Question to practice:

1. First one was to check weather a valid BST exists given its pre-order traversal.
2. Find  maximum number of consecutive days when all the employees were present. Matrix data was given. data[i][j] denotes the j$^{th}$ employee's attendance on i$^{th}$ day.
3. Minimum number of moves required for knight to reach the ending position from given starting position.
4. Find the longest subsequence of S that is palindrome.
5. First question was you are given a array of string followed by two words. You have to find the minimum distance between the two words in the given array of string.
6. Find total number of visible nodes in a binary tree. A node is visible, if it has highest value in the path from root to node. It was done in O(n).
7. Find leftmost unique element in the array. eg. 2 5 2 3 5 6 8 . Answer is 3 for given example. I have done with the help of the hashmap in O(n) time complexity.

8. https://leetcode.com/problems/shortest-word-distance-ii/

9. https://leetcode.com/problems/nested-list-weight-sum-ii/

10. https://leetcode.com/problems/all-oone-data-structure/

11. https://leetcode.com/problems/max-stack/

12. https://leetcode.com/problems/nested-list-weight-sum/

13. https://leetcode.com/problems/minimum-one-bit-operations-to-make-integers-zero/

14. https://leetcode.com/problems/beautiful-arrangement/

15. https://leetcode.com/problems/shortest-word-distance/

16. https://leetcode.com/problems/binary-tree-upside-down/

17. https://leetcode.com/problems/find-leaves-of-binary-tree/

18. https://leetcode.com/problems/can-place-flowers/

19. https://leetcode.com/problems/paint-house/

## Ques 8.

Design a data structure that will be initialized with a string array, and then it should answer queries of the shortest distance between two different strings from the array.

Implement the `WordDistance` class:

- `WordDistance(String[] wordsDict)` initializes the object with the strings array `wordsDict`.
- `int shortest(String word1, String word2)` returns the shortest distance between `word1` and `word2` in the array `wordsDict`.

**Example 1:**

```
Input

["WordDistance", "shortest", "shortest"]

[[["practice", "makes", "perfect", "coding", "makes"]], ["coding", "practice"],
["makes", "coding"]]

Output

[null, 3, 1]



Explanation

WordDistance wordDistance = new WordDistance(["practice", "makes", "perfect",
"coding", "makes"]);

wordDistance.shortest("coding", "practice"); // return 3

wordDistance.shortest("makes", "coding");    // return 1
```

**Constraints:**

- $1 <= wordsDict.length <= 3 * 10^4$
- `1 <= wordsDict[i].length <= 10`
- `wordsDict[i]` consists of lowercase English letters.
- `word1` and `word2` are in `wordsDict`.
- `word1 != word2`
- At most `5000` calls will be made to `shortest`.

## Solution

Before looking at the solution for this problem, let's look at what the problem asks us to do in simpler terms. We have to design a class which receives a list of words as input in the constructor. The class has a function which we need to implement and that function is `shortest` which takes two words as input and returns the minimum distance between the two as the output.

When the problem talks about the distance between two words, it essentially means the absolute gap between the indices of the two words in the list. For e.g. if the first word occurs at a location `i` and the second word occurs at the location `j`, then the distance between the two would be `abs(i - j)`.

The question asks us to find the `minimum` such different between words which clearly indicates that the words can occur at multiple locations. If we have `K` occurrences for the `word1` and `L` occurrences for the `word2`, then iteratively checking every pair of indices will give us a O(N^2)$O(N2)$ algorithm which won't be optimal at all. We won't discuss that algorithm here since it is very straightforward.

The brute-force algorithm would simple consider all possible pairs of indices for (`word1_location`, `word2_location`) and see which one produces the minimum distance. Let's try and build on this idea and see if some pre-processing can help us out reduce the complexity of the brute-force algorithm.

---

## Approach 1: Using Preprocessed Sorted Indices

**Intuition**

A given word can occur multiple times in the original word list. Let's suppose the first word, `word1` in the input to the function `shortest` occurs at the indices `[i1, i2, i3, i4]` in the original list. Similarly, let's assume that the second word, `word2`, appears at the following locations inside the word list `[j1, j2, j3]`.

Now, given these list of indices, we are to simply find the pair of indices `(i, j)` such that their absolute difference is minimum.

The main idea for this approach is that if the list of these indices is in sorted order, we can find such a pair in linear time.
The idea is to use a two pointer approach. Let's say we have a pointer `i` for the sorted list of indices of `word1` and `j` for the sorted list of indices of `word2`. At every iteration, we record the difference of indices i.e. `abs(word1[i] - word2[j])`. Once we've done that, we have two possible choices for progressing the two pointers.

```
word1[i] < word2[j]
```

If this is the case, that means there is no point in moving the `j` pointer forward. The location indices for the words are in a sorted order. We know that `word2[j + 1] > word2[j]` because these indices are sorted. So, if we move `j` forward, then the difference `abs(word1[i] - word2[j + 1])` would be even greater than `abs(word1[i] - word2[j])`. That doesn't help us since we want to find the minimum possible distance (difference) overall.

So, if we have (word1[i] < word2[j]), we move the pointer 'i' one step forward i.e. (i + 1) in the hopes that abs(word1[i + 1] - word2[j]) would give us a lower distance than abs(word1[i] - word2[j]). We say "hopes" because it is not certain this improvement would happen.
Let's look at two different examples. In the first example we will see that moving `i` forward gave us the best difference overall (0). In the second example we see that moving `i` forward leads us to our second case (yet to discuss) but doesn't lead to any improvement in the difference.

**Example-1**

```
word1_locations = [2,4,5,9]

word2_locations = [4,10,11]
```

```
i, j = 0, 0

min_diff = 2 (abs(2 - 4))

word1[i] < word2[j] i.e. 2 < 4

  move i one step forward


i, j = 1, 0 (abs(4 - 4))

min_diff = 0 (We hit the jackpot!)
```

**Example-2**

```
word1_locations = [2,7,15,16]

word2_locations = [4,10,11]


i, j = 0, 0

min_diff = 2 (abs(2 - 4))

word1[i] < word2[j] i.e. 2 < 4

  move i one step forward


i, j = 1, 0

min_diff = 2 (2 < abs(7 - 4))


Here, we did not update out global minimum difference.

That is why we said earlier, moving 'i' forward may or

may not give a lower difference. But moving 'j' forward in

our case would definitely worsen the difference (or keep it same!).
```

Let's move onto our second scenario.

```
word1[i] > word2[j]
```

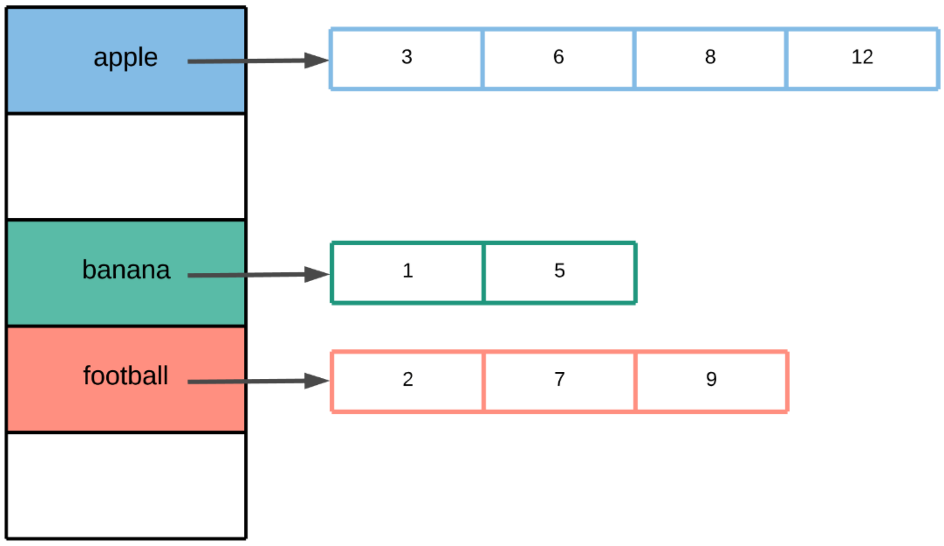
If this is the case, that means there is no point in moving the `i` pointer forward. We know that `word1[i + 1] > word2[j]` because these indices are sorted. So, if we move `i` forward, then the difference `abs(word1[i + 1] - word2[j])` would be even greater than `abs(word1[i] - word2[j])`. That doesn't help us since we want to find the minimum possible distance (difference) overall.

So, along the similar lines of thought as the previous case, if we have (word1[i] > word2[j]), we move the pointer 'j' one step forward i.e. (j + 1) in the hopes that abs(word1[i] - word2[j + 1]) would give us a lower distance than abs(word1[i] - word2[j]). We say "hopes" because as showcased in the previous scenario, it is not certain this improvement would happen.

Now let's formally look at the algorithm for solving this problem.

**Algorithm**

1. In the `constructor` of the class, we simply iterate over the given list of words and prepare a dictionary, mapping a word to all it's locations in the array.
2. Since we process all the words from left to right, we will get all the indices in a sorted order by default for all the words. So, we don't have to sort the indices ourselves.
3. Let's call the dictionary that we build, `locations`.
4. For a given pair of words, obtain the list of indices (appearances inside the original list/array of words). Let's call the two arrays `loc1` and `loc2`.
5. Initialize two pointer variables `l1 = 0` and `l2 = 0`.
6. For a given `l1` and `l2`, we first update (if possible) the minimum difference (distance) till now i.e. `dist = min(dist, abs(loc1[l1] - loc2[l2]))`. Then, we check if `loc1[l1] < loc2[l2]` and if this is the case, we move `l1` one step forward i.e. `l1 = l1 + 1`. Otherwise, we move `l2` one step forward i.e. `l2 = l2 + 1`.
7. We keep doing this until all the elements in the smaller of the two location arrays are processed.
8. Return the global minimum distance between the words.



This represents the locations dictionary that we should build given the original words list in the constructor. The key represents the word and the value is a list containing indices in ascending order of occurrences throughout the array. Let's look at the minimum distance between the words `apple` and `football` in the array. So, we will be considering the two *sorted* lists of indices: `[3, 6, 8, 12]` and `[2, 7, 9]`.

**Complexity analysis**

- Time complexity : The time complexity of the constructor of our class is $\mathrm{O(N)}O(N)$ considering there were $\mathrm{N}N$ words in the original list. We iterate over them and prepare a mapping from key to list of indices as described before. Then, for the function that finds the minimum distance between the two words, the complexity would be $\mathrm{O(max(K, L))}O(max(K,L))$ where $\mathrm{K}K$ and $\mathrm{L}L$ represent the number of occurrences of the two words. However, $\mathrm{K = O(N)}K=O(N)$ and also $\mathrm{L = O(N)}L=O(N)$. Therefore, the overall time complexity would also be $\mathrm{O(N)}O(N)$. The reason the complexity is $\mathrm{O(max(K, L))}O(max(K,L))$ and not $\mathrm{O(min(K, L))}O(min(K,L))$ is because of the scenario where the minimum element of the smaller list is larger than `all` the elements of the larger list. In that scenario, the pointer for the smaller list will not progress at all and the one for the longer list will reach to the very end.

- Space complexity: O(N)$O(N)$ for the dictionary that we prepare in the constructor. The keys represent all the unique words in the input and the values represent all of the indices from 0 … N0…$N$.

## Ques 9.

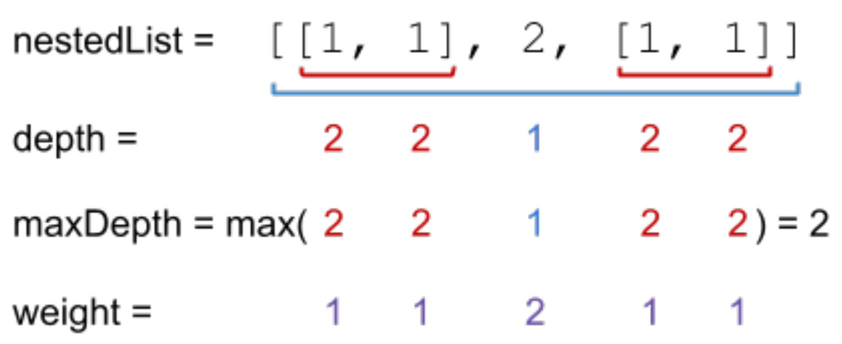
You are given a nested list of integers `nestedList`. Each element is either an integer or a list whose elements may also be integers or other lists.

