- The instructions are the same as in Homework 0, 1, 2, 3.

There are 5 questions for a total of 100 points.

---

1. (15 points) Solve the following recurrence relations.
   (a) $T(n) = 3T(n/3) + cn$, T(1) = c

   a = 3, b = 3, d = 1
   $3 = 3^1$
   $O(n \log_3 n)$

   (b) $T(n) = 3T(n/3) + cn^2$, T(1) = c

   a = 3, b = 3, d = 2
   $3 < 3^2$
   $O(n^2)$

   (c) $T(n) = 3T(n-1) + 1$, T(1) = 1

   Cannot apply Master Theorem
   Notice that there are 3 recursive calls per layer
   Layer decreases by 1 for each iteration, for a total of n layers
   Constant term is not dominating
   Therefore, there are 3^n recursive calls, $O(3^n)$

2. (15 points) An array $A[1...n]$ containing n integers is said to be a *hill-array* iff

   1. $n \geq 3$,
   2. $A[1], A[2], ..., A[n]$ are distinct, and
   3. there is an index $1 < i < n$ such that

   $$A[1] < A[2] < ... < A[i-1] < A[i] > A[i+1] > A[i+2] > ... > A[n]$$

   and such an index i is called the *peak-index*.

   Design an algorithm for finding the peak-index of any given hill-array. You are given as input a hill-array A and the size n of the array. Give a time analysis for your algorithm, and brief explanation for correctness.

   Correctness:
   We use Divide and Conquer to check if the middle element is the peak. If not, we check if its right neighbor is larger, then we know that the peak must be contained in the right half of the array (to satisfy the constraints). Conversely, if the middle element's right neighbor is smaller, the peak must be contained in the left half of the array. Recurse until the peak is found.

   Running Time:
   The algorithm is essentially a modified Binary Search, splitting the array in half every iteration. Therefore, the runtime of the algorithm is logarithmic $O(\log n)$.

```
procedure DNCPeak(A, n):
    if n is 1 then                                          // only one element, is peak
        return 0
    end
    if A[0] ≥ A[1] then                                     // first element violates property, is peak
        return 0
    end
    if A[n − 1] ≥ A[n − 2] then                             // last element violates property, is peak
        return n - 1
    end
    lo = 0, hi = n - 1
    while lo ≤ hi do                                        // can still binary search
        mid = (lo + hi) / 2
        if A[mid] > A[mid − 1] and A[mid] > [mid + 1] then   // mid is peak
            return mid
        end
        else
            if A[mid] < A[mid + 1] then                     // right neighbor is higher, peak in right half
                lo = mid + 1
            end
            else                                            // right neighbor is lower, peak in left half
                hi = mid - 1
            end
        end
    end
    return
```

3. (20 points) Consider the following problem: You are given a pointer to the root r of a binary tree, where each vertex v has pointers $v.lc$ and $v.rc$ to the left and right child, and a value $Val(v) > 1$. The value NIL represents a null pointer, showing that v has no child of that type. You wish to find the path from r to some leaf that minimizes the product of values of vertices along that path. Give an algorithm to find the minimum product of vertices along such a path along with a proof of correctness and runtime analysis.

```
procedure DNCMinProduct(r):
    if r.left is not NIL then
        leftMinHeight = DNCMinProduct(r.left)
    end
    else
        leftMinHeight = 0
    end
    if r.right is not NIL then
        rightMinHeight = DNCMinProduct(r.right)
    end
    else
        rightMinHeight = 0
    end
    return min(leftMinHeight, rightMinHeight) + Val(v)
```

Correctness:
We will prove the correctness of the algorithm through induction.

Base case:

Let us consider a tree of size 1. This means the root has no children, and it is the only leaf in the tree. The only path to the root is through itself, so the algorithm will correctly return Val(v).

Inductive:
Let us assume that DNCMinProduct holds for any tree of size $1 \leq k \leq n-1$. We will show that it also holds for any tree of size $(n-1)+1 = n$.

If the root has both left and right children, the subset of all paths to a leaf will be $r \rightarrow$ *all paths in left child* and $r \rightarrow$ *all paths in right child*. Since each child contains $1 \leq k \leq n-1$ nodes, and we assume DNCMinProduct holds for all such values of k, the minimum path to a leaf from the root is simply $Val(v) + min(DNCMinProduct(r.left, r.right))$.

