Zuha Aqib 26106

Question 1

(a) Suppose $Y[0] = Y[n] = 0$ and $Y[i] > 0$ for every other index i; that is, the endpoints of the path are strictly below every other point on the path. Prove that under these conditions, both of you can meet. You may do this by describing a graph that models all possible locations and transitions for both of you along the path. What are the vertices of this graph? What are the edges? What can you say about the degrees of the vertices?

To prove the Sheikh and I can meet, I need to first make this area a graph. To make it a graph, I see that each point of the Sheikh and me have the same y-coordinate as both are meeting eye level. But, we have different x-coordinates as we are at different positions. When my x-coordinate is equal to the Sheikh's x-coordinate, that means we have met.

So in the question we have been said that the X and Y arrays are of all coordinates that are **corners** of the path. Thus we can say that if the current y is equal to any value in the Y array, then it is a corner, which here we will call a *stop*. A *stop* can be called a traffic light, a place where we cannot move further. In more technical terms, a *stop* can be said is a point (x, y) where y is equal to $Y[i]$ for any i. Every corner is a *stop*.
If I have two points which are *stop*'s, and the line segment between them is contained in the path (and there is no *stop* in between them, then those *stop*'s are *adjacent*.

So to define a graph G of this area, we first say that this graph is undirected, with a set of vertices V and a set of edges E. This means $G = (V, E)$
We can define a vertice v, such that,

(x_M, x_S, y)

where x_M is my x-coordinate,
x_S is the Sheikh's x-coordinate, and
y is both of ours (mine and Sheikh's) y-coordinate.

This vertice v can be also seen as (x_M, y) as my-position and (x_S, y) as the Sheikh's-position. If i have a vertice (x_{M1}, x_{S1}, y_1) and another vertice (x_{M2}, x_{S2}, y_2), then they can be *adjacent vertices* if and only if position 1 is adjacent to position 2, which means (x_{M1}, y_1) *stop* is adjacent to (x_{M2}, y_2) *stop* AND (x_{S1}, y_1) *stop* is adjacent to (x_{S2}, y_2) *stop*. Thus, we define edges between two vertices if they are *adjacent*

So we have successfully defined the graph G, but now we need to prove that if I and the Sheikh start at $y = 0$ vertice, then we always meet. To prove this, each vertex configuration has four possible types of positions for me:

- At an endpoint, meaning I can only move up.
- At a local minimum, meaning I can only move up.
- At a local maximum, meaning I can only move down.
- At a regular point, where I can move either up or down.

The Sheikh has the same four options. Because the path's endpoints are strictly below every other corner, either both I and the Sheikh are at endpoints, or neither is; otherwise, any other combination is possible. This results in ten possible types of vertex configurations, which are as follows:

- (end, end)
- (min, min)
- (min, regular)
- (min, max)
- (regular, min)
- (regular, regular)
- (regular, max)
- (max, min)
- (max, regular)
- (max, max)

The vertex degree depends on their configuration:

- If both me and the Sheikh are at endpoints, we can only move inward, so the four (end, end) vertices have degree 1.
- If both me and the Sheikh are both at local minima, we have four options: I can move left or right, and independently, the Sheikh can also move left or right. Thus, each (min, min) vertex has degree 4. Similarly, each (max, max) vertex has degree 4.
- If I am at a local maximum and the Sheikh is at a local minimum, or vice versa, neither of us can move, so the degree of each (min, max) and (max, min) vertex is 0.
- If I am at a local minimum and the Sheikh is at a regular point, then the Sheikh can only move down, but I have two options. Thus, every (min, regular), (max, regular), (regular, min), and (regular, max) vertex has degree 2.
- In the remaining case where both me and the Sheikh are at regular points, we can either both move right or both move left. Thus, each (regular, regular) vertex has degree 2.

From this, we can conclude that the graph G has exactly four vertices with degree 1, and all other vertices have even degrees. These degree-1 vertices are labeled LL, LR, RL, and RR, indicating which endpoint(s) both me and the Sheikh occupy.