The **depth** of an integer is the number of lists that it is inside of. For example, the nested list `[1,[2,2],[[3],2],1]` has each integer's value set to its **depth**. Let `maxDepth` be the **maximum depth** of any integer.

The **weight** of an integer is `maxDepth - (the depth of the integer) + 1`.

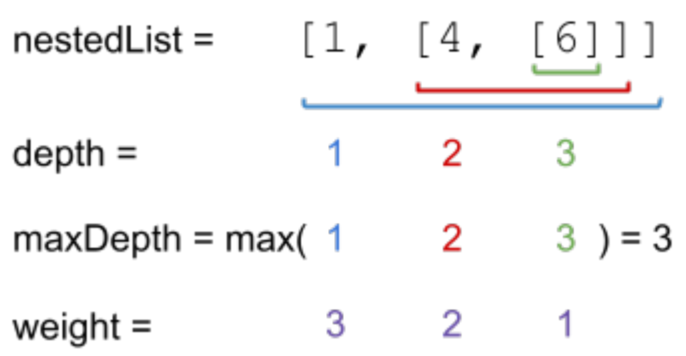
Return *the sum of each integer in* `nestedList` *multiplied by its* ***weight***.

**Example 1:**


nestedList = [[1, 1], 2, [1, 1]]
depth = 2 2 1 2 2
maxDepth = max( 2 2 1 2 2) = 2
weight = 1 1 2 1 1

```
Input: nestedList = [[1,1],2,[1,1]]

Output: 8

Explanation: Four 1's with a weight of 1, one 2 with a weight of 2.

1*1 + 1*1 + 2*2 + 1*1 + 1*1 = 8
```

**Example 2:**



```
Input: nestedList = [1,[4,[6]]]

Output: 17

Explanation: One 1 at depth 3, one 4 at depth 2, and one 6 at depth 1.

1*3 + 4*2 + 6*1 = 17
```

**Constraints:**

- `1 <= nestedList.length <= 50`
- The values of the integers in the nested list is in the range `[-100, 100]`.
- The maximum **depth** of any integer is less than or equal to `50`.

## SOLUTION

Basically we are going to deep whenever we encountered a list other wise we are adding value to the list.
While adding values, we are calculating depth as well.

```
class Solution {

    int maxDepth = 1;

public:

    void dfs(NestedInteger& nObj,int curDepth,vector<int>& depth,vector<int>&
values) {

        maxDepth = max(curDepth,maxDepth);

        for (auto val:nObj.getList()) {

            if (val.isInteger()) {

                values.push_back(val.getInteger());

                depth.push_back(curDepth);

            } else {

                dfs(val,curDepth+1,depth,values);

            }

        }

    }

    int depthSumInverse(vector<NestedInteger>& nestedList) {

        vector<int> depth,weight;

        vector<int> vals;

        int curDepth = 1;

        for(auto val:nestedList) {

            if (val.isInteger()) {

                vals.push_back(val.getInteger());

                depth.push_back(curDepth);

            } else {

                dfs(val,curDepth+1,depth,vals);

            }
        }
        for(auto dep:depth) {
            weight.push_back(maxDepth-dep+1);
        }

        int result = 0;
        for(int index=0; index<vals.size(); index++) {
            result += weight[index]*vals[index];
        }
        return result;
    }
};
```

## QUES 11

Design a max stack data structure that supports the stack operations and supports finding the stack's maximum element.

Implement the `MaxStack` class:

- `MaxStack()` Initializes the stack object.
- `void push(int x)` Pushes element `x` onto the stack.
- `int pop()` Removes the element on top of the stack and returns it.
- `int top()` Gets the element on the top of the stack without removing it.
- `int peekMax()` Retrieves the maximum element in the stack without removing it.
- `int popMax()` Retrieves the maximum element in the stack and removes it. If there is more than one maximum element, only remove the **top-most** one.

**Example 1:**

```
Input

["MaxStack", "push", "push", "push", "top", "popMax", "top", "peekMax", "pop",
"top"]

[[], [5], [1], [5], [], [], [], [], [], []]

Output

[null, null, null, null, 5, 5, 1, 5, 1, 5]



Explanation

MaxStack stk = new MaxStack();

stk.push(5);   // [5] the top of the stack and the maximum number is 5.

stk.push(1);   // [5, 1] the top of the stack is 1, but the maximum is 5.

stk.push(5);   // [5, 1, 5] the top of the stack is 5, which is also the maximum,
because it is the top most one.

stk.top();     // return 5, [5, 1, 5] the stack did not change.

stk.popMax();  // return 5, [5, 1] the stack is changed now, and the top is
different from the max.

stk.top();     // return 1, [5, 1] the stack did not change.

stk.peekMax(); // return 5, [5, 1] the stack did not change.

stk.pop();     // return 1, [5] the top of the stack and the max element is now 5.

stk.top();     // return 5, [5] the stack did not change.
```

**Constraints:**

- $-10^7 <= x <= 10^7$
- At most $10^4$ calls will be made to `push`, `pop`, `top`, `peekMax`, and `popMax`.

- There will be **at least one element** in the stack when `pop`, `top`, `peekMax`, or `popMax` is called.

## SOLUTION

### Approach #1: Two Stacks [Accepted]

**Intuition and Algorithm**

A regular stack already supports the first 3 operations, so we focus on the last two.

For `peekMax`, we remember the largest value we've seen on the side. For example if we add `[2, 1, 5, 3, 9]`, we'll remember `[2, 2, 5, 5, 9]`. This works seamlessly with `pop` operations, and also it's easy to compute: it's just the maximum of the element we are adding and the previous maximum.

For `popMax`, we know what the current maximum (`peekMax`) is. We can pop until we find that maximum, then push the popped elements back on the stack.

Our implementation in Python will showcase extending the `list` class.

```
class MaxStack {

    Stack<Integer> stack;

    Stack<Integer> maxStack;



    public MaxStack() {

        stack = new Stack();

        maxStack = new Stack();

    }



    public void push(int x) {

        int max = maxStack.isEmpty() ? x : maxStack.peek();

        maxStack.push(max > x ? max : x);

        stack.push(x);

    }



    public int pop() {

        maxStack.pop();

        return stack.pop();

    }
    public int top() {

        return stack.peek();

    }


    public int peekMax() {

        return maxStack.peek();

    }


    public int popMax() {

        int max = peekMax();

        Stack<Integer> buffer = new Stack();

        while (top() != max) buffer.push(pop());

        pop();

        while (!buffer.isEmpty()) push(buffer.pop());

        return max;

    }

}
```

**Complexity Analysis**

- Time Complexity: O(N)$O(N)$ for the `popMax` operation, and O(1)$O(1)$ for the other operations, where N$N$ is the number of operations performed.
- Space Complexity: O(N)$O(N)$, the maximum size of the stack.

---

## Approach #2: Double Linked List + TreeMap [Accepted]

**Intuition**

Using structures like Array or Stack will never let us `popMax` quickly. We turn our attention to tree and linked-list structures that have a lower time complexity for removal, with the aim of making `popMax` faster than O(N)$O(N)$ time complexity.

Say we have a double linked list as our "stack". This reduces the problem to finding which node to remove, since we can remove nodes in O(1)$O(1)$ time.

We can use a TreeMap mapping values to a list of nodes to answer this question. TreeMap can find the largest value, insert values, and delete values, all in O(\log N)$O(\log N)$ time.

**Algorithm**

Let's store the stack as a double linked list `dll`, and store a `map` from `value` to a `List` of `Node`.

- When we `MaxStack.push(x)`, we add a node to our `dll`, and add or update our entry `map.get(x).add(node)`.
- When we `MaxStack.pop()`, we find the value `val = dll.pop()`, and remove the node from our `map`, deleting the entry if it was the last one.
- When we `MaxStack.popMax()`, we use the `map` to find the relevant node to `unlink`, and return it's value.

The above operations are more clear given that we have a working `DoubleLinkedList` class. The implementation provided uses `head` and `tail` *sentinels* to simplify the relevant `DoubleLinkedList` operations.

A Python implementation was not included for this approach because there is no analog to *TreeMap* available.

```
class MaxStack {
    TreeMap<Integer, List<Node>> map;
    DoubleLinkedList dll;


    public MaxStack() {
        map = new TreeMap();
        dll = new DoubleLinkedList();
    }


    public void push(int x) {
        Node node = dll.add(x);
        if(!map.containsKey(x))
            map.put(x, new ArrayList<Node>());
        map.get(x).add(node);
    }


    public int pop() {
        int val = dll.pop();
        List<Node> L = map.get(val);
        L.remove(L.size() - 1);
        if (L.isEmpty()) map.remove(val);
        return val;
    }

    public int top() {
        return dll.peek();
    }

    public int peekMax() {
        return map.lastKey();
    }

    public int popMax() {
        int max = peekMax();
        List<Node> L = map.get(max);
        Node node = L.remove(L.size() - 1);
        dll.unlink(node);
        if (L.isEmpty()) map.remove(max);
        return max;
    }
}

class DoubleLinkedList {
    Node head, tail;

    public DoubleLinkedList() {
        head = new Node(0);
        tail = new Node(0);
        head.next = tail;
        tail.prev = head;
    }
    public Node add(int val) {
        Node x = new Node(val);
        x.next = tail;
        x.prev = tail.prev;
        tail.prev = tail.prev.next = x;
        return x;
    }

    public int pop() {
        return unlink(tail.prev).val;
    }

    public int peek() {
        return tail.prev.val;
    }

    public Node unlink(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        return node;
    }
}

class Node {
    int val;
    Node prev, next;
    public Node(int v) {val = v;}
}
```

**Complexity Analysis**

- Time Complexity: O(\log N)$O(\log N)$ for all operations except `peek` which is O(1)$O(1)$, where N$N$ is the number of operations performed. Most operations involving `TreeMap` are O(\log N)$O(\log N)$.

- Space Complexity: O(N)$O(N)$, the size of the data structures used.

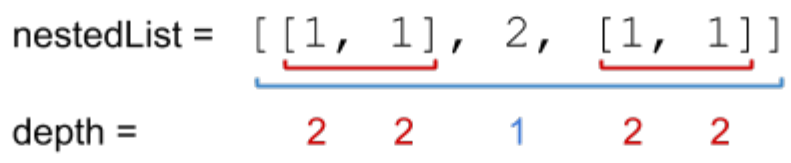
## QUES 12

You are given a nested list of integers `nestedList`. Each element is either an integer or a list whose elements may also be integers or other lists.

The **depth** of an integer is the number of lists that it is inside of. For example, the nested list `[1,[2,2],[[3],2],1]` has each integer's value set to its **depth**.

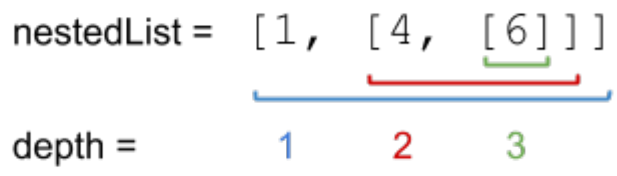
Return *the sum of each integer in* `nestedList` *multiplied by its* ***depth***.

**Example 1:**



```
Input: nestedList = [[1,1],2,[1,1]]

Output: 10

Explanation: Four 1's at depth 2, one 2 at depth 1. 1*2 + 1*2 + 2*1 + 1*2 + 1*2 =
10.
```

**Example 2:**



```
Input: nestedList = [1,[4,[6]]]

Output: 27

Explanation: One 1 at depth 1, one 4 at depth 2, and one 6 at depth 3. 1*1 + 4*2 +
6*3 = 27.
```

**Example 3:**

```
Input: nestedList = [0]

Output: 0
```

**Constraints:**

- `1 <= nestedList.length <= 50`
- The values of the integers in the nested list is in the range `[-100, 100]`.
- The maximum **depth** of any integer is less than or equal to `50`.

## SOLUTION

### Approach 1: Depth-first Search

Because the input is nested, it is natural to think about the problem in a recursive way. We go through the list of nested integers one by one, keeping track of the current depth d$d$. If a nested integer is an integer, n$n$, we calculate its sum as n\times d$n \times d$. If the nested integer is a list, we calculate the sum of this list recursively using the same process but with depth equals d + 1$d+1$.

```
class Solution {

public:

    int depthSum(vector<NestedInteger>& nestedList) {

        return dfs(nestedList, 1);

    }



    int dfs(vector<NestedInteger>& list, int depth) {

        int total = 0;

        for (NestedInteger nested : list) {

            if (nested.isInteger()) {

                total += nested.getInteger() * depth;

            } else {

                total += dfs(nested.getList(), depth + 1);

            }

        }

        return total;

    }

};
```

**Implementation**

**Complexity Analysis**

Let N$N$ be the total number of nested elements in the input list. For example, the list `[ [[[[1]]]], 2 ]` contains 44 nested lists and 22 nested integers (11 and 22), so N = 6$N=6$ for that particular case.

