Name: Ananye Agarwal **Entry Number:** 2017CS10326
Name: Kabir Tomer **Entry Number:** 2017CS50410
Name: Rajat Jaiswal **Entry Number:** 2017CS50415

1. (5 points) Solve the following recurrence relations. You may use the Master theorem wherever applicable.

(a) $T(n) = 3T(n/3) + cn$, T(1) = c

Solution (a) We show by induction that $T(n) = cn(\log_3 n + 1)$. The base case is $n = 1$ and it is easy to see that $T(1) = c$ by our formula. Suppose that $T(k) = ck(\log_3 k + 1)$ for all $k \leq n$. Then, by the recurrence $T(n+1) = 3T\left(\frac{n+1}{3}\right) + c(n+1)$. Since $\frac{n+1}{3} \leq n$ we may write $T\left(\frac{n+1}{3}\right) = c\left(\frac{n+1}{3}\right)\left(\log_3 \frac{n+1}{3} + 1\right)$ by the induction hypothesis. This implies,

$$T(n+1) = 3c\left(\frac{n+1}{3}\right)\left(\log_3 \frac{n+1}{3} + 1\right) + c(n+1)$$

$$\implies T(n+1) = 3c\left(\frac{n+1}{3}\right)\log_3(n+1) + c(n+1)$$

$$\implies T(n+1) = c(n+1)\left(\log_3(n+1) + 1\right)$$

which proves the induction hypothesis for $n+1$. This proves that $T(n) = cn(\log_3 n + 1) \in O(n \log_3 n)$ for all n as desired.

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

Solution (b) We show by induction that $T(n) = \frac{cn(3n-1)}{2}$. The base case is $n = 1$ and it is easy to see that $T(1) = c$ by our formula. Suppose that $T(k) = \frac{ck(3k-1)}{2}$ for all $k \leq n$. Then, by the recurrence $T(n+1) = 3T\left(\frac{n+1}{3}\right) + c(n+1)^2$. Since $\frac{n+1}{3} \leq n$ we may write $T\left(\frac{n+1}{3}\right) = \frac{c\frac{n+1}{3}(3\frac{n+1}{3}-1)}{2}$ by the induction hypothesis. This implies,

$$T(n+1) = 3\left[\frac{c\frac{n+1}{3}(3\frac{n+1}{3} - 1)}{2}\right] + c(n+1)^2$$

$$\implies T(n+1) = \frac{c(n+1)n}{2} + c(n+1)^2$$

$$\implies T(n+1) = \frac{c(n^2 + n + 2n^2 + 2 + 4n)}{2} = \frac{c(3n^2 + 5n + 2)}{2}$$

$$\implies T(n+1) = \frac{c(n+1)(3(n+1) - 1)}{2}$$

which proves the induction hypothesis for $n+1$. This proves that $T(n) = \frac{cn(3n-1)}{2} \in O(n^2)$ for all n as desired.

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

Solution (c) We show by induction that $T(n) = \frac{3^n - 1}{2}$. The base case is $n = 1$ and it is easy to see that $T(1) = 1$ by our formula. Suppose that $T(k) = \frac{3^k - 1}{2}$ for all $k \leq n$. Then, by the recurrence $T(n+1) = 3T(n) + 1$. We may write $T(n-1) = \frac{3^n - 1}{2}$ by the induction hypothesis. This implies,

$$T(n+1) = 3\left(\frac{3^n - 1}{2}\right) + 1$$

$$\implies T(n+1) = \frac{3 \cdot 3^n - 3 + 2}{2} = \frac{3^{n+1} - 1}{2}$$

which proves the induction hypothesis for $n+1$. This proves that $T(n) = \frac{3^n - 1}{2} \in O(3^n)$ for all n as desired.

2. (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 $Val(v) > 0$. 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 total values of vertices along that path. Give an algorithm to find the minimum sum of vertices along such a path along with a proof of correctness and runtime analysis.