Finally, let H be the component of G containing the vertex LR, which corresponds to both me and the Sheikh's starting position. By the Handshake Lemma, H must contain at least one of the other degree-1 vertices:

- If H contains RR, both me and the Sheikh can meet by following a path from LR to RR within H.
- If H contains LL, both me and the Sheikh can meet by following a path from LR to LL.
- If H contains RL, both me and the Sheikh can swap positions by following a path from LR to RL.

Since both me and the Sheikh move continuously, we must meet at some intermediate point. In all cases, me and the Sheikh will eventually meet.

(b) If the endpoints of the path are not below every other vertex, both of you may not be able to meet for the duel. Describe an algorithm to decide whether you can meet, without either of you breaking east-west eye contact or stepping off the path, given the arrays $X[0 \ldots n]$ and $Y[0 \ldots n]$ as input. You may do this by building the graph from the previous part. What problem do you need to solve on this graph? What is the algorithm's running time as a function of n?

Algorithm:

```
        # assuming a graph class is defined
        def bfs(graph, start, target_set):
            # Create a queue for BFS
            queue = deque([start])

            # Set to keep track of visited vertices
            visited = set([start])

            # Perform BFS
            while queue:
                current = queue.popleft()

                # Check if the current vertex is in the target set
                if current in target_set:
                    return True  # Found a path to a target vertex

                # Explore all neighbors of the current vertex
                for neighbor in graph[current]:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append(neighbor)

            # If no path to any target vertex is found
            return False

```

Listing 1: Algorithm to meet on Graph

First, we need to define the *stop*'s on the path. A stop occurs whenever $y = Y[i]$ for a given index i. Since there's at most one *stop* for each index i on each segment of the path, the total number of *stop*'s is proportional to $O(n^2)$. This means we can compute the set of all *stop*'s in $O(n^2)$ time. After determining all the *stop*'s, we sort them by their x-coordinate in $O(n^2 \log(n))$ time. Sorting helps us quickly find the neighbors of each *stop*. With the sorted *stop*'s, we can identify the neighbors in constant time, $O(1)$.

Next, we need to construct the graph, G. For each index i, we need to enumerate the vertex configurations, which consist of pairs of *stop*'s at a given y-coordinate $Y[i]$. Since we have $O(n)$ stops at each $Y[i]$, we'll list all possible pairs of *stop*'s for each configuration. This

is done by brute force, and since there are $O(n^3)$ such configurations, it takes $O(n^3)$ time to generate them. For each configuration, we can find up to 4 neighboring configurations in constant time using the sorted list of *stop*'s. Therefore, constructing the graph takes $O(n^3)$ time.

Now, let's consider how to check if me and the Sheikh can meet. The initial configuration of the Sheikh and I is labeled as s, and we define the set T as all configurations where they are together. We need to find if there's a path in the graph from the starting configuration s to any vertex in T. To do this, we perform a $breadth - first$ search (BFS) or $depth - first$ search (DFS) from the starting configuration. This will mark all vertices that are reachable from s. Then, we simply check if any of the vertices in T are marked as reachable. This final step can be done by brute force.

In total, the algorithm runs in $O(V + E) = O(n^3)$ time, where V represents the number of vertices and E represents the number of edges in the graph. This is the time complexity of the algorithm, which is manageable given the constraints. By following this approach, we can determine if the Sheikh and I can meet without stepping off the path or breaking eye contact.

Question 2

(a) Describe an algorithm to sort an arbitrary pile of n parathas using as few inversions as possible. Exactly how many inversions does your algorithm perform in the worst case?