- Time complexity : \mathcal{O}(N)$O(N)$.

  Recursive functions can be a bit tricky to analyze, particularly when their implementation includes a loop. A good strategy is to start by determining how many times the recursive function is called, and then how many times the loop will iterate *across all calls to the recursive function*.

The recursive function, `dfs(...)` is called exactly **once** for each *nested list*. As N$N$ also includes nested integers, we know that the number of recursive calls has to be *less than* N$N$.

On each nested list, it iterates over all of the nested elements **directly inside that list** (in other words, not nested further). As each nested element can only be directly inside **one** list, we know that there must only be one loop iteration *for each nested element*. This is a total of N$N$ loop iterations.

So combined, we are performing at most 2 \cdot N2$\cdot N$ recursive calls and loop iterations. We drop the 22 as it is a constant, leaving us with time complexity \mathcal{O}(N)O($N$).

- Space complexity : \mathcal{O}(N)O($N$).

  In terms of space, at most O(D)$O(D)$ recursive calls are placed on the stack, where D$D$ is the maximum level of nesting in the input. For example, D=2$D$=2 for the input `[[1,1],2,[1,1]]`, and D=3$D$=3 for the input `[1,[4,[6]]]`.

  In the worst case, D = N$D$=$N$, (e.g. the list `[[[[[[]]]]]]`) so the worst-case space complexity is O(N)$O(N)$.

---

## Approach 2: Breadth-first Search

We can also solve the problem using a breadth-first search. The algorithm for this is closely based on the standard breadth-first search template. The algorithm fully processes each depth before moving to the next one.

```
class Solution {

public:

    int depthSum(vector<NestedInteger>& nestedList) {

        queue<NestedInteger> q;

        for (NestedInteger nested : nestedList) {

            q.push(nested);

        }



        int depth = 1;

        int total = 0;
        while (!q.empty()) {
            size_t size = q.size();
            for (size_t i = 0; i < size; i++) {
                NestedInteger nested = q.front();
                q.pop();
                if (nested.isInteger()) {
                    total += nested.getInteger() * depth;
                } else {
                    for (NestedInteger nested_deeper : nested.getList()) {
                        q.push(nested_deeper);
                    }
                }
            }
            depth++;
        }
        return total;
    }
};
```

**Implementation**

**Complexity Analysis**

- Time complexity : \mathcal{O}(N)O(N).

  Similar to the DFS approach. Each nested element is put on the queue and removed from the queue exactly once.

- Space complexity : \mathcal{O}(N)O(N).

  The worst-case for space complexity in BFS occurs where most of the elements are in a single layer, for example, a flat list such as `[1, 2, 3, 4, 5]` as all of the elements must be put on the queue at the same time. Therefore, this approach also has a worst-case space complexity of \mathcal{O}(N)O(N).

## QUES 14

Suppose you have `n` integers labeled `1` through `n`. A permutation of those `n` integers `perm` (**1-indexed**) is considered a **beautiful arrangement** if for every `i` (`1 <= i <= n`), **either** of the following is true:

- `perm[i]` is divisible by `i`.
- `i` is divisible by `perm[i]`.

Given an integer `n`, return *the **number** of the **beautiful arrangements** that you can construct*.

**Example 1:**

```
Input: n = 2

Output: 2

Explanation:

The first beautiful arrangement is [1,2]:

    - perm[1] = 1 is divisible by i = 1

    - perm[2] = 2 is divisible by i = 2

The second beautiful arrangement is [2,1]:

    - perm[1] = 2 is divisible by i = 1

    - i = 2 is divisible by perm[2] = 1
```

**Example 2:**

```
Input: n = 1

Output: 1
```

**Constraints:**

- `1 <= n <= 15`

## SOLUTION

### Approach #1 Brute Force [Time Limit Exceeded]

**Algorithm**

In the brute force method, we can find out all the arrays that can be formed using the numbers from 1 to N(by creating every possible permutation of the given elements). Then, we iterate over all the elements of every permutation generated and check for the required conditions of divisibility.

In order to generate all the possible pairings, we make use of a function `permute(nums, current_index)`. This function creates all the possible permutations of the elements of the given array.

To do so, `permute` takes the index of the current element $\text{current\_index}$*currentindex* as one of the arguments. Then, it swaps the current element with every other element in the array, lying towards its right, so as to generate a new ordering of the array elements. After the swapping has been done, it makes another call to permute but this time with the index of the next element in the array. While returning back, we reverse the swapping done in the current function call.

Thus, when we reach the end of the array, a new ordering of the array's elements is generated. The following animation depicts the process of generating the permutations.

```
public class Solution {

    int count = 0;

    public int countArrangement(int N) {

        int[] nums = new int[N];

        for (int i = 1; i <= N; i++)

            nums[i - 1] = i;

        permute(nums, 0);

        return count;

    }

    public void permute(int[] nums, int l) {

        if (l == nums.length - 1) {

            int i;

            for (i = 1; i <= nums.length; i++) {

                if (nums[i - 1] % i != 0 && i % nums[i - 1] != 0)

                    break;

            }

            if (i == nums.length + 1) {

                count++;

            }

        }

        for (int i = l; i < nums.length; i++) {

            swap(nums, i, l);

            permute(nums, l + 1);
            swap(nums, i, l);
        }
    }
    public void swap(int[] nums, int x, int y) {
        int temp = nums[x];
        nums[x] = nums[y];
        nums[y] = temp;
    }
}
```

**Complexity Analysis**

- Time complexity : $O(n!)O(n!)$. A total of $n!n!$ permutations will be generated for an array of length $nn$.
- Space complexity : $O(n)O(n)$. The depth of the recursion tree can go upto $nn$. $\mathrm{nums}nums$ array of size $nn$ is used.

---

## Approach #2 Better Brute Force [Accepted]

**Algorithm**

In the brute force approach, we create the full array for every permutation and then check the array for the given divisibilty conditions. But this method can be optimized to a great extent. To do so, we can keep checking the elements while being added to the permutation array at every step for the divisibility condition and can stop creating it any further as soon as we find out the element just added to the permutation violates the divisiblity condition.

```
public class Solution {
    int count = 0;
    public int countArrangement(int N) {
        int[] nums = new int[N];
        for (int i = 1; i <= N; i++)
            nums[i - 1] = i;
        permute(nums, 0);
        return count;
    }

    public void permute(int[] nums, int l) {

        if (l == nums.length) {

            count++;

        }

        for (int i = l; i < nums.length; i++) {

            swap(nums, i, l);

            if (nums[l] % (l + 1) == 0 || (l + 1) % nums[l] == 0)

                permute(nums, l + 1);

            swap(nums, i, l);

        }

    }

    public void swap(int[] nums, int x, int y) {

        int temp = nums[x];

        nums[x] = nums[y];

        nums[y] = temp;

    }

}
```

**Complexity Analysis**

- Time complexity : $O(k)$. $k$ refers to the number of valid permutations.
- Space complexity : $O(n)$. The depth of recursion tree can go upto $n$. Further, $nums$ array of size $n$ is used, where, $n$ is the given number.

---

## Approach #3 Backtracking [Accepted]

**Algorithm**

The idea behind this approach is simple. We try to create all the permutations of numbers from 1 to N. We can fix one number at a particular position and check for the divisibility criteria of that number at the particular position. But, we need to keep a track of the numbers which have already been considered earlier so that they aren't reconsidered while generating the permutations. If the current number doesn't satisfy the divisibility criteria, we can leave all the permutations that can be generated with that number at the particular position. This helps to prune the search space of the permutations to a great extent. We do so by trying to place each of the numbers at each position.

We make use of a visited array of size N$N$. Here, visited[i]$visited[i]$ refers to the i^{th}$i^{th}$ number being already placed/not placed in the array being formed till now(True indicates that the number has already been placed).

We make use of a `calculate` function, which puts all the numbers pending numbers from 1 to N(i.e. not placed till now in the array), indicated by a False$False$ at the corresponding visited[i]$visited[i]$ position, and tries to create all the permutations with those numbers starting from the pos$pos$ index onwards in the current array. While putting the pos^{th}$pos^{th}$ number, we check whether the i^{th}$i^{th}$ number satisfies the divisibility criteria on the go i.e. we continue forward with creating the permutations with the number i$i$ at the pos^{th}$pos^{th}$ position only if the number i$i$ and pos$pos$ satisfy the given criteria. Otherwise, we continue with putting the next numbers at the same position and keep on generating the permutations.

```
public class Solution {

    int count = 0;

    public int countArrangement(int N) {

        boolean[] visited = new boolean[N + 1];

        calculate(N, 1, visited);

        return count;

    }

    public void calculate(int N, int pos, boolean[] visited) {

        if (pos > N)

            count++;

        for (int i = 1; i <= N; i++) {

            if (!visited[i] && (pos % i == 0 || i % pos == 0)) {

                visited[i] = true;

                calculate(N, pos + 1, visited);

                visited[i] = false;

            }

        }

    }

}
```

*Complexity Analysis**

- Time complexity : O(k)$O(k)$. k$k$ refers to the number of valid permutations.
- Space complexity : O(n)$O(n)$. visited$visited$ array of size n$n$ is used. The depth of recursion tree will also go upto n$n$. Here, n$n$ refers to the given integer n$n$.

## QUES 15

Given an array of strings `wordsDict` and two different strings that already exist in the array `word1` and `word2`, return *the shortest distance between these two words in the list*.

**Example 1:**

```
Input: wordsDict = ["practice", "makes", "perfect", "coding", "makes"], word1 =
"coding", word2 = "practice"

Output: 3
```

**Example 2:**

```
Input: wordsDict = ["practice", "makes", "perfect", "coding", "makes"], word1 =
"makes", word2 = "coding"

Output: 1
```

**Constraints:**

- `1 <= wordsDict.length <= 3 * 10`$^4$
- `1 <= wordsDict[i].length <= 10`
- `wordsDict[i]` consists of lowercase English letters.
- `word1` and `word2` are in `wordsDict`.
- `word1 != word2`

## SOLUTION

## Solution

This is a straight-forward coding problem. The distance between any two positions i_1$i_1$ and i_2$i_2$ in an array is |i_1 - i_2|$|i_1 - i_2|$. To find the shortest distance between `word1` and `word2`, we need to traverse the input array and find all occurrences i_1$i_1$ and i_2$i_2$ of the two words, and check if |i_1 - i_2|$|i_1 - i_2|$ is less than the minimum distance computed so far.

---

### Approach #1 (Brute Force)

**Algorithm**

A naive solution to this problem is to go through the entire array looking for the first word. Every time we find an occurrence of the first word, we search the entire array for the closest occurrence of the second word.

```
class Solution {
  public int shortestDistance(String[] words, String word1, String word2) {
    int minDistance = words.length;
    for (int i = 0; i < words.length; i++) {
      if (words[i].equals(word1)) {
        for (int j = 0; j < words.length; j++) {
                if (words[j].equals(word2)) {
                    minDistance = Math.min(minDistance, Math.abs(i - j));
                }
            }
        }
    }
    return minDistance;
  }
}
```

**Complexity Analysis**

The time complexity is $O(n^2)$, since for every occurrence of `word1`, we traverse the entire array in search for the closest occurrence of `word2`.

Space complexity is $O(1)$, since no additional space is used.