Solution: The function MinPath(v) finds the smallest cost path to a leaf in the tree rooted at v.

Algorithm 1 Shortest path to leaf

```
function MinPath($v$)
    if $v$ is NIL then
        return 0
    else
        $a$ = MinPath($v.lc$)
        $b$ = MinPath($v.rc$)
        return $Val(v)$ + MIN($a, b$)
    end if
end function
```

Running time analysis: Notice that MinPath is called exactly once for each vertex v in a tree. This is because each vertex has a unique parent and only a parent of v can call MinPath on it. The time spent inside each call to MinPath is $O(1)$. Therefore, the overall running time is $O(n)$, where n is the set of vertices in the tree.

Correctness It is sufficient to prove that the following statement holds.

Lemma 2.1: MinPath(v) returns the minimum sum of vertices along a path to some leaf in the tree rooted at v.

We prove this theorem by induction on the height of the tree. For the purposes of this question, we define the height of a tree to be the number of nodes on the longest path from a node to some leaf. Therefore, height of a tree with no nodes is 0, and the height of a tree with a single node is 1.

Base Case: When the height is zero, v is NIL and MinPath(v) $= 0$ as expected.

Induction Hypothesis: Suppose lemma 1 holds for all v where the height of the tree rooted at v is atmost k.

Induction Step: Consider a vertex v such that the height of the tree rooted at v is $k+1$. The subtrees $v.lc$ and $v.rc$ have height atmost k, therefore, by the induction hypothesis the minimum sum of vertices along paths to their respective leaves is $a =$ MinPath($v.lc$) and $b =$ MinPath($v.rc$) respectively. In this case, MinPath returns the value $Val(v)$ + MIN(a, b). Suppose the minimum sum of vertices along a path to a leaf of v be S. Let this path be $P = v, v_1, v_2, \ldots$. Notice that the sum of vertices $v_1, v_2, \ldots$ is $S - Val(v)$. Further, $v_1, v_2, \ldots$ is a path from one of the children of v to their leaves. Therefore, by the induction hypothesis, the sum of this path has to be atleast MinPath(v_1). Since v_1 is either $v.lc$ or $v.rc$, this implies $S - Val(v) \geq$ MIN(a, b). Therefore, $S \geq Val(v)$ + MIN(a, b). This implies that $Val(v)$ + MIN(a, b) is indeed the minimum sum of vertices along some path to a leaf in v. This completes the proof of the induction step and of lemma 1 by extension.

3. (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.

The function FindNth finds the n^{th} smallest element in S and T.

Algorithm 2 n^{th} smallest in two sorted arrays of size n

```
 1: function FindNth($S, T$)
 2:     result = BinarySearch($S, T, 1, n$)
 3:     if result is not NIL then
 4:         return result
 5:     else
 6:         result = BinarySearch($T, S, 1, n$)
 7:     end if
 8:     return result
 9: end function
10:
11: function BinarySearch($A, B$, start, end)
12:     while start $\leq$ end do
13:         mid = $\lfloor \frac{\text{start+end}}{2} \rfloor$
14:
15:         if $A[\text{mid}] < B[n - \text{mid}]$ then
16:             start = mid + 1
17:         else if $A[\text{mid}] > B[n - \text{mid} + 1]$ then
18:             end = mid - 1
19:         else
20:             return $A[\text{mid}]$
21:         end if
22:     end while
23:
24:     return NIL
25: end function
```

