**LinkedIn**

Introduction

Top interview questions asked by LinkedIn as voted by the community.

This list will be kept up to date as frequent as possible.

Array and Strings

 Two Sum

 Valid Number

 Text Justification

 Minimum Window Substring

**Shortest Word Distance**

 Integer to English Words

 Find the Celebrity

**Can Place Flowers**

 Isomorphic Strings

Trees and Graphs

 Binary Tree Level Order Traversal

 Maximum Depth of Binary Tree

 Word Ladder

**Binary Tree Upside Down**

 Number of Islands

 Lowest Common Ancestor of a Binary Search Tree

 Lowest Common Ancestor of a Binary Tree

**Closest Binary Search Tree Value II**

**Find Leaves of Binary Tree**

**Second Minimum Node In a Binary Tree**

Recursion and Backtracking

 Combination Sum

 Permutations

 Permutations II

**Factor Combinations**

**Nested List Weight Sum**

**Nested List Weight Sum II**

Heap, Queue, Stack

 Valid Parentheses

 Evaluate Reverse Polish Notation

 Kth Largest Element in an Array

 Exclusive Time of Functions

Sorting and Searching

 Search in Rotated Sorted Array

 Find First and Last Position of Element in Sorted Array

 Pow(x, n)

 Merge Intervals

 Sqrt(x)

 Random Pick with Weight

Dynamic Programming

 Maximum Subarray

**Edit Distance**

 Maximum Product Subarray

**Paint House**

**Paint House II**

**Partition to K Equal Sum Subsets**

Design

 Two Sum III - Data structure design

**Shortest Word Distance II**

 Serialize and Deserialize Binary Tree

 Flatten Nested List Iterator

 Insert Delete GetRandom O(1)

**All O`one Data Structure**

**Max Stack**

 Design HashMap

**Shortest Word Distance**

Given a list of words and two words *word1* and *word2*, return the shortest distance between these two words in the list.

**Example:**  
Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

**Input:** *word1* = “coding”, *word2* = “practice”

**Output:** 3

**Input:** *word1* = "makes", *word2* = "coding"

**Output:** 1

**Note:**  
You may assume that *word1* **does not equal to** *word2*, and *word1* and *word2* are both in the list.

## 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)*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)*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*⋅*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.

**Can Place Flowers**

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in **adjacent** plots.

Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule.

**Example 1:**

**Input:** flowerbed = [1,0,0,0,1], n = 1

**Output:** true

**Example 2:**

**Input:** flowerbed = [1,0,0,0,1], n = 2

**Output:** false

**Constraints:**

* 1 <= flowerbed.length <= 2 \* 104
* flowerbed[i] is 0 or 1.
* There are no two adjacent flowers in flowerbed.
* 0 <= n <= flowerbed.length

## Solution

#### **Approach #1 Single Scan [Accepted]**

The solution is very simple. We can find out the extra maximum number of flowers, count*count*, that can be planted for the given flowerbed*flowerbed* arrangement. To do so, we can traverse over all the elements of the flowerbed*flowerbed* and find out those elements which are 0(implying an empty position). For every such element, we check if its both adjacent positions are also empty. If so, we can plant a flower at the current position without violating the no-adjacent-flowers-rule. For the first and last elements, we need not check the previous and the next adjacent positions respectively.

If the count*count* obtained is greater than or equal to n*n*, the required number of flowers to be planted, we can plant n*n* flowers in the empty spaces, otherwise not.

|  |
| --- |
| public class Solution {  public boolean canPlaceFlowers(int[] flowerbed, int n) {  int i = 0, count = 0;  while (i < flowerbed.length) {  if (flowerbed[i] == 0 && (i == 0 || flowerbed[i - 1] == 0) && (i == flowerbed.length - 1 || flowerbed[i + 1] == 0)) {  flowerbed[i] = 1;  count++;  }  i++;  }  return count >= n;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). A single scan of the flowerbed*flowerbed* array of size n*n* is done.
* Space complexity : O(1)*O*(1). Constant extra space is used.

#### **Approach #2 Optimized [Accepted]**

**Algorithm**

Instead of finding the maximum value of count*count* that can be obtained, as done in the last approach, we can stop the process of checking the positions for planting the flowers as soon as count*count* becomes equal to n*n*. Doing this leads to an optimization of the first approach. If count*count* never becomes equal to n*n*, n*n* flowers can't be planted at the empty positions.

|  |
| --- |
| public class Solution {  public boolean canPlaceFlowers(int[] flowerbed, int n) {  int i = 0, count = 0;  while (i < flowerbed.length) {  if (flowerbed[i] == 0 && (i == 0 || flowerbed[i - 1] == 0) && (i == flowerbed.length - 1 || flowerbed[i + 1] == 0)) {  flowerbed[i++] = 1;  count++;  }  if(count>=n)  return true;  i++;  }  return false;  }  } |

**Complexity Analysis**

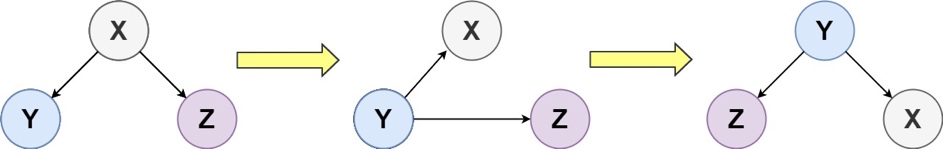
* Time complexity : O(n)*O*(*n*). A single scan of the flowerbed*flowerbed* array of size n*n* is done.
* Space complexity : O(1)*O*(1). Constant extra space is used.

**Binary Tree Upside Down**

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

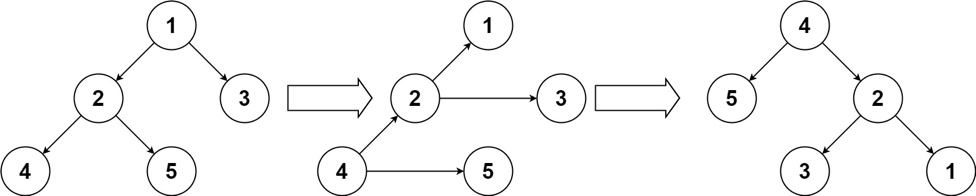
You can turn a binary tree upside down with the following steps:

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 node in the given tree has either **0 or 2 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 node has either 0 or 2 children.

**Closest Binary Search Tree Value II**

Given a non-empty binary search tree and a target value, find *k* values in the BST that are closest to the target.

**Note:**

* Given target value is a floating point.
* You may assume *k* is always valid, that is: *k* ≤ total nodes.
* You are guaranteed to have only one unique set of *k* values in the BST that are closest to the target.

**Example:**

**Input:** root = [4,2,5,1,3], target = 3.714286, and k = 2

4

/ \

2 5

/ \

1 3

**Output:** [4,3]

**Follow up:**  
Assume that the BST is balanced, could you solve it in less than *O*(*n*) runtime (where *n* = total nodes)?

Hide Hint #1

Consider implement these two helper functions:

1. getPredecessor(N), which returns the next smaller node to N.
2. getSuccessor(N), which returns the next larger node to N.

   Hide Hint #2

Try to assume that each node has a parent pointer, it makes the problem much easier.

   Hide Hint #3

Without parent pointer we just need to keep track of the path from the root to the current node using a stack.

   Hide Hint #4

You would need two stacks to track the path in finding predecessor and successor node separately.

## Solution

#### **Overview**

The problem is a BST variation of the "kth-smallest" classical problem. It is popular both in Google and Facebook, but these two companies are waiting for you to show different approaches to this problem. We're proposing 3 solutions here, and it's more an overview.

**Prerequisites**

Because of that, you might want first to check out the list of prerequisites:

* [Inorder traversal of BST is an array sorted in the ascending order.](https://leetcode.com/problems/delete-node-in-a-bst/solution/) To compute inorder traversal follow the direction Left -> Node -> Right.
* [Closest BST value: find one closest element](https://leetcode.com/problems/closest-binary-search-tree-value/solution/).
* [kth-smallest problem for the array could be solved by using heap in \mathcal{O}(N \log k)O(*N*log*k*) time, or by using quickselect in \mathcal{O}(N)O(*N*) time.](https://leetcode.com/problems/top-k-frequent-elements/solution/)

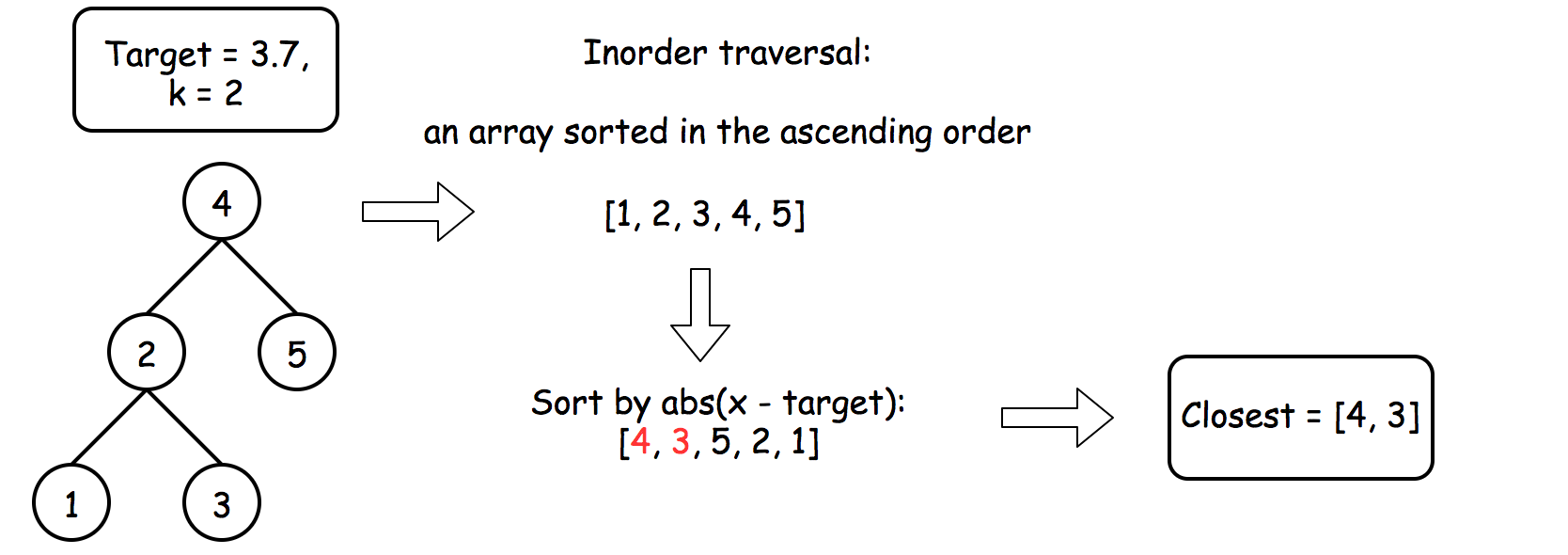
**Google vs. Facebook**

There are three ways to solve the problem:

* Approach 1. Sort, \mathcal{O}(N \log N)O(*N*log*N*) time. The idea is to convert BST into an array, sort it by the distance to the target, and return the k closest elements.
* Approach 2. Facebook-friendly, heap, \mathcal{O}(N \log k)O(*N*log*k*) time. We could use the heap of capacity k, sorted by the distance to the target. It's not an optimal but very straightforward solution - traverse the tree, push the elements into the heap, and then return this heap. Facebook interviewer would insist on implementing this solution because the interviews are a bit shorter than Google ones, and it's important to get problem solved end-to-end.
* Approach 3. Google-friendly, quickselect, \mathcal{O}(N)O(*N*) time. [Here you could find a very detailed explanation of quickselect algorithm.](https://leetcode.com/problems/top-k-frequent-elements/solution/) In this article, we're going to provide a relatively brief implementation. Google guys usually prefer the best-time solutions, well-structured clean skeleton, even if you have no time to implement everything in time end-to-end.

#### **Approach 1: Recursive Inorder + Sort, O(N log N) time**

**Intuition**

 Figure 1. Sort.

The most straightforward approach is to build inorder traversal and then find the k closest elements using build-in sort.

**Algorithm**

* Build an inorder traversal array.
* Find the k closest to the target elements using build-in sort.

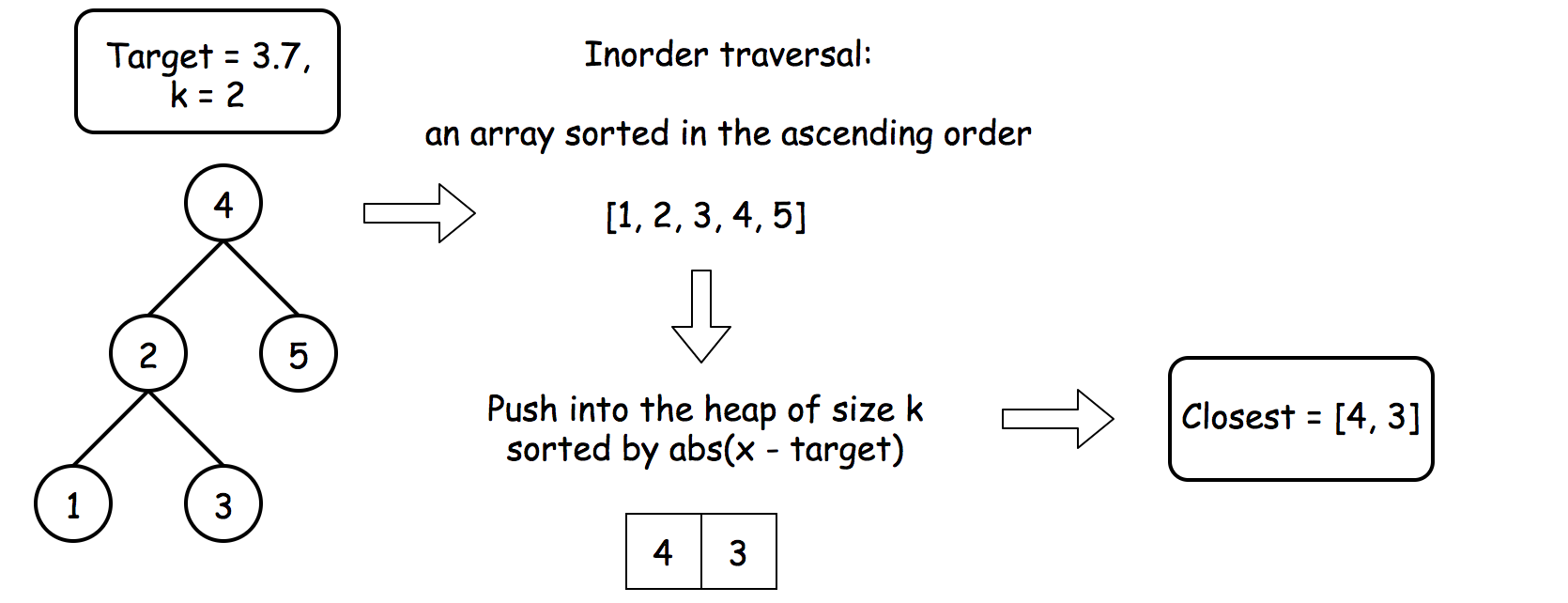
**Implementation**

|  |
| --- |
| class Solution {  public void inorder(TreeNode root, List<Integer> nums) {  if (root == null) return;  inorder(root.left, nums);  nums.add(root.val);  inorder(root.right, nums);  }  public List<Integer> closestKValues(TreeNode root, double target, int k) {  List<Integer> nums = new ArrayList();  inorder(root, nums);    Collections.sort(nums, new Comparator<Integer>() {  @Override  public int compare(Integer o1, Integer o2) {  return Math.abs(o1 - target) < Math.abs(o2 - target) ? -1 : 1;  }  });  return nums.subList(0, k);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N \log N)O(*N*log*N*). \mathcal{O}(N)O(*N*) to build inorder traversal and then \mathcal{O}(N \log N)O(*N*log*N*) to sort it.
* Space complexity: \mathcal{O}(N)O(*N*) to store list nums of N*N* elements.

