1 Max Sum Path

1.1 Define the Subproblems

Let v denote some node on the tree $T(r)$ rooted at r. Let O_v denote the optimal solution(maximum in this case) to the subproblem of a path starting from the node v to one leaf node on the tree $T(v)$. Then $OPT(v)$ is the summed value of those nodes on the O_v path.

1.2 Define the Recurrence

Define the following notations:
V_v: the value of the node v
children(v): the set consisting of all children nodes of the node v
The value of the optimal solution for the path rooted at node v is the sum of the value of the node v and the maximum value of optimal solutions for the paths rooted at a child node of v. We denote that

$$OPT(v) = \max_{n \in \text{children}(v)} (OPT(n)) + V_v$$

1.3 Define the Base Cases

The value of the optimal solution for a path starting from a leaf node(in other words, a node having o children) of $T(r)$ is the value of the leaf node:

$$OPT(v) = V_v \mid v \in \text{leafnodes}(T(r))$$

1.4 Algorithm Description

The main idea is to iterate through the tree firstly starting from the leaf nodes and assign OPT values as the values of the leaf nodes. If the other sibling node of the leaf node has been iterated, use the recurrence defined in section 1.2 to update OPT of its parent node. Here is a concrete recipe:

1. Prepare a memory space to store OPT values. As an efficient data structure, a hash mapping table OPT can map a node v to its corresponding value $OPT(v)$. Because we have assumed that all values are positive, we could prepare a default table initialized with a key-value pair of $v : 0$. Before next step, we have a table $OPT[v] = 0$ for any node v in the given tree $T(r)$.

2. As I have explained a bit, the tree can be iterated in post-order traversal.

3. Every time a node is visited, an operation is done depending on its node type

    - a leaf node
      store the value of the node to OPT table as $OPT[v] = V_v$

- an internal node
  store the sum of the node value and the larger OPT value of its children(given that it's a binary tree).

  $$OPT[v] = V_v + \max_{n \in \{l,r\}} (OPT[n]) \mid l := \text{left child}(v) \text{ and } r := \text{right child}(v)$$

4. The maximum value of the whole tree $T(r)$ is $OPT[r]$, the value stored in the OPT table for the root node r.

1.5 Correctness

The post-order traversal of a tree visits all tree nodes, specifically, a parent node is always visited after all its children nodes are visited. Therefore, the recurrence is always valid because of the visiting order. The base cases are reached since every tree structure spans at leaf nodes. Since I use an iterative dynamic programming technique, the proof of the correctness can be achieved by induction.

- base cases
  A path starting from a leaf node(of course, in this case, the path only goes downwards) is the lead node itself, thus, the summed value is optimal and is equivalent to the value of the node.

- recurrence
  For simplicity, suppose that a node v has exactly two children since the cases, in which a node has only 1 child, are trivial because values assumed to be positive(therefore the sum is always larger). In addition, the values for the pair of sibling nodes $OPT(l)$ and $OPT(r)$ are optimal. Without loss of generosity, suppose that $OPT(l) \geq OPT(r)$. As a result, $OPT(v) = V_v + OPT(l)$ is optimal; otherwise that $OPT(v)$ is not optimal $\Rightarrow V_v + OPT(l) < V_v + OPT(r) \Rightarrow OPT(l) < OPT(r)$ would contradict with the assumption that $OPT(l) \geq OPT(r)$.

1.6 Time Complexity

1. prepare OPT mapping
   O(n) as preparing n key-value pairs

2. traversal
   O(n) as each node in the tree is exactly visited once.

3. update operation at each node
   O(1) if we assume that condition checks, arithmetic operations, hash table lookup(hash table space larger than size of the tree so keys won't collide) use constant time. Specially, checking if a node is a leaf node or an internal node use a constant time since we assume that the tree is given. If such information is not implemented, a pre-process of time complexity O(n) of a tree traversal to store node type information before the post-order traversal OPT updates.

4. look up the optimal value for the root node from the mapping
   O(1) for a hash table

In total, $\in$ O(n) + C·O(n) + O(1) = O(n), where C is a constant.

2 Red-Black Max Sum Subsequence

2.1 Define the Subproblems

Since an extra attribute, color, is required, it is intuitive to define more subproblems in this case than those in the Max Sum Path problem. To create more subproblems for subsets, naturally we can borrow the idea of adding a variable for the colors. Therefore, we have a subproblem for each node v and each color c.
Building upon the prototype in section 1.1, we can let v denote some node on the tree $T(r)$ rooted at r. Let O denote the optimal solution(maximum in this case) to the subproblem of a tree rooted at v with a color reference c(to understand the reference, think about that the shallowest node on the O path has a color of c). Color of nodes on the O path are not the same if they are consecutive, in this case, binary altering from one to the other(red to black or black to red). Then $OPT(v, c)$ is the summed value for nodes on the solution O of which the shallowest node(the closet to v) has a color of c.

2.2 Define the Recurrence

Define the following notations:
V_v: the value of the node v
children(v): the set consisting of all children nodes of the node v

For red color references, we have

$$OPT(v, \text{red}) = \begin{cases} \max_{n \in \text{children}(v)}(OPT(n, \text{black})) + V_v & \text{if the color of } v \text{ is red} \\ \max_{n \in \text{children}(v)}(OPT(n, \text{black})) & \text{if the color of } v \text{ is black} \end{cases}$$

, where $OPT(n, black)$ is chained symmetrically as

$$OPT(v, \text{black}) = \begin{cases} \max_{n \in \text{children}(v)}(OPT(n, \text{red})) + V_v & \text{if the color of } v \text{ is black} \\ \max_{n \in \text{children}(v)}(OPT(n, \text{red})) & \text{if the color of } v \text{ is red} \end{cases}$$

2.3 Define the Base Cases

For any leaf node v

$$OPT(v, \text{black}) = \begin{cases} V_v & \text{if the color of } v \text{ is black} \\ 0 & \text{if the color of } v \text{ is red} \end{cases}$$

and

$$OPT(v, \text{red}) = \begin{cases} V_v & \text{if the color of } v \text{ is red} \\ 0 & \text{if the color of } v \text{ is black} \end{cases}$$

2.4 Algorithm Design

Analogous to section 1.4, we use a key of a pair tuple instead and modify the operations when a node is being visited. And finally we chose the larger value between the optimal solution starting from a red node and the optimal solution starting from a black node.