Running Time: Notice that in each iteration of the while loop in lines 12-22, the quantity end - start + 1 reduces by at least a factor of 2. Further, the while loop can only execute until start $\leq$ end. This implies that the while loop executes $O(\log(\text{end} - \text{start} + 1))$ times. Since each iteration of the loop is constant time, the overall runtime of BinarySearch is $O(\log(\text{end} - \text{start} + 1))$. Therefore, in the function FindNth, the runtime of lines 2, 6 is $O(\log n)$. Since all other lines are constant time, the total runtime of FindNth is $O(\log n)$.
Correctness: Consider the following lemma
Lemma 3.1: For two sorted arrays A, B each of size n, BinarySearch($A, B, 1, n$) returns the n^{th} smallest element in these arrays if it is present in A, otherwise it returns NIL.
Notice that lemma 3.1 straightforwardly implies the correctness of FindNth. This is because the n^{th} smallest element in S, T must be present in either S or T. FindNth first executes line 2 : result = BinarySearch($S, T, 1, n$). Lemma 3.1 implies that if the n^{th} smallest element is in S, the result variable is non-NIL and the desired answer. Accordingly, FindNth checks if result is not NIL and returns it. Otherwise, the n^{th} smallest element is in T. Accordingly, the function calls BinarySearch($T, S, 1, n$) and returns the result.

Proof of lemma 3.1: We show that the while loop in lines 12-22 maintains the following loop invariant.

Loop invariant: The n^{th} smallest element in the arrays A, B, if present in A, lies in the subarray A[start..end].

Suppose the invariant holds at the start of the loop. We will show that it continues to hold at the end of the loop (line 25). There are three cases,
Case 1: $A\,[\text{mid}] < B\,[n - \text{mid}]$ – In this case, no element in A[start..mid] can be the n^{th} smallest element. Indeed, for $k \leq \text{mid}$, the number of elements smaller than it in A is atmost $k - 1$. Further, since $A[k] \leq A\,[\text{mid}] < B\,[n - \text{mid}]$, the number of elements smaller than $A[k]$ in B is less than $n - \text{mid}$. Therefore, the total number of elements smaller than $A[k]$ is less than $k - 1 + n - \text{mid} < n - 1$ which implies k cannot be the n^{th} smallest element. Therefore, the n^{th} smallest element can only lie in A[(mid + 1)..end]. Thus, the invariant is maintained in this case since the algorithm sets start = mid + 1.
Case 2: $A\,[\text{mid}] > B\,[n - \text{mid} + 1]$ – In this case, no element in A[mid..end] can be the n^{th} smallest element. Indeed, for $k \geq \text{mid}$, the number of elements smaller than it in A is atleast $k - 1$. Further, since $A[k] \geq A\,[\text{mid}] > B\,[n - \text{mid} + 1]$, the number of elements smaller than $A[k]$ in B is greater than $n - \text{mid} + 1$. Therefore, the total number of elements smaller than $A[k]$ is greater than $k - 1 + n - \text{mid} + 1 > n$ which implies k cannot be the n^{th} smallest element. Therefore, the n^{th} smallest element can only lie in A[(start)..mid-1]. Thus, the invariant is maintained in this case since the algorithm sets end = mid - 1.
Case 3: $B\,[n - \text{mid}] \leq A\,[\text{mid}] \leq B\,[n - \text{mid} + 1]$ – In this case line 20 is executed and the loop exits. Therefore, the invariant holds vacuously.
Now, notice that when BinarySearch is called, the loop invariant holds, since the n^{th} smallest element in each case (if present) can only lie between indices $1, n$. Therefore, we conclude that this invariant holds as long as the loop executes. BinarySearch can exit either from line 20 or from line 24. If it exits from line 20, we have found an index such that $B\,[n - \text{mid}] \leq A\,[\text{mid}] \leq B\,[n - \text{mid} + 1]$. However, this implies that there are mid $- 1$ elements smaller than A[mid] in A and $n - \text{mid}$ elements smaller than it in B. Therefore, the total number of elements smaller than it is $n - 1$ implying that A[mid] is indeed the n^{th} smallest element. Therefore, BinarySearch returns the correct answer in this case.
If the function returns from line 24, start > end. This means that the n^{th} smallest element cannot be present in A since otherwise the loop invariant would be violated. Accordingly, BinarySearch returns NIL. This proves the correctness of BinarySearch and also proves lemma 3.1.

