5800 Recurrences Exercises and Solutions

Alan Jamieson

May 2022

Instructions: This document lists several exercises similar to those that you would be tackling as in-class exercises as well as those that are similar to what you might expect on a homework assignment. My suggestion is to work through the exercises yourself or with peers (preferred) and then compare it against my worked solutions. As you are working through these if you have questions, feel free to reach out via Discord (preferred), Teams, or email, or book an office hour appointment with me or one of the TAs.

All exercises from Dasgupta et al. *Algorithms*, Goddard, Skiena *The Algorithm Design Manual*, or my own brain.

1 Exercises

1. Determine the Big-O of the following recurrences:
   (a) $T(n) = 3T(n/2) + n^2$
   (b) $T(n) = 4T(n/2) + n^2$
   (c) $T(n) = 16T(n/4) + n$
   (d) $T(n) = T(n/2) + 2^n$ a special exercise!
   (e) $T(n) = 9T(n/4) + n^2$
   (f) $T(n) = 2T(n/2) + n(logn)^2$ another special one!
2. Give the Big-O analysis of the following pseudocode function:

```
//Given an array of integers, A
findMax(A):
    if A.length <= 1: return A[0]
    leftmax = findMax(left half of A)
    rightmax = findMax(right half of A)
    if leftmax >= rightmax
        return leftmax
    else
        return rightmax
```

3. A classic recursive problem to give beginner programmers is calculating the factorial of an integer n. The pseudocode typically looks something like:

```
fact(n):
  if n < 2: return n
  return fact(n-1) * n
```

   However, this pseudocode doesn't quite fit the format for the Master Theorem. Come up with the recurrence to describe this algorithm, and determine the Big-O of that recurrence. Detail reasoning.

4. CHALLENGE PROBLEM: that won't have a solution detailed below - the classic version of generating the Fibonacci sequence is the horrendously inefficient:

```
fib(n):
  if n <= 2: return n
  return fib(n-1) + fib(n-2)
```

   For this function, the recurrence looks something like $T(n) = T(n-1) + T(n-2) + O(1)$. How would you analyze this recurrence to get the Big-O of this version of the Fibonacci generating function?

2 Solutions

1. (a) Apply the Master Theorem here - $a = 3, b = 2, d = 2$. $log_2 3$ is approximately 1.6 which is less than 2, so the first condition of the Master Theorem applies, meaning $T(n) = O(n^d) = O(n^2)$.

   (b) Master Theorem (MT) again - $a = 4, b = 2, d = 2$. $log_2 4$ is 2, which is equal to d. The second condition holds, meaning $T(n) = O(n^d logn) = O(n^2 logn)$.

   (c) Another MT application - $a = 16, b = 4, d = 1$. $log_4 16$ is 4 which is greater than d. The third condition holds, meaning $T(n) = O(n^{log_b a}) = O(n^2)$

   (d) This one is a little different. The Master Theorem doesn't apply here (at least not in the traditional sense - see a note later), but we can do some thinking to figure out what's going on. Notice that the recursive part of this problem, $T(n/2)$ ends up representing a log factor $log_2 n$ because it divides our data set by 2 each time. But also notice that this is something that grows significantly less than the other part of our recurrence 2^n. Intuitively, this will tell us that the whole thing is $O(2^n)$.

   Now, there is a more general version of the Master Theorem can be used in this case. A nice writeup of that version can be found here: https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/.

   (e) Apply the MT. $a = 9, b = 4, d = 2$. $log_4 9 \approx 1.6$, which is less than d. The first condition holds, meaning $T(n) = O(n^2)$.

   (f) The question here is can we apply the MT as we know it? Turns out, yes! Remember that the MT wants recurrence equations in the form $T(n) = aT(n/b) + O(n^d)$. We know that we can bound the second half of our recurrence $(n(logn)^2)$ by n^3 since each $logn$ is bounded by n, giving us a $O(n^3)$ for that portion of our recurrence. With that change, we end up with a slightly adjusted recurrence, swapping out our $n(logn)^2$ with $O(n^3)$, leaving us $T(n) = 2T(n/2) + O(n^3)$, which we can apply the MT to easily. $a = 2, b = 2, d = 3$. $log_2 2 = 1$ which is less than d, so the first condition holds and $T(n) = O(n^3)$.

2. The first thing we need to do is to decompose this into the appropriate parts. Our base case here doesn't really cost us anything extra based on the size of the input (A in this case), so that's O(1). Same with our comparison at the end (the if-then-else block). The "work" here is based on the two recursive calls in the middle. Notice that each splits our input in half, so that gives an $n/2$ to put in the recurrence. Since there are two recursive calls, we end up with something like $T(n) = 2T(n/2)+O(1)+O(1)$, which shrinks to $T(n) = 2T(n/2) + O(1)$.

Taking that recurrence, we can apply the Master Theorem - $a = 2, b = 2, d = 0$ (NOTE: $n^0 = 1$ which is why $d = 0$).

3. The recurrence for this problem is $T(n) = T(n-1) + O(1)$. The $T(n-1)$ comes directly from the recursive call as part of the return, and the $O(1)$ is for the math operations, return, and comparison (all abstracted to constant time functions). Of course, there is a concern that the larger that the factorial result becomes, this $O(1)$ will eventually land as $O(n)$ in some implementations due to the time requirements for multiplication.

   The Big-O of this function should be $O(n)$. Notice that the $T(n-1)$ basically steps through our input n, meaning that the various math and comparison operations will be run n times.

4. No solution provided because I'm a mean person.