#### **Approach 2: Recursive Inorder + Heap, O(N log k) time**

 Figure 2. Heap.

**Algorithm**

* Instantiate the heap with "less close element first" strategy so that the heap contains the elements that are closest to the target.
* Use inorder traversal to traverse the tree following the direction Left -> Node -> Right.
  + Push all elements into heap during the traversal, keeping the heap size less than or equal to k*k*.
* As a result, the heap contains k*k* elements that are closest to target. Convert it into a list and return.

**Implementation**

|  |
| --- |
| class Solution {  public void inorder(TreeNode r, List<Integer> nums, Queue<Integer> heap, int k) {  if (r == null)  return;    inorder(r.left, nums, heap, k);  heap.add(r.val);  if (heap.size() > k)  heap.remove();  inorder(r.right, nums, heap, k);  }  public List<Integer> closestKValues(TreeNode root, double target, int k) {  List<Integer> nums = new ArrayList();    // init heap 'less close element first'  Queue<Integer> heap = new PriorityQueue<>((o1, o2) -> Math.abs(o1 - target) > Math.abs(o2 - target) ? -1 : 1);  inorder(root, nums, heap, k);  return new ArrayList<>(heap);  }  } |

**Optimisations**

One could optimize the solution by adding the stop condition. Inorder traversal pops the elements in the sorted order. Hence once the distance of the current element to the target becomes greater than the distance of the first element in a heap, one could stop the computations. The overall worst-case time complexity would be still \mathcal{O}(N \log k)O(*N*log*k*), but the average time could be improved to \mathcal{O}(H \log k)O(*H*log*k*), where H*H* is a tree height.

**Complexity Analysis**

* Time complexity: \mathcal{O}(N \log k)O(*N*log*k*) to push N elements into the heap of the size k*k*.
* Space complexity: \mathcal{O}(k + H)O(*k*+*H*) to keep the heap of k elements and the recursion stack of the tree height.

#### **Approach 3: QuickSelect, O(N) time.**

**Hoare's selection algorithm**

Quickselect is a [textbook algorithm](https://en.wikipedia.org/wiki/Quickselect) typically used to solve the problems "find kth something": kth smallest, kth largest, etc. Like quicksort, quickselect was developed by [Tony Hoare](https://en.wikipedia.org/wiki/Tony_Hoare), and also known as Hoare's selection algorithm.

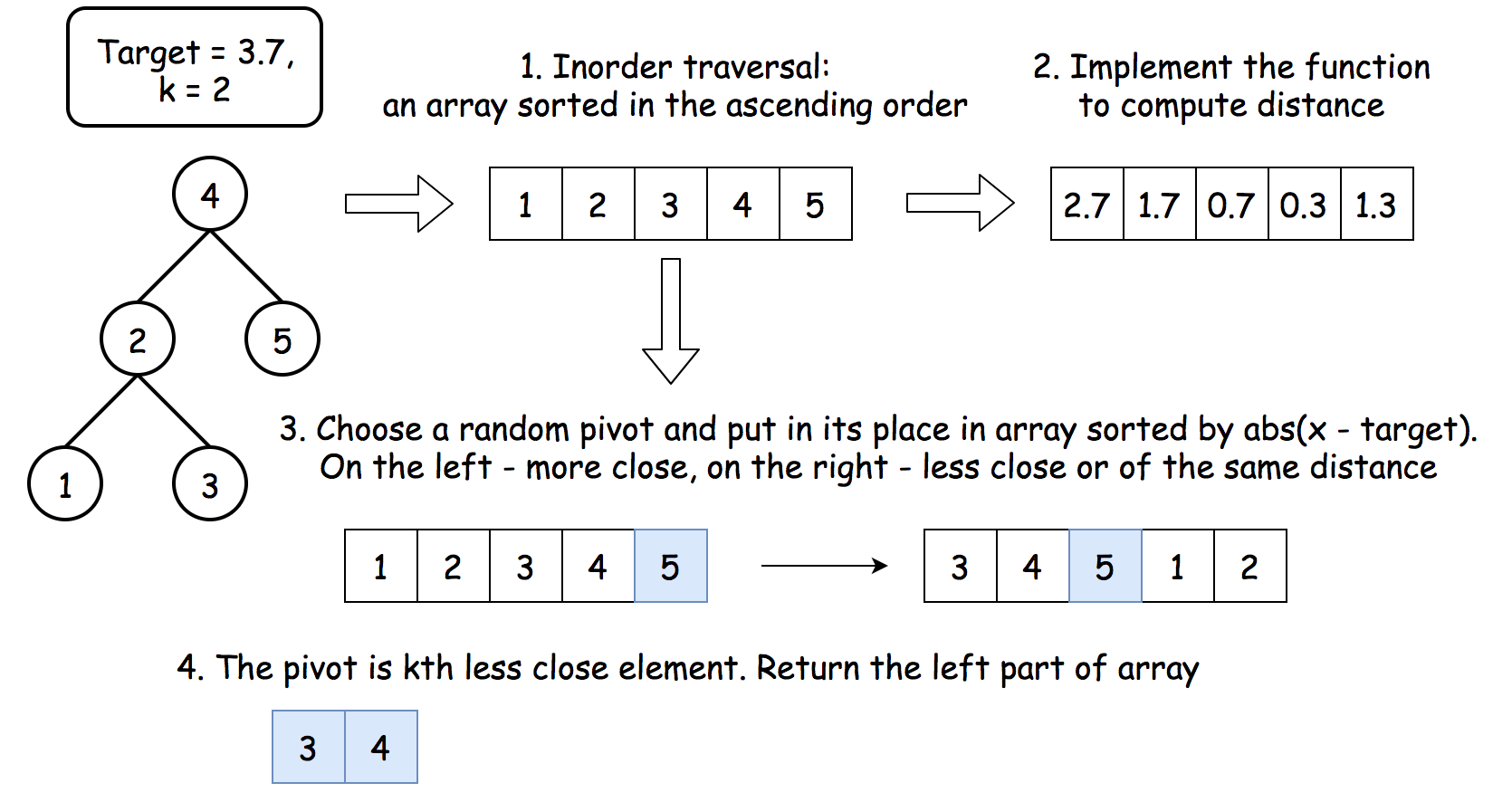
It has \mathcal{O}(N)O(*N*) average time complexity and widely used in practice. It is worth to note that its worst-case time complexity is \mathcal{O}(N^2)O(*N*2), although the probability of this worst-case is negligible.

The approach is the same as for quicksort.

One chooses a pivot and defines its position in a sorted array in a linear time using the so-called partition algorithm.

As an output, we have an array where the pivot is on its perfect position in the ascending sorted array, sorted by the frequency. All elements on the left of the pivot are more close to the target than the pivot, and all elements on the right are less close or on the same distance from the target.

The array is now split into two parts. If by chance, our pivot element took kth final position, then k*k* elements on the left are these k*k* closest elements we're looking for. If not, we can choose one more pivot and place it in its perfect position.

 Figure 3. Quickselect.

If that were a quicksort algorithm, one would have to process both parts of the array. That would result in \mathcal{O}(N \log N)O(*N*log*N*) time complexity. In this case, there is no need to deal with both parts since one knows in which part to search for kth closest element, and that reduces the average time complexity to \mathcal{O}(N)O(*N*).

**Algorithm**

The algorithm is relatively straightforward:

* Traverse the tree and convert it into array nums.
* Implement the simple function to compute the distance to the target. Note that the distance is not unique. That means we need a partition algorithm that works fine with duplicates.
* Work with nums array. Use a partition scheme (please check the next section) to place the pivot into its perfect position pivot\_index in the sorted array, move more close elements to the left of the pivot, and less close or of the same distance - to the right.
* Compare pivot\_index and k.
  + If pivot\_index == k, the pivot is the kth less close element, and all elements on the left are the k*k* closest elements to the target. Return these elements.
  + Otherwise, choose the side of the array to proceed recursively.

**Hoare's Partition vs. Lomuto's Partition**

There is a zoo of partition algorithms. The most simple one is [Lomuto's Partition Scheme](https://en.wikipedia.org/wiki/Quicksort#Lomuto_partition_scheme).

The drawback of Lomuto's partition is that it fails with duplicates.

Here we work with an array of unique elements, but they are compared by the distances to the target, which are not unique. That's why we choose Hoare's Partition here.

Hoare's partition is more efficient than Lomuto's partition because it does three times fewer swaps on average, and creates efficient partitions even when all values are equal.

Here is how it works:

* Move pivot at the end of the array using swap.
* Set the pointer at the beginning of the array store\_index = left.
* Iterate over the array and move all more close elements to the left swap(store\_index, i). Move store\_index one step to the right after each swap.
* Move the pivot to its final place, and return this index.

**Implementation**

|  |
| --- |
| class Solution {  List<Integer> nums;  double target;    public void swap(int a, int b) {  int tmp = nums.get(a);  nums.set(a, nums.get(b));  nums.set(b, tmp);  }    public void inorder(TreeNode r, List<Integer> nums) {  if (r == null)  return;    inorder(r.left, nums);  nums.add(r.val);  inorder(r.right, nums);  }  public int partition(int left, int right, int pivotIndex) {  double pivotDist = dist(pivotIndex);  // 1. move pivot to end  swap(pivotIndex, right);  int storeIndex = left;  // 2. move more close elements to the left  for (int i = left; i <= right; i++) {  if (dist(i) < pivotDist) {  swap(storeIndex, i);  storeIndex++;  }  }  // 3. move pivot to its final place  swap(storeIndex, right);  return storeIndex;  }    public void quickselect(int left, int right, int kSmallest) {  /\*  Sort a list within left..right till kth less close element  takes its place.  \*/  // base case: the list contains only one element  if (left >= right) return;    // select a random pivot\_index  Random randomNum = new Random();  int pivotIndex = left + randomNum.nextInt(right - left);  // find the pivot position in a sorted list  pivotIndex = partition(left, right, pivotIndex);  // if the pivot is in its final sorted position  if (kSmallest == pivotIndex) {  return;  } else if (kSmallest < pivotIndex) {  // go left  quickselect(left, pivotIndex - 1, kSmallest);  } else {  // go right  quickselect(pivotIndex + 1, right, kSmallest);  }  }    public double dist(int idx) {  return Math.abs(nums.get(idx) - target);  }  public List<Integer> closestKValues(TreeNode root, double target, int k) {  nums = new ArrayList();  this.target = target;  inorder(root, nums);  quickselect(0, nums.size() - 1, k);  return nums.subList(0, k);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), \mathcal{O}(N^2)O(*N*2) in the worst case. [Please refer to this card for the good detailed explanation of Master Theorem](https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/2871/). Master Theorem helps to get an average complexity by writing the algorithm cost as T(N) = a T(N / b) + f(N)*T*(*N*)=*aT*(*N*/*b*)+*f*(*N*). Here we have an example of Master Theorem case III: T(N) = T \left(\frac{N}{2}\right) + N*T*(*N*)=*T*(2*N*​)+*N*, that results in \mathcal{O}(N)O(*N*) time complexity. That's the case of random pivots.

In the worst-case of constantly bad chosen pivots, the problem is not divided by half at each step, it becomes just one element less, that leads to \mathcal{O}(N^2)O(*N*2) time complexity. It happens, for example, if at each step you choose the pivot not randomly, but take the rightmost element. For the random pivot choice, the probability of having such a worst-case is negligibly small.

* Space complexity: \mathcal{O}(N)O(*N*) to store nums.

**Find Leaves of Binary Tree**

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

**Example:**

**Input:** [1,2,3,4,5]

  1

/ \

2 3

/ \

4 5

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

**Explanation:**

1. Removing the leaves [4,5,3] would result in this tree:

1

/

2

 2. Now removing the leaf [2] would result in this tree:

1

 3. Now removing the leaf [1] would result in the empty tree:

[]

[[3,5,4],[2],[1]], [[3,4,5],[2],[1]], etc, are also consider correct answers since per each level it doesn't matter the order on which elements are returned.

