Part 1

For each of the following two recurrences, construct a recursion tree and use it to solve the recurrence. Your solution should include
• a sketch of the tree that clearly shows the branching factor of each node, the depth of the base-case nodes, and work/node for the first 2 levels of the tree;
• a chart similar to examples in lecture and studio with columns for the depth, number of nodes, work per node, and work per level at each level of the tree;
• a summation giving the total work of the recurrence; and
• a closed-form asymptotic solution to the recurrence.

You may assume for simplicity that the input size n is a power of b, the denominator of the size in the recursive term, and that T(1) = d for some constant d.

1. T(n) = 3T(n/2) + n^2; T(1) = d

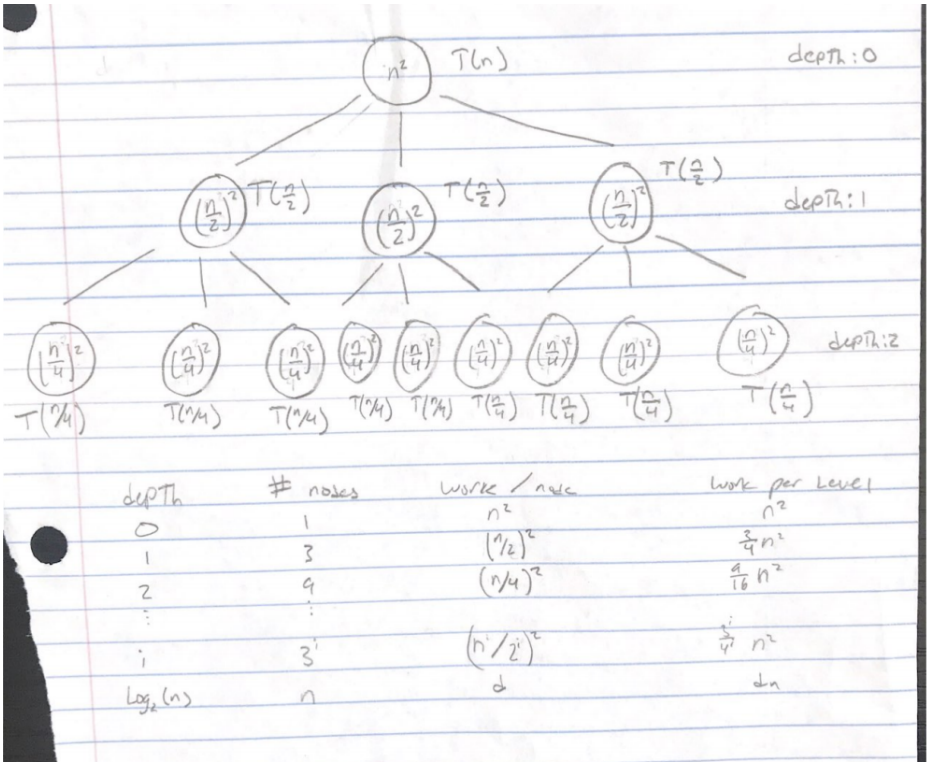

Figure 1:

Total Work: $\sum_{i=0}^{Log(n)-1}(n^2) + dn = n^2 log(n) + dn$
$T(n) = \Theta(n^2 log(n))$

