Nicholas Cheng
Tuesday 9/28/2021

HW 4

Q1.

a) $T(n) = 8T\left(\frac{n}{4}\right) + O(n)$

Master Theorem:

$$a = 8, b = 4, d = 1$$

$$\log_4 8 > 1\ Therefore \log_4 8 > d$$

Therefore, $O\left(n^{\log_b a}\right)$

$T(n) = 8T\left(\frac{n}{4}\right) + O(n)$ => $O\left(n^{\log_4 8}\right) = O(n^{1.5})$

b) $T(n) = 2T\left(\frac{n}{4}\right) + O\left(n^{\frac{1}{2}}\right)$

Master Theorem:

$$a = 2, b = 4, d = \frac{1}{2}$$

$$\log_4 2 = \frac{1}{2}, Therefore \log_4 2 = d$$

Therefore, $O(n^d \log(n))$

$T(n) = 2T\left(\frac{n}{4}\right) + O(n^{\frac{1}{2}})$ => $O\left(n^{\frac{1}{2}} \log(n)\right)$

c) $T(n) = T(n-4) + O(n^2)$

$T(n-4) = T(n-8) + O((n-4)^2)$

$$T(n) = n^2 + (n-4)^2 + (n-8)^2 + (n-12)^2 + \cdots$$

$$\sum_{k=0}^{for\ integers\ of\ k\ until\ k=n/4} (n-4k)^2$$

Therefore, in the long run the above summation can be bounded by $\frac{n}{4} * n^2$

$$Therefore, \quad O(n^2 * n) = O(n^3)$$

Nicholas Cheng
Tuesday 9/28/2021

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

$$O(n) + O\left(n^{\frac{1}{2}}\right) + O\left(n^{\frac{1}{4}}\right) + T\left(n^{\frac{1}{8}}\right) + \cdots$$

Intuitively, the solution to the above recurrence is O(n), attempt to show it.

$$n = 2^m, T(2^m) = T\left(2^{m/2}\right) + O(2^m)$$

$$n = 2^m$$

$$\log_2 n = m$$

$$m = \log_2 n$$

$$S(m) = T(2^m)$$

$$S(m/2) = T(2^{m/2})$$

$$T(2^m) = T\left(2^{m/2}\right) + O(2^m) <=> S(m) = S\left(\frac{m}{2}\right) + O(2^m)$$

$$S(m) = S\left(\frac{m}{2}\right) + O(2^m)$$

$$S(m/2) = S\left(\frac{m}{4}\right) + O\left(2^{m/2}\right)$$

$$S\left(\frac{m}{4}\right) = S\left(\frac{m}{8}\right) + O\left(2^{m/4}\right)$$

$$S(m) = 2^m + 2^{m/2} + 2^{m/4} + \cdots + 2^{m/2^{\log_2 m}}$$

$$\sum_{i=0}^{\log_2 m} 2^{m/2^i}$$

$$S(m) = 2^m + 2^{m/2} + 2^{m/4} \ldots < 2^m + 2^{m-1} + 2^{m-2} \ldots$$

$$\sum_{i=0}^{\log_2 m} 2^{m/2^i} < \sum_{i=0}^{\log_2 m} 2^{m-i}$$

$$S(m) = O(2^m * \log_2 m)$$

$$T(n) = O(2^m * \log_2 m) = O\left(2^{\log_2 n} * \log_2 \log_2 n\right)$$

$$T(n) = O(n * \log_2 \log_2 n)$$

However, the solution is intuitively $O(n)$, therefore the above is not tight enough, we can approach this differently by using an overestimation that we know how to evaluate:

$$\sum_{i=0}^{\log_2 m} 2^{m/2^i} < \sum_{i=0}^{\log_2 m} 2^{m-i} = \sum_{i=0}^{\log_2 m} 2^m * \left(\frac{1}{2}\right)^i = 2^m * \sum_{i=0}^{\log_2 m} \left(\frac{1}{2}\right)^i$$