Final Solution

```
    def flip(paratha_stack, k):
        # flip the top k parathas, including k
        paratha_stack[:k] = reversed(paratha_stack[:k])

    def sort_parathas(paratha_stack, n):
        for i in range(1, n):  # Loop from 1 to n-1
            # find position of the i-th largest paratha
            A = paratha_stack.index(max(paratha_stack[:n - i])) + 1

            # flip the top A parathas to bring the largest to the top
            flip(paratha_stack, A)

            # flip the top (n - i + 1) parathas to move it to the
    correct position
            flip(paratha_stack, n - i + 1)

        return paratha_stack

    # Main code
    print("Hello, welcome to my paratha solver!")
    # paratha_stack = [5, 2, 8, 4, 7]  # define a tuple of numbers
    here
    n = len(paratha_stack)
    sorted_parathas = sort_parathas(paratha_stack, n)
    print("Sorted parathas:", sorted_parathas)

```

Listing 2: Algorithm to sort n parathas

Inversions in Worst Case

$$2n - 2$$

The calculation is shown below at the end.

Explanation and Testing

To solve and deduce an algorithm to sort these parathas smallest to largest, we need to devise a plan on which way we will sort them. Either we sort them smallest to largest, or largest to smallest. Lets go over both ways.

Smallest to Largest:

Lets devise some steps to follow to try and come up with an algorithm:

i. I locate the smallest paratha in the stack and call it paratha A.

ii. I flip all the parathas from the top to paratha A, including A.

iii. Now my smallest paratha (A) is at the top. Now I locate the second smallest paratha in the stack and name it paratha B. This paratha B will essentially be below paratha A as A was at the top.

iv. Now if i flip all the parathas from the top to paratha B and including B, the paratha B comes at the top however paratha A is now at B's previous position, when it should be above B.

So I have reached a stop point, as if I am sorting smallest to largest, I am unable to stack them as the smallest paratha (A) will always be on the other side of the stack from the second smallest paratha (B) and I am unable to bring them together.
So **this way has failed**. Lets try the second way:

Largest to Smallest:

Lets devise some steps to follow to try and come up with an algorithm:

i. I locate the largest paratha in the stack and call it paratha C.

ii. I flip all the parathas from the top to paratha C, including C.

iii. Now my largest paratha (C) is at the top. So i flip all the parathas (n parathas) to make my largest paratha C at the bottom of the stack.

iv. Now I locate the second largest paratha in the stack which is bound to be ABOVE the largest paratha C as C was at the bottom. I name the second largest paratha D.

v. Now I repeat step 2, i flip all the parathas from top to paratha D including D, and it is confirmed that paratha C would not be in the flip as it is at the bottom.

vi. So paratha D is now at the top, I get all the parathas apart from paratha C which is $n-1$ parathas, and I flip them, which makes paratha D at the second-last-bottom, just above largest paratha C. Which is correct.

I can see that this could be an iterative process as each time my searching of largest paratha gets one paratha less, and the parathas are stacking correctly from bottom.

Tentative Algorithm

1) for every paratha i in the stack of n parathas:

2) locate the i^{th} largest paratha, name this paratha j

3) flip all the parathas from the top to the j^{th} paratha (including j), so flip j parathas to make the paratha j at the top

4) flip all the parathas from the $(n-i)^{th}$ paratha to the top, making sure the i^{th} largest paratha is at its correct position in terms of large-ness

Before we write this in proper algorithmic and code format, lets test it using an example:

Figure 1: 4 parathas with widths 3 cm, 6 cm, 5 cm and 4 cm from top to bottom

Solving an Example to test Algorithm

Step 1) Started for-loop with $i = 1$ which will run till $n = 4$ so it will run 4 times

Step 2) Identify the $1st$ (i^{th}) largest paratha: identified at $2nd$ position

Step 3) Flip all parathas from top to the i^{th} position which is flip the 1st and 2nd paratha

Step 4) Now largest is at top. flip all 4 parathas. BUT, we had said we would flip $n-i$ parathas which in this case is $4-1=3$ which is wrong, so lets **tentatively revise the formula** of flipping parathas to $n-i+1$.

Step 5) So now increment i to 2, and identify the $2nd$ largest paratha: identified at position 2

Step 6) Flip all parathas from top to i^{th} paratha, which is flip the 1st and 2nd paratha

Step 7) (using the new revised formula), flip $n-i+1$ parathas from top, which is $4-2+1=3$ which is correct. flip the top 3 parathas.

