CS302 - Problem Set 1

1. Read over the Syllabus. (If you are also in 333, the main difference is in the Assessments section.) You don't need to follow links - you can go back to them later as needed. Then go to the Syllabus Discussion Channel on Teams, and post a comment or a question or respond to an existing post by replying. If someone has already posted your question or comment, you can give it a thumbs-up.

2. In your personal section of OneNote, under the Reflection section, you should find the Page "Advice from Last Year's Students." Please read over their advice, and add your own response on the page, either following the prompt, or based on what is meaningful to you. (This page is only visible to you and to me, not to other students.)

3. The following are videos/resources to fill in or refresh material that is helpful for solving the rest of the problem set (and are topics that you were very likely exposed to in 200/201, and that we will continue to practice throughout the semester). You are not required to watch/read them, but if you are getting stuck as work you on the problem set, you should check out the relevant topic.

   - Video: Deriving a recurrence relation for the runtime of a recursive algorithm
   - Video: Proving correctness of recursive algorithms using regular and strong induction
   - Log Review resources
   - Textbook Chapter on Big-O

4. This problem will help you to review and recall logarithms, exponentiation, summation, trees, recurrence relations, geometric series, and big-O notation, all of which will come up with some regularity over the course of the semester.

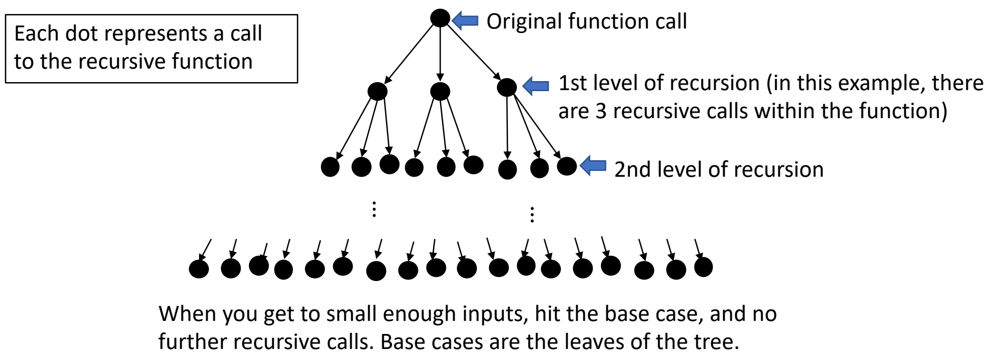
   The runtime of many recursive algorithms takes the following form:

   $$T(1) = O(1), \qquad T(n) = aT(n/b) + O(n^d), \tag{1}$$

   where $T(n)$ is the runtime of the function on an input of size n, and a, b, and d are constants and that do not depend on n.

   (a) What do a, b, and d represent in terms of the structure of the algorithm? (If you are stuck on this, take a look at the recurrence relation video and try to generalize.)

   (b) We can represent the structure of a recursive algorithm with runtime $T(n) = aT(n/b) + O(n^d), T(1) = O(1)$ using a tree:


Each dot represents a call to the recursive function
Original function call
1st level of recursion (in this example, there are 3 recursive calls within the function)
2nd level of recursion
When you get to small enough inputs, hit the base case, and no further recursive calls. Base cases are the leaves of the tree.

Each call to the algorithm is represented by a vertex, the initial call of the function is the root, and each recursive call produces a child vertex from the vertex of the function that called it. In terms of n (the original input size), a, b, and/or d, how many levels will this tree have (from the root to the leaves)? In other words, how many levels of recursion will there be?

(c) In terms of n (the original input size), a, b, d, and/or k, how many function calls are there at the kth level of recursion? In terms of the tree, we can rephrase this question as how many vertices are there in level k below the root?

(d) At each level of recursion, the size of the input to the function decreases. If the original call had an input of size n, in terms of n, a, b, d, and/or k, what is the size of the input to the function calls at level k?