2. T(n) = 4T(n/2) + n^2; T(1) = d

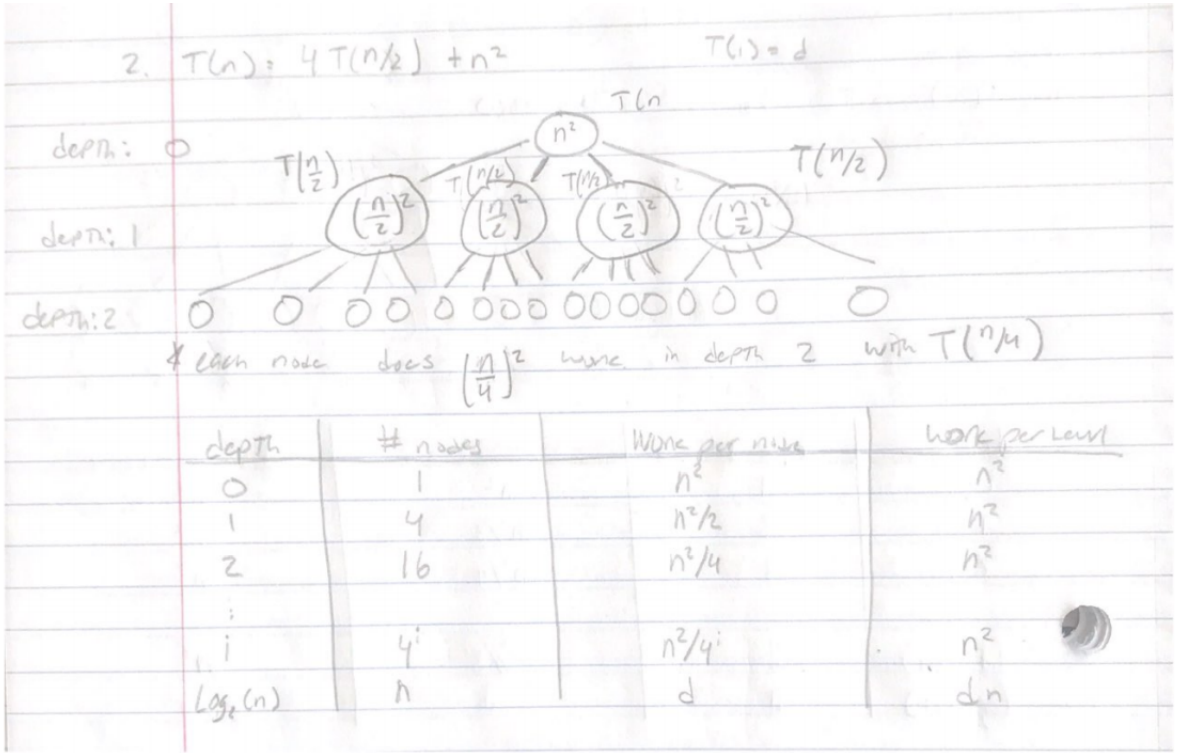

Figure 2:

Total Work: $\sum_{i=0}^{Log(n)-1}(n^2) + dn = n^2 log(n) + dn$
$T(n) = \Theta(n^2 log(n))$

3. T(n) = T(n/3) + log(n)
$a = 1; b = 3; f(n) = log(n), so C_crit = 0, k = 1$
$C = \frac{log(a)}{log(b)} = \frac{log(1)}{log(3)} = 0$
$C = C_critandk > 1$ so this is case 2a
This results in a run time of $T(n) = \Theta(log^2(n))$

4. T(n) = 7T(n/2) + n^2
$a = 7; b = 2; f(n) = n^2, so C_crit = 2, k = 0$
$C = \frac{log(a)}{log(b)} = \frac{log(7)}{log(2)} = 2.81$
$C > C_critandk = 0$ so this is case 3
This case gives a condition of $af(n/b) \leq kf(n)$ with k < 1 for n >> 1. This results in $7f(n/2) \leq kf(n)$ where f(n) is n^2 This results in not being able to yield a run time since there is no k which makes the inequality true
.

5. T(n) = 21T(n/5) + $10n^2$
   $a = 21; b = 5; f(n) = 10n^2, soC_crit = 2, k = 0$
   $C = \frac{log(a)}{log(b)} = \frac{log(21)}{log(5)} = 1.89$
   $C < C_critandk = 0$ so this is case 1
   This results in a run time of $T(n) = \Theta(n^2)$

6. T(n) = 9T(n/3) - $6n^2log^2(n)$

   Since f(n) is negative the master theorem can not be applied.

7. T(n) =2 T(n/2) + $nlog^{-1}(n)$
   $a = 2; b = 2; f(n) = nlog^{-1}(n), soC_crit = 1, k = -1$
   $C = \frac{log(a)}{log(b)} = \frac{log(2)}{log(2)} = 1$
   $C = C_critandk = -1$ so this is case 2b
   This results in a run time of $T(n) = \Theta(nlog(n))$

8. T(n) = 11T(n/4) + $2n^{log_3(11)}$
   $a = 11; b = 4; f(n) = 2n^{log_3(11)}$, so $C_crit = log_3(11) = 2.18, k = 0$
   $C = \frac{log(a)}{log(b)} = \frac{log(11)}{log(4)} = 1.73$
   $C < C_critandk = 0$ so this is case 1
   This results in a run time of $T(n) = \Theta(n^{log_3(11)})$