Step 8) Now we increment i to 3, and identify the $3rd$ largest paratha: identified at position 2

Step 9) Flip the top parathas till position 2 i.e. flip the 1st and 2nd paratha

Step 10) Now flip the top $n - i + 1$ which is $4 - 3 + 1 = 2$ parathas, which is correct

Step 11) So now after incrementing i, it is 4, which is equal to n, so only one paratha is left. We do not require anymore flips or computations. Thus another change to the algorithm is to make the $for - loop$ from $i = 1$ to $n - 1$ iterations.

Changes to the Tentative Algorithm

1) for-loop iterations reduced to $n - 1$
2) paratha flipping of $n - i + 1$ parathas

Final Algorithm

The proper runnable algorithm is written in *Python* at the start of the answer. A pseudocode is written here:

I. for $i = 1$ to $n - 1$:
II. locate the position of the i^{th} largest paratha as A
III. flip the top A parathas
IV. flip the top $n - i + 1$ parathas

Exact Flips Calculation

We have a for-loop which iterates $n-1$ times. At each iteration, we select the largest paratha which is $O(1)$. Then we do 2 flips at each iteration. So that is,

2 flips at each of $n - 1$ iterations
which is $2(n - 1) = 2n - 2$ flips at the worst case.

Method 2

Another way is to do $n-2$ iterations but an additional check is performed that if the top 2 parathas are out of order, they are flipped to make it in order.

The Algorithm is as follows:

I. for $i = 1$ to $n-2$:
II. locate the position of the i^{th} largest paratha as A
III. flip the top A parathas
IV. flip the top $n-i+1$ parathas
V. if the top 2 parathas are out of order:
VI. flip the top 2 parathas

The number of flips is 1 less than Method 1:

$$2 \text{ flips at each of } n-2 \text{ iterations } +1$$
$$\text{which is } 2(n-2)+1 = 2n-4+1 = 2n-3 \text{ flips at the worst case.}$$

(b) Now suppose one side of each paratha is burnt. Describe an algorithm to sort an arbitrary pile of n parathas, so that the burnt side of every paratha is facing down, using as few inversions as possible. Exactly how many inversions does your algorithm perform in the worst case?

Final Solution

```
    def flip(paratha_stack, k):
        # Flip the top k parathas, reversing order and burnt side
        paratha_stack[:k] = [(size, not burnt) for size, burnt in
reversed(paratha_stack[:k])]

    def sort_parathas(paratha_stack, n):
        for i in range(1, n):  # Loop from 1 to n-1
            # Find position of the i-th largest paratha
            A = max(range(n - i), key=lambda x: paratha_stack[x][0])
+ 1  # Compare by size

            # Flip to bring the largest to the top
            flip(paratha_stack, A)

            # Flip if the top paratha has the burnt side down
            if not paratha_stack[0][1]:  # Burnt side down? False?
                flip(paratha_stack, 1)

            # Flip to move it to the correct position
            flip(paratha_stack, n - i + 1)

        # If the top paratha has burnt side up, flip it:
        if paratha_stack[0][1]:
            flip(paratha_stack, 1)

        return paratha_stack

    # Main execution
    print("Hello, Welcome to my Paratha Solver!")
    paratha_stack = [(5, False), (2, True), (8, False), (4, True),
(7, False)]
    n = len(paratha_stack)
    sorted_parathas = sort_parathas(paratha_stack, n)
    print("Sorted parathas:", sorted_parathas)

```

Listing 3: Algorithm to sort n parathas

Inversions in Worst Case

$3n - 2$

The calculation is shown below at the end.

Explanation and Testing

Using the same algorithm as before, we can make one modification, that if the i^{th} largest paratha we flipped to the top, and if its burnt is downwards, we make it upwards, so that when it flips down, the burnt side flips downwards and inside, so that it is no longer visible. we also add one more condition that if after all are sorted and the top paratha is still burnt up, we make it burnt down and flip single paratha.