#### **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 ith height will be appear in the ith 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))

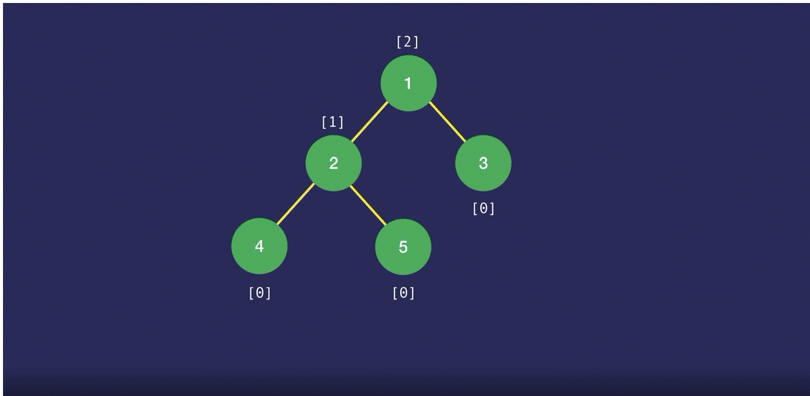
Where \text{root.left}root.left and \text{root.right}root.right are left and right children of the root respectively

**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.

Attached below is the video which shows the calculation of height in a height-first-search manner for the binary tree given in the example.



|  |
| --- |
| class Solution {  private List<Pair<Integer, Integer>> pairs;    private int getHeight(TreeNode root) {    // return -1 for null nodes  if (root == null) 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 = Math.max(leftHeight, rightHeight) + 1;  // collect the pair -> (height, val)  this.pairs.add(new Pair<Integer, Integer>(currHeight, root.val));  // return the height of the current node  return currHeight;  }    public List<List<Integer>> findLeaves(TreeNode root) {  this.pairs = new ArrayList<>();    getHeight(root);    // sort all the (height, val) pairs  Collections.sort(this.pairs, Comparator.comparing(p -> p.getKey()));    int n = this.pairs.size(), height = 0, i = 0;  List<List<Integer>> solution = new ArrayList<>();    while (i < n) {  List<Integer> nums = new ArrayList<>();  while (i < n && this.pairs.get(i).getKey() == height) {  nums.add(this.pairs.get(i).getValue());  i++;  }  solution.add(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 List<List<Integer>> solution;    private int getHeight(TreeNode root) {    // return -1 for null nodes  if (root == null) {  return -1;  }    // first calculate the height of the left and right children  int leftHeight = getHeight(root.left);  int rightHeight = getHeight(root.right);    int currHeight = Math.max(leftHeight, rightHeight) + 1;    if (this.solution.size() == currHeight) {  this.solution.add(new ArrayList<>());  }    this.solution.get(currHeight).add(root.val);    return currHeight;  }    public List<List<Integer>> findLeaves(TreeNode root) {  this.solution = new ArrayList<>();    getHeight(root);    return this.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 and storing all the pairs at the correct position also takes O(N)*O*(*N*) time. Hence overall time complexity of this approach is O(N)*O*(*N*).
* Space Complexity: O(N)*O*(*N*), the space used by solution array.

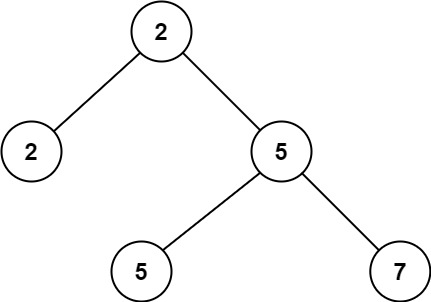
**Second Minimum Node In a Binary Tree**

Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes. More formally, the property root.val = min(root.left.val, root.right.val) always holds.

Given such a binary tree, you need to output the **second minimum** value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

**Example 1:**

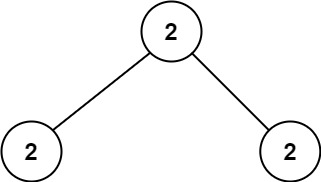


**Input:** root = [2,2,5,null,null,5,7]

**Output:** 5

**Explanation:** The smallest value is 2, the second smallest value is 5.

**Example 2:**



**Input:** root = [2,2,2]

**Output:** -1

**Explanation:** The smallest value is 2, but there isn't any second smallest value.

**Constraints:**

* The number of nodes in the tree is in the range [1, 25].
* 1 <= Node.val <= 231 - 1
* root.val == min(root.left.val, root.right.val) for each internal node of the tree.

## Solution

#### **Approach #1: Brute Force [Accepted]**

**Intuition and Algorithm**

Traverse the tree with a depth-first search, and record every unique value in the tree using a Set structure uniques.

Then, we'll look through the recorded values for the second minimum. The first minimum must be \text{root.val}root.val.

|  |
| --- |
| class Solution {  public void dfs(TreeNode root, Set<Integer> uniques) {  if (root != null) {  uniques.add(root.val);  dfs(root.left, uniques);  dfs(root.right, uniques);  }  }  public int findSecondMinimumValue(TreeNode root) {  Set<Integer> uniques = new HashSet<Integer>();  dfs(root, uniques);  int min1 = root.val;  long ans = Long.MAX\_VALUE;  for (int v : uniques) {  if (min1 < v && v < ans) ans = v;  }  return ans < Long.MAX\_VALUE ? (int) ans : -1;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the total number of nodes in the given tree. We visit each node exactly once, and scan through the O(N)*O*(*N*) values in unique once.
* Space Complexity: O(N)*O*(*N*), the information stored in uniques.

#### **Approach #2: Ad-Hoc [Accepted]**

**Intuition and Algorithm**

Let \text{min1 = root.val}min1 = root.val. When traversing the tree at some node, \text{node}node, if \text{node.val > min1}node.val > min1, we know all values in the subtree at \text{node}node are at least \text{node.val}node.val, so there cannot be a better candidate for the second minimum in this subtree. Thus, we do not need to search this subtree.

Also, as we only care about the second minimum \text{ans}ans, we do not need to record any values that are larger than our current candidate for the second minimum, so unlike Approach #1 we can skip maintaining a Set of values(uniques) entirely.

|  |
| --- |
| class Solution {  int min1;  long ans = Long.MAX\_VALUE;  public void dfs(TreeNode root) {  if (root != null) {  if (min1 < root.val && root.val < ans) {  ans = root.val;  } else if (min1 == root.val) {  dfs(root.left);  dfs(root.right);  }  }  }  public int findSecondMinimumValue(TreeNode root) {  min1 = root.val;  dfs(root);  return ans < Long.MAX\_VALUE ? (int) ans : -1;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the total number of nodes in the given tree. We visit each node at most once.
* Space Complexity: O(N)*O*(*N*). The information stored in \text{ans}ans and \text{min1}min1 is O(1)*O*(1), but our depth-first search may store up to O(h) = O(N)*O*(*h*)=*O*(*N*) information in the call stack, where h*h* is the height of the tree.

**Factor Combinations**

Numbers can be regarded as product of its factors. For example,

8 = 2 x 2 x 2;

= 2 x 4.

Write a function that takes an integer *n* and return all possible combinations of its factors.

**Note:**

1. You may assume that *n* is always positive.
2. Factors should be greater than 1 and less than *n*.

**Example 1:**

**Input:** 1

**Output:** []

**Example 2:**

**Input:** 37

**Output:**[]

**Example 3:**

**Input:** 12

**Output:**

[

[2, 6],

[2, 2, 3],

[3, 4]

]

**Example 4:**

**Input:** 32

**Output:**

[

[2, 16],

[2, 2, 8],

[2, 2, 2, 4],

[2, 2, 2, 2, 2],

[2, 4, 4],

[4, 8]

]

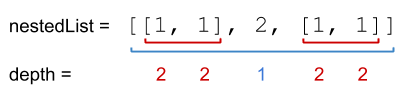
**Nested List Weight Sum**

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:**

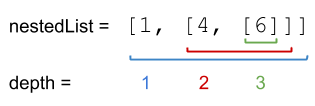


**Input:** nestedList = [[1,1],2,[1,1]]

**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 Article

#### **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*×*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.

**Implementation**

|  |
| --- |
| class Solution {  public int depthSum(List<NestedInteger> nestedList) {  return dfs(nestedList, 1);  }  private int dfs(List<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;  }  } |

**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 *NN*.

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⋅*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](https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/1372/). The algorithm fully processes each depth before moving to the next one.

**Implementation**

|  |
| --- |
| class Solution {  public int depthSum(List<NestedInteger> nestedList) {  Queue<NestedInteger> queue = new LinkedList<>();  queue.addAll(nestedList);  int depth = 1;  int total = 0;  while (!queue.isEmpty()) {  int size = queue.size();  for (int i = 0; i < size; i++) {  NestedInteger nested = queue.poll();  if (nested.isInteger()) {  total += nested.getInteger() \* depth;  } else {  queue.addAll(nested.getList());  }  }  depth++;  }  return total;  }  } |

**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*).

**Nested List Weight Sum II**

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Different from the [previous question](https://leetcode.com/problems/nested-list-weight-sum/) where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

**Example 1:**

**Input:** [[1,1],2,[1,1]]

**Output:** 8

**Explanation:** Four 1's at depth 1, one 2 at depth 2.

**Example 2:**

**Input:** [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.

**Edit Distance**

Given two strings word1 and word2, return the minimum number of operations required to convert *word1* to *word2*.

You have the following three operations permitted on a word:

* Insert a character
* Delete a character
* Replace a character

**Example 1:**

**Input:** word1 = "horse", word2 = "ros"

**Output:** 3

**Explanation:**

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

**Example 2:**

**Input:** word1 = "intention", word2 = "execution"

**Output:** 5

**Explanation:**

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

**Constraints:**

* 0 <= word1.length, word2.length <= 500
* word1 and word2 consist of lowercase English letters.

## Solution

#### **Intuition**

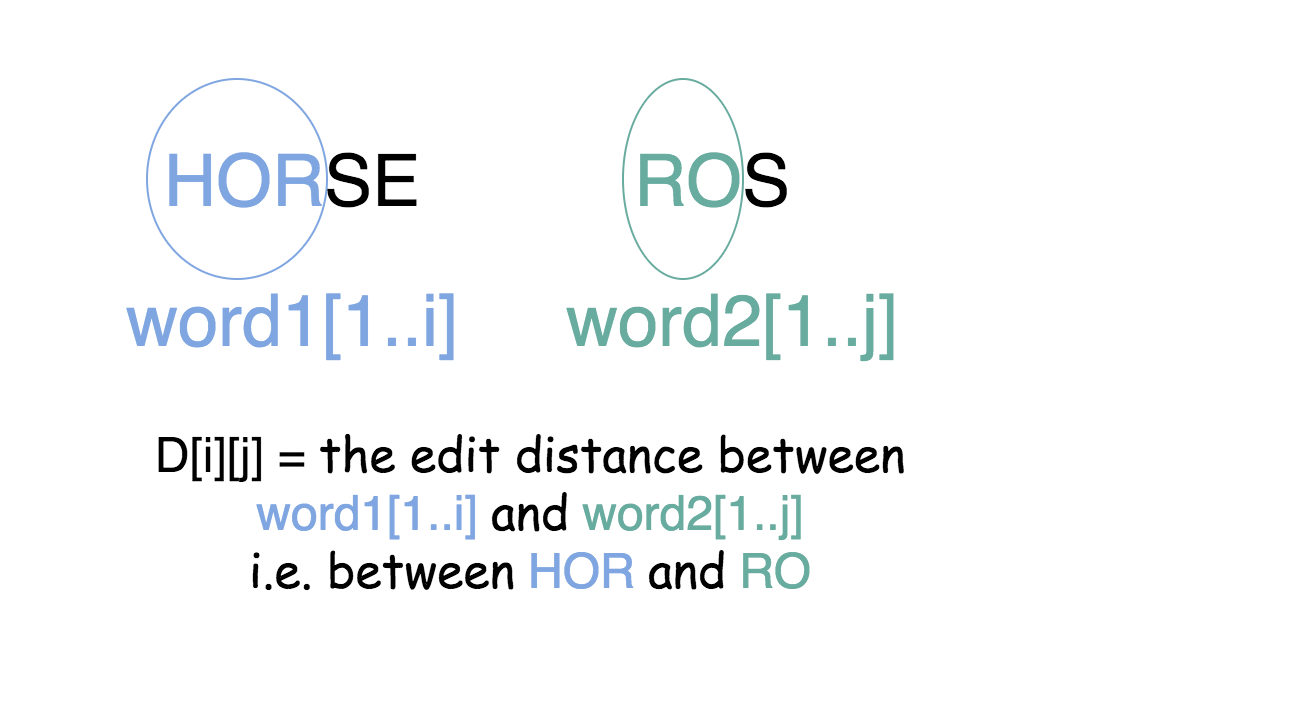
The edit distance algorithm is very popular among the data scientists. It's one of the basic algorithms used for evaluation of machine translation and speech recognition.

The naive approach would be to check for all possible edit sequences and choose the shortest one in-between. That would result in an exponential complexity and it's an overkill since we actually don't need to have all possible edit sequences but just the shortest one.

#### **Approach 1: Dynamic Programming**

The idea would be to reduce the problem to simple ones. For example, there are two words, horse and ros and we want to compute an edit distance D for them. One could notice that it seems to be more simple for short words and so it would be logical to relate an edit distance D[n][m] with the lengths n and m of input words.

Let's go further and introduce an edit distance D[i][j] which is an edit distance between the first i characters of word1 and the first j characters of word2.



It turns out that one could compute D[i][j], knowing D[i - 1][j], D[i][j - 1] and D[i - 1][j - 1].

There is just one more character to add into one or both strings and the formula is quite obvious.