1. Prepare a memory space to store OPT values. As an efficient data structure, a hash mapping table OPT can map a tuple of a node v and a color reference c to a corresponding value $OPT(v, c)$. Because we have assumed that all values are positive ($\forall v \in T(r) : V_v > 0$), we could prepare a default table initialized with a key-value pair of $\{(v, c) : 0\}$. Before next step, we have a table $OPT[(v, c)] = 0$ for any combination of a node v in the given tree $T(r)$ and a color reference c. Here is the recipe.

2. The entire tree is iterated through in post-order traversal assuming that whether a node is a leaf node or an internal one is given by the tree itself. (If the assumption is not valie, traverse the whole tree to get such information before this step).

3. Every time a node is visited during the traversal in step 2, an operation is done depending on its node type

   - a leaf node
     store the value of the node to OPT table with two color references as:

$$OPT[(v, \text{red})] = \begin{cases} V_v & \text{if the color of } v \text{ is red} \\ 0 & \text{if the color of } v \text{ is black} \end{cases}$$

$$OPT[(v, \text{black})] = \begin{cases} V_v & \text{if the color of } v \text{ is black} \\ 0 & \text{if the color of } v \text{ is red} \end{cases}$$

   - an internal node
     For combinations of the node and two different color references, use the recurrence defined previously and store the value in the OPT hash map.

4. The global optimal value is the larger one of $OPT[(r, \text{red})]$ and $OPT[(r, \text{black})]$.

2.5 Correctness Analysis

- Sufficiency
  For each node in the tree, it is either an internal node or a leaf node. Per the definition of an internal node, an internal node is not a leaf node and it always has a leaf node below somewhere in the tree. Thus in terms of the recurrence, it always ends at a base case for a leaf node. Concretely, $\forall v \in T(r), \forall c \in \{\text{black}, \text{red}\} : OPT(v, c)$ ends up with either $OPT(l, \text{black})$ or $OPT(l, \text{red})$ where l is a leaf node in the tree.

- Base case values
  If there's only one node, and we assume a single node can be regarded as an alternating path, the max sum is trivially the value of that node. For an alternating path starting with the color of the node, OPT value is the value of the node consequently. For alternating paths starting with other colors, we define OPT to be o since there are no such nodes with the colors.

- Recurrence relationships
  For simplicity meanwhile without loss of generosity, here we only analyze a red node v and omit black nodes as they are symmetrical.
  Situation 1 Given the internal node v is red and to find a max sum alternating subsequence starting from a red node $OPT(v, \text{red})$

  $$\max_{n \in \text{children}(v)} (OPT(n, \text{black})) + V_v$$

  we have to add up V_v since $V_v > 0$. The next node on the subsequence has to be black(so we check $OPT(n, \text{black})$ and not the node itself(so we restrict n as descendants of v). In fact, we only need to check children of v instead of descendants of v because if a solution is optimal for a children node, a part of the solution path has to include an optimal path for non-children descendant nodes. In other words, $\forall d \in \text{descendant}(a) : OPT(a, c) \geq OPT(d, c)$. Otherwise, it contradicts to the definition of a maximum or a is not an ancestor of d.
  Situation 2 Given the internal node v is red and to find a max sum alternating subsequence starting from a black node $OPT(v, \text{black})$.

  $$\max_{n \in \text{children}(v)} (OPT(n, \text{red}))$$

  In this case, the colors don't match. So we have to use OPTs of its children starting with a red node(opposite to black).

- algorithm correctness
  By a post-order traversal, all OPT values for descendants of a node have been processed before visiting the node. Thus, the operands for OPT for the nodes have already prepared and can be got from a hash map looking up.

- global maximum
  The optimal solution has to either start with a red node or a black node. Therefore, global optimal value $GOPT \geq OPT[(r, \text{red})]$ and $GOPT \geq OPT[(r, \text{black})]$ as $GOPT = \max(OPT[(r, \text{red})], OPT[(r, \text{black})])$, is optimal, augmented with the proof in the section Recurrence relationships.

  $$\forall d \in \text{descendant}(r) \forall c \in \{\text{red}, \text{black}\} \mid GOPT \geq OPT(r, c) \geq OPT(d, c)$$

2.6 Time Complexity

1. prepare OPT mapping
   O(n) as preparing n key-value pairs

2. traversal
   O(n) as each node in the tree is exactly visited once.

3. update operation at each node
   O(1) if we assume that condition checks, arithmetic operations, hash table lookup(hash table space larger than size of the tree so keys won't collide) use constant time. Specially, checking if a node is a leaf node or an internal node use a constant time since we assume that the tree is given. If such information is not implemented, a pre-process of time complexity O(n) of a tree traversal to store node type information before the post-order traversal OPT updates.

4. look up and compare
   Looking up for the optimal values for (root node, red) and (root node, black) from the hash table OPT O(1). $GOPT = \max(OPT[(r, \text{red})], OPT[(r, \text{black})])$ O(1)

In total, $\in$ O(n) + C·O(n) + O(1) = O(n), where C is a constant.
If a node type preprocessing is required (O(n)), the overall time complexity is still O(n).