9. T(n) = 8T(n/3) + $n^2log(log(n))$
   $a = 8; b = 3; f(n) = n^2log(log(n)), soC_crit = 2, k = 1$
   $C = \frac{log(a)}{log(b)} = \frac{log(8)}{log(3)} = 1.89$
   $C < C_critandk = 0$ so this is case 1
   This results in a run time of $T(n) = \Theta(n^2)$

   Part 2

10. Supply blocks of pseudocode to replace the lines "FOO" and "BAR" to complete the implementation of the merge operation. Consider which parts of the regular merge algorithm are missing, and when/how to fetch more data from the cloud.
    $FOO = u \leftarrow A[++imodb]$
    $BAR = v \leftarrow B[++jmodb]$

11. Exactly how many times must the merge function call each of read and write to merge two arrays of size n/2 into an array of size n, assuming that n/2 is divisible by b? (Hint: Beware off-by-1 errors when thinking about how many elements must be examined!)

    Since merge method takes two array of n/2 and combines them the read function is called $\frac{n}{2b}$ time to get all the elements in an array. Then an extra read is necessary to check for any other elements less than inf. This yields the number of times read merge for 1 array = $\frac{n}{2b} + 1$
    So to get both array you multiply by 2 to get the number of times the merge function calls read to be

$\frac{n}{b} + 2$.

Since the write function is called once per merge call the amount of times merge calls write is $\frac{n}{b}$.

12. What is the total number of cloud read and write operations performed by MergeSort to sort an array A of size n stored in the cloud? Give an asymptotic answer in terms of n, the number of elements in A, and b, the chunk size. How does this cost compare to the asymptotic cost of merge-sorting an array of size n held entirely in your computer's memory?

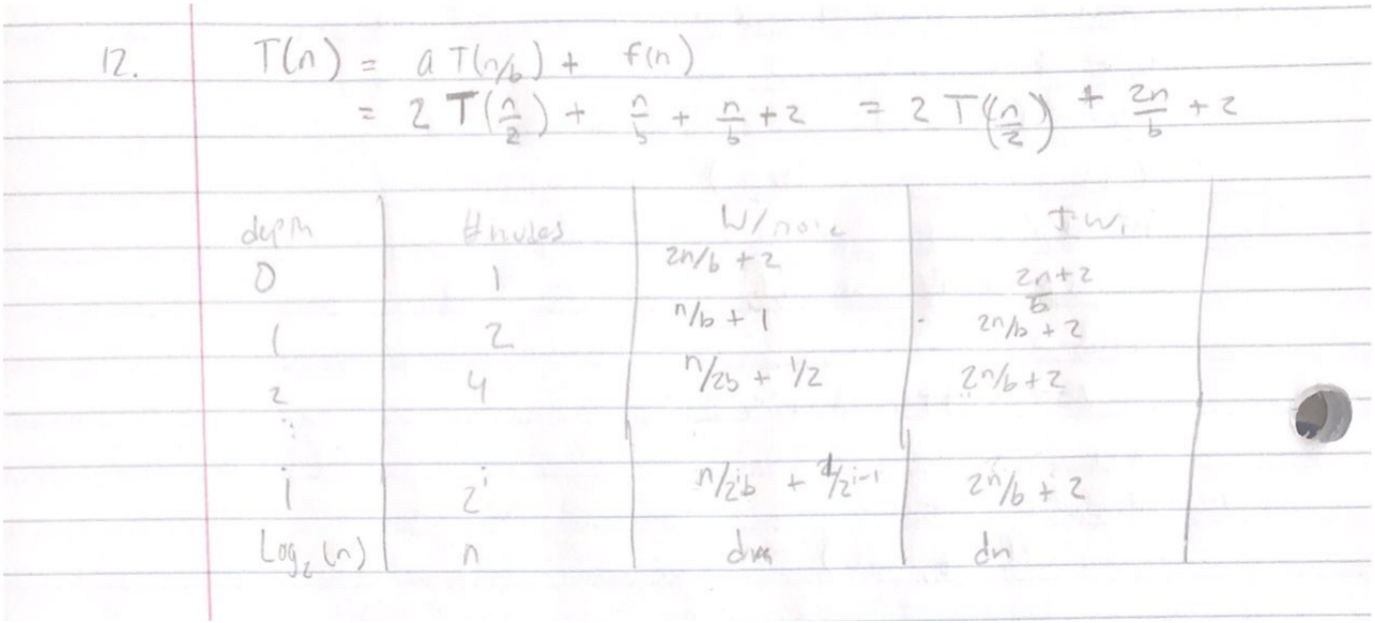

Figure 3:

From the chart above this results in a summation of: $\sum_{i=0}^{log(n)-1}(\frac{2n}{b} + 2) + dn$. This yields a $T(n) = \Theta(nlog(n))$.

13. Why might MergeSort be preferable to HeapSort in the cloud sorting model, where the cost is based on the number of chunk reads and writes? If there is an asymptotic difference between MergeSort and HeapSort in the cloud, include it in your argument.

Both methods have the same asymptotic time complexity. MergeSort is preferable in cloud sorting models because it the cost is based on the number of chunks read and writes so it can sort through sections of a huge array faster allowing the method to complete the sorting at a faster rate than HeapSort which would need to look through the whole array every time it checks for the next highest or lowest element in the array.

Part 3

Consider the problem we studied in M5's Studio: searching a sorted array A[1..n] for the leftmost occurrence of a query value x. Recall that the binary search algorithm solves this problem and returns

the index of the element in the array if it is present, or "not found" if it is not present. Binary search is one algorithm for this problem, but is it the fastest possible?

14. How many different outcomes does the search problem have for an array of size n? How did you arrive at this number?
    For an array of size n their are n unique outcomes when finding an element in the array and then if it can not be found then the search will return not found. So there are a total of n + 1 outcomes.

15. As for sorting, we will use the comparison model of computation, but this time the only comparisons permitted are of the form "is A[i] ¡ x?", where x is the query value. How many different outcomes can such a comparison have?
    There are only two different outcomes when using this comparison. The outcome can either be less than or greater than/equal to the query value.

16. Sketch a decision tree for solving this problem in this model on arrays of size 4, using the tree notation shown in the example from lecture videos/slides. Your tree must exactly match the sequence of comparisons performed by the (corrected!) binary search code at the end of M5's Studio, Part B. Include only comparisons of A[i] against x for some i. (You can treat "=" as another permitted comparison when building your tree.)