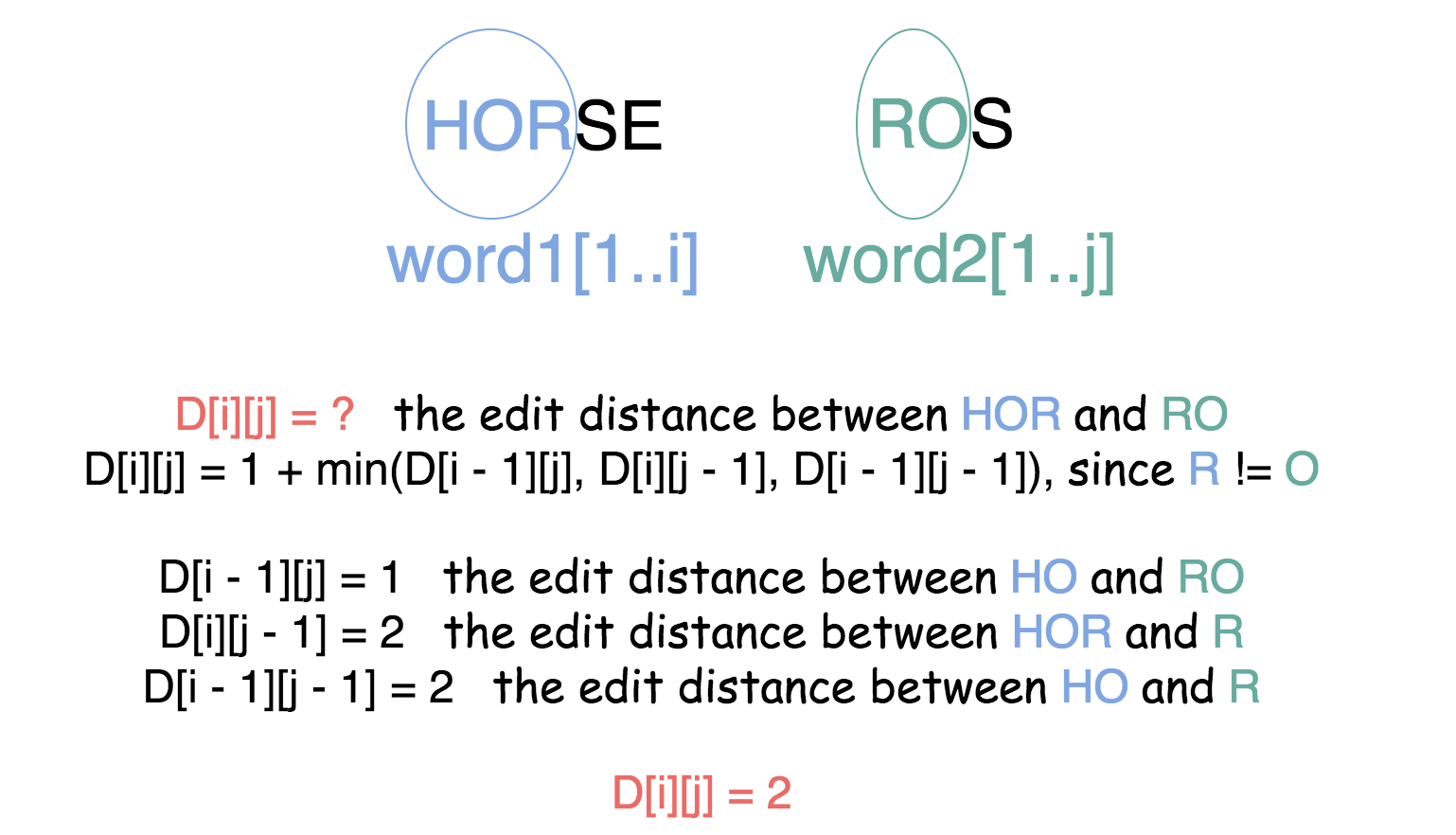
If the last character is the same, i.e. word1[i] = word2[j] then

D[i][j] = 1 + \min(D[i - 1][j], D[i][j - 1], D[i - 1][j - 1] - 1)*D*[*i*][*j*]=1+min(*D*[*i*−1][*j*],*D*[*i*][*j*−1],*D*[*i*−1][*j*−1]−1)

and if not, i.e. word1[i] != word2[j] we have to take into account the replacement of the last character during the conversion.

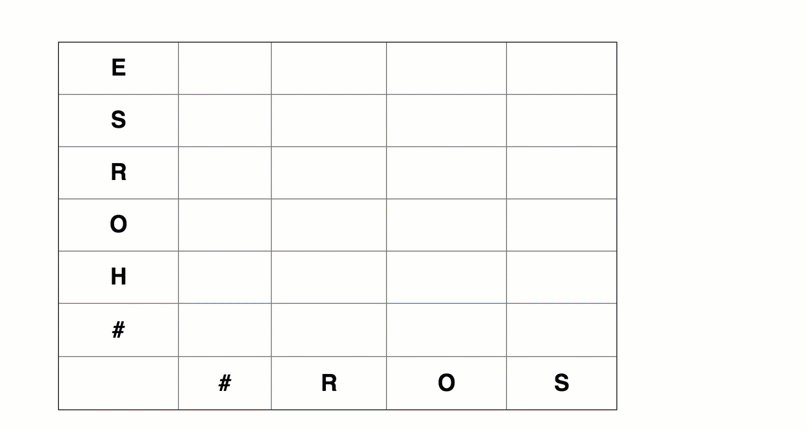
D[i][j] = 1 + \min(D[i - 1][j], D[i][j - 1], D[i - 1][j - 1])*D*[*i*][*j*]=1+min(*D*[*i*−1][*j*],*D*[*i*][*j*−1],*D*[*i*−1][*j*−1])

So each step of the computation would be done based on the previous computation, as follows:



The obvious base case is an edit distance between the empty string and non-empty string that means D[i][0] = i and D[0][j] = j.

Now we have everything to actually proceed to the computations



|  |
| --- |
| class Solution {  public int minDistance(String word1, String word2) {  int n = word1.length();  int m = word2.length();  // if one of the strings is empty  if (n \* m == 0)  return n + m;  // array to store the convertion history  int [][] d = new int[n + 1][m + 1];  // init boundaries  for (int i = 0; i < n + 1; i++) {  d[i][0] = i;  }  for (int j = 0; j < m + 1; j++) {  d[0][j] = j;  }  // DP compute  for (int i = 1; i < n + 1; i++) {  for (int j = 1; j < m + 1; j++) {  int left = d[i - 1][j] + 1;  int down = d[i][j - 1] + 1;  int left\_down = d[i - 1][j - 1];  if (word1.charAt(i - 1) != word2.charAt(j - 1))  left\_down += 1;  d[i][j] = Math.min(left, Math.min(down, left\_down));  }  }  return d[n][m];  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(m n)O(*mn*) as it follows quite straightforward for the inserted loops.
* Space complexity : \mathcal{O}(m n)O(*mn*) since at each step we keep the results of all previous computations.

**Paint House**

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 a *n* x *3* cost matrix. 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... Find 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 = []

**Output:** 0

**Example 3:**

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

**Output:** 2

**Constraints:**

* costs.length == n
* costs[i].length == 3
* 0 <= 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](https://leetcode.com/explore/learn/card/n-ary-tree/) or the [Binary Trees module](https://leetcode.com/explore/learn/card/data-structure-tree/). 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.

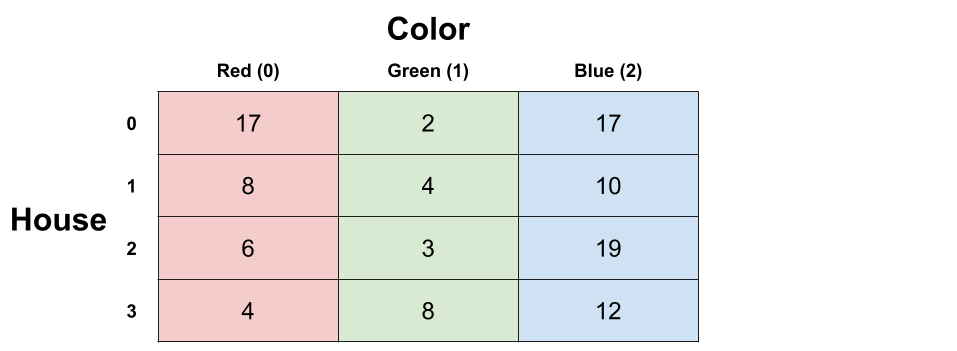
#### **Approach 1: Brute force**

**Intuition**

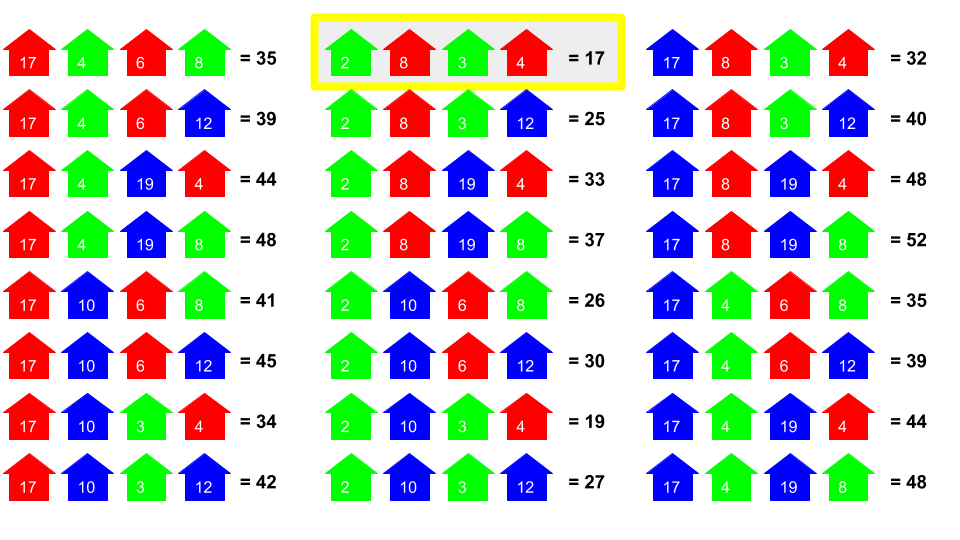
The brute force approach is often a good place to start. From there, we can identify unnecessary work and further optimize. In this case, the brute force algorithm would be to generate every valid permutation of house colors (or all permutations and then remove all the invalid ones, e.g. ones that have 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]]



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)*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)*O*(2*n*) or O(3^n)*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)*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)*O*(*n*3) such permutations in total.

* Space complexity : Anywhere from O(n)*O*(*n*) to O(n \cdot 3^n)*O*(*n*⋅3*n*).

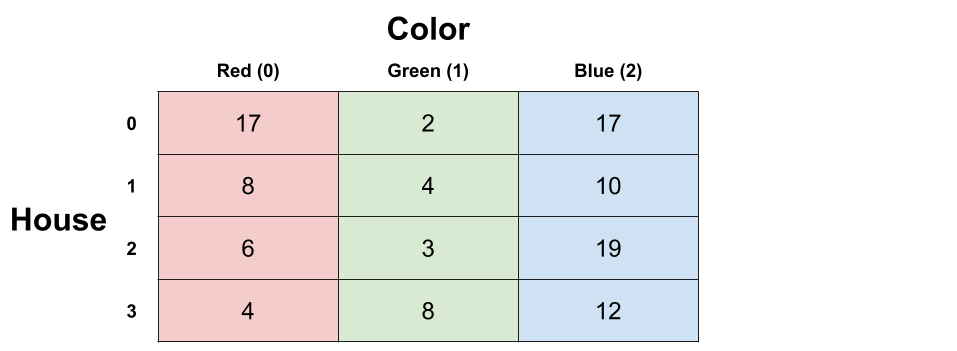
This would depend entirely on the implementation. If you generated all the permutations at the same time and put them in a massive list, then you'd be using O(n \* 2^n)*O*(*n*∗2*n*) or O(n \* 3^n)*O*(*n*∗3*n*) space. If you generated one, processed it, generated the next, processed it, etc, without keeping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

#### **Approach 2: Brute force with a Recursive Tree**

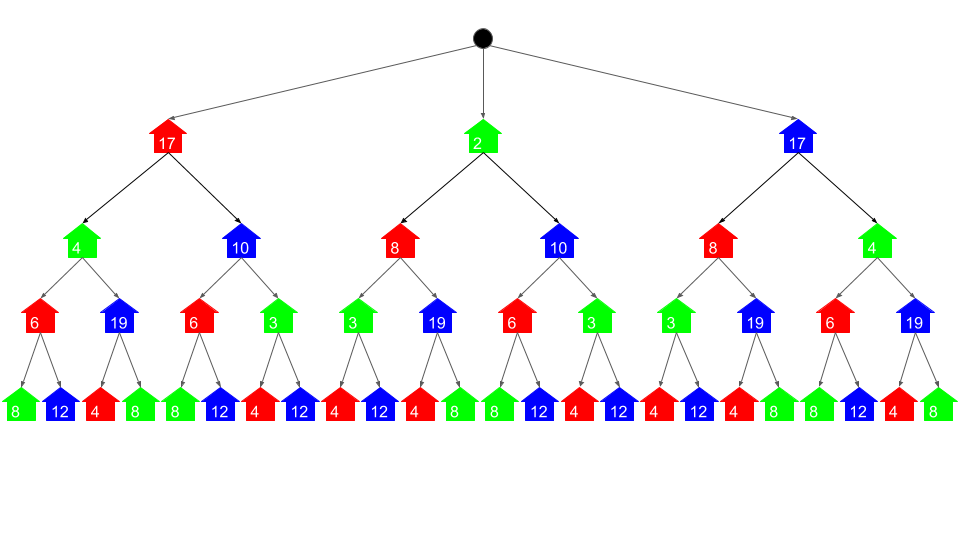
**Intuition**

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:



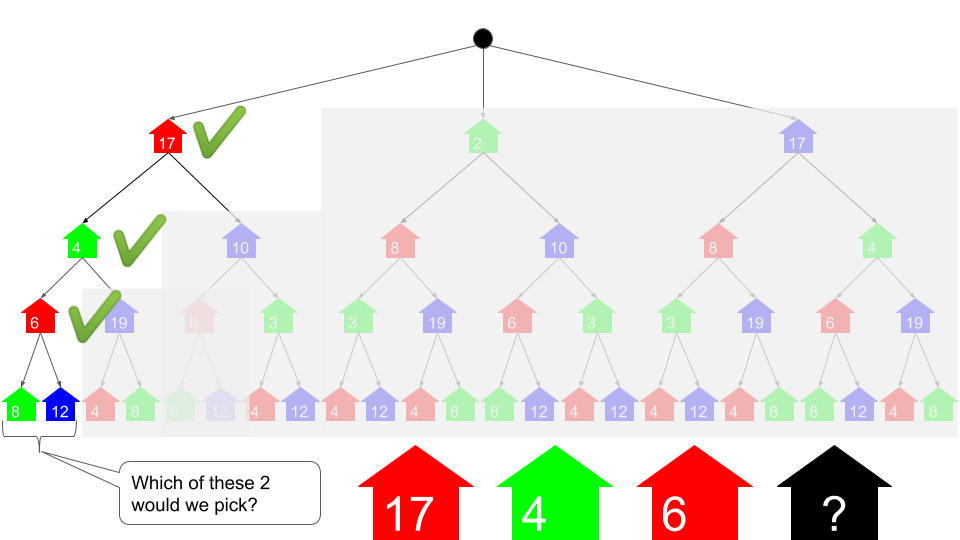
And here is how we can represent it using a tree. Each path from root to leaf represents a different possible permutation of house colors. There are 24 leaf nodes on the tree, just like there was 24 permutations identified in the brute force approach.



