CPSC 320 2017W1: Assignment 4

November 2, 2017

Please submit this assignment via GradeScope at `https://gradescope.com`. Be sure to identify everyone in your group if you're making a group submission (which we encourage!).

Submit by the deadline **Friday 17 Nov at 10PM**. For credit, your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**. Your group's submission **must**:

- Be on time.
- Consist of a single, clearly legible file uploadable to GradeScope with clearly indicated solutions to the problems. (PDFs produced via LaTeX, Word, Google Docs, or other editing software work well. Scanned documents will likely work well. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout (not the individual quizzes). Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the **ugrad.cs.ubc.ca e-mail addresses** of each member of your team. (No names are necessary.)
- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and GradeScope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff." (Go read those guidelines!)
- Include at the start of the document your outside-group collaborators' ugrad.cs.ubc.ca e-mail addresses. (Be sure to get those when you collaborate!)

1 Making Every Second Count

Aconcagua.ca sells storage on both an auction and fixed-price basis. They want to use historical auction data to investigate their fixed price choices.

For n seconds, they have the price point reached in each second in their auctions. They want to find the largest price-over-time stretch in their data. That is, given an array A of n price points, they want to find the largest possible value of $f(i,d) = d * \min(A[i], A[i+1], \ldots, A[i+d-1])$ where i is the index of the left end of a stretch of seconds, d is the duration in seconds of that stretch, and so the function f computes the duration multiplied by the minimum price over that period. (Prices are positive, $d \geq 0$, for all values of i, $f(i,0) = 0$, and for legal indexes i into A, $f(i,1) = A[i]$.)

For example, the best stretch is underlined in the following price array: $[8, 2, \underline{9, 5, 6, 5}, 3, 1]$. Using 1-based indexing, the value for this optimal stretch starting at index 3 and running for 4 seconds is $f(3,4) = 4 * \min(9,5,6,5) = 4 * 5 = 20$.

1. Write out two trivial instances of the Aconcagua problem.

   (a) First instance:

   (b) Second instance:

2. Fill in the blanks below with the optimal values for i and d, and $f(i,d)$ in each array. You **MUST** use 1-based indexing. (The first element of each array has an index of 1.)

   (a) $[1, 9, 4, 7, 4, 2, 8]$

   Value of i: ☐

   Value of d: ☐

   Value of $f(i,d)$: ☐

   (b) $[18, 6, 8, 3, 4, 5, 11]$

   Value of i: ☐

   Value of d: ☐

   Value of $f(i,d)$: ☐

   (c) $[2, 2, 2, 2]$

   Value of i: ☐

   Value of d: ☐

   Value of $f(i,d)$: ☐

3. Fill in **best** choice of circles among each set below to complete design of a **brute force** algorithm to solve the "Aconcagua" problem and analyse its runtime in terms of n, the length of A.

   A valid solution to the Aconcagua problem is ○ a pair of values for i and d ○ a value for f(i, d) ○ a subset of A . However, a better quantity to consider in designing a brute force approach is ○ a pair of values for i and d ○ a value for f(i, d) ○ a subset of A . There are

○ $O(n)$
○ $O(n \lg n)$
○ $O(n^2)$
○ $O(n^3)$
○ $O(2^n)$

of these quantities. Given a candidate quantity, a brute force approach to computing its f value would take

○ $O(1)$
○ $O(d)$
○ $O(f)$
○ $O(i)$
○ $O(n)$

to run. Overall the resulting natural brute force algorithm will take

○ $O(n)$
○ $O(n \lg n)$
○ $O(n^2)$
○ $O(n^3)$
○ $O(2^n)$

time to run, which is

○ polynomial
○ exponential

.

4. Consider again the example array: $[8, 2, 9, 5, 6, 5, 3, 1]$. Imagine that you are told that the solution **must** use the elements at indexes 4 and 5 (i.e., elements 5 and 6, using 1-based indexing). You begin by considering the smallest stretch of the array that includes these two elements: $i = 4, d = 2$ (i.e., elements 5 and 6 only) and repeatedly expand that candidate "stretch" to include one more element either to the left or right of the existing stretch while maintaining the invariant that the stretch you have chosen is the best of that length (value of d) that includes indexes 4 and 5 until the stretch includes the entire array.

