**November 2020 Challenge**

Week 1: November 1st - November 7th

~~+35~~

Meeting Rooms

~~+10~~

**Convert Binary Number in a Linked List to Integer**

~~+10~~

**Insertion Sort List**

~~+10~~

**Consecutive Characters**

~~+10~~

**Minimum Height Trees**

~~+10~~

**Minimum Cost to Move Chips to The Same Position**

~~+10~~

**Find the Smallest Divisor Given a Threshold**

~~+10~~

Add Two Numbers II

Week 2: November 8th - November 14th

~~+35~~

**Two Sum Less Than K**

~~+10~~

**Binary Tree Tilt**

~~+10~~

**Maximum Difference Between Node and Ancestor**

~~+10~~

**Flipping an Image**

~~+10~~

 Valid Square

~~+10~~

Permutations II

~~+10~~

Populating Next Right Pointers in Each Node

~~+10~~

**Poor Pigs**

Week 3: November 15th - November 21st

~~+35~~

 **Remove Interval**

~~+10~~

**Range Sum of BST**

~~+10~~

**Longest Mountain in Array**

~~+10~~

**Mirror Reflection**

~~+10~~

 Merge Intervals

~~+10~~

Decode String

~~+10~~

**Search in Rotated Sorted Array II**

~~+10~~

**Numbers At Most N Given Digit Set**

Week 4: November 22nd - November 28th

~~+35~~

Longest Substring with At Most Two Distinct Characters

~~+10~~

**Unique Morse Code Words**

~~+10~~

**House Robber III**

~~+10~~

 Basic Calculator II

~~+10~~

**Smallest Integer Divisible by K**

~~+10~~

**Longest Substring with At Least K Repeating Characters**

~~+10~~

**Partition Equal Subset Sum**

~~+10~~

Sliding Window Maximum

Week 5: November 29th - November 30th

~~+35~~

**Maximum Average Subarray II**

~~+10~~

**Jump Game III**

~~+10~~

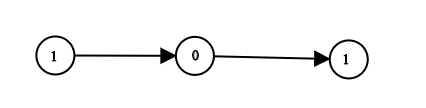
 The Skyline Problem

**Convert Binary Number in a Linked List to Integer**

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the *decimal value* of the number in the linked list.

**Example 1:**



**Input:** head = [1,0,1]

**Output:** 5

**Explanation:** (101) in base 2 = (5) in base 10

**Example 2:**

**Input:** head = [0]

**Output:** 0

**Example 3:**

**Input:** head = [1]

**Output:** 1

**Example 4:**

**Input:** head = [1,0,0,1,0,0,1,1,1,0,0,0,0,0,0]

**Output:** 18880

**Example 5:**

**Input:** head = [0,0]

**Output:** 0

**Constraints:**

* The Linked List is not empty.
* Number of nodes will not exceed 30.
* Each node's value is either 0 or 1.

   Hide Hint #1

Traverse the linked list and store all values in a string or array. convert the values obtained to decimal value.

   Hide Hint #2

You can solve the problem in O(1) memory using bits operation. use shift left operation ( << ) and or operation ( | ) to get the decimal value in one operation.