(e) In terms of n (the original input size), a, b, d, and/or k, at a single vertex (function call) at level k, how much time (how many operations) is used by that function call, excluding any operations done in the recursive calls it makes. For example, in MergeSort, if the input to a function call is size m, the work done at that call and ignoring the two recursive calls is $O(m)$, because we only look at non-recursive parts of the algorithm, which take time $O(m)$.

(f) In terms of n (the original input size), a, b, d, and/or k, how much time (how many operations) is used by all function calls at level k, excluding any operations done by the further recursive calls they make? (Combine your answers to (c) and (e).)

(g) In terms of n (the original input size), a, b, and/or d, how much time (how many operations) is used by all function calls in the algorithm (at all levels of recursion)? (Please leave in summation notation.)

(h) Use the formula for geometric series:

$$\sum_{k=0}^{t} r^k = \begin{cases} t+1 & \text{if } r = 1 \\ \frac{r^{t+1}-1}{r-1} & \text{else} \end{cases} \tag{2}$$

to evaluate the sum from the previous question.

(i) If $\frac{a}{b^d} = 1$, what is the big-O runtime of the algorithm? You should assume a, b, and d are constants, and n is the variable that gets large.

(j) If $\frac{a}{b^d} < 1$ what is the big-O runtime of the algorithm? You should assume a, b, and d are constants, and n is the variable that gets large.

(k) If $\frac{a}{b^d} > 1$ what is the big-O runtime of the algorithm? You should assume a, b, and d are constants, and n is the variable that gets large.

(l) You should find that the runtime behaves differently depending on the 3 possible relationships between $\frac{a}{b^d}$ and 1. Qualitatively explain the behavior in each case. I'll do one case as an example: if $\frac{a}{b^d} < 1$, that means that a tends to be small relative to b and d. If b and d are large, that means that recursive calls happen on inputs that are much smaller than the original input, and a lot of time/operations are spent *not* in recursive calls. If a is small, that means there are not a lot of recursive calls. Putting these ideas together, we expect that in this case the runtime will *not* be strongly dependent on the recursive calls. From our analysis in part (j), we see that when $\frac{a}{b^d} < 1$, the whole runtime of the algorithm is $O(n^d)$. However, note that the original function call uses $O(n^d)$ time, excluding recursive calls! This means that all of the recursive calls in the whole algorithm are basically not adding anything significant to the runtime, and the majority of the runtime is spent at the top of the tree, at the root. This makes sense, given our qualitative explanation that we expect the runtime to not be strongly affected by recursive calls. (Now you should do a similar analysis for the other two cases: $\frac{a}{b^d} > 1$ and $\frac{a}{b^d} = 1$.)

5. The elements of a bi-tonic list either only increase, only decrease, or first increase and then decrease. For example: $[1, 4, 5, 10, 14]$, $[1, 8, 5, 4]$, $[1, 0]$, and $[3]$ are all bitonic, while $[5, 4, 3, 4, 5]$ and $[3, 5, 3, 5]$ are not bitonic.

(a) Write pseudocode for a recursive algorithm `BMax` that finds the maximum value of a bi-tonic list of n distinct integers that runs in time $O(\log n)$.

```
Algorithm 1: BMax($A$)
Input  : Bi-tonic list $A$ of distinct integers of size $n$
Output: The maximum value of $A$)
// Your pseudocode here!
```

(b) Prove your algorithm is correct. (In the video, I use a more formal inductive proof style with a predicate $P(n)$. You don't have to use a predicate if you prefer to set up your inductive proof less formally, as long as the structure and logic are still there.)

(c) Create a recurrence relation for the runtime of your algorithm.

(d) Evaluate the recurrence relation using your result from problem 1, showing that the algorithm indeed has runtime $O(\log n)$, and also provide an intuitive explanation for why the runtime is what it is.

(e) (Challenge) How would each part of the problem change if we additionally allow lists that first decrease and then increase? What about if we allow lists that change direction twice (first increase, then decrease, then increase; or first decrease, then increase, then decrease) or change direction k times?