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.

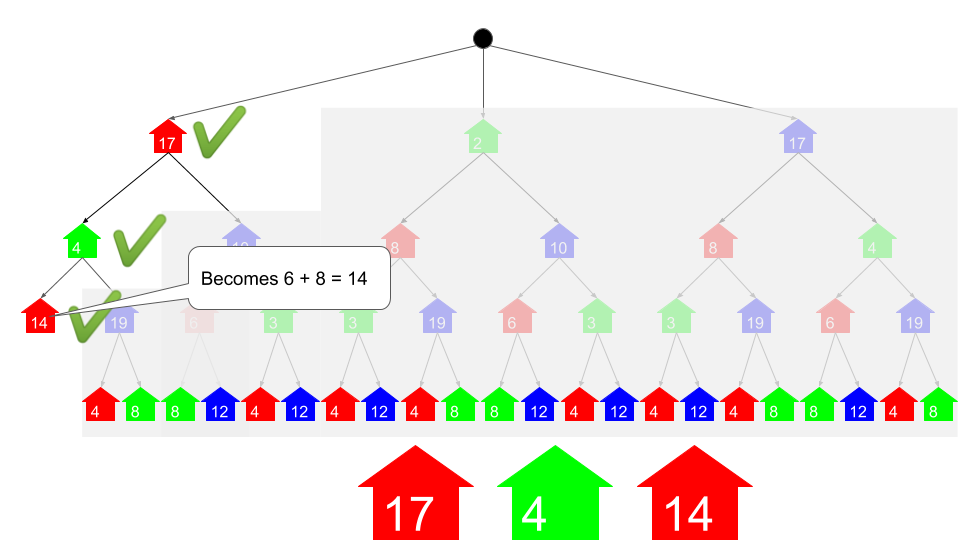
Without worrying yet about how we would actually implement it, we'll now explore a straightforward algorithm that can be used to solve the problem using this tree.

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?

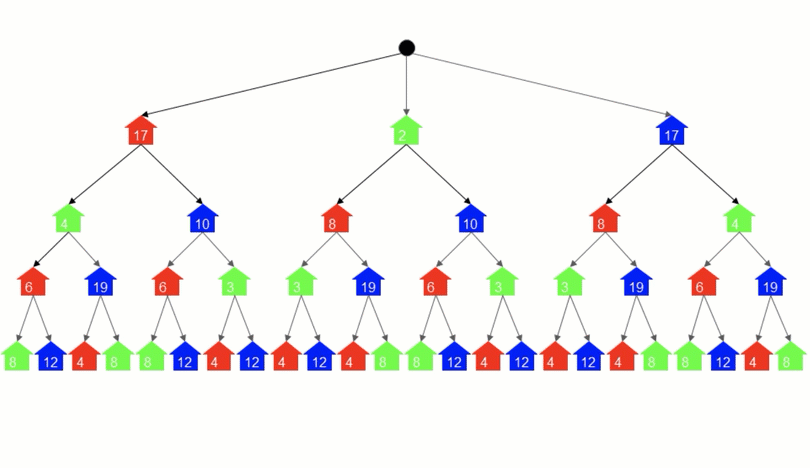


To minimize cost, we'd choose green. This is because green is 8, and blue is 12. Under the assumption that we'd already decided that the first 3 houses would be red, green, and red, this decision is definitely optimal. We know that there's no way we could do better.

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 can repeatedly remove leaf nodes following this same process, as shown in this animation.



We are left with the conclusion that:

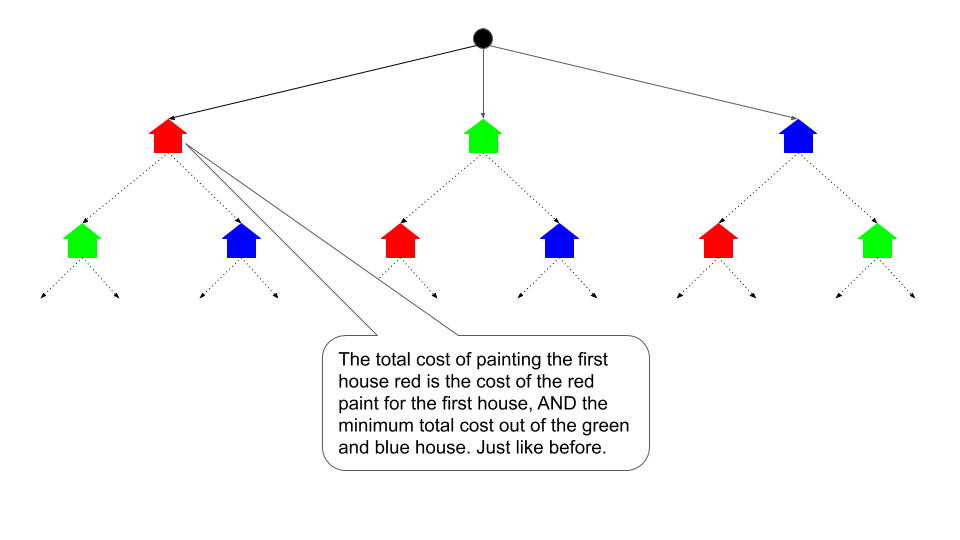
* Painting the first house red would have a total cost of 34.
* Painting the first house green would have a total cost of 17.
* Painting the first house blue would have a total cost of 32.

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?



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):

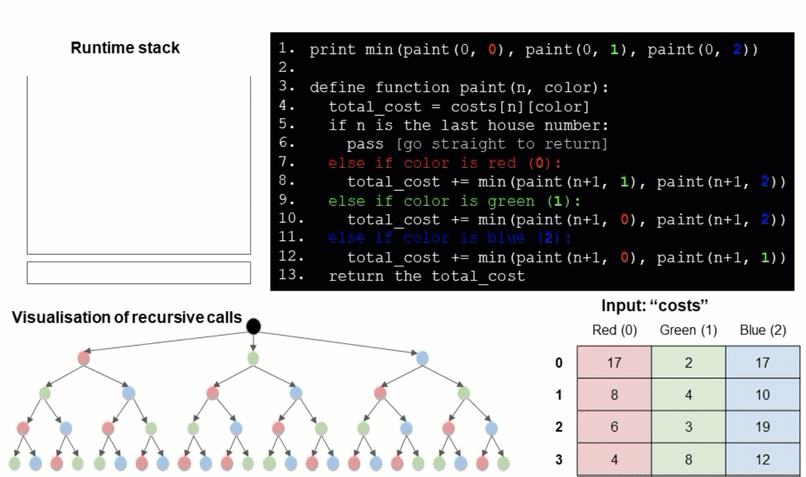
total\_cost += min(paint(n+1, 0), paint(n+1, 2))

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^n2*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*⋅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*).

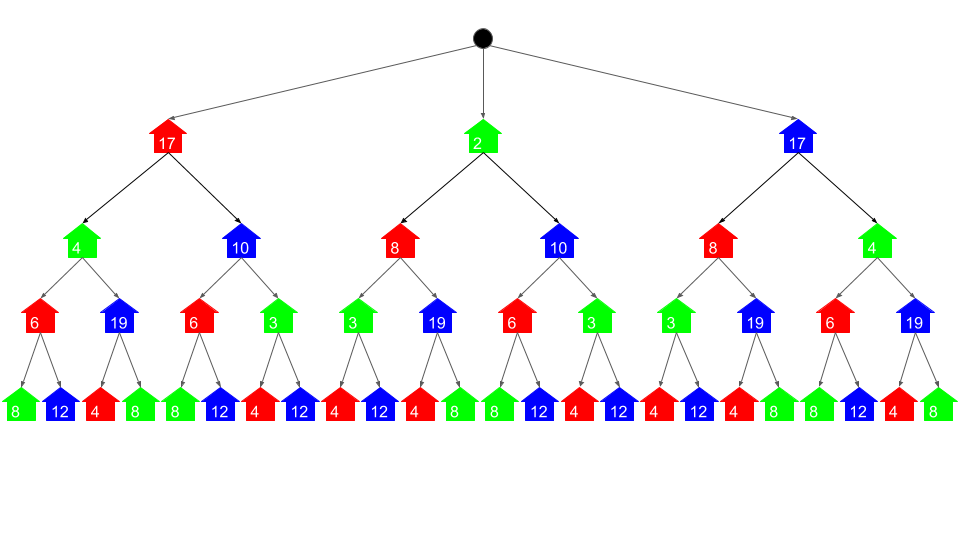
This algorithm might initially appear to be O(1)*O*(1), because we are not allocating any new data structures. However, we need to take into account space usage on the **run-time stack**. The run-time stack was shown in the animation. Whenever we are processing the last house (house number n - 1), there 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**

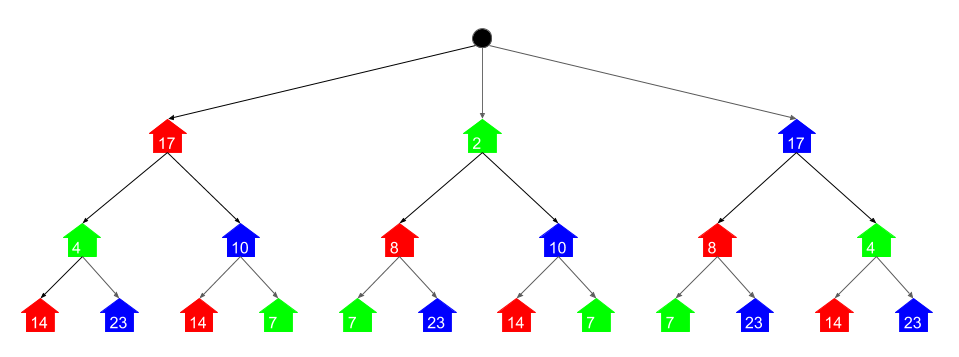
You may have noticed a very important pattern while we were working on the previous approach. Let's take a closer look.

This is the tree before we removed any layers.



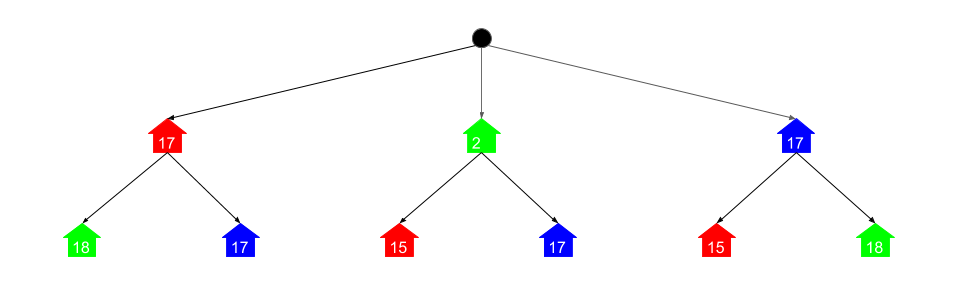
Look at the leaf nodes. All the red houses cost 4, the green houses 7, and the blue houses 23. This makes sense, as the original input told us the costs of painting the 4th house red, green, or blue were 4, 7, and 23 respectively.

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))

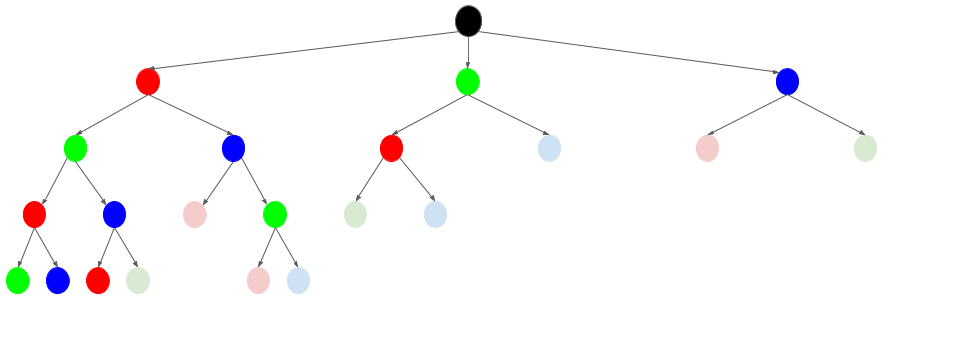
else if color is blue (2):

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)*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)*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)*O*(*n*).

* Space complexity : O(n)*O*(*n*).

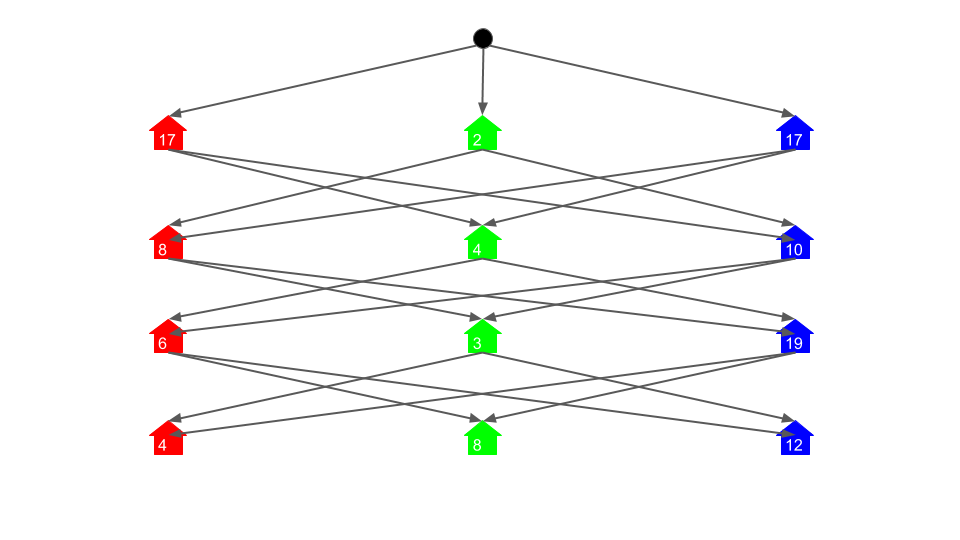
Like the previous approach, the main space usage is on the stack. When we go down the first branch of function calls (see the tree visualization), we won't find any results in the dictionary. Therefore, every house will make a stack frame. Because there are n houses, this gives a worst case space usage of O(n)*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.