Solution

Overview

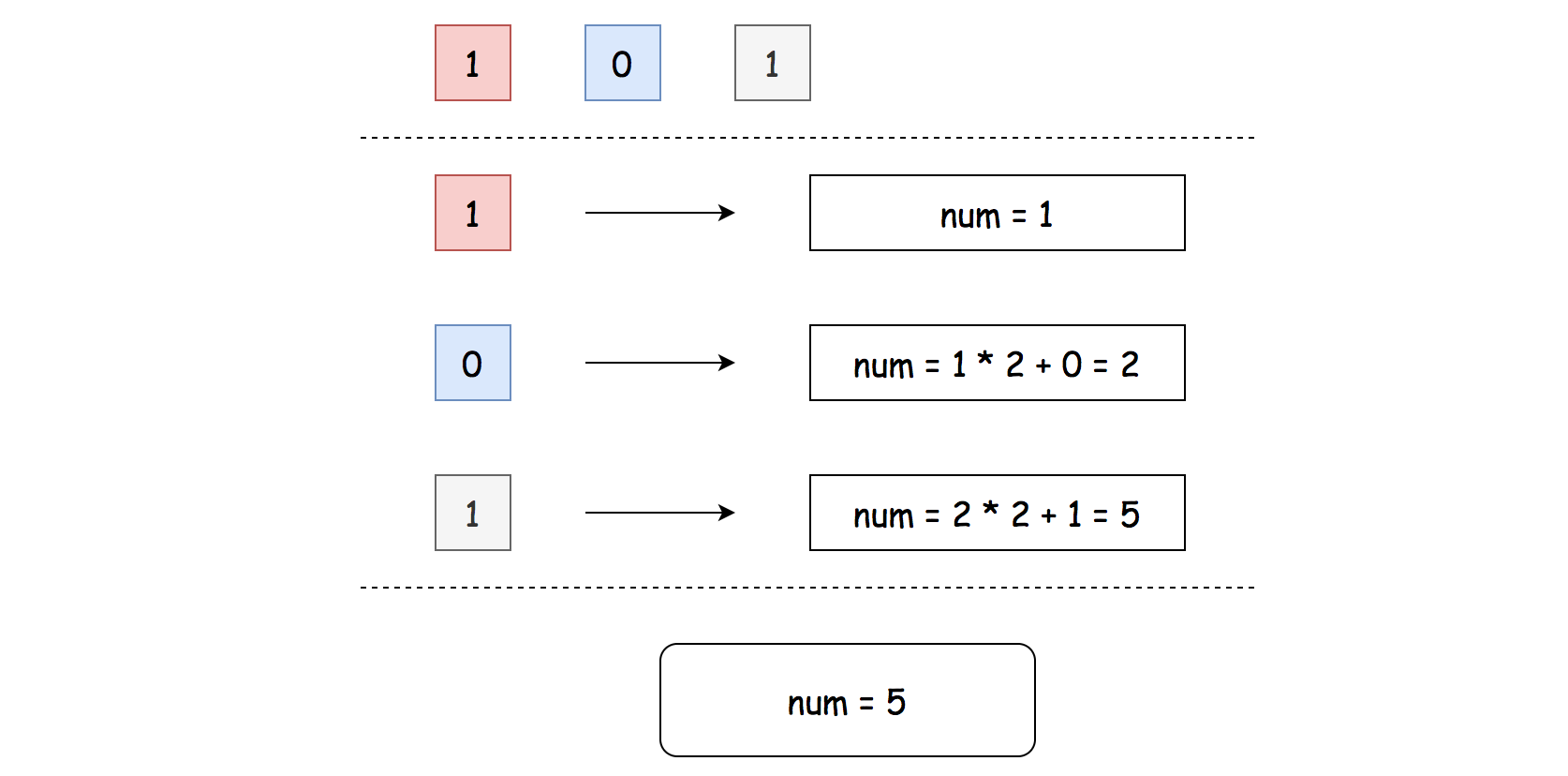
Here we have two subproblems:

* To parse non-empty linked list and to retrieve the digit sequence which represents a binary number.
* To convert this sequence into the number in decimal representation.