4. 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) (10 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) (10 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)*

Solution (a):

Algorithm 3 To find the majority element in a given array.

```
function isMajority($A[1,..,n], x$)
    $n \leftarrow Length(A)$
    $count \leftarrow 0$
    for $i = 1$ to $n$ do
        if A[i] == x then
            $count++$
        end if
    end for
    return $count > \frac{n}{2}$
end function

function findMajority($A[1,..,n]$)
    $n \leftarrow Length(A)$
    if n == 1 then
        return $A[1]$
    end if

    $L,\ R \leftarrow$ Left and Right halves of $A$
    $L_{majority} \leftarrow findMajority(L)$
    $R_{majority} \leftarrow findMajority(R)$
    if $isMajority(A, L_{majority})$ then
        return $L_{majority}$
    else if $isMajority(A, R_{majority})$ then
        return $R_{majority}$
    else
        return $None$
    end if
end function

```

Running time analysis: The function `isMajority` takes $O(n)$ time. It basically iterates through the entire and calculates the frequency of element x. If the frequency is greater than $\frac{n}{2}$ then it returns true, otherwise false. The function `findMajority` recursively calls itself on the left and right halves of the array and then checks if either of the returned value is a majority in the original array or not. So, the recurrence relation for this function turns out to be $T(n) = 2(\frac{n}{2}) + O(n)$. Solving this recurrence relation using the master's theorem gives the running time of the algorithm $T(n) \in O(n \log n)$.

Proof of Correctness: If an element x occurs $> \frac{n}{2}$ times in an array A then it is a majority element as per the definition. If x is a majority element in A, then it occurs $> \frac{n}{2}$ times. If it occurs $\leq \frac{n}{2}$ times then it is not a majority element according to the definition.
If x appears $> \frac{n}{2}$ times in A, then it appears $> \frac{n}{4}$ times in L or R or both. If it occurs $\leq \frac{n}{4}$ times in both L and R then it appears $\leq \frac{n}{2}$ times in A, hence contradiction.
If x occurs $> \frac{n}{4}$ times in L or R or both then it is a majority element in L or R or both and vice-versa. Size of L or R is $\frac{n}{2}$ and according to the definition of majority element, a majority element of L or R will have to occur $> \frac{n}{4}$ times in respective sub-array. If it occurs $\leq \frac{n}{4}$ times then it is not a majority element.

This establishes the relation: $isMajority(A, x) \iff Count(x, A) > \frac{n}{2} \Rightarrow Count(x, L) > \frac{n}{4} \mid\mid Count(x, R) > \frac{n}{4} \iff isMajority(L, x) \mid\mid isMajority(R, x)$.
This is precisely what we are doing in the `findMajority` function. We can also use induction.

Proposition: *P(n):* `findMajority` returns the majority element(if exists) for any array of size n.
Base Case: When n = 1, we have just one element and that is the majority element.
Induction Hypothesis: P(k) is true for all $1 \leq k \leq n$.
Inductive Step: For any array of size n+1, the algorithm returns correct majority element for the left and right halves, whose sizes are less than n. Induction hypothesis holds. If the array had a majority element, it would be returned by the recursive calls on either L or R. And therefore it will also be returned by the original call to the function. Hence Proved.

Solution (b):

Algorithm 4 To find the majority element in a given array.

```
function majorityFromPairs($A[1,..,n]$)
    $n \leftarrow Length(A)$
    if $n == 0$ then
        return $None$
    else if $n == 1$ then
        return $A[1]$
    end if
    if $n$ is odd then
        if $isMajority(A, A[n])$ then
            return $A[n]$
        end if
        $n \leftarrow n - 1$
    end if
    $L \leftarrow [.]$
    for $i = 1$ to $\frac{n}{2}$ do
        if $A[2i] == A[2i-1]$ then
            Append $A[2i]$ to $L$.
        end if
    end for
    return $majorityFromPairs(L)$
end function

function findMajority2($A[1,..,n]$)
    $M \leftarrow majorityFromPairs(A)$
    if $isMajority(A, M)$ then
        return $M$
    else
        return $None$
    end if
end function
```