Lets write this in algorithm format:

I. for $i = 1$ to $n - 1$:
II. locate the position of the i^{th} largest paratha as A
III. flip the top A parathas
IV. if the top paratha is burnt side down:
V. flip the top paratha to make the burnt side up
VI. flip the top $n - i + 1$ parathas
VII. if the top 2 parathas are out of order:
VIII. flip the top 2 parathas

Lets test this algorithm using the same example as before:

Figure 2: 4 parathas with widths 3 cm, 6 cm, 5 cm and 4 cm from top to bottom, where black depicts burnt on top and blue depicts burnt down

Step 1) Started for-loop with $i = 1$ which will run till $n - 1 = 4 - 1 = 3$ so it will run 3 times
Step 2) Identify the $1st$ (i^{th}) largest paratha: identified at $2nd$ position
Step 3) Flip all parathas from top to the i^{th} position which is flip the 1st and 2nd paratha

Figure 3: The 6cm-paratha was burnt upwards, when flipped, it is now burnt downwards, so its blue

Step 4) Now largest is at top. Is the top paratha burnt downwards? No. Is it burnt upwards? Yes. No need to flip.

Step 5) Now flip the $n - i + 1$ parathas which is $4 - 1 + 1 = 4$ parathas.

Step 6) So now increment i to 2, and identify the $2nd$ largest paratha: identified at position 2

Step 7) Flip all parathas from top to i^{th} paratha, which is flip the 1st and 2nd paratha

Step 8) is the top paratha burnt downwards? no. is it burnt upwards? yes. so no need to flip.

Step 9) flip $n - i + 1$ parathas from top, which is $4 - 2 + 1 = 3$ which is correct. flip the top 3 parathas.

Step 10) Now we increment i to 3, and identify the $3rd$ largest paratha: identified at position 2

Step 11) Flip the top parathas till position 2 i.e. flip the 1st and 2nd paratha

Step 12) is the top paratha burnt downwards? no. is it burnt upwards? yes. so no need to flip.

Step 13) Now flip the top $n - i + 1$ which is $4 - 3 + 1 = 2$ parathas, which is correct

Step 14) We have completed 3 iterations. Done, and the parathas are sorted, with all burnt down.

Exact Flips Calculation

We have a for-loop which iterates $n-1$ times. At each iteration, we select the largest paratha which is $O(1)$. Then we do 3 flips at each iteration. And then at the end we perform a single flip. So that is,

$$\text{3 flips at each of } n-1 \text{ iterations } +1$$
$$\text{which is } 3(n-1)+1 = 3n-3+1 = 3n-2 \text{ flips at the worst case.}$$

Question 3

(a) How you can decide whether you should accept the challenge, that is, there exists a winning strategy. What is the complexity of your algorithm?

Algorithm:

```
    # Node class defined
    class TreeNode:
        # constructor
        def __init__(self, color=None, left=None, right=None,
    is_player_turn=True):
            self.color = color  # 'white' or 'black' for leaf nodes
            self.left = left
            self.right = right
            self.is_player_turn = is_player_turn  # Determines if
    it's the player's turn

        def is_leaf(self):
            return self.left is None and self.right is None

    # function to recursively search every node in tree
    def evaluate_tree(root):
        if root.is_leaf():
            return 1 if root.color == "white" else 0

        left_value = evaluate_tree(root.left)
        right_value = evaluate_tree(root.right)

        if root.is_player_turn:
            # Player chooses the best path (OR operation)
            return left_value or right_value
        else:
            # Sheikh forces worst-case scenario (AND operation)
            return left_value and right_value

    def should_accept_challenge(tree_root):
        return evaluate_tree(tree_root) == 1

    # MAIN CODE: Creating a sample tree (depth=2)
    leaf1 = TreeNode(color="white")
    leaf2 = TreeNode(color="black")
    leaf3 = TreeNode(color="black")
    leaf4 = TreeNode(color="white")

    node1 = TreeNode(left=leaf1, right=leaf2, is_player_turn=False)
```

```
    node2 = TreeNode(left=leaf3, right=leaf4, is_player_turn=False)
    root = TreeNode(left=node1, right=node2, is_player_turn=True)

    print("Winning Strategy:", should_accept_challenge(root))

```

Listing 4: Algorithm to find winning strategy against Sheikh

Complexity:

There are 4^n leaves and we spend $O(1)$ time at each one, so the complexity is $O(4^n)$.