If the root has only one child, the subset of all paths to a leaf will be $r \rightarrow$ *all paths in only child*. DNCMinProduct will return the minimum path to a leaf in r.onlyChild, while the other (nonexistent) child will return a value of 0. Now running DNCMinProduct on the root will return $Val(v) + min(DNCMinProduct(r.onlyChild, 0))$, which is indeed correct. Thus, the inductive step holds for n.

Running Time:
The algorithm recurses over every subtree in the tree. It stops recursing once a subtree's leaf nodes have been identified. It must traverse every node in a subtree once to have completely visited all of its leaf node. We can view the entire problem as recursing over a subtree with root r, so the runtime is all the nodes contained in the tree, $O(n)$.

4. (20 points) Let S and T be sorted arrays each containing n elements. Design an algorithm to find the n^{th} smallest element out of the $2n$ elements in S and T . Discuss running time, and give proof of correctness.

```
procedure DNCNthSmallest(S, T, n):
    lo = 0, hi = n - 1
    i = 0, j = 0
    while lo ≤ hi do
        i = (lo + hi) / 2                                   // get midpoint of S
        j = k - i - 1                  // for a total of n elements covered by both indices
        if T[j − 1] > S[i] then            // S is too small for tightest pairing, increase
            lo = i + 1
        else if i > 0 and S[i − 1] > T[j] then                  // S is too large, decrease
            hi = i - 1
        else
            return min(S[i], T[j])
        end
    end
```

Correctness:
We are trying to find the n^{th} smallest element in S and T. This can be done by considering the pair of elements at $S[i]$ and $T[j]$, such that $i + j = n$. We want this relation because it means there are n elements smaller than i and j (inclusive), so if the n^{th} smallest element exists in this subset of n elements, it must be either $S[i]$ or $T[j]$. This arises from the fact that both arrays are sorted, so all elements before i and j cannot possibly be the n^{th} smallest element (there are only n elements in this subset), and don't need to be considered.

Now all we need to do is find the tightest pairing of $S[i]$ and $T[j]$ that doesn't violate the sorted property. If $S[i]$ and $T[j]$ are not the tightest pairings, the n^{th} smallest element may be contained beyond i and j. We may not be considering elements past i that are less than $T[j]$. Similarly, we may be skipping elements past j that are less than $S[i]$. Our subset of $i + j = n$ elements may be sorted, but if they are not the *tightest* sorting possible, finding the n^{th} smallest element on this subset is meaningless in the context of the entire problem.

To find this tightest pairing of $S[i]$ and $T[j]$, we check if $T[j-1]$ is larger than $S[i]$. If so, element $T[j-1]$ is closer to $T[j]$ than $S[i]$ to $T[j]$. This is not the tightest pairing, and we need to reconsider $S[i]$. We need to increase $S[i]$ so it can potentially "overtake" $T[j-1]$ and be closer to $T[j]$. We do this by setting the new $S[i]$ to be the midpoint of the remaining larger terms in S. The opposite applies if $S[i-1]$ is larger than $T[j]$, where element $S[i-1]$ is closer to $S[i]$ than $T[j]$ to $S[i]$. Again, this is not the tightest pairing. We decrease $S[i]$ by setting it to the midpoint of the remaining smaller terms in S. Finally, we readjust j so the equality $i + j = n$ is maintained.

The n^{th} smallest element is found when we have achieved the tightest pairing of $S[i]$ and $T[j]$. This means n elements are covered by i and j, and these elements are the tightest sorting possible. From here, we just pick the smaller of $S[i]$ and $T[j]$. This is because arrays are zero-indexed, so we overshoot our tightest pairing by one element.

Running Time:
We are essentially just running binary search on S. Since both arrays are of equal size n, the total running time is logarithmic $O(log\ n)$.

5. An array $A[1...n]$ is said to have a majority element if more than half (i.e., $> n/2$) of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] \geq A[j]$?" (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.
   (a) (15 points) Show how to solve this problem in $O(n \log n)$ time. Provide a runtime analysis and proof of correctness.
   (*Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.*)
   (b) (15 points) Design a linear time algorithm. Provide a runtime analysis and proof of correctness.
   (*Hint: Here is another divide-and-conquer approach:*
   - *Pair up the elements of A arbitrarily, to get n/2 pairs (say n is even)*
   - *Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them*
   - *Show that after this procedure there are at most n/2 elements left, and that if A has a majority element then it's a majority in the remaining set as well)*