M6: Lab

Part I

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. $T(n) = 3T(n/2) + n^2$
2. $T(n) = 4T(n/2) + n^2$

Now, for each of the following recurrences, what (if any) asymptotic solution does the Master Method give? You should use the *extended* version of the Master Theorem (found on Wikipedia) that we referenced in studio. If this version of the Master Theorem gives no solution, say so — in which case you do not need to solve the recurrence. You must *show your work* for full credit, including:

- the values of a, b, and $f(n)$;
- the test you performed to determine which case of the Master Theorem applies, if any;
- the result of the test (i.e. which case of the Master Theorem applies);
- the conclusion about running times you draw from the result.

3. $T(n) = T\left(\frac{n}{3}\right) + \log n$
4. $T(n) = 7T\left(\frac{n}{2}\right) + n^2$
5. $T(n) = 26T\left(\frac{n}{5}\right) + 12n^2$
6. $T(n) = 9T\left(\frac{n}{3}\right) - 6n^2 \log^2 n$
7. $T(n) = 9T\left(\frac{n}{3}\right) + \frac{n^2}{\log n}$
8. $T(n) = 11T\left(\frac{n}{4}\right) + 2n^{\log_3 11}$
9. $T(n) = 3T\left(\frac{n}{2}\right) + n^2 \log\log n$

Part II

While HeapSort (as we saw in M6's Studio) is more amenable than MergeSort to an in-place implementation, MergeSort has its own advantages. For this problem, suppose you want to perform MergeSort on a really *huge* array A. The array is so big that it doesn't fit in your computer's memory and has to be stored in the cloud.

More specifically, assume that our computer has enough memory to hold $3b$ elements, for some constant b, but A has size n much greater than b. We can call `read(X, i, B)` to read a chunk of b elements from an array X (in the cloud) starting at index i into a local array B. Similarly, we can call `write(C, X, i)` to write a chunk of b elements stored in local array C to a cloud array X starting at index i.

Here's a proposed (incomplete!) implementation of the merge operation that merges cloud arrays X and Y into cloud array Z. The code uses local arrays A, B, and C, each of size b, to cache X, Y, and Z. For simplicity, we assume that the input arrays X and Y have sizes a multiple of b, and that reading past the end of either X or Y returns values ∞ as in the studio. "mod" is the integer modulo operator (`%` in Java).

```
MERGE(X, Y, Z)
    i ← 0
    j ← 0
    k ← 0
    READ(X, 0, A)
    READ(Y, 0, B)
    while A[i mod b] < ∞ or B[j mod b] < ∞ do
        u ← A[i mod b]
        v ← B[j mod b]
        C[k mod b] = min(u, v)
        if u < v
            FOO
        else
            BAR
        k++
        if k mod b = 0
            WRITE(C, Z, k − b)
```

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.

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!)

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?

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.

Part III

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?

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?

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.)

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.

Part IV

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 $\Theta(d(n+k))$. 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.

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.

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