Explanation:

In our Introduction to Artificial Intelligence course, we learned an AI-algorithm called "$Mini-Max$", which was used when we have two player games, when one is finding a path to win, we maximize one player and minimize the other.
This problem **does not use this algorithm**, however it uses the **same concept** of this algorithm.

The Concept:

In this algorithm, we want the Sheikh to always land in the end on a branch that has 2 whites. Even if it is 1 black and 1 white, the Sheikh would always go for black to make us lose. In 2 blacks, its a 100% we will lose. But even if theres 1 black, then 100% we will lose.
So we have reached a conclusion that we always, always want the Sheikh to come at 2 white leafs. If 2 white leafs don't exist, then we don't go down that path, or if they don't exist at all, we do not accept the challenge.
Since we are starting, we always want to pick the branch (left or right) that has 2 whites. If any of the two branches contain it, we can pick it.

Thus we are reaching towards a conclusion, that we don't want to give the Sheikh a choice, thus on its turn, we can say that both branches should have a winning point for us (2-whites) so we AND the value of both branches. But if its our turn, then if any of the two branches contain 2-whites, we can move forward, thus we OR the value of both branches.

To formulate this better, we can give each node a value. If a node is not a leaf, it is dependent on its children nodes. Each leaf can be either black ($value = 0$) or white ($value = 1$). The node gets assigned its value dependent if its a Sheikh turn or my turn. If its my turn, both the children's value's will be OR-ed and the node is assigned, and if its the Sheikh's turn, the children's value's will be AND-ed and the node is assigned.

But because I'm exploring every node and its children first, we backtrack up, just like in the "$Mini-Max$" algorithm. So this is a **recursive** algorithm, in which we are **backtracking** for all the values. We start from the leaves and work our way up, OR and AND-ing appropriately, to the root, and if the root is 1, it means from the root, I can take such a move that will always make me win, regardless of what the Sheikh takes. But if the root is 0, then there exists no such path that would make me win and I am dependent on the Sheikh's turns.

So lets first articulate the concept into a pseudocode algorithm:

```
I.   evaluate_tree(node):
II.      if node is leaf:
III.        if node is white:
IV.           return 1
V.          else:
VI.           return 0
VII.     left_node = evaluate_tree(node.left)
VIII.    right_node = evaluate_tree(node.right)
IX.      if node is my turn:
X.          return left_node OR right_node
XI.      else:
XII.        return left_node AND right_node
```

Lets try and solve the tree given in the question using this algorithm:

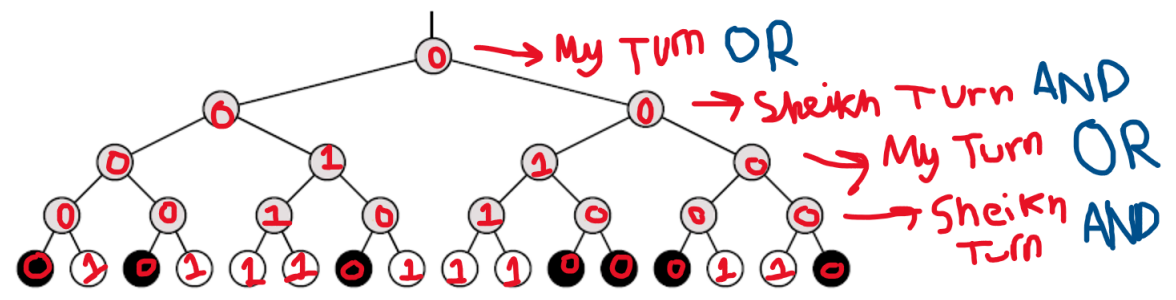

Figure 4: Solving this graph using the algorithm we concluded

So solving this tree using our algorithm, we found that this tree is not winnable for us. Whether we choose right or left at the start, we are dependent on the Sheikh for winning or losing (we have a higher probability of loosing as the Sheikh would choose a black path). We want to play **if and only if** the node we choose has all paths going to 2-whites to make us win.