Nicholas Cheng
Tuesday 9/28/2021

$$2^m * \sum_{i=0}^{\log_2 m} \left(\frac{1}{2}\right)^i = Geometric\ Series = 2^m\left(\frac{1-\left(\frac{1}{2}\right)^{\log_2 m}}{1-\frac{1}{2}}\right)$$

$$= 2^m * \frac{\left(1-\frac{1^{\log_2 m}}{2^{\log_2 m}}\right)}{\frac{1}{2}}$$

$$= 2^m * (2 * (1-\frac{1}{m}))$$

$$= 2 * 2^m - \frac{2 * 2^m}{m}$$

Therefore, $S(m)$ is upper bounded by:

$$S(m) = T(2^m) = O(2^{m+1} - \frac{2^{m+1}}{m})$$

$$since\ m = \log_2 n$$

$$T(2^m) = 2 * 2^m - \frac{2 * 2^m}{m}$$

$$T(n) = 2 * n - \frac{2 * n}{\log_2 n}$$

$$T(n) = O\left(n - \frac{n}{\log_2 n}\right)$$

Therefore, $T(n) = O(n)$

Nicholas Cheng
Tuesday 9/28/2021

Q2.

Array A of length n and R is the range of values within A

```
Rcounts = []
For i in range R+1
        Rcounts.append(0)
Min = A[0]
For i in A
        If i < min
                min = i
For i in A
        Rcounts[i-min] += 1
numberValue = []
countValue = []
For i in range R+1
        If(Rcounts[i] > 0)
                numberValue.append(i+1)
                countValue.append(Rcounts[i])

setsCurrent = 0
For i in range n
        If(countValue[setsCurrent] < 1)
                setsCurrent += 1
        A[i] = numberValue[setsCurrent]
        countValue[setsCurrent] -= 1
```

//Creating an array to store the amount of times a specific value in the range R appears in A

//Finding the minimum value within A

//Using the minimum to determine the value in A's corresponding Rcounts index and then incrementing its occurrence data by 1 for every value in A

//Stores a number in the range
//Stores the amount of times it appears

//Removing garbage values from Rcounts and assigning the data to new clean arrays in order to retain linear time when reconstructing the array

//Finally, we reconstruct the sorted list from the new clean arrays linearly.

Nicholas Cheng
Tuesday 9/28/2021

Q3.

a.

Every comparison removes 1 element. The algorithm is just a structured way to call comparisons that each remove 1 element.

At the end of any call to the algorithm only 1 element will remain. Thus we eliminated $n - 1$ elements

Therefore, we have called $n - 1$ comparisons for every case including worst case.

b.

A simple way: search for the minimum value, and then remove that minimum value from the original list and run it again.

Better way that is in part c:

Since the algorithm always results in only the minimum value, the second minimum value must have been eliminated, but the only number that can eliminate it is the smallest value.

So if you can keep track of what numbers were compared with the smallest value then one of those numbers will be the second smallest value.

Therefore, if you can store or keep track of a list/array/dictionary for the numbers that were compared with the smallest number, then you can run the algorithm again on this smaller list that we have proven to contain the second smallest number as its minimum value to get the second smallest number.

c.

As an extension of the idea presented in part b, where we proved the list of numbers (logList) that were compared with the smallest number (k_1) contain the second smallest number (k_2) and thus when you run logList through the

algorithm we obtain k_2, the size logList would be equal to the amount of times k_1 was compared.

Since k_1 can only be compared a maximum of once per series of comparisons after which the size of the list shrinks to ceil(n/2), k_1 will only be compared a maximum of $ceil(\log_2 n)$ times, meaning the maximum size logList would be $ceil(\log_2 n)$

Therefore, if we keep track of all the numbers that were compared with k_1 during the original comparison process, which would take $(n-1)$ comparisons, we can obtain logList by using for example:

a dictionary where every distinct integer in the original list corresponds to a list of values that was compared with that distinct integer

A list of length 0 arrays for every distinct integer such that other integers compared with it are appended to the corresponding arrays, but if the integer is eliminated its corresponding array is deallocated, such that the list of numbers that were compared with the last remaining value k_1 is the last remaining list

Then we can run logList through the algorithm, which would take $(\lceil \log_2 n \rceil - 1)$ comparisons, to obtain k_2.

Therefore, considering the algorithm's worst case time presented in part a. and the worst case size of logList, by using the way presented in this answer we can find k_2 in a worst case of

$$n - 1 + \lceil \log_2 n \rceil - 1$$

$$= n + \lceil \log_2 n \rceil - 2 \quad comparisons$$