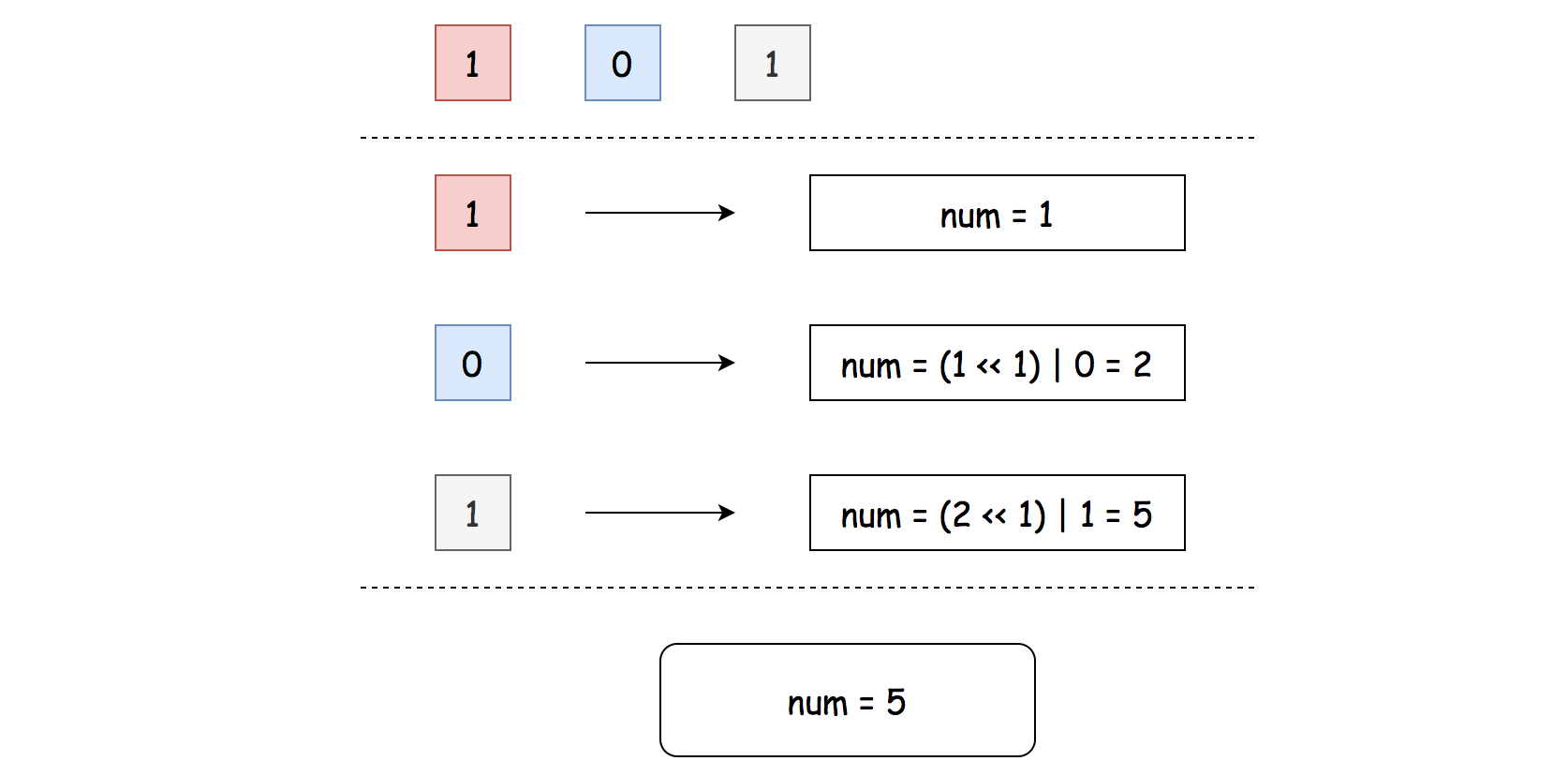
The first subproblem is easy because the linked list is guaranteed to be non-empty.

|  |
| --- |
| class Solution {  public int getDecimalValue(ListNode head) {  while (head.next != null) {  head = head.next;  // TODO  }  }  } |