In this example, the leaves were all AND-ed to get the fourth rows nodes values as it was the Sheikh's Turns. We don't want to go to a node that has even a single black - so all those nodes were made of value 0. Then in the third row, the nodes were OR-ed as it was our turn and even if there is one path that has whites, we can go to it. Then in the second row, it was again Sheikh's turn, so all were AND-ed. Lastly, in the first row, the root node, is our turn which we OR-ed.

Thus, this algorithm works.

As we are exploring all leaves and nodes, the Time Complexity is $O(4^n)$

(b) Unfortunately, you do not have enough time to look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $O(3^n)$ expected time.

Algorithm:

```
    def WINNING(v):
        # Base Case: If it's a leaf, return whether it's white (winning)
        if v.is_leaf():
            return v.color == "white"

        # Randomly decide the order of visiting children
        if random.random() < 0.5:
            left_first = True
        else:
            left_first = False

        if v.is_your_turn():  # OR Gate: You can win if at least one child is winning
            if left_first:
                if WINNING(v.left): return True
                return WINNING(v.right)
            else:
                if WINNING(v.right): return True
                return WINNING(v.left)

        else: # Sheikh Chilly's Turn: AND Gate, you win only if both children lead to a win
            if left_first:
                if not WINNING(v.left): return False
                return WINNING(v.right)
            else:
                if not WINNING(v.right): return False
                return WINNING(v.left)

        return False  # Default case (should never reach)

```

Listing 5: Algorithm to randomize tree

Complexity:

$O(3^n)$. Explanation is below.

Explanation:

Here I'll stick to the original view of the tree as alternating between AND and OR gates. To speed up the algorithm, we make two changes. First, we implement the logic more carefully to avoid redundant recursive calls. Second, we randomly decide whether to visit the grandchildren of v in left-to-right order (as in our deterministic algorithm) or in right-to-left order.

```
I.   Game(v):
II.    if v is a leaf:
III.     return [v is white]
IV.    with probability 1/2
V.       if Game(v.left.left) = True
VI.        if Game(v.left.right) = True
VII.         return True
VIII.    if Game(v.right.left) = True
IX.        return Game(v.right.right)
X.       return False
XI.    else
XII.     if Game(v.right.right) = True
XIII.      if Game(v.right.left) = True
XIV.         return True
XV.      if Game(v.left.right) = True
XVI.       return Game(v.left.left)
XVII.    return False
```

The left-to-right algorithm makes four recursive calls in exactly two cases:

- The grandchildren have values $True$, $False$, $True$, $False$. In this case, the right-to-left algorithm makes only two recursive calls (both returning $False$).
- The grandchildren have values $True$, $False$, $True$, $True$. In this case, the right-to-left algorithm makes only two recursive calls (the rightmost $Trues$).

Symmetrically, in both two cases where the right-to-left algorithm makes four recursive calls, the left-to-right algorithm makes only two. In every other case, both algorithms make at most three recursive calls. Thus, no matter what values the grandchildren have, the expected number of recursive calls is at most 3.

Let $T(n)$ denote the worst-case running time of this algorithm when v is the root of a tree with depth $2n$. Because random decisions at different levels of recursion are independent, this function satisfies the recurrence

$$T(n) \leq O(1) + 3 * T(n-1).$$

We conclude that our algorithm runs in $O(3^n)$ expected time, as required. (Moreover, the analysis is actually tight.)