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.

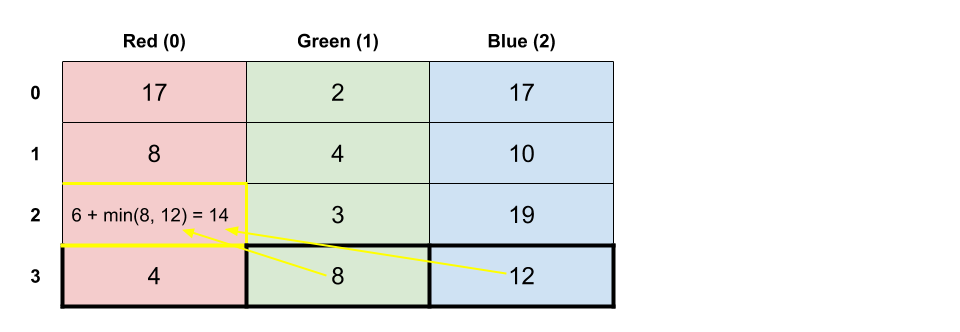


We can, therefore, calculate the cost of each subproblem, starting from the ones with the highest house numbers, and write the results directly into the input array. In effect, we will replace each single-house cost value in the array with the cost of painting the house that color and the minimum cost 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.

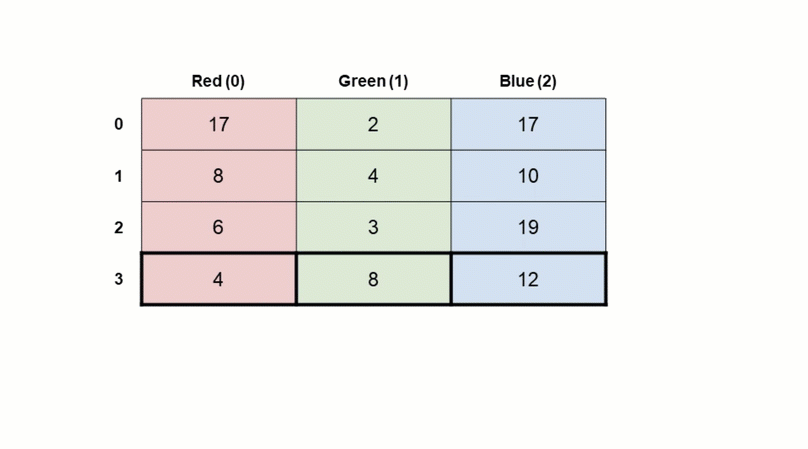
First thing to realize is that we don't need to do anything to the last row. Like in the tree, these costs are the total costs because there are no further houses after them.



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.



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. Here is an animation showing the process.



**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 shown in the animation.

|  |
| --- |
| 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⋅(*n*−1) cells in the grid. Expanding that out, we get 3 \cdot n - 33⋅*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*m* colors. For this particular problem we were told there's only 33 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)*O*(*n*⋅*m*), because m*m* is not a constant, whereas 33 is. If this confused you, I'd recommend reading up on big-oh notation.

* Space Complexity : O(1)*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)*O*(*n*) (for the same reason the time complexity is O(n)*O*(*n*), the constants are dropped in big-oh notation).

Using O(n)*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*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](https://leetcode.com/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 66 calculations at a time (using 2 x length-3 arrays). Because this is actually a constant, the space complexity is still O(1)*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*m* colors, then the space complexity would become O(m)*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.

**Paint House II**

There are a row of *n* houses, each house can be painted with one of the *k* colors. 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 a *n* x *k* cost matrix. For example, costs[0][0] is the cost of painting house 0 with color 0; costs[1][2] is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

**Note:**  
All costs are positive integers.

**Example:**

**Input:** [[1,5,3],[2,9,4]]

**Output:** 5

**Explanation:** Paint house 0 into color 0, paint house 1 into color 2. Minimum cost: 1 + 4 = 5;

  Or paint house 0 into color 2, paint house 1 into color 0. Minimum cost: 3 + 2 = 5.

**Follow up:**  
Could you solve it in *O*(*nk*) runtime?

## Solution

**Paint House II** is a follow up question of [Paint House](https://leetcode.com/problems/paint-house/). In the original Paint House problem, k was always 3. In this problem, k is no longer fixed and instead can be any non-negative integer.

If you haven't yet attempted the original [Paint House](https://leetcode.com/problems/paint-house/) question and are having trouble with this question, go attempt Paint House first and come back. There is also an in-depth [Paint House Solution Article](https://leetcode.com/articles/paint-house/). This solution article will assume you are already comfortable with the memoization and dynamic programming solutions for Paint House.

#### **Approach 1: Memoization**

**Intuition**

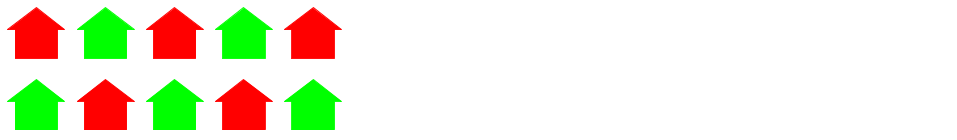
Remembering that we already know how to solve this problem using memoization when k = 3 (check the [Paint House Solution Article](https://leetcode.com/articles/paint-house/)) if you can't remember how), let's think through some of the other possible values of k.

For this explanation, we'll call a way of painting the houses valid if, and only if, there are no adjacent houses painted the same color. We'll call an input valid if it is possible to paint the houses in a valid way. **The test cases here on Leetcode are all valid inputs**. In an interview however, you'd need to ensure that it is safe to assume that the input is always valid though.

If k = 0, then this means we have no colors. If there are no colors, it's probably reasonable to assume there are no houses either, i.e. n = 0. In other words, the input is []. **For this question here on Leetcode, this is a safe assumption**. In an interview though it could be a good idea to ask the interviewer whether or not the input is guaranteed to be valid. For example, could you get a test case such as [[],[],[],[]]? This would be k = 0 and n = 4. Of course, this case doesn't make much sense, because we are supposed to be painting houses, but can't with no paint. Either you'd be told it could never happen, or that you needed to do something special for it, such as returning -1.

If k = 1 (all houses have to be the same color), then it's probably safe to assume that n = 1. Otherwise, the problem would be impossible to solve without breaking the adjacent color rule. Again, this is a safe assumption here, but do consider asking the interviewer whether or not you could get an invalid input that had k = 1 and n > 1. So, assuming that k = 1 and n = 1, the total cost will be the cost of painting that one house the only color available.

If k = 2 (there are two colors), then we know the problem is always solvable, because we can simply paint the houses alternating colors. For example, when n = 5 and k = 2, here are the only 2 valid ways of painting the houses. Anything else would be invalid.

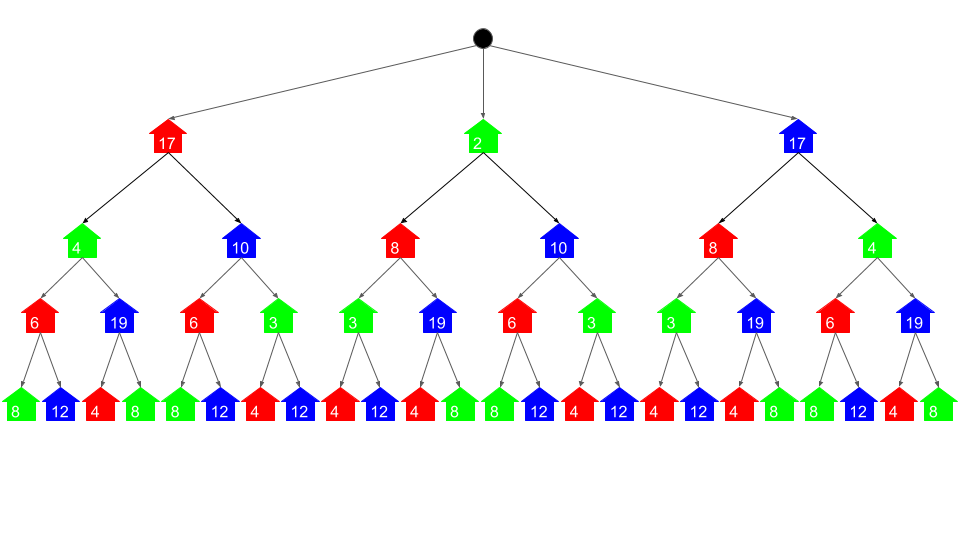


The answer will be the one that leads to the lowest cost. It'd be easy to check both.

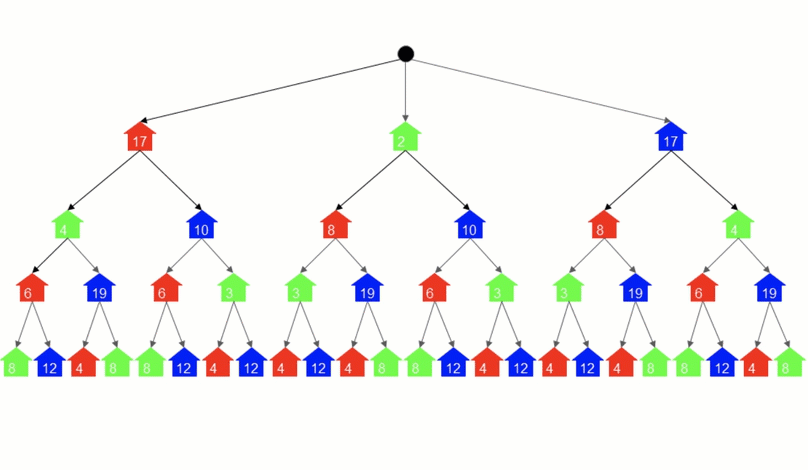
When k = 3, the problem is equivalent to [Paint House](https://leetcode.com/problems/paint-house/). In the Solution Article for that question, we worked through an example where n = 4.

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

A good way to visualize all of the valid painting permutations is to use a tree. Each root-to-leaf path represents one valid way of painting the houses.



The cheapest cost of painting the houses is, therefore, the root-to-leaf path with the lowest total sum of its nodes. This animation shows the algorithm we used to solve this problem for Paint House.



Luckily, we didn't actually need to create the tree itself—there is a simpler way using recursion.

Say we have a paint function that takes 2 parameters: a house number and a color to paint that house. The output is the ***total*** **cost of painting that house and all the ones after it**. For example paint(1, red) would be the cost of painting house 1 red, along with the cost of painting the houses after it (taking into account restrictions caused by painting house 1 red).

Therefore the cheapest way of painting all the houses can be expressed as follows, where 0 is the first house.

min(paint(0, "red"), paint(0, "green"), paint(0, "blue"))

The paint function has a recursive implementation. costs refers to the *input* table.

def paint(i, color):

### BASE CASE ###

if i is the last house number:

return costs[i][color]

### RECURSIVE CASE ###

lowest\_cost = Infinity

for each next\_color in ["red", "green", "blue"]:

if next\_color != color: # No adjacent houses can be same color.

this\_cost = costs[i][color] + paint(i + 1, next\_color) # <- Recursive call

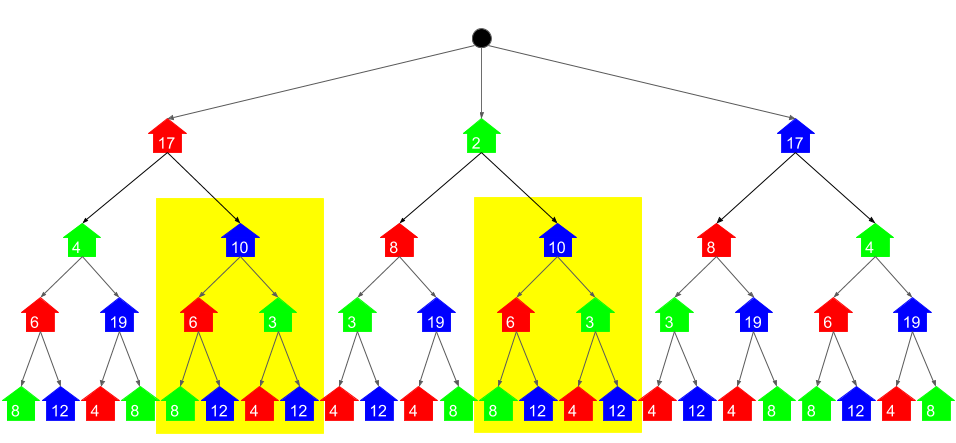
lowest\_cost = min(lowest\_cost, this\_cost)

return lowest\_cost

The **base case** is where i refers to the *last house*. Painting the last house a particular color can be obtained from the costs table.

The **recursive case** is where we also need to consider the houses after i. It is obtained by looking up the cost of painting house i the given color (in the costs table) and then by determining the cost of painting the houses after it. The cost of painting the houses after requires making recursive paint calls to determine the cost of painting house i + 1 each of the 2 other colors and then finding the minimum of those 2 values.