---

## Approach #2 (One-pass)

**Algorithm**

We can greatly improve on the brute-force approach by keeping two indices `i1` and `i2` where we store the *most recent* locations of `word1` and `word2`. Each time we find a new occurrence of one of the words, we do not need to search the entire array for the other word, since we already have the index of its most recent occurrence.

```
class Solution {

  public int shortestDistance(String[] words, String word1, String word2) {

    int i1 = -1, i2 = -1;

    int minDistance = words.length;

    for (int i = 0; i < words.length; i++) {

      if (words[i].equals(word1)) {

        i1 = i;

      } else if (words[i].equals(word2)) {

        i2 = i;

      }



      if (i1 != -1 && i2 != -1) {

        minDistance = Math.min(minDistance, Math.abs(i1 - i2));

      }

    }
```

```
        return minDistance;

    }

}
```

**Complexity Analysis**

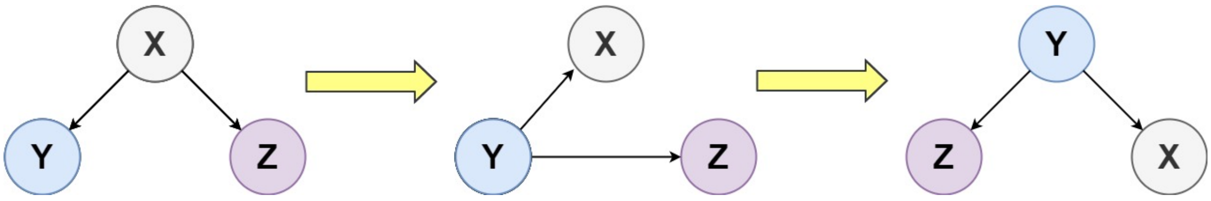
- Time complexity: O(N \cdot M)$O(N \cdot M)$ where N$N$ is the number of words in the input list, and M$M$ is the total length of two input words.
- Space complexity: O(1)$O(1)$, since no additional space is allocated.

## QUES 16

Given the `root` of a binary tree, turn the tree upside down and return *the new root*.

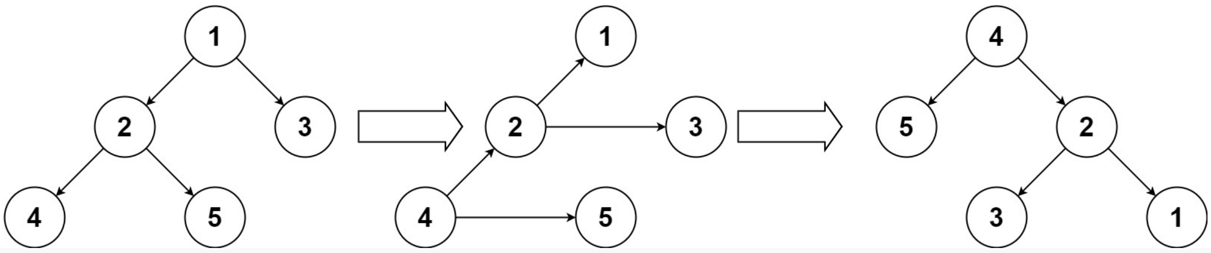
You can turn a binary tree upside down with the following steps:

1. The original left child becomes the new root.
2. The original root becomes the new right child.
3. The original right child becomes the new left child.



The mentioned steps are done level by level. It is **guaranteed** that every right node has a sibling (a left node with the same parent) and has no children.

**Example 1:**



```
Input: root = [1,2,3,4,5]

Output: [4,5,2,null,null,3,1]
```

**Example 2:**

```
Input: root = []

Output: []
```

**Example 3:**

```
Input: root = [1]

Output: [1]
```

**Constraints:**

- The number of nodes in the tree will be in the range `[0, 10]`.
- `1 <= Node.val <= 10`
- Every right node in the tree has a sibling (a left node that shares the same parent).
- Every right node in the tree has no children.

## SOLUTION

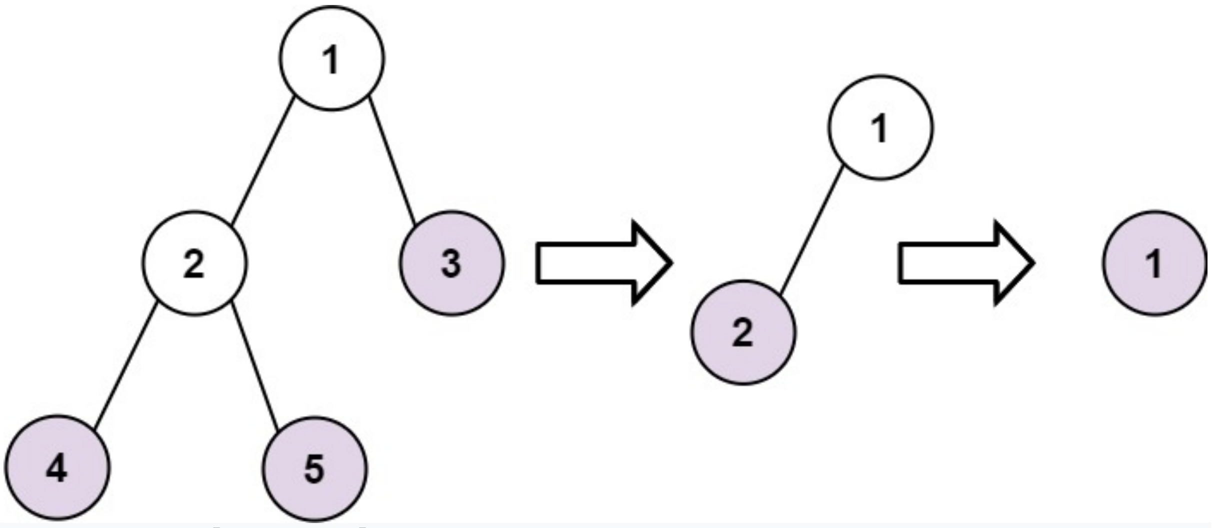
```
def upsideDownBinaryTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

        self.new_root = None


        def dfs(node, prev):

            if not node: return


            # if left leaf reached, make it a root

            if not node.left and not node.right:

                self.new_root = node


            # first turn the left child

            dfs(node.left, node)


            # turn left and right child based on previous if it exists

            if prev:

                node.right = prev

                node.left = prev.right

            else:

                node.right = node.left = None


        dfs(root, None)

        return self.new_root
```

## QUES 17

Given the `root` of a binary tree, collect a tree's nodes as if you were doing this:

- Collect all the leaf nodes.
- Remove all the leaf nodes.
- Repeat until the tree is empty.

**Example 1:**



```
Input: root = [1,2,3,4,5]

Output: [[4,5,3],[2],[1]]

Explanation:

[[3,5,4],[2],[1]] and [[3,4,5],[2],[1]] are also considered correct answers since
per each level it does not matter the order on which elements are returned.
```

**Example 2:**

```
Input: root = [1]

Output: [[1]]
```

**Constraints:**

- The number of nodes in the tree is in the range `[1, 100]`.
- `-100 <= Node.val <= 100`

## SOLUTION

### Approach 1: DFS (Depth-First Search) with sorting

**Intuition**

The order in which the elements (nodes) will be collected in the final answer depends on the "height" of these nodes. The height of a node is the number of edges from the node to the deepest leaf. The nodes that are located in the $i^{th}$ height will be appear in the $i^{th}$ collection in the final answer. For any given node in the binary tree, the height is obtained by adding 1 to the maximum height of any children. Formally, for a given node of the binary tree \text{root}root, it's height can be represented as

\text{height(root)} = \text{1} + \text{max(height(root.left), height(root.right))}height(root)=1+max(height(root.left), height(root.right))

Where \text{root.left}root.left and \text{root.right}root.right are left and right children of the root respectively

**Algorithm**

In our first approach, we'll simply traverse the tree recursively in a depth first search manner using the function `int getHeight(node)`, which will return the height of the given node in the binary tree. Since height of any node depends on the height of it's children node, hence we traverse the tree in a post-order manner (i.e. height of the childrens are calculated first before calculating the height of the given node). Additionally, whenever we encounter a null node, we simply return -1 as it's height.

Next, we'll store the pair `(height, val)` for all the nodes which will be sorted later to obtain the final answer. The sorting will be done in increasing order considering the height first and then the val. Hence we'll obtain all the pairs in the increasing order of their height in the given binary tree.

```
class Solution {
public:

    vector<pair<int, int>> pairs;

    int getHeight(TreeNode *root) {

        // return -1 for null nodes
        if (!root) return -1;

        // first calculate the height of the left and right children
        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        // based on the height of the left and right children, obtain the height of the current
(parent) node
        int currHeight = max(leftHeight, rightHeight) + 1;

        // collect the pair -> (height, val)
        this->pairs.push_back({currHeight, root->val});

        // return the height of the current node
        return currHeight;
    }

    vector<vector<int>> findLeaves(TreeNode* root) {
        this->pairs.clear();

        getHeight(root);

        // sort all the (height, val) pairs
        sort(this->pairs.begin(), this->pairs.end());

        int n = this->pairs.size(), height = 0, i = 0;
        vector<vector<int>> solution;
        while (i < n) {
            vector<int> nums;
            while (i < n && this->pairs[i].first == height) {
                nums.push_back(this->pairs[i].second);
                i++;
            }
            solution.push_back(nums);
            height++;
        }
        return solution;
    }
};
```

**Complexity Analysis**

- Time Complexity: Assuming N$N$ is the total number of nodes in the binary tree, traversing the tree takes O(N)$O(N)$ time. Sorting all the pairs based on their height takes O(N \log N)$O(N\log N)$ time. Hence overall time complexity of this approach is O(N \log N)$O(N\log N)$

- Space Complexity: O(N \log N)$O(N\log N)$, the space used by `pairs` and `solution`.

---

## Approach 2: DFS (Depth-First Search) without sorting

We've seen in approach 1 that there is an additional sorting that is being performed, which increases the overall time complexity to O(N \log N)$O(N\log N)$. The question we can ask here is, can we do better than this? To answer this, we try to remove the sorting by directly placing all the values in their respective positions, i.e. instead of using the `pairs` array to collect all the `(height, val)` pairs and then sorting them based on their heights, we'll directly obtain the solution by placing each element (`val`) to its correct position in the solution array. To clarify, in the given binary tree, `[4, 3, 5]` goes into the first position, `[2]` goes into the second position and `[1]` goes into the third position in the solution array.

To do this, we modify our `getHeight` method to directly insert the node's value in the solution array at the correct location. Solution array is kept empty in the beginning and as we encounter elements with increasing height, we'll keep increasing the size of the solution array to accomodate for these elements. For example, if our solution array currently is `[[4, 3, 5]]` and if we want to insert 2 at the second position, we first create the space for 2 by increasing the size of the solution array by 1 and then insert 2 at it's correct location.

- `[[4, 3, 5]] -> [[4, 3, 5], []] # increase the size of solution array`
- `[[4, 3, 5], []] -> [[4, 3, 5], [2]] # insert 2 at it's correct location`

Below is the implementation of the above mentioned approach.

```
class Solution {
private:

    vector<vector<int>> solution;

public:

    int getHeight(TreeNode *root) {

        // return -1 for null nodes
        if (!root) {
            return -1;
        }

        // first calculate the height of the left and right children
        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        // based on the height of the left and right children, obtain the height of the current
(parent) node
        int currHeight = max(leftHeight, rightHeight) + 1;

        // create space for node located at `currHeight` if not already exists
        if (this->solution.size() == currHeight) {
            this->solution.push_back({});
        }

        // insert the value at the correct position in the solution array
        this->solution[currHeight].push_back(root->val);

        // return the height of the current node
        return currHeight;
    }

    vector<vector<int>> findLeaves(TreeNode* root) {
        this->solution.clear();

        getHeight(root);

        return this->solution;
    }
};
```

**Complexity Analysis**

- Time Complexity: Assuming $N$ is the total number of nodes in the binary tree, traversing the tree takes $O(N)$ time and storing all the pairs at the correct position also takes $O(N)$ time. Hence overall time complexity of this approach is $O(N)$.
- Space Complexity: $O(N)$, the space used by `solution` array.

## QUES 19

There is a row of `n` houses, where each house can be painted one of three colors: red, blue, or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by an `n x 3` cost matrix `costs`.

- For example, `costs[0][0]` is the cost of painting house `0` with the color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on...

Return *the minimum cost to paint all houses*.

**Example 1:**

```
Input: costs = [[17,2,17],[16,16,5],[14,3,19]]

Output: 10

Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2 into blue.

Minimum cost: 2 + 5 + 3 = 10.
```

**Example 2:**

```
Input: costs = [[7,6,2]]

Output: 2
```

**Constraints:**

- `costs.length == n`
- `costs[i].length == 3`
- `1 <= n <= 100`
- `1 <= costs[i][j] <= 20`

## SOLUTION

For those already familiar with memoization and dynamic programming, this question will be easy. For those who are very new to Leetcoding, it might seem like a medium, or even a hard. ***For those who are starting to learn about memoization and dynamic programming, this question is a great one for getting started***!

This article is aimed at those of you getting started with dynamic programming and memoization. I'll assume you have already worked through prerequisite concepts such as n-ary trees (or binary trees), including with recursion. If you haven't, then I strongly recommend that you come back to this question after working through either the N-ary Trees module or the Binary Trees module. The intuition behind memoization and dynamic programming is best understood using trees, so that is what I've done in this article. Understanding how to recognize and then approach memoization and dynamic programming problems is essential for interview success.