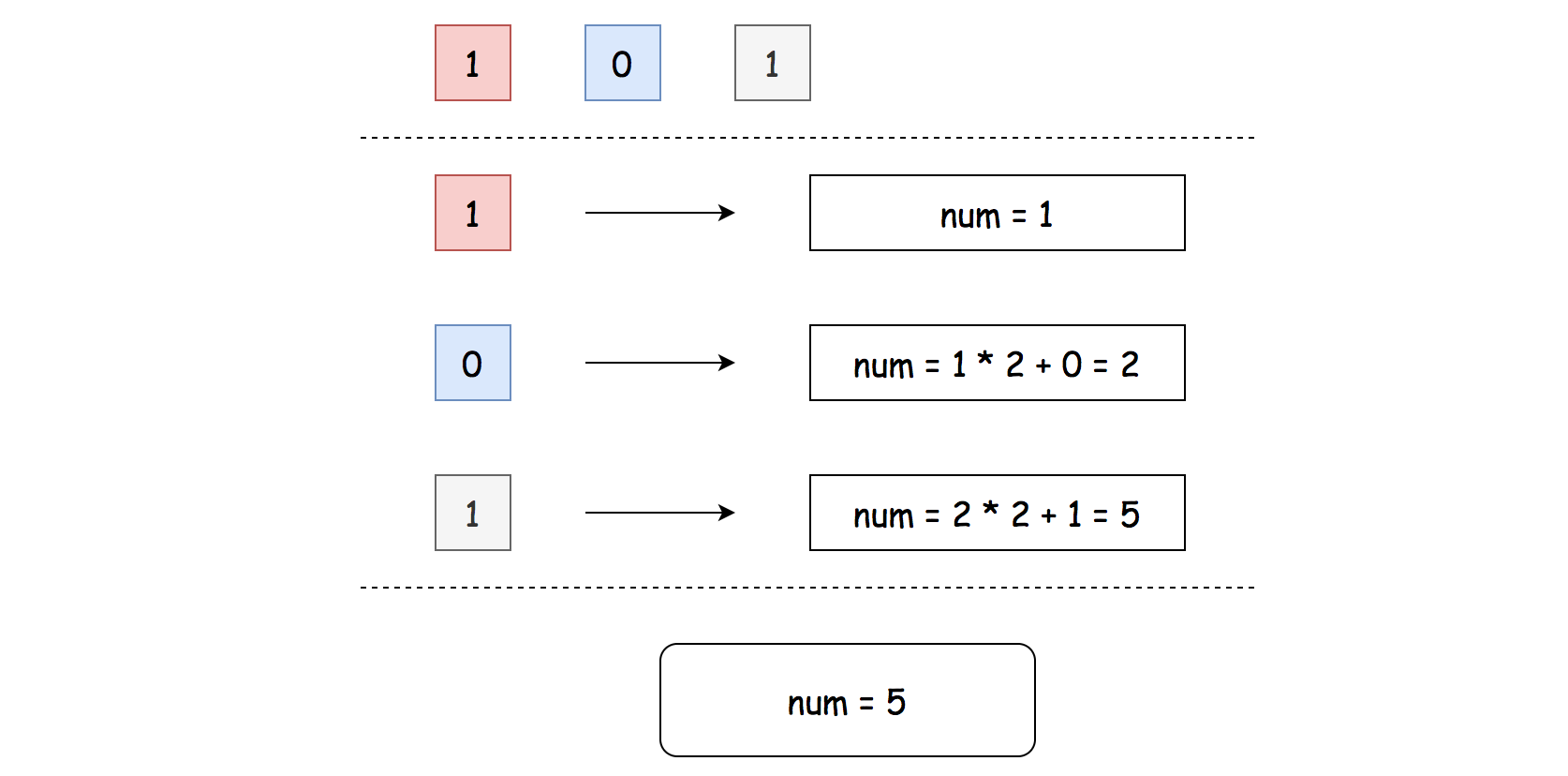
The second subproblem is to convert (101)\_2(101)2​ into 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 51×22+0×21+1×20=5. It could be solved in two ways. To use classical arithmetic is more straightforward