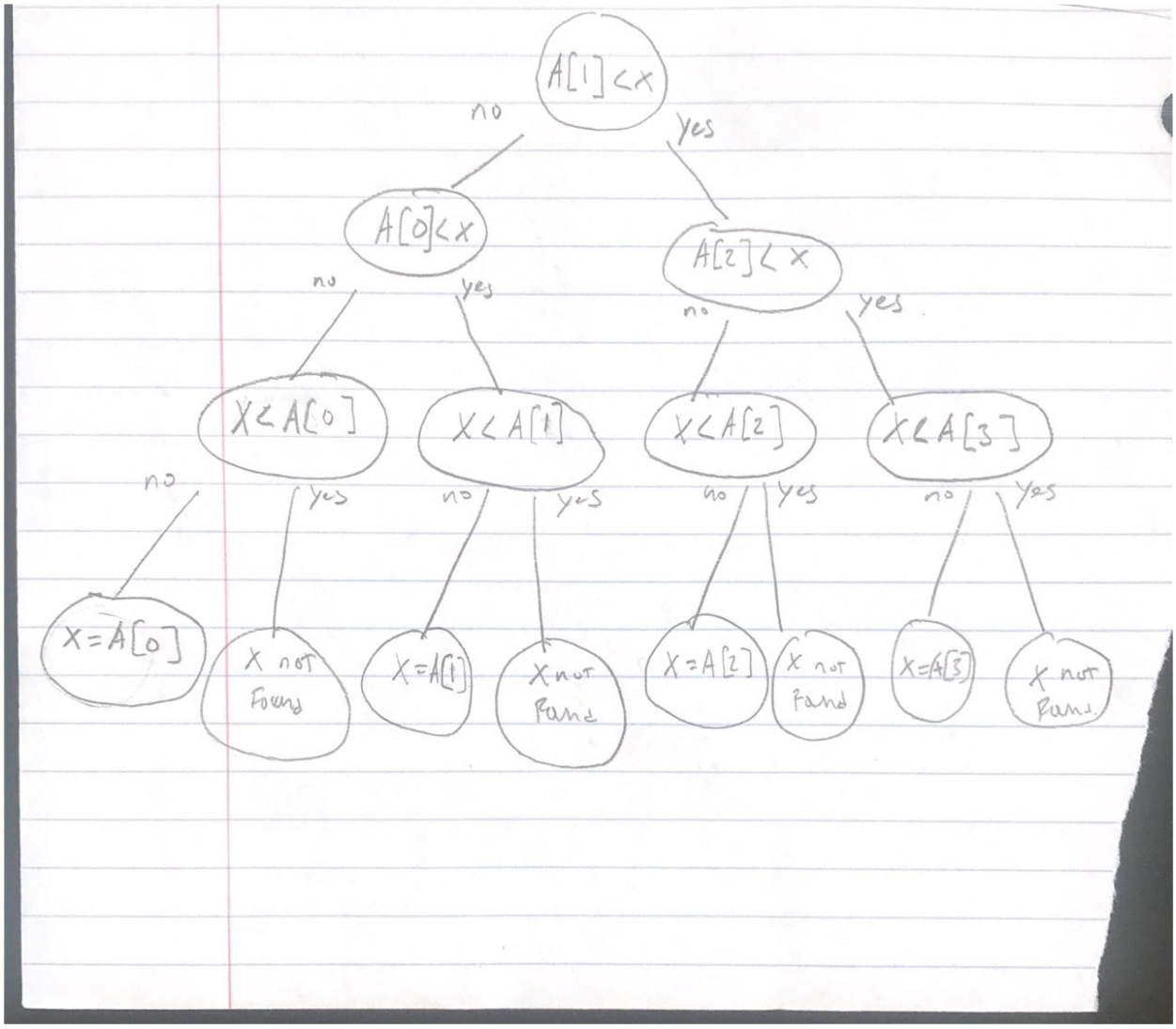
A[1]<x
no
yes
A[0]<x
A[2]<x
no
yes
no
yes
X<A[0]
X<A[1]
X<A[2]
X<A[3]
no
yes
no
yes
no
yes
no
yes
X=A[0]
X not Found
X=A[1]
X not Found
X=A[2]
X not Found
X=A[3]
X not Found

Figure 4:

17. Using similar decision-tree reasoning to what we used for sort, derive an asymptotic lower bound on the cost of any algorithm for searching a sorted array in the comparison model. Justify your answer.

    Assuming each comparison to be constant time, then the summation for the time complexity is $\sum_{i=0}^{log(n)-1} n = nlog(n) - 1$, so $T(n) = \Theta(nlog(n))$

    Part 4

    In lectures and the zyBook, we reviewed the radix sort algorithm for sorting an array of n d-digit integers, with each digit in base k, in linear time (d(n + k)). The basic algorithm is as follows:
    for j=1..d do
    sort the input stably by each element's jth least-significant digit
    The sort we used in each pass through the loop was a simple bucket sort, but any stable sort will work (albeit perhaps with different overall complexity). We tried this algorithm and saw that it worked on an example. Your job is to prove inductively that this algorithm works in general. The class notes suggest proving the following loop invariant: after j passes through the loop, the input is sorted according to the integers formed by each element's j least-significant digits.

18. State and prove a suitable base case for the proof.

    Base Case: Let d = 1, and see that the radix sort is sorted based on the first digit.

19. Now state and prove an inductive case for the proof. You may assume that the per-digit sort used in each iteration is (1) correct and (2) stable.

    Now assume j - 1 < d, meaning the second to last digit, is also sorted. So now we sort the j = d case, which is the last digit of the numbers. If the last digit from the one number is less than another then it will be before the other digit. If the last digits of the number are the same then they stay in the same order as before, which is based on the sort of the second to last digit.

    This shows that the numbers will be sorted correctly into the right order and maintain the stability because any two numbers that share a common digit when comparing that digit will stay in the same order.

20. Why does the invariant imply that the radix sort as a whole is correct?

    The invariant implies that the whole radix sort is correct because at the d = j case it sorts the last digit of each number. Since every other digit before this has been sorted already, as seen from the proof in question 19, this last case will complete the sort of the whole array so that it is correct.