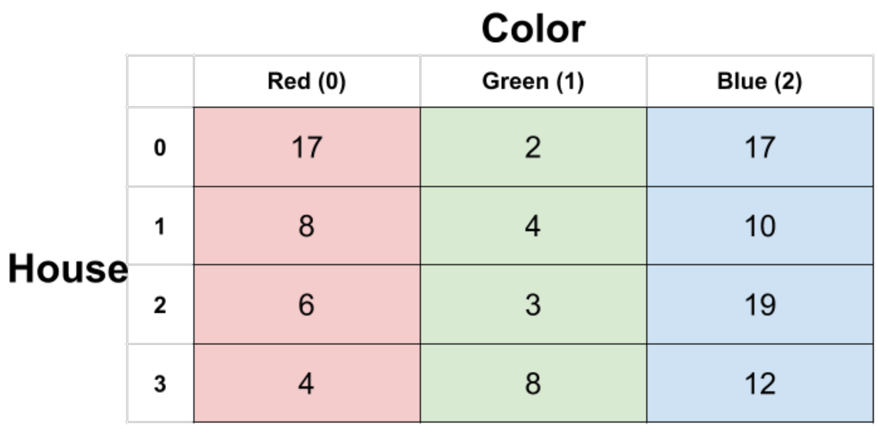
## Approach 1: Brute force

**Intuition**

The brute force approach is often a good place to start. From there, we can identify unnecessary work and further optimize. In this case, the brute force algorithm would be to generate every valid permutation of house colors (or all permutations and then remove all the invalid ones, e.g. ones that have 2 red houses side-by-side) and score them. Then, the lowest score is the value we need to return.

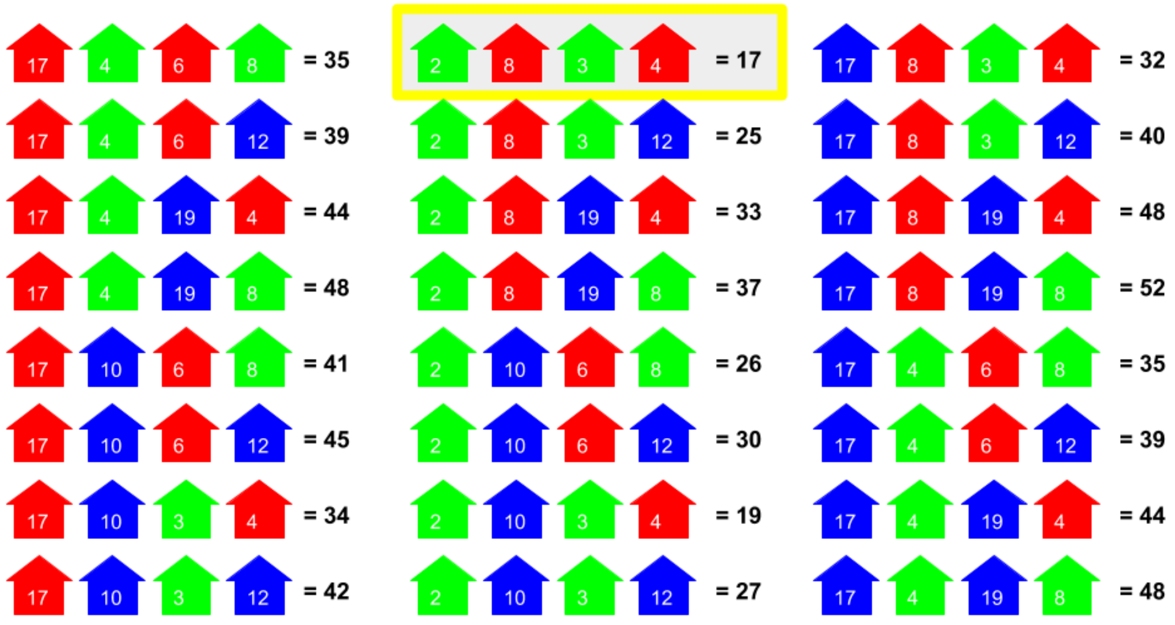
For this article, we'll use the following input. It is for 4 houses.

```
[[17, 2, 17], [8, 4, 10], [6, 3, 19], [4, 8, 12]]
```



| | | Color | | |
|---|---|---|---|---|
| | | Red (0) | Green (1) | Blue (2) |
| House | 0 | 17 | 2 | 17 |
| | 1 | 8 | 4 | 10 |
| | 2 | 6 | 3 | 19 |
| | 3 | 4 | 8 | 12 |

These are all the valid sequences you can get with 4 houses. In total, there are 24 of them. The one with the lowest total cost is highlighted.



The best option is to paint the first house green, second house red, third house green, and fourth house red. This will cost a total of `17`.

**Algorithm**

It's not worth worrying about how you'd implement the brute force solution—it's completely infeasible and useless in practice. Additionally, the latter approaches move in a different direction, and the permutation code actually takes some effort to understand (which would be a distraction for you). Therefore, I haven't included code for it. You wouldn't be writing code for it in an interview either, instead you'd simply describe a possible approach and move onto optimizing, and then write code for a more optimal algorithm.

There are many different approaches to it. All are based on permutation generation, but some only generate permutations that follow the color rules, and others generate all permutations and then remove the non-valid ones afterwards. Some are recursive, and others are iterative. Some use $O(n)$ space by only generating one permutation at a time and then processing it before generating the next, and others use a lot more (discussed below) from generating all the sequences first and then processing them.

The simplest is probably to generate every possible length-n string of `0`, `1`, and `2`, remove any that have the same digit twice in a row, and then score those that are left, keeping track of the smallest cost seen so far.

**Complexity Analysis**

- Time complexity : $O(2^n)$ or $O(3^n)$.

  Without writing code, we can get a good idea of the cost. We know that at the very least, we'd have to process every valid permutation. The number of valid permutations doubles with every house added. With `4` houses, there were `24` permutations. If we add another house, then all of our permutations for 4 houses could be extended with 2 different colors for the 5th house, giving `48` permutations. Because it doubles every time, this is $O(n^2)$.

  It'd be even worse if we generated all permutations of `0`, `1`, and `2` and then pruned out the invalid ones. There are $O(n^3)$ such permutations in total.

- Space complexity : Anywhere from $O(n)$ to $O(n \cdot 3^n)$.

  This would depend entirely on the implementation. If you generated all the permutations at the same time and put them in a massive list, then you'd be using $O(n * 2^n)$ or $O(n * 3^n)$ space. If you generated one, processed it, generated the next, processed it, etc, without keeping the long list, it'd require $O(n)$ space.

---

## Approach 2: Brute force with a Recursive Tree

**Intuition**

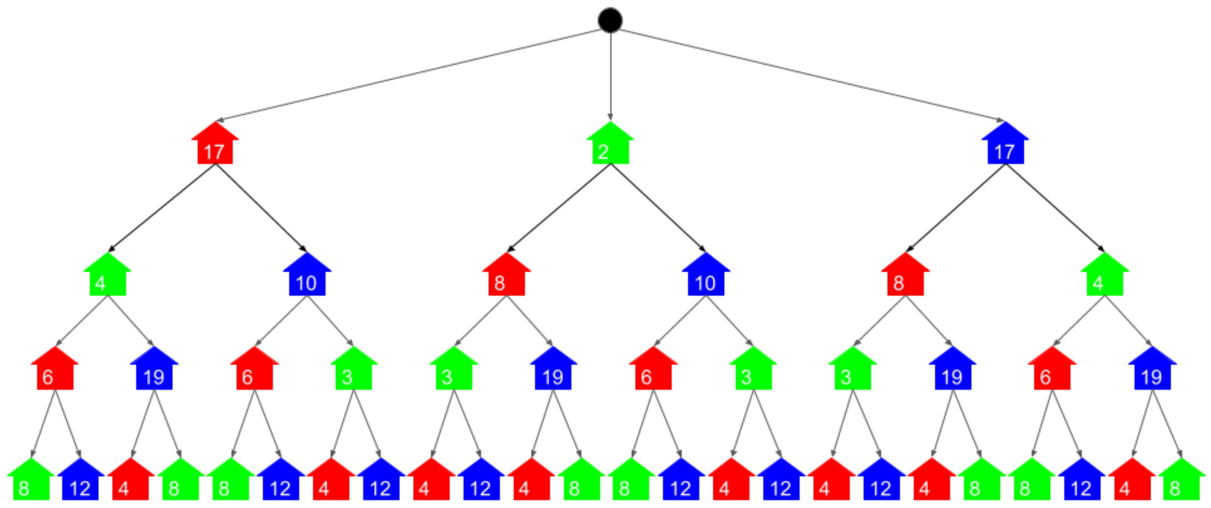
Like the first approach, this approach still isn't good enough. However, it bridges the gap between approach 1 and approach 3, with approach 3 further building on it. So make sure you understand it well.

When we have permutations, we can think of them as forming a big tree of all the options. Drawing out the tree (or part of it) can give useful insights and reveal other possible algorithms. We'll continue using the sample example that we did above:

**Color**

| House | | Red (0) | Green (1) | Blue (2) |
|---|---|---|---|---|
| | 0 | 17 | 2 | 17 |
| | 1 | 8 | 4 | 10 |
| | 2 | 6 | 3 | 19 |
| | 3 | 4 | 8 | 12 |

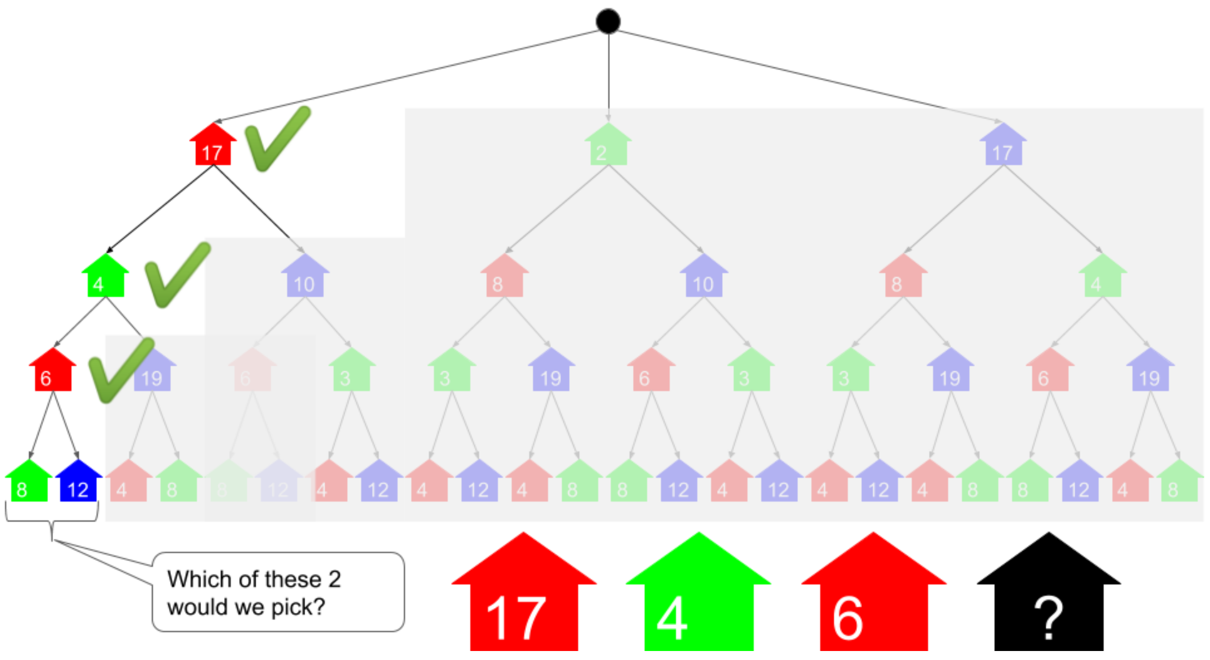
And here is how we can represent it using a tree. Each path from root to leaf represents a different possible permutation of house colors. There are 24 leaf nodes on the tree, just like there was 24 permutations identified in the brute force approach.



The tree representation gives a useful model of the problem and all the possible permutations. It shows that, for example, if we paint the first house red, then we have 2 options for the second house: green or blue. And then if we choose green for the second house, we could choose red or blue for the 3rd house. And so forth.