 Figure 1. Approach 1: num = num \* 2 + x

and to use bitwise operators is faster

 Figure 2. Approach 2: num = (num << 1) | x

#### Approach 1: Binary Representation

 Figure 3. Approach 1: num = num \* 2 + x.

* Initialize result number to be equal to head value: num = head.val. This operation is safe because the list is guaranteed to be non-empty.
* Parse linked list starting from the head: while head.next:
  + The current value is head.next.val. Update the result by shifting it by one to the left and adding the current value: num = num \* 2 + head.next.val.
* Return num.

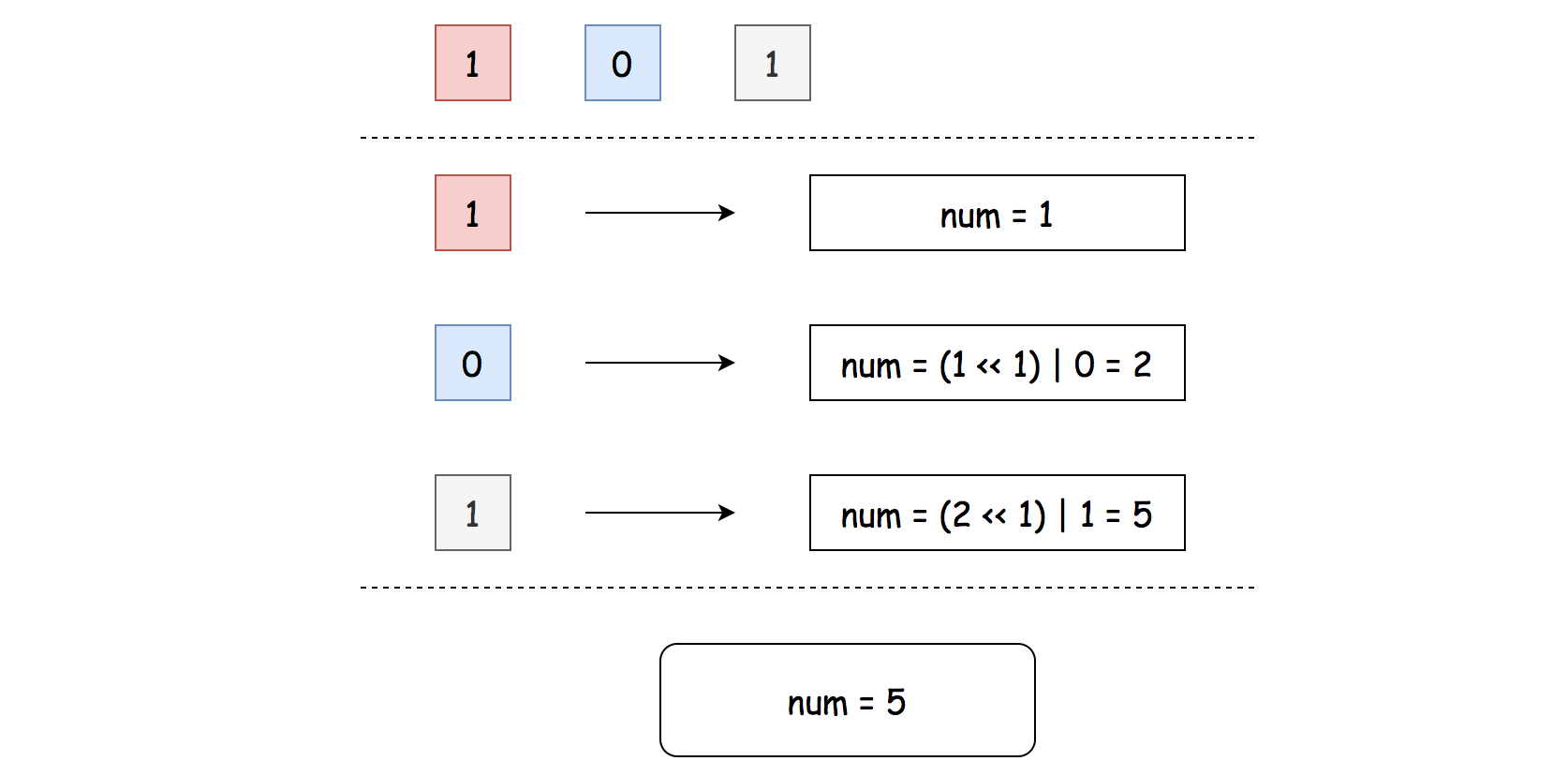
**Implementation**

|  |
| --- |
| class Solution {  public int getDecimalValue(ListNode head) {  int num = head.val;  while (head.next != null) {  num = num \* 2 + head.next.val;  head = head.next;  }  return num;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(1)O(1).

#### Approach 2: Bit Manipulation

 Figure 4. Approach 2: num = (num << 1) | x

* Initialize result number to be equal to head value: num = head.val. This operation is safe because the list is guaranteed to be non-empty.
* Parse linked list starting from the head: while head.next:
  + The current value is head.next.val. Update the result by shifting it by one to the left and adding the current value using logical OR: num = (num << 1) | head.next.val.
* Return num.

**Implementation**

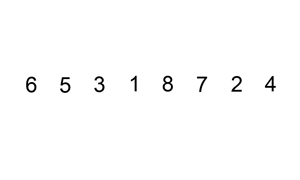
|  |
| --- |
| class Solution {  public int getDecimalValue(ListNode head) {  int num = head.val;  while (head.next != null) {  num = (num << 1) | head.next.val;  head = head.next;  }  return num;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(1)O(1).

**Insertion Sort List**

Sort a linked list using insertion sort.

  
A graphical example of insertion sort. The partial sorted list (black) initially contains only the first element in the list.  
With each iteration one element (red) is removed from the input data and inserted in-place into the sorted list

**Algorithm of Insertion Sort:**

1. Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.
2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.
3. It repeats until no input elements remain.

**Example 1:**

**Input:** 4->2->1->3

**Output:** 1->2->3->4

**Example 2:**

**Input:** -1->5->3->4->0

**Output:** -1->0->3->4->5

## Solution Article

#### Overview

[Insertion sort](https://en.wikipedia.org/wiki/Insertion_sort) is an intuitive sorting algorithm, although it is much less efficient than the more advanced algorithms such as quicksort or merge sort.

Often that we perform the sorting algorithm on an [Array](https://leetcode.com/explore/learn/card/fun-with-arrays) structure, this problem though asks us to perform the insertion sort on a **linked list** data structure, which makes the implementation a bit challenging.

In this article, we will present some tricks to manipulate the linked list, which would help us to simplify the logics of implementation.