   Finish the following table describing this expansion, thinking carefully about why you make the choice you do at each point:

i	d	minimum	$f(i,d)$
4	2	5	10
3	3	5	15
3	4	5	20
1	8	1	8

1.1 Quiz Solution

1. There are many reasonable trivial solutions to the problem. For example:
   - A length 0 array has a maximum f value of 0.
   - A length 1 array has a maximum f value of $A[1]$.
   - An array of any length $n \geq 1$ filled with all identical elements has a maximum f value of $A[1] * n$. (You may or may not consider that last trivial.)

2. (a) $[1, \underline{9, 4, 7, 4}, 2, 8]$
   $i = 2, d = 4, f(i, d) = 16$
   (b) $[\underline{18, 6, 8, 3, 4, 5, 11}]$
   $i = 1, d = 7, f(i, d) = 21$
   (c) $[\underline{2, 2, 2, 2}]$
   $i = 1, d = 4, f(i, d) = 8$

3. A valid solution to the Aconcagua problem is **a value for** $f(i, d)$. However, a better quantity to consider in designing a brute force approach is **a pair of values for** i **and** d. There are $O(n^2)$ of these quantities. Given a candidate quantity, a brute force approach to computing its f value would take $O(d)$ to run. Overall the resulting natural brute force algorithm will take $O(n^3)$ time to run, which is **polynomial**.

4. Consider again the example array: $[8, 2, 9, 5, 6, 5, 3, 1]$. ...

   Finish the following table describing this expansion, thinking carefully about why you make the choice you do at each point:

i	d	minimum	$f(i, d)$
4	2	5	10
3	3	5	15
3	4	5	20
3	**5**	**3**	**15**
2	**6**	**2**	**12**
1	**7**	**2**	**14**
1	8	1	8

   (At each step, we just expand the stretch in the direction of the larger value on the border of the stretch, or the only allowable stretch once we reach a boundary.)

1.2 Assignment

1. Give pseudo-code for the natural brute force algorithm to solve the Aconcagua problem in $\Theta(n^3)$ time.

```
BruteSolveAcon(A, n):
```

2. Give pseudocode for an efficient ~~divide-and-conquaguar~~ divide-and-conquer algorithm for finding the optimal price stretch based on the insight in quiz question 4.

```
Conquaguar(A, n):
```

3. Give and briefly justify a good asymptotic bound on the runtime of your algorithm.

4. Prove that your algorithm implementing the insight from quiz question 4 has (in general, not just for the array from that question) the property described in that question. Specifically: at each step, it finds the best stretch of the length (value of d) under consideration that includes the middle indexes of the array.

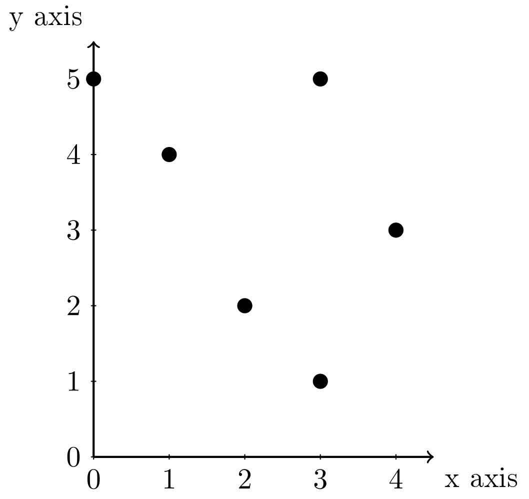
2 Quadsires

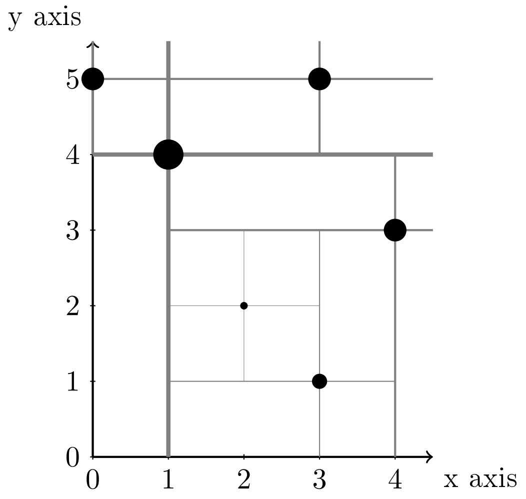
The popular augmented reality game Lokimon Go maintains a large map of sites of interest—like Lokistops and Gyms—each with x and y coordinates. (Longitude and latitude, but for our purposes, we'll take these to be integer x and y on a Cartesian grid.)