Running time analysis: The function `majorityFromPairs` takes $O(n)$ time. It checks if the last element is the majority element or not in case array size is odd. As discussed in last part `isMajority` takes $O(n)$ time. It also recursively iterates on array L, whose size is $\leq \frac{n}{2}$. So, the recurrence relation for this function turns out to be $T(n) \leq T(\frac{n}{2}) + O(n)$. Solving this recurrence relation using the master's theorem gives the running time of the algorithm $T(n) \in O(n)$. The function `findMajority2` calls `majorityFromPairs` and `ismajority` once, both of which are linear, so the overall running time of the algorithm is linear i.e. $O(n)$.

Proof of Correctness: The function `majorityFromPairs` maintains the invariant that if an element x is a majority element of array A, then it is also present in the newly created array L, and is the majority element of L. If x is the last element of A and size of A is odd, then it is correctly returned by `isMajority(A, x)`. In one call of `majorityFromPairs` we break the array into pairs. If in a pair both the elements are different then we discard that pair. Otherwise, we just keep one of the elements. Since x is the majority element of A, then it will be present $> \frac{n}{2}$ times in A. Therefore, not all the pairs will be of different-element type (pigeon-hole principle). Hence, x will be present in L. One element from each same-element pair is present in L. x must be the majority element in same-type pairs. Let a be the number of same-element type pairs and b be the number of different-element type pairs. $a + b = \frac{n}{2}$, and $x > \frac{n}{2}$. The number of x in different-element type pairs can be at most a. If any other element y is the majority element in l, then the count of x in same-element type pairs is at most b. Hence, in total the total count of x in A becomes at most $a+b(=\frac{n}{2})$. Hence, contradiction.

We will now use induction.
Proposition: *P(n):* `findMajority2` returns the majority element(if exists) for any array of size n.
Base Case: When n = 1, we have just one element and that is the majority element.
Induction Hypothesis: P(k) is true for all $1 \leq k \leq n$.
Inductive Step: For any array A of size n+1, if x is the majority element, then it is also a majority element of L in `majorityFrompairs(A)` as proved in the invariant above. Since size of L is smaller than n+1, we will apply induction hypothesis to get x as the returned element from the recursive call. Therefore, if we have a majority element in A, then that element is returned by the algorithm. P(n+1) holds. Hence Proved.

5. Consider the following algorithm for deciding whether an undirected graph has a *Hamiltonian Path* from x to y, i.e., a simple path in the graph from x to y going through all the nodes in G exactly once. ($N(x)$ is the set of neighbors of x, i.e. nodes directly connected to x in G).

```
HamPath(graph $G$, node $x$, node $y$)
  - If $x = y$ is the only node in $G$ return $True$.
  - If no node in $G$ is connected to $x$, return $False$.
  - For each $z \in N(x)$ do:
      - If HamPath($G - \{x\}, z, y$), return $True$.
  - return $False$
```

(a) (7 points) Give proof of correctness of the above backtracking algorithm.
Assume that `HamPath` is always given x to G (atleast on initial call).

Solution: We prove correctness by induction on the size of graph.

Base Case: For graphs with a single node, if destination node and source node are same as the single graph node, the path exists trivially, and `HamPath` returns *True*. If not then no node is connected to x in G, and `HamPath` returns *False*

Induction Hypothesis: `HamPath` is correct for all graphs of size $< n$.

We now prove correctness for graphs of size n under the above assumption.

Induction Step:
Completeness: If $n = 1$ the base case applies. If a simple path from x to y of length n exists, the first nodes of the path must be x and z where z is a neighbour of x (because consecutive nodes in a path are neighbours). The subpath defined by removing x from the beginning of the above path is a simple path containing every node in $G - \{x\}$, and therefore it is a Hamiltonian path for $G - \{x\}$ from z to y.