#### Approach 1: Insertion Sort

**Intuition**

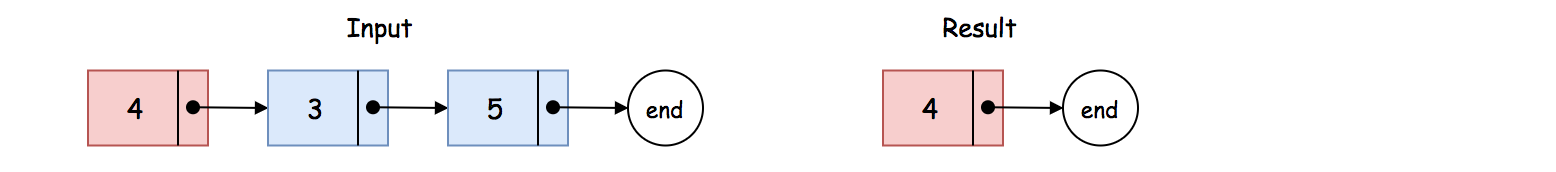
Let us first review the idea of insertion sort algorithm, which can be broke down into the following steps:

* First of all, we create an empty list which would be used to hold the results of sorting.
* We then iterate through each element in the input list. For each element, we need to find a proper position in the resulting list to insert the element, so that the order of the resulting list is maintained.
* As one can see, once the iteration in the above step terminates, we will obtain the resulting list where the elements are ordered.

Now, let us walk through a simple example, by applying the above intuition.

Given the input list input=[4, 3, 5], we have initially an empty resulting list result=[].

* We then iterate over the input list. For the first element 4, we need to find a proper position in the resulting list to place it. Since the resulting list is still empty, we then simply append it to the resulting list, i.e. result=[4].



* Now for the second element (i.e. 3) in the input list, similarly we need to insert it properly into the resulting list. As one can see, we need to insert it right before the element 4. As a result, the resulting list becomes [3, 4].



* Finally, for the last element (i.e. 5) in the input list, as it turns out, the proper position to place it is the tail of the resulting list. With this last iteration, we obtain a sorted list as result=[3, 4, 5].



**Algorithm**

To translate the above intuition into the implementation, we applied two **tricks**.

The first trick is that we will create a dummy (pseudo\_head) node which serves as a pointer pointing to the resulting list.

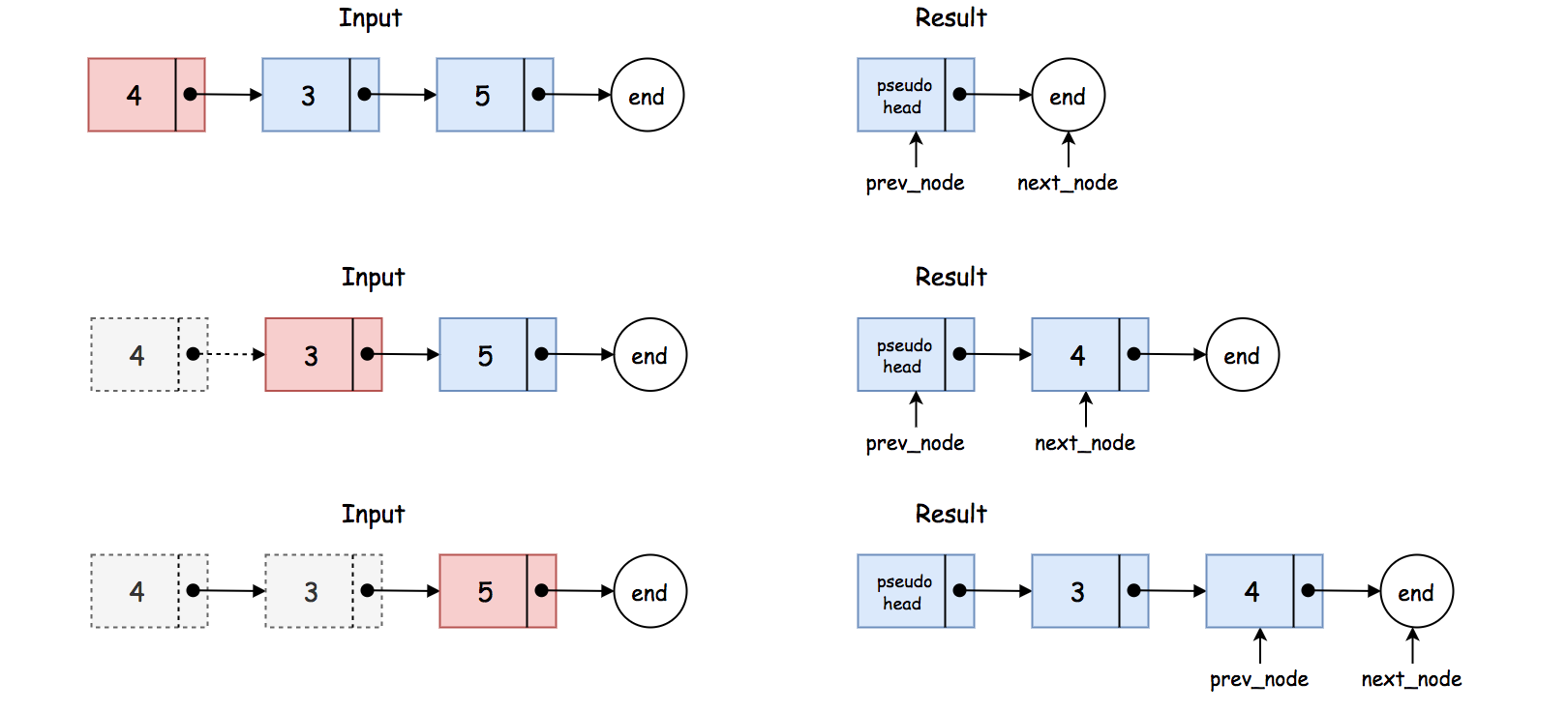
More precisely, this node facilitates us to always get a hold on the resulting list, especially when we need to insert a new element to the head of the resulting list. One will see later in more details how it can greatly simplify the logic.