These are stored in a type of "quad tree" data structure. Specifically, the structure is a 4-ary tree. Each node `n` is associated with a Lokimon Go site of interest `n.site`, represented simply as a (x, y) coordinate pair giving the location of the site (`n.site` for the pair or `n.site.x` and `n.site.y` for the individual coordinates). The node's four subtrees are `n.NE`, `n.SE`, `n.SW`, and `n.NW`. These are (respectively): the area to the northeast (higher x and y values), southeast (higher x and lower y values), southwest (lower x and y values), and northwest (lower x and higher y values). Some of these subtrees may be empty (represented by the special quad tree value `EMPTY`) if they contain no sites of interest.

There are some boundary cases: No two sites have the **same** coordinates. Sites that are due north or due east are both included in the `NE` subtree, due south in the `SE` subtree, and due west in the `NW` subtree.

For example, on the left below is a set of sites we might want to put into a Lokimon Go quad tree. On the right is that set in a possible quad tree. The root is the largest dot at $(1, 4)$ with lines separating out its NE, SE, SW, and NW quadrants. It has three non-empty subtrees, and each root of these subtrees is drawn with a slightly smaller dot. The NE is rooted at $(3, 5)$, the SE at $(4, 3)$, and the NW at $(0, 5)$. Only the SE subtree has further children. It has a SW subtree (rooted at $(3, 1)$) which in turn has a NW subtree (rooted at $(2, 2)$).





We could then search for a site like $(2, 2)$ by accessing the root, which is at $(1, 4)$, observing that $(2, 2)$ is to the SW of the root, and recursing into the SW subtree of the root.

We also define a rectangle $((x_{lo}, y_{lo}), (x_{hi}, y_{hi}))$ to contain the set of sites with coordinates (x, y) such that x is between x_{lo} and x_{hi} and y is between y_{lo} and y_{hi}. (We are intentionally vague on the boundary conditions for the moment. However, if $x_{lo} > x_{hi}$ or $y_{lo} > y_{hi}$, the rectangle is empty.)

We gave a rough outline of a good algorithm for searching a quadtree for a site above. Assume that `Search(qt, pt)` is an efficient and correct implementation of this algorithm that takes a quadtree `qt` and (x, y) coordinate pair `pt` and produces the quad tree node containing that site (if any) or the special empty quad tree value `EMPTY` (otherwise).

1. This algorithm will have two base cases. What will they be? Fill in the boxes next to the **TWO** that apply:

☐ `qt` is the root node of the quad tree
☐ `pt` is (0, 0)
☐ `qt` is `EMPTY`
☐ `qt.site` is equal to `pt`
☐ `qt.site.value` is 0

2. Let `pt` be (x_p, y_p) and the coordinates of `qt.site` be (x_q, y_q). If $x_p < x_q$ and $y_p \geq y_q$, which subtree should we recurse into? Fill in the circle next to the correct answer.

   ○ NE
   ○ SE
   ○ SW
   ○ NW

3. Give a good big-O bound on the worst-case runtime of `Search` in terms of each of the following variables or write "none" if no such bound can be given.

   (a) The number of nodes e in `qt` that are to the east of (have x coordinates *at least as large* as) `pt`: ☐

   (b) The number of nodes n in `qt`: ☐

   (c) The height h of `qt`: ☐

4. We can use a rectangle to define the portion of the (x, y) plane contained in each subtree of the quadtree. So, the root contains $((-\infty, -\infty), (\infty, \infty))$, but the root's SE subtree contains only $((root.site.x, -\infty), (\infty, root.site.y))$.

   If a given subtree contains the rectangle $((x_{lo}, y_{lo}), (x_{hi}, y_{hi}))$, we can ask whether the subtree (and its rectangle) should contain sites on its northern ($y = y_{hi}$), eastern ($x = x_{hi}$), southern ($y = y_{lo}$), and western ($x = x_{lo}$) boundaries. For each of the following indicate the **best** choice about whether a subtree should contain sites on these boundaries (**YES**) or should not (**NO**) based on the definitions above. Do not worry about infinite values (∞ and $-\infty$) in these choices.

