In theory, you have already seen things on this assignment in previous classes such as Data Structures and Discrete Math. However, very few students remember everything here, and that's OK. Use this assignment as an opportunity to help you figure out some things you should review.

Start early!

Ask for help when you get stuck!

Work together, but remember to write up **your own solutions**.

Have fun!

I **expect** you to ask me for help on homework assignments in this class. If you never ask for any help, you are either very smart or very foolish.

1. How many times do you have to repeatedly halve 32 in order to reach 1 (or, put another way, how many times do you have to repeatedly double 1 in order to reach 32)? What about 8192? Consider the function which given an input n, outputs the number of times n can be repeatedly halved before falling below 1. What is the common mathematical name for this function?

2. For each of the following, answer with the **best** (smallest) upper bound from this list: $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n)$. Give a **brief justification** for each.

   (a) number of leaves in a depth-n balanced binary tree

   (b) depth of an n-node balanced binary tree

   (c) number of edges in an n-node tree

   (d) time needed to sort a list of n items using merge sort

   (e) number of distinct subsets of a set of n items

   (f) number of bits needed to represent the number n in binary

   (g) time needed to find the closest pair of points among n points in Euclidean space by simply listing all the pairs

   (h) time needed to insert n items into a binary heap

   (i) time needed to find the second largest number in a *sorted* list of n distinct numbers

   (j) time needed to find the second largest number in an *unsorted* list of n distinct numbers

Note: students coming out of data structures often think that big-O is specifically about measuring *how long* algorithms take, but it is not. Big-O is a tool for measuring the rate of growth of one thing relative to another. We often do use it measure the rate of growth of the time needed to run an algorithm relative to the size of the input, but we can use it to measure rates of growth of other things as well, such as the amount of memory used relative to the size of the input, the height of a tree relative to the width, and so on. Therefore, you should not assume that these are asking about *time* unless they explicitly say so. For example, part (a) is asking *how many leaves* there are in a tree with a depth of n, that is, how fast does the number of leaves grow relative to the depth? It is not asking *how long* it would take to find those leaves.

3. Let p_n be defined for all $n \geq 0$ by

$$p_0 = 0$$
$$p_n = 2p_{n-1} + 1 \qquad (n > 0)$$

   State and prove a closed (that is, non-recursive) formula for p_n.

If you need a refresher on recursion and induction, there are a lot of resources posted on the course website!

4. We will not use the syntax of any particular programming language in this course to specify algorithms. In class, I will tend to use Python-like *pseudocode*, which is like real code except that it can include English phrases to avoid specifying too much detail. For example, Algorithm 1 below gives some pseudocode for finding the smallest integer in an array. Notice that it uses the phrase "a is smaller than m" in the condition of the **if**. (Of course, in this example it would have actually been shorter to just write "$a < m$"; the point is that we can describe things informally if we want.)

Algorithm 1: FINDMIN(A, n)

Require: A sequence of integers A.

```
1: m = +∞
2: for a in A :
3:     if a is smaller than m :
4:         m = a
5: return m
```

Your pseudocode does not have to look exactly like mine; for example, you could use syntax that feels more familiar to you, such as Java-like or Haskell-like syntax. The important point is that it be readable and understandable to someone who knows how to program. That is, your pseudocode should have enough structure and detail so that a competent programmer would be able to take it and translate it into working code in a programming language of their choice.

A binary tree T is either NULL, or else it has a left child $T.left$, a right child $T.right$, and an integer value $T.value$. Write some pseudocode to find the smallest integer value in a binary tree T, or return $+\infty$ if T is null.

Note you should not assume T is a binary *search* tree; that is, there need not be any relationship between the value at a node and the values at its children, so the smallest integer value could be anywhere in the tree.