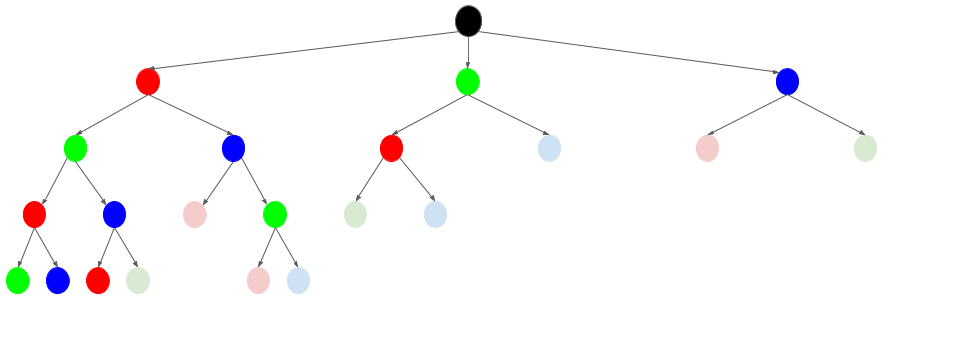
This algorithm is inefficient though. There is a lot of repetition in the tree, meaning we're doing the same calculations over and over again. For example, the total cost of painting the *second* house blue will be the same regardless of whether the first house was red or green. For example, both these branches of the tree are identical.



This should also be apparent from the definition of the paint function. The parameters are simply a house number and a color. It doesn't require any information about *where* exactly in the tree it is.

To solve this problem, we add **memoization** to the paint function. Recall that memoization is where before returning an answer, the recursive function writes the answer into a dictionary with the input parameters as the key and the answer as the value. Then before doing a calculation, it checks whether or not that particular calculation has already been done. For the example above, the function would only calculate the cost of painting the second house blue *once*, and then the second time it would look it up in the dictionary.

Here is a visualization that shows the calculations that need to be done when using memoization. The brighter circles show where the function body runs, and the duller circles show where a lookup was done and the function immediately returned. These are the only times the function is called.



**If this explanation wasn't thorough enough for you**, then please check out the full [Paint House Solution Article](https://leetcode.com/articles/paint-house/). In that article I go into a lot more depth about memoization and how this algorithm was derived.

The k > 3 case is really no different to this. The only difference is that instead of only considering 2 possible colors for the *next* house, we're considering k - 1 colors (all colors except for the color of the current house).

**Algorithm**

Unlike the pseudocode above, we don't need to worry about the names of the colors. Instead, they are represented by numbers between 0 and k - 1. Additionally, we're also using memoization (lru\_cache in Python, and a dictionary in Java).

|  |
| --- |
| class Solution {  private int n;  private int k;  private int[][] costs;  private Map<String, Integer> memo;  public int minCostII(int[][] costs) {  if (costs.length == 0) return 0;  this.k = costs[0].length;  this.n = costs.length;  this.costs = costs;  this.memo = new HashMap<>();  int minCost = Integer.MAX\_VALUE;  for (int color = 0; color < k; color++) {  minCost = Math.min(minCost, memoSolve(0, color));  }  return minCost;  }  private int memoSolve(int houseNumber, int color) {  // Base case: There are no more houses after this one.  if (houseNumber == n - 1) {  return costs[houseNumber][color];  }  // Memoization lookup case: Have we already solved this subproblem?  if (memo.containsKey(getKey(houseNumber, color))) {  return memo.get(getKey(houseNumber, color));  }  // Recursive case: Determine the minimum cost for the remainder.  int minRemainingCost = Integer.MAX\_VALUE;  for (int nextColor = 0; nextColor < k; nextColor++) {  if (color == nextColor) continue;  int currentRemainingCost = memoSolve(houseNumber + 1, nextColor);  minRemainingCost = Math.min(currentRemainingCost, minRemainingCost);  }  int totalCost = costs[houseNumber][color] + minRemainingCost;  memo.put(getKey(houseNumber, color), totalCost);  return totalCost;  }  // Convert a house number and color into a simple string key for the memo.  private String getKey(int n, int color) {  return String.valueOf(n) + " " + String.valueOf(color);  }  } |

**Complexity Analysis**

* Time complexity : O(n \cdot k ^ 2)*O*(*n*⋅*k*2).

Determining the total time complexity of a recursive memoization algorithm requires looking at how many calls are made to the paint function, and how much each call costs (remember that the memoization lookups are O(1)*O*(1)). The function is called once for each possible pair of house number and color. This gives n \cdot k*n*⋅*k* calls. Then, each call has a loop that loops over each of the k*k* colors. Therefore, we have n \cdot k \cdot k = n \cdot k ^2*n*⋅*k*⋅*k*=*n*⋅*k*2 which is O(n \cdot k ^ 2)*O*(*n*⋅*k*2).

The part outside of the recursive function is O(k)*O*(*k*) and therefore does not impact the overall complexity.

* Space complexity : O(n \cdot k)*O*(*n*⋅*k*).

There are 2 different places memory is being used that we need to consider.

Firstly, the memoization is storing the answers for each pair of house number and color. There are n \cdot k*n*⋅*k* of these, and so O(n \cdot k)*O*(*n*⋅*k*) memory used.

Secondly, we need to consider the memory used on the run-time stack. In the worst case, there's a stack frame for each house number on the stack. This is a total of O(n)*O*(*n*).

The O(n)*O*(*n*) is insignficant to the O(n \cdot k)*O*(*n*⋅*k*), so we're left with a total of O(n \cdot k)*O*(*n*⋅*k*).

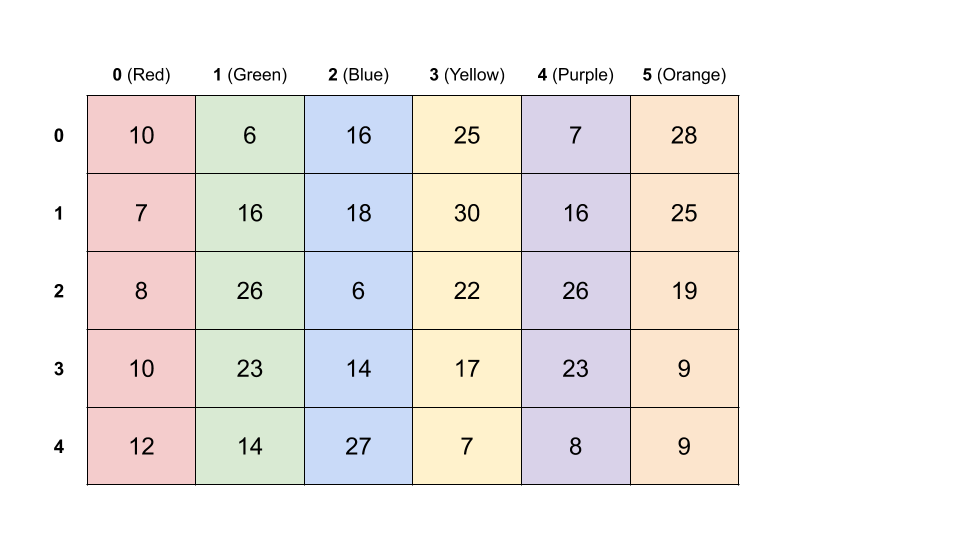
#### **Approach 2: Dynamic Programming**

**Intuition**

Let's look at a bigger example now, and view the problem in a different way to how we did before. For this example, k = 6 and n = 5.

[[10, 6, 16, 25, 7, 28], [7, 16, 18, 30, 16, 25], [8, 26, 6, 22, 26, 19], [10, 23, 14, 17, 23, 9], [12, 14, 27, 7, 8, 9]]

And here is a diagram of the input grid.



Each row represents the different colors a house could be. Remember that the colors are represented by numbers. The actual colors are only to make the table easier to read.

The problem we're trying to solve is equivalent to the following: **pick exactly one number from each row** such that the **sum of those numbers is minimized**. Because 2 adjacent houses cannot be the same color, **adjacent rows must be picked from different columns**. This is a straightforward variant of one of those "classic" minimum-path-in-a-grid dynamic programming problems.

The way that we solve it is to iterate over the cells and determine what the cheapest way of getting to that cell is. We'll work from top to bottom.

To begin with, we say the first row (house 0) is already completed. We don't need to make any changes to it.

Then, for each cell in the second row, we work out the cheapest way of getting to it from the first row is. For example, to get to [1][red] we have to go through any of the non-red cells from the row above. We want to go through the minimum.



We show our decision by updating [1][red] to 7 + 6 = 13.

We can repeat this for the rest of the second row, and then work down each of the remaining rows.

Here's an animation of the algorithm being carried out.

When we're finished, the final answer is the **minimum value in the last row**.

**Algorithm**

We'll do this in the same way we did in the animation above—an in-place algorithm that over-writes the input grid.

|  |
| --- |
| class Solution {  public int minCostII(int[][] costs) {  if (costs.length == 0) return 0;  int k = costs[0].length;  int n = costs.length;  for (int house = 1; house < n; house++) {  for (int color = 0; color < k; color++) {  int min = Integer.MAX\_VALUE;  for (int previousColor = 0; previousColor < k; previousColor++) {  if (color == previousColor) continue;  min = Math.min(min, costs[house - 1][previousColor]);  }  costs[houseNumber][color] += min;  }  }  // Find the minimum in the last row.  int min = Integer.MAX\_VALUE;  for (int c : costs[n - 1]) {  min = Math.min(min, c);  }  return min;  }  } |

**Complexity Analysis**

* Time complexity : O(n \cdot k ^ 2)*O*(*n*⋅*k*2).

We iterate over each of the n \cdot k*n*⋅*k* cells. For each of the cells, we're finding the minimum of the k*k* values in the row above, excluding the one that is in the same column. This operation is O(k)*O*(*k*). Multiplying this out, we get O(n \cdot k ^ 2)*O*(*n*⋅*k*2).

* Space complexity : O(1)*O*(1) if done in-place, O(n \cdot k)*O*(*n*⋅*k*) if input is copied.

We're not creating any new data structures in the code above, and so it has a space complexity of O(1)*O*(1). This is, however, overwriting the given input, which might not be ideal in some situations.

If we don't want to overwrite the input, we could instead create a copy of it first and then do the calculations in the copy. This will require an additional O(n \cdot k)*O*(*n*⋅*k*) space.

#### **Approach 3: Dynamic Programming with O(k) additional Space.**

**Intuition**

Implementing the algorithm in-place meant that we only needed O(1)*O*(1) additional space. This, however required modifying the input, which could be a problem in some situations.

The easiest solution is to make a copy of the input array and then do the calculations in that instead. This would require O(n \cdot k)*O*(*n*⋅*k*) additional space.

There is a way that uses less space though. We're only ever working with 2 rows at a time: the current row, and the row before it. The rows before that are never looked at again, and the rows after are still the same as the input array. Therefore, we can take advantage of this to only use O(k)*O*(*k*) space.

**Algorithm**

Instead of writing the updated costs into the input array, the algorithm writes them into a k-length array. The k-length array from the previous row is held onto in-order to do these calculations.

|  |
| --- |
| class Solution {  public int minCostII(int[][] costs) {  if (costs.length == 0) return 0;  int k = costs[0].length;  int n = costs.length;  int[] previousRow = costs[0];  for (int house = 1; house < n; house++) {  int[] currentRow = new int[k];  for (int color = 0; color < k; color++) {  int min = Integer.MAX\_VALUE;  for (int previousColor = 0; previousColor < k; previousColor++) {  if (color == previousColor) continue;  min = Math.min(min, previousRow[previousColor]);  }  currentRow[color] += costs[house][color] += min;  }  previousRow = currentRow;  }  // Find the minimum in the last row.  int min = Integer.MAX\_VALUE;  for (int c : previousRow) {  min = Math.min(min, c);  }  return min;  }  } |

**Complexity Analysis**

* Time complexity : O(n \cdot k ^ 2)*O*(*n*⋅*k*2).

Same as above.

* Space complexity : O(k)*O*(*k*).

The previous row and the current row are represented as k-length arrays.

This approach does not modify the input grid.

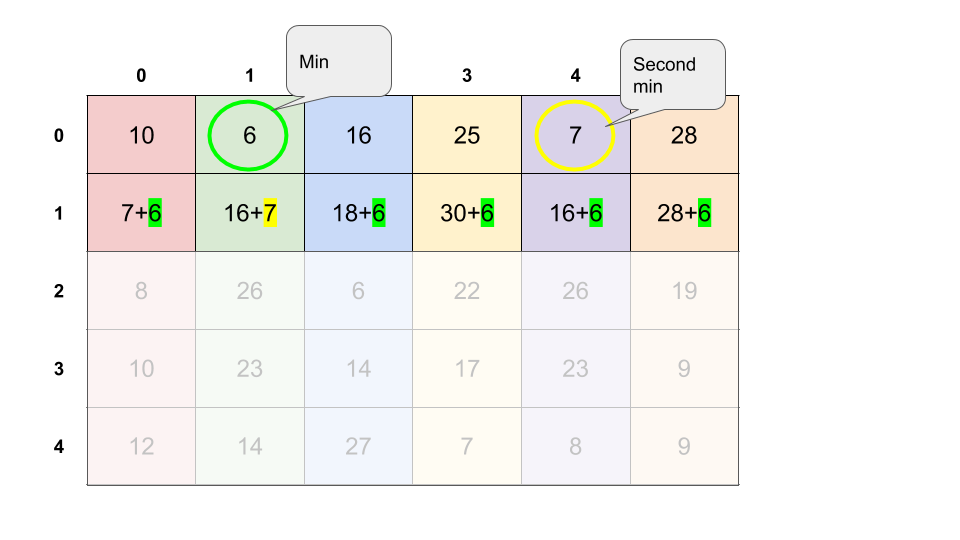
#### **Approach 4: Dynamic programming with Optimized Time**

**Intuition**

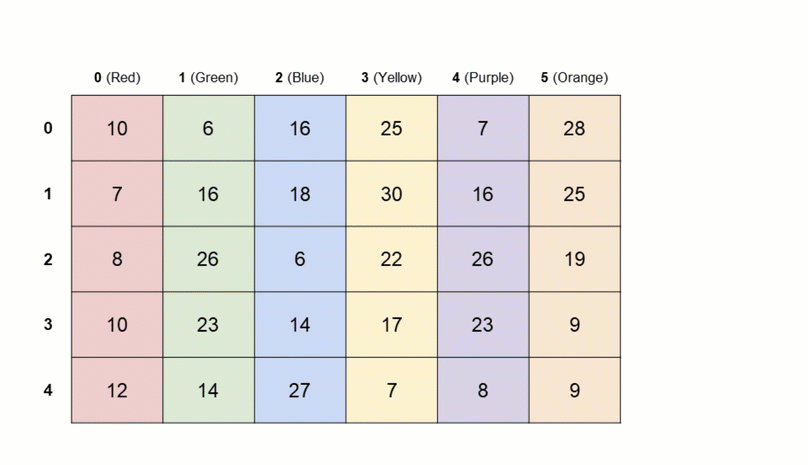
Despite Paint House II being listed as a hard question, and the problem statement listing O(n \cdot k)*O*(*n*⋅*k*) time as a "follow up", you'd possibly be expected to come up with this solution at top companies as it's still a fairly basic dynamic programming algorithm. You should, therefore, ensure you're comfortable with this approach and could identify and apply similar observations in other dynamic programming problems. At the very least, it'll make you look awesome!

