possibly).

Answer:

The quick sort algorithm invovles the focus on a pivot point. The pivot partitions the array into two parts, everything to the left of the pivot is less than the pivot, everything to the right of the pivot is greater than the pivot. From Cormen text, page 172. Everything in the light shaded area represents the left array and everything in the dark array represents the right side of the array.



[3 1 2 4 5 8 7 6 9]

Examining the above array means that there are multpile potential pivot points. Three could not be the pivot because there are numbers less than three to the right of three. One could not be a pivot because the element to the left, 3, is greater than 1. Two could not be a pivot beause three is to the left and greater. Four could have been a pivot because everything to the left of four is less than four and everything to the right of four is greater. Five and eight could also be a potential pivot for this same reason. Seven and six could not be because of 9. Nine could be because everything to the left is less. So overall we have potential pivots: Four, 5, 8, 9.

Question 7:

Insertion sort can be expressed as a recursive procedure as follows: In order to sort A[1...n], we recursively sort A[1...n-1] and then insert A[n] into the sorted array A[1.. n-1]. Write a recurrence (T(n)) as a function of the input size n for

the running time of this recursive version of insertion sort.

Answer:

Let us take the array [5 2 9 7 14 8] and implement the recursive sort mentioned in the text. Let us write psuedocode for what is happening:

recursiveInsertionSort(arr[ ],n)

if n >= 1 then return;

recursiveInsertionSort(arr[ ],n-1)

It would sort [5 2 9 7 14] then insert the 6th node in the sorted array. It would then operate on [5 2 9 7] and insert the 5th node in the sorted array. It would continue on in this way until it meets the base case of just 5. Now the function will recursively operate and insert the element into the array and put it in the correct place. Using this algorithm we can write the recurrence relationship as

$$T(n) = \begin{cases} O(1) & \text{if } n = 1. \\ T(n-1) + O(n) & if n > 1. \end{cases} \quad (1)$$

The first case in the recurrence relationship is when we meet our base case. Next, the (n-1) is the recursive call to sort the n-1 elements in the array. Lastly, we have the O(n) to insert the element into the correct position.

Question 8:
For input strings s1 and s2, write a function to determine if s1 is an anagram of s2.

Answer: See Python Programming Files

Question 9:
Determine for a given string, if it is a palindrome. Consider only alphanumeric characters (ignore others) and ignore cases. Note, empty string is a valid palindrome

Answer: See Python Programming Files

Question 10:
An array A is always increasing if for all i <= j, A[i] <= A[j]. An array A is always decreasing if for all i <= j, A[i] >= A[j]. An array is monotonic if it is either always increasing or or always decreasing. Return true if and only if the given array is monotonic.

Answer: See Python Programming Files