Boundary	**YES**	**NO**
Northern:	○	○
Eastern:	○	○
Southern:	○	○
Western:	○	○

5. Assume we add a field `box` to each node that is a four-tuple: `(xlo, xhi, ylo, yhi)`, where `n.box` should contain the boundaries for the rectangle representing the portion of the (x, y) plane contained in the subtree rooted at `n`. Complete the following algorithm for setting correct `box` boundaries on the nodes of a quad-tree, given that the given parameters `(xlo, xhi, ylo, yhi)` are correct boundaries for the subtree `qt`. Note that we only ask for the NE and SW subtrees, leaving ... for the parameters for the recursive calls on the NW and SE subtrees. You should assume that the SE and NW subtrees are handled correctly.

```
procedure SetBox(qt, xlo, xhi, ylo, yhi)
    x = qt.site.x
```

```
        y = qt.site.y
        if root is not EMPTY then
            qt.box = (xlo, xhi, ylo, yhi)
            SetBox(qt.NE, $\boxed{\phantom{xx}}$, $\boxed{\phantom{xx}}$, $\boxed{\phantom{xx}}$, $\boxed{\phantom{xx}}$)
            SetBox(qt.SE, ..., ..., ..., ...)
            SetBox(qt.SW, $\boxed{\phantom{xx}}$, $\boxed{\phantom{xx}}$, $\boxed{\phantom{xx}}$, $\boxed{\phantom{xx}}$)
            SetBox(qt.NW, ..., ..., ..., ...)
        end if
    end procedure
```

6. Give a good big-O bound on the worst-case runtime of a correct and efficient implementation of `SetBox` in terms of each of the following variables or write "none" if no such bound can be given.
   (a) The number of nodes e in `qt` that are to the east of (have x coordinates *at least as large* as) `xlo`: $\boxed{}$
   (b) The number of nodes n in `qt`: $\boxed{}$
   (c) The height h of `qt`: $\boxed{}$

2.1 Quiz Solution

1. Base cases: `qt` is `EMPTY`, `qt.site` is equal to `pt`.
2. NW (because smaller x is W and larger y is N)
3. The algorithm traverses a single path from root to target or leaf.
   (a) The number of nodes e in `qt` that are to the east of (have x coordinates *at least as large* as) `pt`: "none"
   (b) The number of nodes n in `qt`: $O(n)$
   (c) The height h of `qt`: $O(h)$
4.

Boundary	**SOLUTION**
Northern:	**NO**
Eastern:	**NO**
Southern:	**YES**
Western:	**YES**

(Why? Consider which directions from a root node are included in each of the subtrees. E.g., because sites due north of the root are included in the NE subtree, the NE subtree must include its western boundary (which is that due north line).)

5.
```
    procedure SetBox(qt, xlo, xhi, ylo, yhi)
        x = qt.site.x
        y = qt.site.y
    if root is not EMPTY then
        qt.box = (xlo, xhi, ylo, yhi)
        SetBox(qt.NE, qt.site.x, xhi, qt.site.y, yhi)
        SetBox(qt.SE, ..., ..., ..., ...)
        SetBox(qt.SW, xlo, qt.site.x, ylo, qt.site.y)
        SetBox(qt.NW, ..., ..., ..., ...)
    end if
end procedure
```

6. The algorithm spends constant time at each node.
   (a) The number of nodes e in `qt` that are to the east of (have x coordinates *at least as large* as) `xlo`: $O(e)$
   (b) The number of nodes n in `qt`: $O(n)$
   (c) The height h of `qt`: $O(4^h)$

2.2 Assignment

We add to each node a non-negative integer `freq` field representing the frequency at which we expect that node's site to be accessed in Lokimon Go. We want higher-frequency nodes to be closer to the root of the tree. In particular, if a node is k steps from the root (where for the root, $k = 0$), then we assign it a cost of $k * \texttt{freq}$. The cost of a whole tree is the sum of the costs of each of its nodes.

1. Give efficient pseudocode for a function `Cost` that computes the total cost of a tree `qt` (that has `freq` fields).

   ```
   Cost(qt):
   ```

2. Given a set of sites and their frequencies, we can ask which is the optimal quadtree (i.e., a quadtree that has minimal cost among all possible quadtrees built from the same sites).

   Give pseudocode for a brute-force, recursive algorithm to find the total cost of the optimal quadtree for a given set of sites with their frequencies.

   Hint: Add two "fake" zero-frequency sites at $(-\infty, -\infty)$ and (∞, ∞). A subproblem can be described by a rectangle that could be the box associated with a subtree of the final quadtree. A rectangle is an (x, y) coordinate for the lower-left corner and an (x, y) coordinate for the upper-right corner. Now, how can you concisely describe a subproblem?