Method 2:

Another way to solve this problem which I thought of was is that instead of going to every leaf and backtracking (which is time complexity $O(4^n)$, we would randomly pick a path. How? we would pick it by assigning 0.5 probability to every node. generate a random number and go for the any one of the children nodes. We do this till we get to a leaf. We also keep track of what nodes we went to by pushing each node into a stack. Then when we get to a leaf, if we reach a white leaf, meaning it is good for us and makes us win. Thus, this path was good. So we backtrack the path by popping the stack and updating the probabilities of all the nodes we passed - we increased their probabilities as these nodes were good for us. If we had reached a black leaf, it made us lose, thus this path was not good for us - and we do the same, we backtrack and decrease the probabilities of all the nodes.

We repeat this k times, we generate random numbers, go down a random path, check the leaf, backtrack from the leaf to the root on the same path, updating probabilites as we go.

If the majority of the paths we found were wins for us, then we say that we can win this game. If we found more losses than wins, then we say that we cannot win this game.

I found this similar to ada-boost algorithm in Introduction to Machine Learning course. However, I was not able to prove the time complexity of this algorithm. I think it would be:

each path is of 2n nodes, and each path requires (1) random number generation and then going backwards, (2) pop the node from stack, (3) update the probability. so $O(3)$ for each path. we run this algorithm for example n times. so $2n * 3 = 6n$ and then at the end we do 1 calculation for win-or-not, so $6n + 1$.

However, I am not sure of this algorithm. It makes sense to me however it may not be right as the time complexity is too small and there are no recursive calls.

Question 4

(a) Prove that this algorithm uniformly shuffles the deck, meaning each permutation of the deck has equal probability. You may do this by proving that at all times, the cards below card $n-1$ are uniformly shuffled.

Understanding:

So to begin this problem, we need to clarify what "**uniformly shuffled**" means. It means that during the algorithm, every possible permutation of cards is possible to come, and there is no bias. This means that if there are n cards, then there exists $n!$ permutations of them. The existense and occuring of each permutation has $\frac{1}{n!}$ probability.

Here we are given in question that we are holding one card out. That leaves $n-1$ cards, and we can insert that card into any of the n locations.
When the top card is removed and randomly re-inserted, each possible position for the inserted card is equally likely. This ensures that the relative order of the cards below it remains random. When card $n-1$ is inserted, it does not alter the relative order of cards below it, and it is randomly inserted into those cards. This maintains the uniform shuffling. The base case would be when there is only one card, or no cards, below card $n-1$, in which case the statement is trivially true.

For each permutation of these cards to exist has probability $\frac{1}{(n-1)!}$. Each permutation has equal probability of happening. The probability of insertion of this new card to any of the n locations is $1/n$. Thus the probability of every permutation-insertion is

$$= \frac{1}{(n-1)!} * \frac{1}{n}$$
$$= \frac{1}{n(n-1)!}$$
$$= \frac{1}{n!}$$

Final Statement:

Because the cards below card $n-1$ are uniformly shuffled at all times, and because the card $n-1$ is inserted randomly, the final arrangement of the cards (including card $n-1$) will be uniformly random.

Therefore, every permutation of the deck has equal probability.

(b) What is the exact expected number of steps executed by the algorithm?

As suggested by the hint, we break the execution of the algorithm into $n-1$ phases as in part (a). Let T_i denote the number of steps in the i^{th} phase, and let $T = \sum_{i=1}^{n-1} T_i$ denote the total number of steps. We immediately have $T_{n-1} = 1$, but the other T_i's are random variables.

Consider the ith phase, for some $i < n-1$. In any step in this phase, the top card is inserted uniformly at random into one of n locations, and exactly $i+1$ of these locations are under card $n-1$. Thus, we have:

$$E[Ti] = 1 + \frac{n-i-1}{n} E[T_i],$$

which implies $E[T_i] = \dfrac{n}{i+1}$.
Linearity of expectation now implies that

$$E[T] = \sum_{i=1}^{n-1} E[T_i] = \sum_{i=1}^{n-2} \frac{n}{i+1} + 1 = nH_{n-1} - n + 1$$