`HamPath` is correct for $G - \{x\}$ since it is of size $n-1$. Therefore, `HamPath`$(G - \{x\}, z, y)$ returns True (Note that z belongs to the graph, since it is a neighbour of x in G, and our initial assumption holds). Since z is a neighbour of x, `HamPath` returns True for G. Therefore if a Hamiltonian path exists, True is returned, and if False is returned then no Hamiltonian path exists.
Soundness: If $n = 1$ the base case applies. $G - \{x\}$ is a graph of size $n-1$, therefore by induction hypothesis `HamPath` is correct for $G - \{x\}$. If for some neighbour z of x, `HamPath`$(G - \{x\}, z, y)$ returns True (Note that z is in the graph as above), then by the induction hypothesis a Hamiltonian path from z to y exists for $G - \{x\}$. The path formed by adding x to the beginning of this path is therefore a simple path ($x \notin G - \{x\}$) of length n passing through every node in G so a Hamiltonian path exists. Therefore if True is returned a Hamiltonian path exists.

(b) (8 points) If every node of the graph G has degree (number of neighbors) at most 4, how long will this algorithm take at most? *(Hint: you can get a tighter bound than the most obvious one.)*

Solution: In each call to `HamPath`, the function spends $O(\deg(x))$ time executing lines 1, 2, 3 and then makes a recursive call for each neighbor z of x. Since $\deg(x)$ is atmost 4, the overall runtime is proportional to the number of recursive calls made. Consider the tree T of recursive calls for `HamPath`. Notice that any node in this tree (other than the root) has atmost 3 children since when it is called one of its edges has already been removed. The total number of edges in graph $G = (V, E)$ is $\frac{1}{2}\sum_v \deg(v) \le 2n$. Suppose the tree T branches into 3 a times and branches into 2 b times. Notice that at each branching, those many edges are removed. This implies that $3a + 2b \le 2n$. Further, the total number of leaves is atmost $4 \cdot 3^a \cdot 2^b$. Since each path to a leaf from the root of T has atmost n nodes (in each recursive call we remove one node), the overall runtime is bounded by $n \cdot 4 \cdot 3^a \cdot 2^b$. Notice that

$$3^a \cdot 2^b = 3^{\frac{3a+2b}{3}} \left(\frac{2}{3^{2/3}}\right)^b \le 3^{\frac{2n}{3}} \cdot (0.96)^b \le 3^{\frac{2n}{3}} = 2.08^n$$

Therefore, the overall runtime is bounded by $O(n \cdot 2.08^n)$.

6. (20 points) Design an $O(poly(n) \cdot 2^{n/4})$-time backtracking algorithm for the maximum independent set problem for graphs with bounded degree 3 (these are graphs where all vertices have degree at most 3). Give running time analysis and proof of correctness.

```
function CLEAN(G, L)
    for each node in G do
        if node has no neighbours add to L and remove from G
        if node has 1 neighbour add to L and remove node and neighbour from G
    end for
    repeat above until all nodes have atleast 2 neighbours or G empty
    return G, L
end function
```

```
function MIS(G, L)
    G, L = Clean(G, L)
    G, L = MIS3(G,L)
    return length(L)
end function
```

```
function MIS3(G, L) n = node with 3 neighbours
    if no such n then
        return MIS2(G, L)
    end if
    return Larger (by list length) of MIS3(G - n, L) and MIS3(G - n - N(n), L + [n])
end function
```

```
function MIS2(G, L)
    G, L = Clean(G,L) n = node with 2 neighbours
    if no such n then
        return MIS2(G, L)
    end if
    return Larger (by list length) of MIS3(G - n, L) and MIS3(G - n - N(n), L + [n])
end function
```

Any node of degree 1 or less may safely be added to the MIS. Clean does so i^2

If MIS2 receives a Graph with nodes of degree 2 or less, it outputs MIS of that graph in poly(n) time.