   ```
   BruteBuildQT(sites, freqs, n):
   ```

3. Give pseudocode for a memoized version of your brute-force algorithm. Your memoized algorithm must run in worst-case $O(n^3)$ time (for n sites).

   ```
   MemoizedBuildQT(sites, freqs, n):
   ```

3 Shreddin' the Gnar

Enough of all this hard work, it's time to start thinking about the snowboarding season! Cypress is opening a huge new run with a sequence of jumping ramps designed to delight. Every day, they'll hold a contest to see who can catch the most air on the way down the mountain. You are tasked with planning the winning route, according to the following rules: whenever you reach a ramp while on the ground, you can either use that ramp to jump through the air, possibly flying over one or more ramps, or shred past that ramp and stay on the ground. Obviously, if you fly *over* a ramp, you cannot use that ramp to increase your air time.

Based on observation and experimentation, you have assembled the following data: an array $R[1 \dots n]$, where $R[i]$ is the distance from the top of the hill to the i^{th} ramp, and an array $L[1 \dots n]$, where $L[i]$ is the distance any boarder who takes the i^{th} ramp will travel through the air. For simplicity, you may assume that array R is sorted by distance, so that the ramps are numbered $1 \dots n$ down the mountain.

In this part of the problem we will design an algorithm to find an optimal solution. Don't forget the definition of $N(i)$: For any index i, let $N(i)$ denote the smallest index j such that $R[j] > R[i] + L[i]$. You may also find it convenient to define $R[n+1] = \infty$.

3.1 Quiz and Solution

1. A reasonable definition for an algorithm to solve this problem would be $MA(i)$:

   [✓ The maximum distance spent in the air starting on the ground at ramp i.]
   ○ The maximum number of ramps taken before ramp i.
   ○ The maximum number of ramps taken after starting on the ground at ramp i.
   ○ The minimum number of ramps missed after starting on the ground at ramp i.
   ○ The single longest jump remaining after and including ramp i.
   ○ The minimum total distance spent in the air before ramp i.

2. We can solve the problem by calling the algorithm as: [✓ $MA(1)$] ○ $MA(n)$ (choose one).

3. A base case for the problem is: $MA(i) = \boxed{}$ if [✓ $i > n$] ○ $i < n$ (choose one).

4. Suppose you choose not to take ramp i. Then, $MA(i)$ is just:

   ○ $N(i)$
   ○ $MA(i)$
   [✓ $MA(i+1)$]
   ○ $MA(N(i))$
   ○ None of these.

5. Suppose you choose to take ramp i. In that case, $MA(i)$ is:

   ○ $L[i] + MA(i+1)$
   ○ $R[i+1] + MA(i+1)$
   ○ $L[N(i)] + MA(i+1)$
   [✓ $L[i] + MA(N(i))$]
   ○ None of these.

6. The recursive case of the dynamic programming algorithm you can use to solve this problem takes the [✓ max] ○ min (choose one) of the values in the last two parts.

7. Given the algorithm you have designed by your selections above, a reasonable approach to memoizing the computation would require ______ space.

○ $O(1)$
○ $O(\log n)$
[✓ $O(n)$]
○ $O(n \log n)$
○ $O(n^2)$
○ We don't have enough information to answer the question.

3.2 Assignment

Putting together the results from the quiz...

Let MA(i) denote the maximum distance any snowboarder can spend in the air starting on the ground at the i^{th} ramp. We need to compute MA(1). This function satisfies the following recurrence:

$$\text{MA}(i) = \begin{cases} 0 & i > n \\ \max \left\{ \begin{array}{l} \text{MA}(i+1) \\ L[i] + \text{MA}(N(i)) \end{array} \right\} & 1 \le i \le n \end{cases}$$

1. Complete the pseudocode below to define a dynamic programming (iterative memoized) algorithm for computing MA(i). Assume $N(i)$ is already implemented, but think about how it works and about its running time on data of size i. Write **clearly** and **carefully** and follow the existing notation, as some minor differences in notation can also indicate substantial misunderstandings. (Please do use the typical max function available in most languages.)

```
function MAXAIR($R[1..n], L[1..n]$)
    $R[n+1] \leftarrow \infty$
    MA$[n+1] \leftarrow$ $\boxed{\phantom{xx}}$
    for $i = n$ down to 1 do
        MA$[i] \leftarrow$ $\boxed{\phantom{xxxxxxxxxxxxxxxxxxxx}}$
    end for
    return MA[1]
```

2. Give a good upper bound on the running time of MaxAir on data of size n.

3. Uh Oh. The Cypress ski patrol heard about the contest and protested that it would result in too many injuries. In response, Cypress imposed a limit on the number of jumps that any snowboarder can take as they navigate the slope. Reformulate the problem and its solution using this new information:

   (a) The problem is now parameterized by two input variables. Rewrite the definition of the function MA(i, j) in English.
   Let MA(i, j) denote...