So far, all of our approaches have had a O(n \cdot k^2)*O*(*n*⋅*k*2) time complexity. This is because calculating the new value for each of the O(n \cdot k)*O*(*n*⋅*k*) cells required looking at each of the k*k* cells in the row immediately below.

However, we don't need to look at the entire previous row for every cell. Let's look again at the large example from above. When we're calculating the values for the second row, we're adding the minimum from the first row onto them. The only cell we can't do this for is the one that was directly below the minimum, as this would break the adjacency rule. For this one, it makes sense to add the second minimum.



Here's an animation of the entire algorithm.



**Algorithm**

The simplest way of implementing this algorithm is to base it on the animation above. This requires overwriting the input.

|  |
| --- |
| class Solution {  public int minCostII(int[][] costs) {  if (costs.length == 0) return 0;  int k = costs[0].length;  int n = costs.length;  for (int house = 1; house < n; house++) {  // Find the minimum and second minimum color in the PREVIOUS row.  int minColor = -1; int secondMinColor = -1;  for (int color = 0; color < k; color++) {  int cost = costs[house - 1][color];  if (minColor == -1 || cost < costs[house - 1][minColor]) {  secondMinColor = minColor;  minColor = color;  } else if (secondMinColor == -1 || cost < costs[house - 1][secondMinColor]) {  secondMinColor = color;  }  }  // And now calculate the new costs for the current row.  for (int color = 0; color < k; color++) {  if (color == minColor) {  costs[house][color] += costs[house - 1][secondMinColor];  } else {  costs[house][color] += costs[house - 1][minColor];  }  }  }  // Find the minimum in the last row.  int min = Integer.MAX\_VALUE;  for (int c : costs[n - 1]) {  min = Math.min(min, c);  }  return min;  }  } |

**Complexity Analysis**

* Time complexity : O(n \cdot k)*O*(*n*⋅*k*).

The first loop that finds the minimums of the first row is O(k)*O*(*k*) because it looks at each of the k*k* values in the first row exactly once. The second loop is O(n \cdot k)*O*(*n*⋅*k*) because the outer loop loops n*n* times, and the inner loop loops k*k* times. O(n \cdot k) + O(k) = O(n \cdot k)*O*(*n*⋅*k*)+*O*(*k*)=*O*(*n*⋅*k*). We know it is impossible to ever do better here, because we cannot solve the problem without at least looking at each of the n \cdot k*n*⋅*k* cells once.

* Space complexity : O(1)*O*(1).

Like approach 2, this approach also modifies the input instead of allocating its own space.

#### **Approach 5: Dynamic programming with Optimized Time and Space**

**Intuition**

There is another way we can still solve the problem in O(1)*O*(1) space and O(n \cdot k)*O*(*n*⋅*k*) time complexity, and preserving the input.

The only thing the algorithm in the previous approach is really doing is going through the rows, and finding the 2 minimums of each row. It does this by calculating all the new costs for the row, writing them into the input, and then finding the minimums. This overwriting isn't necessary though—we can simply keep track of the 2 smallest values we've seen so far, as we go, in the current row. We also need to remember the 2 from the previous row.

**Algorithm**

The approach is a hybrid of approach 3 and 4. Like approach 4, it finds the minimums once instead of repeatedly. Like approach 3, it keeps track of information only from the current and previous rows. Unlike approach 3 though, the only information kept is the minimums.

|  |
| --- |
| class Solution {  public int minCostII(int[][] costs) {  if (costs.length == 0) return 0;  int k = costs[0].length;  int n = costs.length;  /\* Firstly, we need to determine the 2 lowest costs of  \* the first row. We also need to remember the color of  \* the lowest. \*/  int prevMin = -1; int prevSecondMin = -1; int prevMinColor = -1;  for (int color = 0; color < k; color++) {  int cost = costs[0][color];  if (prevMin == -1 || cost < prevMin) {  prevSecondMin = prevMin;  prevMinColor = color;  prevMin = cost;  } else if (prevSecondMin == -1 || cost < prevSecondMin) {  prevSecondMin = cost;  }  }  // And now, we need to work our way down, keeping track of the minimums.  for (int house = 1; house < n; house++) {  int min = -1; int secondMin = -1; int minColor = -1;  for (int color = 0; color < k; color++) {  // Determine the cost for this cell (without writing it in).  int cost = costs[house][color];  if (color == prevMinColor) {  cost += prevSecondMin;  } else {  cost += prevMin;  }  // Determine whether or not this current cost is also a minimum.  if (min == -1 || cost < min) {  secondMin = min;  minColor = color;  min = cost;  } else if (secondMin == -1 || cost < secondMin) {  secondMin = cost;  }  }  // Transfer current mins to be previous mins.  prevMin = min;  prevSecondMin = secondMin;  prevMinColor = minColor;  }  return prevMin;  }  } |

There are many ways to compact the code a bit more, particularly in the case of the Python. I haven't done this here as it could be problematic for those less familiar with the 2 languages I have provided solutions in, however feel free to post your own solutions in the comments. I'm excited to see the elegance you can come up with!

**Complexity Analysis**

* Time complexity : O(n \cdot k)*O*(*n*⋅*k*).

Same as the previous approach.

* Space complexity : O(1)*O*(1).

The only additional working memory we're using is a constant number of single-value variables to keep track of the 2 minimums in the current and previous row, and to calculate the cost of the current cell. Because the memory usage is constant, we say it is O(1)*O*(1). Unlike the previous approach one though, this one does not overwrite the input.

**Partition to K Equal Sum Subsets**

Given an array of integers nums and a positive integer k, find whether it's possible to divide this array into k non-empty subsets whose sums are all equal.

**Example 1:**

**Input:** nums = [4, 3, 2, 3, 5, 2, 1], k = 4

**Output:** True

**Explanation:** It's possible to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with equal sums.

**Note:**

* 1 <= k <= len(nums) <= 16.
* 0 < nums[i] < 10000.

   Hide Hint #1

We can figure out what target each subset must sum to. Then, let's recursively search, where at each call to our function, we choose which of k subsets the next value will join.

#### **Approach #1: Search by Constructing Subset Sums [Accepted]**

**Intuition**

As even when k = 2, the problem is a "Subset Sum" problem which is known to be NP-hard, (and because the given input limits are low,) our solution will focus on exhaustive search.

A natural approach is to simulate the k groups (disjoint subsets of nums). For each number in nums, we'll check whether putting it in the i-th group solves the problem. We can check those possibilities by recursively searching.

**Algorithm**

Firstly, we know that each of the k group-sums must be equal to target = sum(nums) / k. (If this quantity is not an integer, the task is impossible.)

For each number in nums, we could add it into one of k group-sums, as long as the group's sum would not exceed the target. For each of these choices, we recursively search with one less number to consider in nums. If we placed every number successfully, then our search was successful.

One important speedup is that we can ensure all the 0 values of each group occur at the end of the array groups, by enforcing if (groups[i] == 0) break;. This greatly reduces repeated work - for example, in the first run of search, we will make only 1 recursive call, instead of k. Actually, we could do better by skipping any repeated values of groups[i], but it isn't necessary.

Another speedup is we could sort the array nums, so that we try to place the largest elements first. When the answer is true and involves subsets with a low size, this method of placing elements will consider these lower size subsets sooner. We can also handle elements nums[i] >= target appropriately. These tricks are not necessary to solve the problem, but they are presented in the solutions below.

|  |
| --- |
| class Solution {  public boolean search(int[] groups, int row, int[] nums, int target) {  if (row < 0) return true;  int v = nums[row--];  for (int i = 0; i < groups.length; i++) {  if (groups[i] + v <= target) {  groups[i] += v;  if (search(groups, row, nums, target)) return true;  groups[i] -= v;  }  if (groups[i] == 0) break;  }  return false;  }  public boolean canPartitionKSubsets(int[] nums, int k) {  int sum = Arrays.stream(nums).sum();  if (sum % k > 0) return false;  int target = sum / k;  Arrays.sort(nums);  int row = nums.length - 1;  if (nums[row] > target) return false;  while (row >= 0 && nums[row] == target) {  row--;  k--;  }  return search(new int[k], row, nums, target);  }  } |

**Complexity Analysis**

* Time Complexity: O(k^{N-k} k!)*O*(*kN*−*kk*!), where N*N* is the length of nums, and k*k* is as given. As we skip additional zeroes in groups, naively we will make O(k!)*O*(*k*!) calls to search, then an additional O(k^{N-k})*O*(*kN*−*k*) calls after every element of groups is nonzero.
* Space Complexity: O(N)*O*(*N*), the space used by recursive calls to search in our call stack.

#### **Approach #2: Dynamic Programming on Subsets of Input [Accepted]**

**Intuition and Algorithm**

As in Approach #1, we investigate methods of exhaustive search, and find target = sum(nums) / k in the same way.

Let used have the i-th bit set if and only if nums[i] has already been used. Our goal is to use nums in some order so that placing them into groups in that order will be valid. search(used, ...) will answer the question: can we partition unused elements of nums[i] appropriately?

This will depend on todo, the sum of the remaining unused elements, not crossing multiples of target within one number. If for example our target is 10, and our elements to be placed in order are [6, 5, 5, 4], this would not work as 6 + 5 "crosses" 10 prematurely.

If we could choose the order, then after placing 5, our unused elements are [4, 5, 6]. Using 6 would make todo go from 15 to 9, which crosses 10 - something unwanted. However, we could use 5 since todo goes from 15 to 10; then later we could use 4 and 6 as those placements do not cross.

It turns out the maximum value that can be chosen so as to not cross a multiple of target, is targ = (todo - 1) % target + 1. This is essentially todo % target, plus target if that would be zero.

Now for each unused number that doesn't cross, we'll search on that state, and we'll return true if any of those searches are true.

Notice that the state todo depends only on the state used, so when memoizing our search, we only need to make lookups by used.

In the solutions below, we present both a top-down dynamic programming solution, and a bottom-up one. The bottom-up solution uses a different notion of state.

|  |
| --- |
| enum Result { TRUE, FALSE }  class Solution {  boolean search(int used, int todo, Result[] memo, int[] nums, int target) {  if (memo[used] == null) {  memo[used] = Result.FALSE;  int targ = (todo - 1) % target + 1;  for (int i = 0; i < nums.length; i++) {  if ((((used >> i) & 1) == 0) && nums[i] <= targ) {  if (search(used | (1<<i), todo - nums[i], memo, nums, target)) {  memo[used] = Result.TRUE;  break;  }  }  }  }  return memo[used] == Result.TRUE;  }  public boolean canPartitionKSubsets(int[] nums, int k) {  int sum = Arrays.stream(nums).sum();  if (sum % k > 0) return false;  Result[] memo = new Result[1 << nums.length];  memo[(1 << nums.length) - 1] = Result.TRUE;  return search(0, sum, memo, nums, sum / k);  }  } |

|  |
| --- |
| class Solution {  public boolean canPartitionKSubsets(int[] nums, int k) {  int N = nums.length;  Arrays.sort(nums);  int sum = Arrays.stream(nums).sum();  int target = sum / k;  if (sum % k > 0 || nums[N - 1] > target) return false;  boolean[] dp = new boolean[1 << N];  dp[0] = true;  int[] total = new int[1 << N];  for (int state = 0; state < (1 << N); state++) {  if (!dp[state]) continue;  for (int i = 0; i < N; i++) {  int future = state | (1 << i);  if (state != future && !dp[future]) {  if (nums[i] <= target - (total[state] % target)) {  dp[future] = true;  total[future] = total[state] + nums[i];  } else {  break;  }  }  }  }  return dp[(1 << N) - 1];  }  } |

**Complexity Analysis**

* Time Complexity: O(N \* 2^N)*O*(*N*∗2*N*), where N*N* is the length of nums. There are 2^N2*N* states of used (or state in our bottom-up variant), and each state performs O(N) work searching through nums.
* Space Complexity: O(2^N)*O*(2*N*), the space used by memo (or dp, total in our bottom-up variant).

**All O`one Data Structure**

Implement a data structure supporting the following operations:

1. Inc(Key) - Inserts a new key with value 1. Or increments an existing key by 1. Key is guaranteed to be a **non-empty** string.
2. Dec(Key) - If Key's value is 1, remove it from the data structure. Otherwise decrements an existing key by 1. If the key does not exist, this function does nothing. Key is guaranteed to be a **non-empty** string.
3. GetMaxKey() - Returns one of the keys with maximal value. If no element exists, return an empty string "".
4. GetMinKey() - Returns one of the keys with minimal value. If no element exists, return an empty string "".

Challenge: Perform all these in O(1) time complexity.

|  |
| --- |
| class AllOne {  /\*\* Initialize your data structure here. \*/  public AllOne() {    }    /\*\* Inserts a new key <Key> with value 1. Or increments an existing key by 1. \*/  public void inc(String key) {    }    /\*\* Decrements an existing key by 1. If Key's value is 1, remove it from the data structure. \*/  public void dec(String key) {    }    /\*\* Returns one of the keys with maximal value. \*/  public String getMaxKey() {    }    /\*\* Returns one of the keys with Minimal value. \*/  public String getMinKey() {    }  }  /\*\*  \* Your AllOne object will be instantiated and called as such:  \* AllOne obj = new AllOne();  \* obj.inc(key);  \* obj.dec(key);  \* String param\_3 = obj.getMaxKey();  \* String param\_4 = obj.getMinKey();  \*/ |

**Max Stack**

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:**

* -107 <= x <= 107
* At most 104 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.

**Follow up:** Could you come up with a solution that supports O(1) for each top call and O(logn) for each other call?

#### **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.