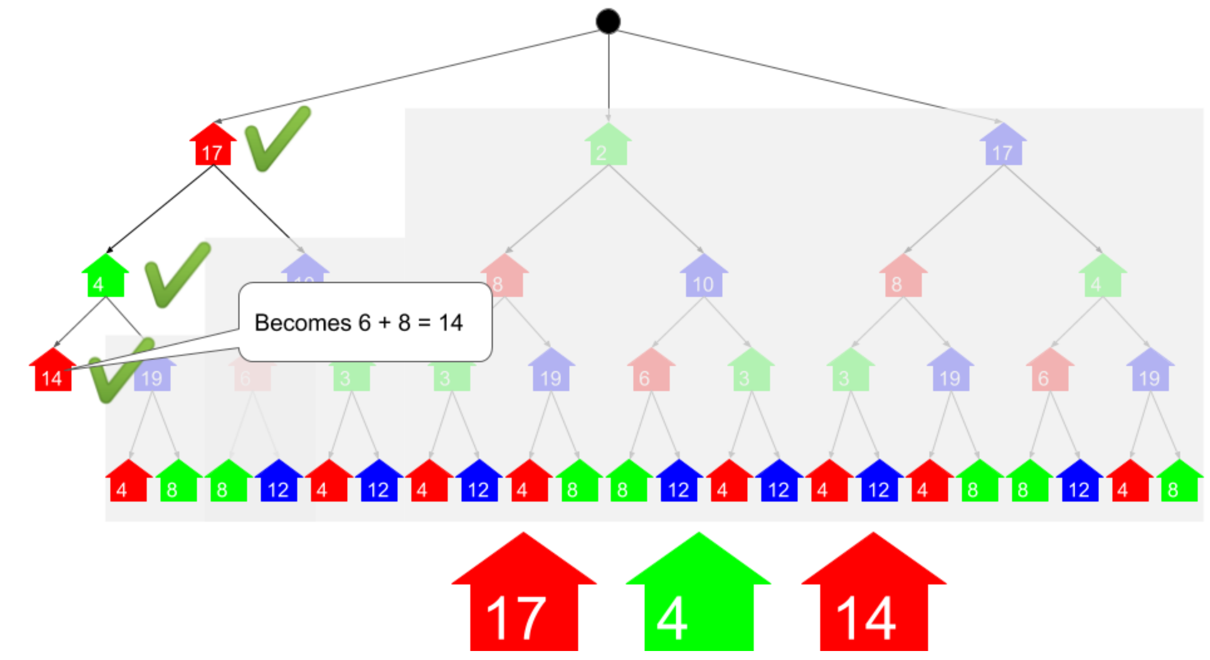
Without worrying yet about how we would actually implement it, we'll now explore a straightforward algorithm that can be used to solve the problem using this tree.

If the first 3 houses were red, green, and red then we could paint the 4th house green or blue. Which would we want to choose?



To minimize cost, we'd choose green. This is because green is `8`, and blue is `12`. *Under the assumption that we'd already decided that the first 3 houses would be red, green, and red*, this decision is definitely optimal. We know that there's no way we could do better.

What we were effectively doing was deciding which was cheaper out of 2 permutations: `red, green, red, green` or `red, green, red, blue`. Because the former is cheaper, we have completely ruled out the latter. We can simplify our tree with this new information by adding the cost of the 4th house to the cost of the 3rd house on that branch.



We are left with the conclusion that:

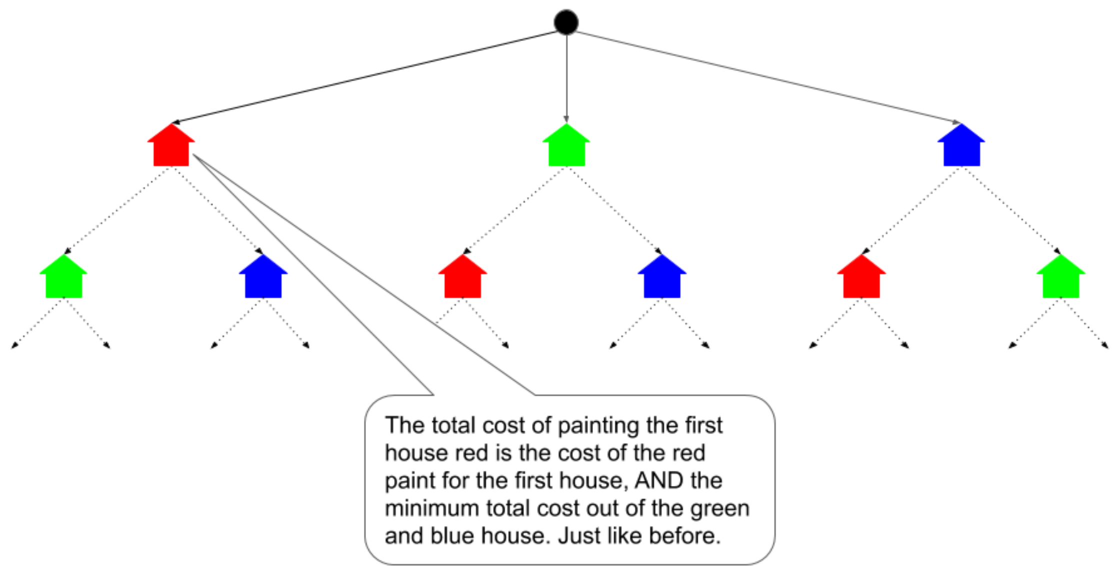
- Painting the first house red would have a *total* cost of `34`.
- Painting the first house green would have a *total* cost of `17`.
- Painting the first house blue would have a *total* cost of `32`.

So, it makes sense to paint the first house green. This gives a total cost of `17`, which was the same answer our brute force in approach 1 arrived at.

**Algorithm**

To actually implement it, we'll need to change the way we think about it. What we did here was a *bottom-up* algorithm, meaning that we started by processing leaf nodes and then worked our way up. When we implement algorithms like this though, we almost always do it *top-down*. This allows us to use an *implicit tree* with recursion, instead of actually making a tree (i.e. having to work with `TreeNode`'s'). The recursive calls all form a tree structure. If you're not too familiar with this idea yet, don't panic, there is an animation of the algorithm and the code in the next section. The best way to get your head around recursion is to look at examples and recognise common patterns.

Let's get started. Remember how we determined the cost of painting each house in the tree?


The total cost of painting the first house red is the cost of the red paint for the first house, AND the minimum total cost out of the green and blue house. Just like before.

By *total cost*, we mean the cost of painting that house a particular color *and* painting the ones after it optimally.

In pseudocode the top-down recursive algorithm looks like this:

```
print min(paint(0, 0), paint(0, 1), paint(0, 2))



define function paint(n, color):

  total_cost = costs[n][color]

  if n is the last house number:

    pass [go straight to the return]

  else if color is red (0):

    total_cost += min(paint(n+1, 1), paint(n+1, 2))

  else if color is green (1):

    total_cost += min(paint(n+1, 0), paint(n+1, 2))

  else if color is blue (2):

    total_cost += min(paint(n+1, 0), paint(n+1, 1))

  return the total_cost
```

Here is an animation/ walkthrough of the algorithm. It also shows how the recursive calls make the same structure as the tree we were playing around with before, without actually building a tree. While this algorithm might be a bit to get your head around if you're not too familiar with recursion, doing so is essential to understanding approach 3.

And here is the code. While you're reading over it, have an initial think about how you could optimize it so that it no longer takes exponential time. Hint: look closely at the parameters of the recursive function. Are we actually repeating the same thing over and over? Fixing this problem will be what we tackle in Approach 3.

```
class Solution {



    private int[][] costs;



    public int minCost(int[][] costs) {

        if (costs.length == 0) {

            return 0;

        }

        this.costs = costs;

        return Math.min(paintCost(0, 0), Math.min(paintCost(0, 1), paintCost(0, 2)));

    }



    private int paintCost(int n, int color) {

        int totalCost = costs[n][color];

        if (n == costs.length - 1) {

        } else if (color == 0) { // Red

            totalCost += Math.min(paintCost(n + 1, 1), paintCost(n + 1, 2));

        } else if (color == 1) { // Green

            totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 2));

        } else { // Blue

            totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 1));

        }

        return totalCost;

    }
```

}

**Complexity Analysis**

- Time complexity : O(2^n)$O(2^n)$.

  While this approach is an improvement on the previous approach, it still requires exponential time. Think about the number of leaf nodes. Each permutation has its own leaf node. The number of internal nodes is the same as the number of leaf nodes too. Remember how there are 2^n$2^n$ different permutations? Each effectively adds `2` nodes to the tree, so dropping the constant of `2` gives us O(2^n)$O(2^n)$.

  This is better than the previous approach, which had an additional factor of `n`, giving O(n \cdot 2 ^ n)$O(n \cdot 2^n)$. That extra factor of `n` has disappeared here because the permutations are now "sharing" their similar parts, unlike before. The idea of "sharing" similar parts can be taken much further for this particular problem, as we will see with the remaining approaches that knock the time complexity all the way down to O(n)$O(n)$.

- Space complexity : O(n)$O(n)$.

  This algorithm might initially appear to be O(1)$O(1)$, because we are not allocating any new data structures. However, we need to take into account space usage on the **run-time stack**. The run-time stack was shown in the animation. Whenever we are processing the last house (house number `n - 1`), there are `n` stack frames on the stack. This space usages counts for complexity analysis (it's memory usage, like any other memory usage) and so the space complexity is O(n)$O(n)$.

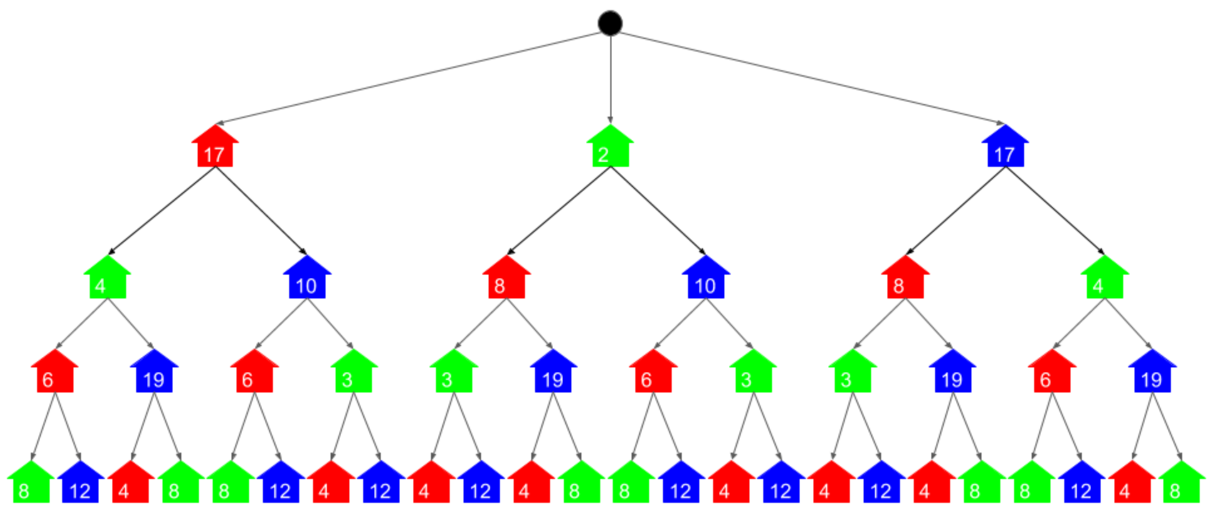
---

## Approach 3: Memoization

### Intuition

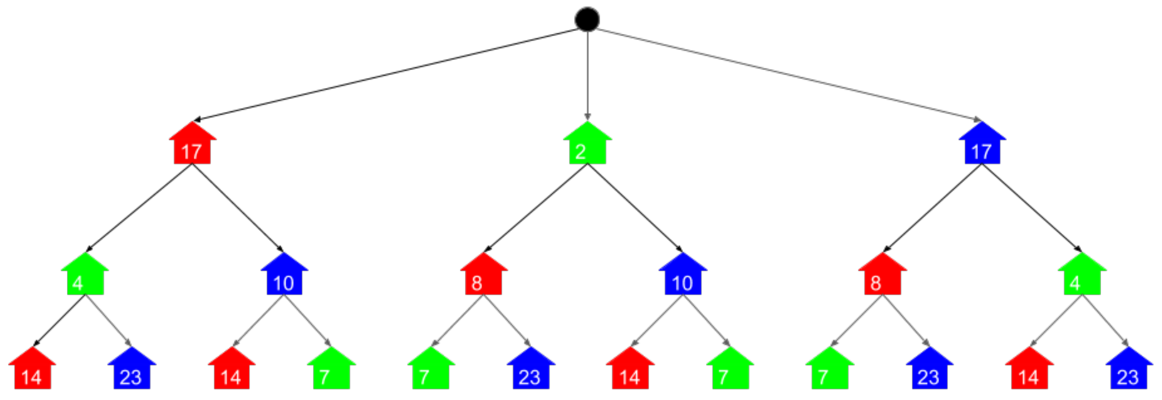
You may have noticed a very important pattern while we were working on the previous approach. Let's take a closer look.