   (b) Define the recursive function that could be used to compute MA(i, j).

   (c) Write pseudocode to define a dynamic programming (iterative memoized) algorithm for computing MA(i, j).

   (d) What is the running time of your algorithm?

4 Recurring Nightmare

4.1 Quiz and Answers

Below you will find five recurrences that should be familiar to you.

Identify the problem context in which you've seen each recurrence and write it in the answer box, and also to discuss how each function relates to the problem that it solves. Annotate your answer with an explanation for any recurrence which is not a worst case analysis of an upper bound on the running time.

1. Running time of $\boxed{\qquad\qquad\qquad}$ on an array of size n:

   $$T(n) = \begin{cases} 0 & n = 0 \\ T(3n/4) + T(n/4) + n & n > 0 \end{cases}$$

   This is much like the average-case analysis (or the similar expected analysis) for quicksort.

   Asymptotic closed form: $T(n) = \Theta(n \log n)$

2. Running time of $\boxed{\qquad\qquad\qquad}$ on an array of size n:

   $$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + T(1) + n & n > 0 \end{cases}$$

   This is much like the worst-case analysis for quicksort.

   Asymptotic closed form: $T(n) = \Theta(n^2)$

3. Running time of $\boxed{\qquad\qquad\qquad}$ on an array of size n:

   $$T(n) = \begin{cases} 0 & n = 0 \\ T(3n/4) + T(n/5) + n & n > 0 \end{cases}$$

   This is much like the worst-case analysis for deterministic select.

   Asymptotic closed form: $T(n) = \Theta(n)$

4. Running time of $\boxed{\qquad\qquad\qquad}$ on an array of size n:

   $$T(n) = \begin{cases} 0 & n = 0 \\ 2T(n/2) + n & n > 0 \end{cases}$$

   This is much like the worst-case analysis for mergesort.

   Asymptotic closed form: $T(n) = \Theta(n \log n)$

5. Running time of $\boxed{\qquad\qquad\qquad}$ on input of magnitude n:

$$T(n) \geq \begin{cases} 1 & n \in \{0, 1, \ldots, 24\} \\ 3T(n-25) + 1 & n > 24 \end{cases}$$

   This is much like the lower-bound we developed on the runtime of our "brute force" recursive coin-changing algorithm. Note that since we only know that $T(n) \geq$ the cases on the right, this is an Ω bound, not a Θ bound.

   Asymptotic closed form: $T(n) = \Omega(c^n), c > 1$

   (A more specific function than c^n would be fine, but the key point on this recurrence in class was that its solution is exponential.)

4.2 Assignment

1. Give an asymptotic solution to the recurrence below. In particular, follow these steps: draw a recurrence tree and show your work as you develop a *guess* at the solution; then, prove that your guess is correct via induction.

$$T(n) = \begin{cases} 1 & n = 1 \\ 3T(n/2) + n & n > 1 \end{cases}$$

2. Give an asymptotic solution to the recurrence below by substituting $U(n) = n \cdot T(n)$, noting the well-known bound on the more familiar recurrence you find, and then deriving the bound on the original function $T(n)$. **Show** your steps using the substitution to get a more familiar recurrence and then transforming this back into a solution for $T(n)$.

$$T(n) = \begin{cases} 1 & n = 1 \\ \frac{1}{4}T(\frac{n}{4}) + \frac{3}{4}T(\frac{3n}{4}) + 1 & n > 1 \end{cases}$$

3. Give an asymptotic solution to the recurrence below by narrowing in on a tight bound. Loose bounds—which you need not prove—of $\Omega(n)$ and $O(n \log n)$ are easy to establish. What functions lie between those? Guess and prove! *Hint:* consider using functions like $\sqrt{\lg x}$, $\lg \lg x$, and $\lg^* x$.

$$T(n) = \begin{cases} 2 & n = 2 \\ \sqrt{n} \cdot T(\sqrt{n}) + n & n > 2 \end{cases}$$