Intro to Algorithms HW 4

Greg Stewart

February 16, 2018

Q1 Compute the tight big-O bounds for the following:

Half of these are in the form $T(n) = aT(n/b) + O(n^d)$, where a, b, d are constant, so we can use one of the Master Theorem formulas discussed in lecture to find the order notation.

(a) $T(n) = 8T(n/4) + O(n)$.

Here, $a = 8$ and $b^d = 4$, so $a > b^d$, which means we have the form

$$T(n) = O(n^{\log_b(a)})$$

and in this case we have

$$T(n) = O(n^{\log_4(8)})$$

(b) $T(n) = 2T(n/4) + O(\sqrt{n})$.

$a = 2$ and $b^d = 4^{1/2} = 2$, so $a = b^d$, and we use the form

$$T(n) = O(n^d \cdot \log_b n)$$

which in this case is

$$T(n) = O(\sqrt{n} \cdot \log_4 n)$$

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

This is not divide-and-conquer in the same way the first two parts are, so the Master Theorem does not apply. However, there is a limit on the number of recurrences. Writing out the first few recurrences,

$$\begin{aligned} T(n) &= T(n-4) + O(n) \\ &= [T(n-8) + O(n)] + O(n) \\ &= \{[T(n-12) + O(n)] + O(n)\} + O(n) \end{aligned}$$

And so on. What's obvious here is that, until $n - k = 0$, each recurrence takes $O(n)$ time. As each recurrence takes off 4 from n, the depth of the recursion is $\frac{n}{4}$, so we find

$$T(n) = \frac{n}{4} \cdot O(n) = O(n^2)$$

(d) $T(n) = T(\sqrt{n}) + O(n)$

This can be solved similarly to the previous part. Each recurrence requires $O(n)$ time and reduces n by a square root, so we can write

$$T(n) = O(n) \cdot \sum_{i=1}^{k} n^{\frac{1}{2^i}}$$

where a ceiling can be put on at $n^{\frac{1}{2^k}} = 2$. Solving this for k,

$$\begin{aligned} n^{\frac{1}{2^k}} &= 2 \\ \frac{1}{2^k} \cdot \log n &= 1 \\ \log n &= 2^k \\ \log\log n &= k \end{aligned}$$

So in the end for the summation we have

$$T(n) = O(n) \cdot \log\log n = O(n \log\log n)$$

Q2 Give an $O(n + R)$ algorithm to sort all the integers in an array A.

```
function sort(A):
  input: array A of integers
  output: sorted array (same array)!

  minimum = 2**40
  maximum = -2**40

  for i in 0..length(A)
    if A(i) < minimum
      minimum = A(i)
    if A(i) > maximum
      maximum = A(i)

  // make a new array for the counts of values
  counts_arr = new array(maximum - minimum + 1)

  for i in 0..length(A)
    counts_arr(A(i) - minimum) += 1

  k = 0
  for i in max..min
    for j in 0..(counts_arr(i - minimum))
      A(k) = i
      k += 1
```

Q3

(a) How many comparisons are done in the above algorithm in the worst case?

There are $n/2$ comparisons for each recurrence, which we can express in summation notation all the way down as

$$\sum_{i=1}^{k} \frac{n}{2^i} = n \sum_{i=1}^{k} (\frac{1}{2})^i$$

where in the final step,

$$\frac{n}{2^k} = \frac{n}{n/2}$$

So we can find k by solving

$$\begin{aligned} 2^k &= \frac{n}{2} \\ 2^{k+1} &= n \\ k+1 &= \log n \\ k &= \log n - 1 \end{aligned}$$

Anyway, as the aforementioned summation is a geometric series, we simply use the formula for the sum of a geometric series to obtain

$$\text{comparisons} = n \cdot \frac{1}{1 - 1/2} = 2n$$

There are $2n$ comparisons by this formula.

However, in reality there are n comparisons, as this is a discrete application of the summation and we start at $n/2$ and end at $n/(n/2)$. Think about this as such:

$$comparisons = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{n/2}$$

This approaches n as a sum, so there are n comparisons in the worst case.

(b) Show how to modify/extend this method to find the second smallest element

To find the second smallest element in the array, we must keep track of all the elements in a tree as comparisons are made. We first make the tree in the same fashion as the original algorithm, obtaining the minimum element. Create a variable to store the second minimum element. Then, move back through the tree toward the leaf nodes (the original array) and compare the minimum element to the root of each subtree you encounter. If the root of the subtree is less than the currently stored second minimum element, update the second minimum element, then discard the subtree with the larger root and continue down the one with the smaller root. Do this until you reach the leaf nodes and you will have the second minimum element.

(c) Prove we find the second smallest element in $n + \log n - 2$ comparisons in the worst case

The height of the tree created for part (b) is $\log n$, which is where that term comes from. From (a), we know that there are n comparisons to find the smallest element, but realistically, there is no need to do final 2 comparisons, as we can just apply the formula from (b) to the existing subtrees to figure out which subtree to go down, without having a single minimum root for the tree. Thus, we can subtract 2 from the total number of comparisons to make the tree, giving $n - 2$. Adding these together, we get

$$n + \log n - 2$$