In a singly-linked list, each node has only one pointer that points to the next node. If we would like to insert a new node (say B) before certain node (say A), we need to know the node (say C) that is currently before the node A, i.e. C -> A. With the reference in the node C, we could now insert the new node, i.e. C -> B -> A.

Given the above insight, in order to insert a new element into a singly-linked list, we apply another trick.

The idea is that we use a ***pair of pointers*** (namely prev -> next) which serve as place-holders to guard the position where in-between we would insert a new element (i.e. prev -> new\_node -> next).

With the same example before, i.e. input=[4, 3, 5], we illustrate what the above helper pointers look like at the moment of insertion, in the following graph:



Here are some sample implementations based on the above ideas:

|  |
| --- |
| class Solution {  public ListNode insertionSortList(ListNode head) {  ListNode dummy = new ListNode();  ListNode curr = head;  while (curr != null) {  // At each iteration, we insert an element into the resulting list.  ListNode prev = dummy;  // find the position to insert the current node  while (prev.next != null && prev.next.val < curr.val) {  prev = prev.next;  }  ListNode next = curr.next;  // insert the current node to the new list  curr.next = prev.next;  prev.next = curr;  // moving on to the next iteration  curr = next;  }  return dummy.next;  }  } |

**Complexity Analysis**

Let N*N* be the number of elements in the input list.

* Time Complexity: \mathcal{O}(N^2)O(*N*2)
  + First of all, we run an iteration over the input list.
  + At each iteration, we insert an element into the resulting list. In the worst case where the position to insert is the tail of the list, we have to walk through the entire resulting list.
  + As a result, the total steps that we need to walk in the worst case would be \sum\_{i=1}^{N} i = \frac{N(N+1)}{2}∑*i*=1*N*​*i*=2*N*(*N*+1)​.
  + To sum up, the overall time complexity of the algorithm is \mathcal{O}(N^2)O(*N*2).
* Space Complexity: \mathcal{O}(1)O(1)
  + We used some pointers within the algorithm. However, their memory consumption is constant regardless of the input.
  + **Note**, we did not create new nodes to hold the values of input list, but simply reorder the existing nodes.

**Consecutive Characters**

Given a string s, the power of the string is the maximum length of a non-empty substring that contains only one unique character.

Return the power of the string.

**Example 1:**

**Input:** s = "leetcode"

**Output:** 2

**Explanation:** The substring "ee" is of length 2 with the character 'e' only.

**Example 2:**

**Input:** s = "abbcccddddeeeeedcba"

**Output:** 5

**Explanation:** The substring "eeeee" is of length 5 with the character 'e' only.

**Example 3:**

**Input:** s = "triplepillooooow"

**Output:** 5

**Example 4:**

**Input:** s = "hooraaaaaaaaaaay"

**Output:** 11

**Example 5:**

**Input:** s = "tourist"

**Output:** 1

**Constraints:**

* 1 <= s.length <= 500
* s contains only lowercase English letters.

   Hide Hint #1

Keep an array power where power[i] is the maximum power of the i-th character.

   Hide Hint #2

The answer is max(power[i]).