This is the tree before we removed any layers.



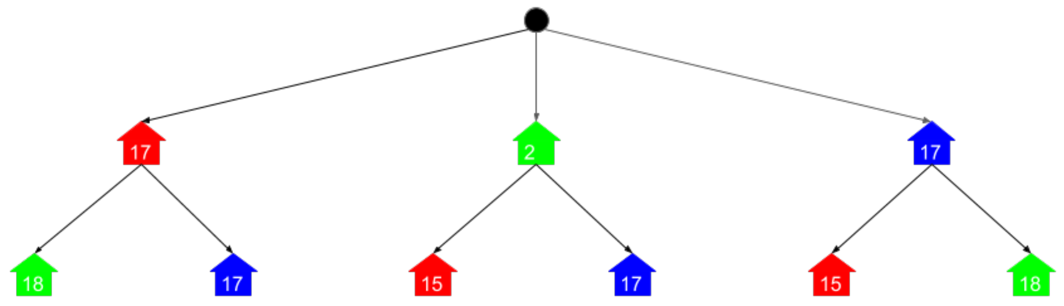
Look at the leaf nodes. All the red houses cost `4`, the green houses `7`, and the blue houses `23`. This makes sense, as the original input told us the costs of painting the `4th` house red, green, or blue were `4`, `7`, and `23` respectively.

But look at what happens when we remove those leaf nodes in the way we described in the previous section.



Again, all the red houses are the same at `14`, the green houses are `7`, and the blue houses are `23`. Why has this happened? Well, we were always adding the cheapest of the 2 children to the parent, before deleting the 2 children. Painting the 3rd house itself red *always* costs `6`. And then we can *always* choose between painting the 4th house green or blue. It only ever made sense to choose green, as that was `8` (compared to `12` to paint it blue) Therefore, all those branches became `6 + 8 = 14`. Similar arguments apply to painting the 3rd house blue or green.

And here's the tree when we'd removed another layer again.



Unsurprisingly, the pattern still continues.

This pattern is important, because it shows us that we're actually doing the same few calculations over and over again. Instead of repeatedly doing the same (expensive) calculations, we should instead save and re-use results where possible.

For example, imagine if in school you'd been given this math homework (and were *not* allowed to use a calculator). How would you approach it?

```
1) 345 * 282 = ?

2) 43 + (345 * 282) = ?

3) (345 * 282) + 89 = ?

4) (345 * 282) * 5 + 19 = ?
```

Unless you really, really love arithmetic, I think you would have done the working for `345 * 282` just *once* and then inserted it into all the other equations. You probably wouldn't have done the long multiplication 4 separate times for it!

And it's the same for calculating the costs for painting these houses. We only need to calculate the cost of painting the 2nd house red *once*.

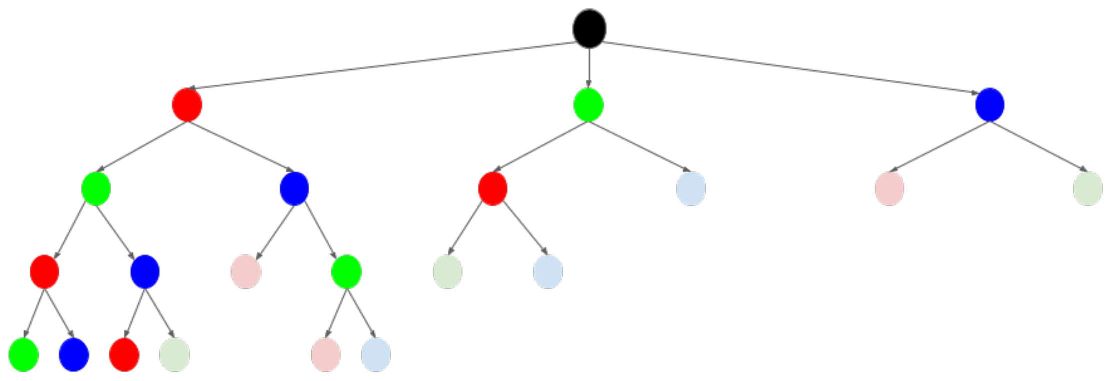
So to do this, we'll use **memoization**. Immediately before returning a value we've finished computing, we'll write it into a dictionary with the input values as the key and the return value as the result. Then at the start of the function, we'll first check if the answer is already in the dictionary. If it is, we can immediately return it. If not, then we need to continue like before and compute it.

**Algorithm**

The algorithm is almost the same as before. The only difference is that we create an empty dictionary at the start, write the return values into it, and check it first to see if we've already found the answer for a particular set of input parameters.

```
print min(paint(0, 0), paint(0, 1), paint(0, 2))

memo = a new, empty dictionary

define function paint(n, color):
  if (n, color) is a key in memo:
     return memo[(n, color)]
  total_cost = costs[n][color]
  if n is the last house number:
    pass [go straight to return]
  else if color is red (0):
    total_cost += min(paint(n+1, 1), paint(n+1, 2))
  else if color is green (1):
    total_cost += min(paint(n+1, 0), paint(n+1, 2))
  else if color is blue (2):
    total_cost += min(paint(n+1, 0), paint(n+1, 1))
  memo[(n, color)] = total_cost
  return the total_cost
```

Remember how the previous approach made a recursive function call for every node in the tree we drew? Well this approach only needs to do the calculations shown. The brighter circles represent where it needed to actually calculate the answer and the dull circles show where an answer was looked up in the dictionary.



```
class Solution {


    private int[][] costs;

    private Map<String, Integer> memo;


    public int minCost(int[][] costs) {

        if (costs.length == 0) {

            return 0;

        }

        this.costs = costs;

        this.memo = new HashMap<>();

        return Math.min(paintCost(0, 0), Math.min(paintCost(0, 1), paintCost(0, 2)));

    }


    private int paintCost(int n, int color) {

        if (memo.containsKey(getKey(n, color))) {

            return memo.get(getKey(n, color));

        }

        int totalCost = costs[n][color];

        if (n == costs.length - 1) {

        } else if (color == 0) { // Red

            totalCost += Math.min(paintCost(n + 1, 1), paintCost(n + 1, 2));
        } else if (color == 1) { // Green
            totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 2));
        } else { // Blue
            totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 1));
        }
        memo.put(getKey(n, color), totalCost);


        return totalCost;
    }


    private String getKey(int n, int color) {
        return String.valueOf(n) + " " + String.valueOf(color);
    }
}
```

**Complexity Analysis**

- Time complexity : $O(n)$.

  Analyzing memoization algorithms can be tricky at first, and requires understanding how recursion impacts the cost differently to loops. The key thing to notice is that the full function runs once for each possible set of parameters. There are `3 * n` different possible sets of parameters, because there are `n` houses and `3` colors. Because the function body is $O(1)$ (it's simply a conditional), this gives us a total of `3 * n`. There can't be more than `3 * 2 * n` searches into the memoization dictionary either. The tree showed this clearly—the nodes representing lookups had to be the child of a call where a full calculation was done. Because the constants are all dropped, this leaves $O(n)$.

- Space complexity : $O(n)$.

  Like the previous approach, the main space usage is on the stack. When we go down the first branch of function calls (see the tree visualization), we won't find any results in the dictionary. Therefore, every house will make a stack frame. Because there are `n` houses, this gives a worst case space usage of $O(n)$. Note that this could be a problem in languages such as Python, where stack frames are large.

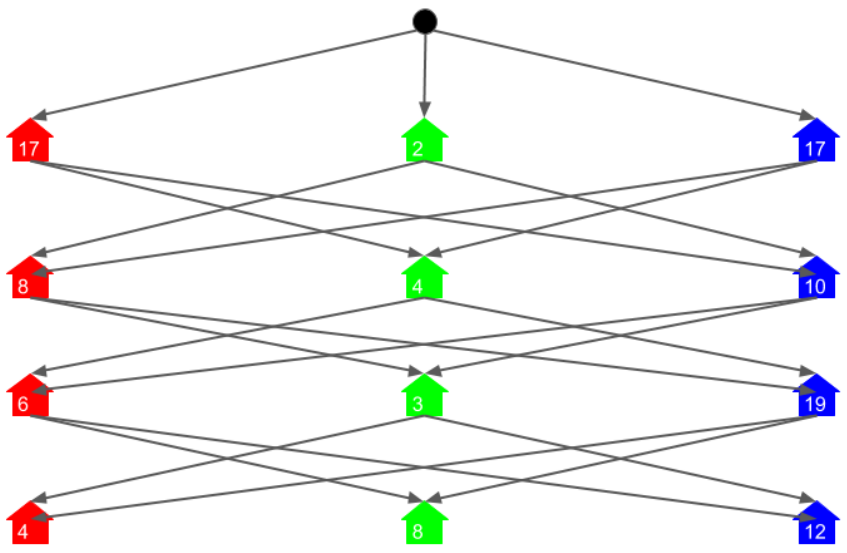
---

## Approach 4: Dynamic Programming

**Intuition**

In approach 2, we started with, although didn't actually implement, a bottom up algorithm. The reason we didn't implement it is because we would have had to generate an actual tree which would have been a lot of work, and unnecessary for what we were trying to accomplish. However, there is another way of writing an iterative bottom-up algorithm to solve this problem. It utilizes the same pattern that we identified in approach 3.

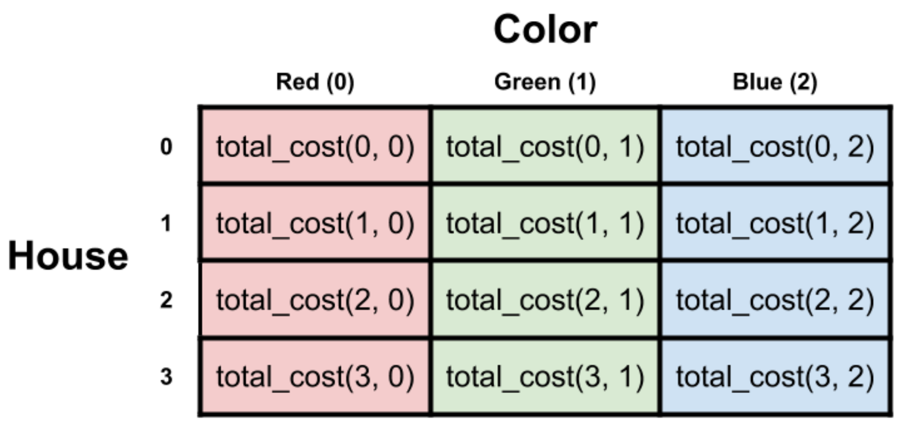
As a starting point, what would the tree look like if we converted it into a directed graph without the repetition? In other words, if we made it so that the 2nd house being blue was pointed to by both the 1st house being green and the 1st house being red? Well, it'd look something like this.



Directly generating this graph (i.e. not generating the massive tree first) and then using the same algorithm from approach 2 would achieve comparable time complexity to approach 3. But there's a far simpler way that doesn't even require generating the graph: dynamic programming! Dynamic programming is iterative, unlike memoization, which is recursive.

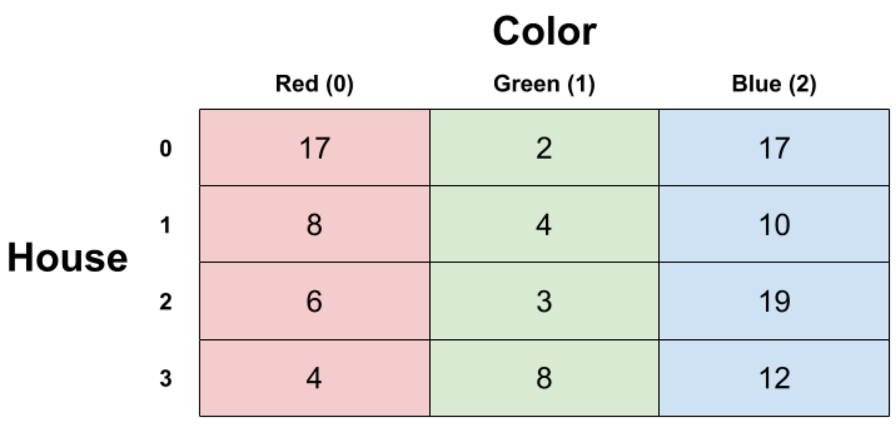
We'll define a subproblem to be calculating the total cost for a particular house position and color.

