2.)

1.) OPT(p) represents the maximum amount of donations we can from the subtree rooted at person p.

2.A) RECURSIVE FORMULA

Let…

include(p)🡐 OPT (T[p].left.right) + OPT(T[p].left.left) + OPT(T[p].right.left) + OPT(T[p].right.right) + D[p]
include_left(p)🡐 OPT (T[p].left.right) + OPT(T[p].left.left) +D[p]
exclude(p)🡐 OPT(T[p].left)+ OPT(T[p].right)

Then…

	{ **D[p]**	if T[p].right ==null and T[p].left ==null
OPT(p)=	{ **max{ include_left(p); exclude(p)}**	if T[p].right==null
	{ **max{ include (p); exclude(p)}**	else

2.B) PROOF

<u>BASE CASE:</u> When the node being considered is a leaf node, we choose the max of excluding the node and including the node, max{D[p],0}= D[p], so the base case holds true.

<u>INDUCTIVE HYPOTHESIS:</u> The assertion holds true for all subtrees with parent p in T. Consider the proof by cases.

CASE 1 (p has one child): In this instance, we choose the max of including p or not. Given by the problem that a node with one child has solely a left child, we compare the donations received by choosing the child and excluding p, versus choosing p and including it's grandchildren. Since this operation maximizes donations at p, the formula holds.

CASE 2(p has two children): In this instance, we choose the max of including p or not. Given that a node can have at most two children, we compare the donations received by choosing both children, versus choosing p and all (possibly) four grandchildren. Since this operation maximizes donations at p, the formula holds.

<u>CAVEATS:</u> We assume that if a node doesn't exist, the formula returns zero for its optimal donation, as is with the base case. Additionally, we see each step maximizes the donations while adhering to the rules outlined in the problem where no direct parent, child pairs are both donating.

<u>INDUCTIVE STEP:</u> By induction, we prove that the formula produces an optimal output at each subtree with parent p. Therefore, the formula is correct for all nodes in the tree, which optimizes donations from the company.

3.A) ALGORITHM

```
def MaxDonations (T,D):
        M ←{};
        M[none] ←0; #for all null nodes, this way looking at null nodes won't cause issues
        return recMaxDonations('CEO', T, D, M);
```

```
def recMaxDonations(p,T,D,M):
        #already computed values or null nodes return immediately
        if p in M do:
                return M[p];
        #fill include values with appropriate donations
        include← D[p]

        if T[p].left do:
                include += recMaxDonations(T[p].left.left,T,D,M) +
                                recMaxDonations(T[p].left.right,T,D,M);
        if T[p].right do:
                include += recMaxDonations(T[p].right.left,T,D,M) +
                                recMaxDonations(T[p].right.right,T,D,M);
        #similarly, fill exclude
        exclude ← recMaxDonations(T[p].left,T,D,M) + recMaxDonations(T[p].right,T,D,M);
        #only choose max to add to memorization table
        M[p]= max(include, exclude);
        return M[p]
```

3.B) COMPLEXITY

SPACE COMPLEXITY: To analyze space complexity, we look at all temporary structures and the recursive stacks.

1) OPT array: takes O(n) space since we store one optimal donation amount per person node in T
2) Recursion stack: takes O(logn) space because we recurse through a balanced tree of height logn.

Overall, the space complexity is O(n + logn) = O(n) considering the largest term.

TIME COMPLEXITY: To analyze time complexity, we look at each step of the algorithm

1) Include←: filling the included value makes four recursive calls in the worst case, 2 on every child. Therefore, in the worst case, these operations will take O(4n) = O(n) time.
2) Exclude←: filling the excluded value makes two recursive calls in the worst case, one each child. Therefore, in the worst case, these operations will take O(2n) = O(n) time.

Since all other operations run in constant, or O(1) time, the overall time complexity O(n).

4.A) ALGORITHM

```
def get_donation_list (T, D):
    people ← empty set
    backtracking('CEO', T, D, M, people);
    return people;
```

```
def backtracking (p,T, D, M, people):
    if p ==null do:
        return;

    #fill include values with appropriate donations
    include← D[p]
    if T[p].left do:
        include += M[T[p].left.left] + M[T[p].left.right];
    if T[p].right do:
        include += M[T[p].right.left] +  M[T[p].right.right];

    #similarly, fill exclude
    exclude ← M[T[p].left] + M[T[p].right];

    if include>exclude do:
        #p was chosen
        people.append(p);
        If T[p].left do:
            Backtracking(T[p].left.left, T, D, M, people);
            Backtracking(T[p].left.right, T, D, M, people);
        If T[p].right do:
            Backtracking(T[p].right.left, T, D, M, people);
            Backtracking(T[p].right.right, T, D, M, people);
    else do: #p was not chosen
        Backtracking(T[p].left, T, D, M, people);
        Backtracking(T[p].left, T, D, M, people);
```

4.B) COMPLEXITY

SPACE COMPLEXITY: To analyze space complexity, we look at all structures and the recursive stack. Like the other algorithm, all strucutres take at most O(n) space, while the recursive stack reaches at most O(logn) space, so taking the largest term we say space complexity is given by O(n).

TIME COMPLEXITY: To analyze time complexity, we analyze each step of the algorithm. In the worst case, the backtracking visits each node once through the recursive calls. All other operations including building include and exclusive operations in constant O(1) time; therefore, the overall time complexity can be given as O(n).