For the 4-house example, the memoization approach needed to solve a total of 12 different subproblems. We know this, because there were 3 possible values for the color (0, 1, 2), and 4 possible values for the house number (0, 1, 2, 3). In total, this gave us 12 different possibilities. The dynamic programming approach will need to solve these same subproblems, except in an iterative manner.



| | | Color | | |
|---|---|---|---|---|
| | | Red (0) | Green (1) | Blue (2) |
| House | 0 | total_cost(0, 0) | total_cost(0, 1) | total_cost(0, 2) |
| | 1 | total_cost(1, 0) | total_cost(1, 1) | total_cost(1, 2) |
| | 2 | total_cost(2, 0) | total_cost(2, 1) | total_cost(2, 2) |
| | 3 | total_cost(3, 0) | total_cost(3, 1) | total_cost(3, 2) |

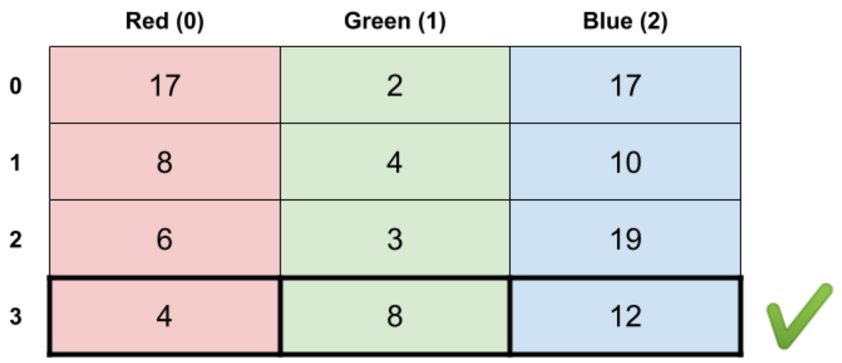
Now, remember the size of the input array? It's the same! Also, notice how it maps onto the tree. Again, it's the same.



| House | | Color | | |
|---|---|---|---|---|
| | | **Red (0)** | **Green (1)** | **Blue (2)** |
| | **0** | 17 | 2 | 17 |
| | **1** | 8 | 4 | 10 |
| | **2** | 6 | 3 | 19 |
| | **3** | 4 | 8 | 12 |

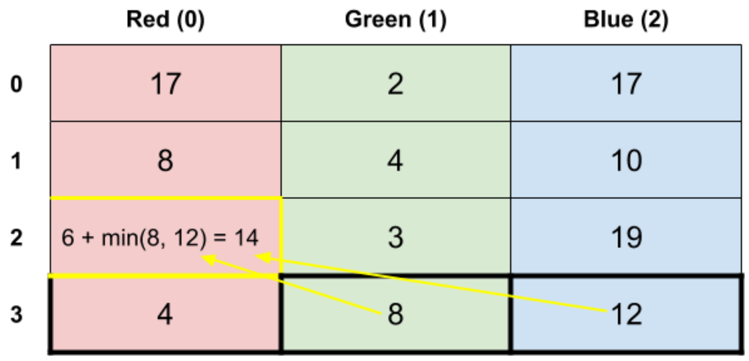
We can, therefore, calculate the cost of each subproblem, starting from the ones with the highest house numbers, and write the results directly into the input array. In effect, we will replace each single-house cost value in the array with the cost of painting the house that color and the minimum cost to paint all the houses after it. This is almost the same as what we did on the tree. The only difference is that we are only doing each calculation once and we are writing results directly into the input table. It is bottom up, because we are solving the "lower" problems first, and then the "higher" ones once we've solved all the lower ones that they depend on.

First thing to realize is that we don't need to do anything to the last row. Like in the tree, these costs are the total costs because there are no further houses after them.



| | **Red (0)** | **Green (1)** | **Blue (2)** | |
|---|---|---|---|---|
| **0** | 17 | 2 | 17 | |
| **1** | 8 | 4 | 10 | |
| **2** | 6 | 3 | 19 | |
| **3** | 4 | 8 | 12 | ✓ |

Now, what about the second-to-last row? Well, we know that if we painted that house red, that it'd cost itself and the cheapest out of blue and green from the next row, which is 8. So the total cost there would be 14, and we can put that into the cell.



| | **Red (0)** | **Green (1)** | **Blue (2)** |
|---|---|---|---|
| **0** | 17 | 2 | 17 |
| **1** | 8 | 4 | 10 |
| **2** | 6 + min(8, 12) = 14 | 3 | 19 |
| **3** | 4 | 8 | 12 |

Just like we did with the tree, we can work our way up through the grid, repeatedly applying the same algorithm to determine the total value for each cell. Once we have updated all the cells, we then simply need to take the minimum value from the first row and return it.

**Algorithm**

The algorithm is straightforward. We iterate backwards over all the rows in the grid (starting from the second-to-last) and calculate a total cost for each cell in the way

```
class Solution {

    public int minCost(int[][] costs) {



        for (int n = costs.length - 2; n >= 0; n--) {

            // Total cost of painting the nth house red.

            costs[n][0] += Math.min(costs[n + 1][1], costs[n + 1][2]);

            // Total cost of painting the nth house green.

            costs[n][1] += Math.min(costs[n + 1][0], costs[n + 1][2]);

            // Total cost of painting the nth house blue.

            costs[n][2] += Math.min(costs[n + 1][0], costs[n + 1][1]);

        }



        if (costs.length == 0) return 0;



        return Math.min(Math.min(costs[0][0], costs[0][1]), costs[0][2]);

    }

}
```

You could also avoid the hardcoding of the colors and instead iterate over the colors. This approach will be covered in the solution article for the follow up question where there are `m` colors instead of just 3.

**Complexity Analysis**

- Time Complexity : O(n)$O(n)$.

  Finding the minimum of two values and adding it to another value is an O(1)$O(1)$ operation. We are doing these O(1)$O(1)$ operations for 3 \cdot (n - 1)$3 \cdot (n-1)$ cells in the grid. Expanding that out, we get 3 \cdot n - 3$3 \cdot n - 3$. The constants don't matter in big-oh notation, so we drop them, leaving us with O(n)$O(n)$.

*A word of warning:* This would *not* be correct if there were $m$ colors. For this particular problem we were told there's only $3$ colors. However, a logical follow-up question would be to make the code work for any number of colors. In that case, the time complexity would actually be $O(n \cdot m)$, because $m$ is not a constant, whereas $3$ is. If this confused you, I'd recommend reading up on big-oh notation.

- Space Complexity : $O(1)$

  We don't allocate any new data structures, and are only using a few local variables. All the work is done directly into the input array. Therefore, the algorithm is in-place, requiring constant extra space.

---

## Approach 5: Dynamic Programming with Optimized Space Complexity

**Intuition**

Overwriting the input array isn't always desirable. What if, for example, other functions also needed to use that same array?

We could allocate our own array and then continue in the same way as approach 4. This would bring our space complexity up to $O(n)$ (for the same reason the time complexity is $O(n)$, the constants are dropped in big-oh notation).

Using $O(n)$ space isn't necessary though—we can further optimize the space complexity. Remember how the dynamic programming animation blanked out rows to show we'd no longer be looking at them? We only needed to look at the previous row, and the row we're currently working on. The rest could have been thrown away. So to avoid overwriting the input, we keep track of the previous row and the current row as length-3 arrays.

This space-optimization technique applies to many dynamic programming problems. As a general rule, I'd recommend first trying to come up with an algorithm that has optimal time complexity, and then looking at if you can trim down the space complexity.

**Algorithm**

It's up to you whether you do this using length-3 arrays or variables. Arrays are better in terms of writing clean code though. They will also be easier to adapt if you were asked to make the algorithm work with $m$ colors. I have chosen to use arrays here as keeping track of 6 seperate variables is too messy.

The `previous_row` starts as being the last row of the input array. The `current_row` is the row `n` is currently up to (starts as the second to last row). At each step we update the values in `current_row` by adding values from `previous_row`. We then set `previous_row` to be `current_row` and go on to the next value of `n` where we repeat the process. At the end, the first row will be sitting in the `previous_row` variable, so we find the minimum like we did before.

Note that we have to be careful about not overwriting the `costs` array inadvertently. Any rows we take out of the array that will be *written* into will need to be copies. This can be done using `clone` in Java (suitable for an array of primitive types such as integers) and `copy.deepcopy` in Python.

```
class Solution {

    public int minCost(int[][] costs) {
        if (costs.length == 0) return 0;

        int[] previousRow = costs[costs.length -1];

        for (int n = costs.length - 2; n >= 0; n--) {

            int[] currentRow = costs[n].clone();
            // Total cost of painting the nth house red.
            currentRow[0] += Math.min(previousRow[1], previousRow[2]);
            // Total cost of painting the nth house green.
            currentRow[1] += Math.min(previousRow[0], previousRow[2]);
            // Total cost of painting the nth house blue.
            currentRow[2] += Math.min(previousRow[0], previousRow[1]);
            previousRow = currentRow;
        }

        return Math.min(Math.min(previousRow[0], previousRow[1]), previousRow[2]);
    }
}
```

Thanks so much to @bitbleach for pointing out that the original code I had here was over writing the input array! Because this is such an easy mistake to make, **I've kept the original code for reference.**

```
/* This code OVERWRITES the input array! */

class Solution {
    public int minCost(int[][] costs) {

        if (costs.length == 0) return 0;
        int[] previousRow = costs[costs.length -1];


        for (int n = costs.length - 2; n >= 0; n--) {


            /* PROBLEMATIC CODE IS HERE
             * This line here is NOT making a copy of the original, it's simply
             * making a reference to it Therefore, any writes into currentRow
             * will also be written into "costs". This is not what we wanted!
             */
            int[] currentRow = costs[n];


            // Total cost of painting the nth house red.
            currentRow[0] += Math.min(previousRow[1], previousRow[2]);
            // Total cost of painting the nth house green.
            currentRow[1] += Math.min(previousRow[0], previousRow[2]);
            // Total cost of painting the nth house blue.
            currentRow[2] += Math.min(previousRow[0], previousRow[1]);
            previousRow = currentRow;
        }


        return Math.min(Math.min(previousRow[0], previousRow[1]), previousRow[2]);
    }
}
```

**Complexity Analysis**

- Time Complexity : O(n)$O(n)$.

  Same as previous approach.

- Space Complexity : O(1)$O(1)$

We're "remembering" up to $6$ calculations at a time (using 2 x length-3 arrays). Because this is actually a constant, the space complexity is still $O(1)$.

Like the time complexity though, this analysis is dependent on there being a constant number of colors (i.e. 3). If the problem was changed to be $m$ colors, then the space complexity would become $O(m)$ as we'd need to keep track of a couple of length-m arrays.

---

## Justifying why this is a Dynamic Programming Problem

Many dynamic programming problems have very straightforward solutions. As you get more experience with them, you'll gain a better intuition for when a problem might be solvable with dynamic programming, and you'll also get better at quickly identifying the overlapping subproblems (e.g. that painting the 3rd house green will have the same total cost regardless of whether the 2nd house was blue or red). Thinking about the tree structure can help too for identifying those subproblems, although you won't always need to draw it out fully like we did here.

Remember that a **subproblem** is any call to the recursive function. Subproblems are solved either as a base case (in this case a simple lookup from the table and no further calculations) or by looking at the solutions of a bunch of lower down subproblems. In dynamic programming lingo, we say that this problem has an **optimal substructure**. This means that the optimal cost for each **subproblem** is constructed from the **optimal cost** of **subproblems** below it. This is the same property that must be true for greedy algorithms to work.

If, for example, we hadn't been able to choose the minimum and know it was optimal (perhaps because it would impact a choice further up the tree) then there would *not* have been **optimal substructure**.

In addition this problem also had **overlapping subproblems**. This just means that the lower subproblems were often shared (remember how the tree had lots of branches that looked the same?)

Problems that have **optimal substructure** can be solved with greedy algorithms. If they *also* have **overlapping subproblems**, then they can be solved with dynamic programming algorithms.