## Solution

### **Overview**

This problem is very similar to [674. Longest Continuous Increasing Subsequence](https://leetcode.com/problems/longest-continuous-increasing-subsequence/), and the only difference is that we need a substring with the same characters instead of an increasing one. Therefore, similar methods can be applied. Below, a similar and simple approach is introduced.

### **Approach #1: One Pass**

**Intuition and Algorithm**

Recall the problem, we need to find "the maximum length of a non-empty substring that contains only one unique character".

In other words, we need to find the Longest Substring with **the same characters**.

We can iterate over the given string, and use a variable count to record the length of that substring.

When the next character is the same as the previous one, we increase count by one. Else, we reset count to 1.

With this method, when reaching the end of a substring with the same characters, count will be the length of that substring, since we reset the count when that substring starts, and increase count when iterate that substring.

Therefore, the maximum value of count is what we need. Another variable is needed to store the maximum while iterating.

|  |
| --- |
| class Solution {  public int maxPower(String s) {  int count = 0;  int maxCount = 0;  char previous = ' ';  for (int i = 0; i < s.length(); i++) {  char c = s.charAt(i);  if (c == previous) {  // if same as previous one, increase the count  count++;  } else {  // else, reset the count  count = 1;  previous = c;  }  maxCount = Math.max(maxCount, count);  }  return maxCount;  }  } |

**Complexity Analysis**

Let N*N* be the length of s.

* Time Complexity: O(N)*O*(*N*), since we perform one loop through s.
* Space Complexity: O(1)*O*(1), since we only have two integer variables count and max\_count(maxCount), and one character variable previous.

**Minimum Height Trees**

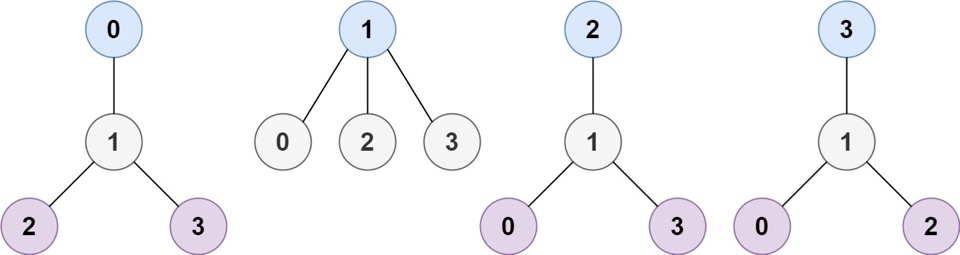
A tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h))  are called **minimum height trees** (MHTs).

Return a list of all ***MHTs'*** root labels. You can return the answer in **any order**.

The **height** of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

**Example 1:**

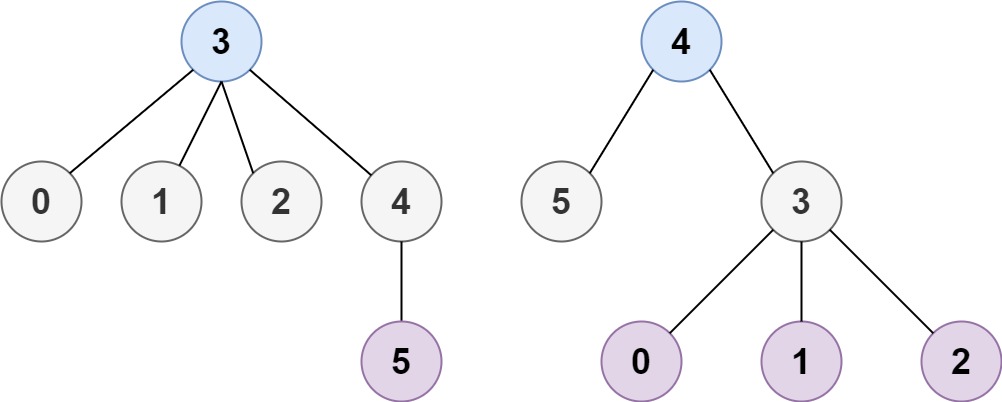


**Input:** n = 4, edges = [[1,0],[1,2],[1,3]]

**Output:** [1]

**Explanation:** As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.

**Example 2:**



**Input:** n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]

**Output:** [3,4]

**Example 3:**

**Input:** n = 1, edges = []

**Output:** [0]

**Example 4:**

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

**Output:** [0,1]

**Constraints:**

* 1 <= n <= 2 \* 104
* edges.length == n - 1
* 0 <= ai, bi < n
* ai != bi
* All the pairs (ai, bi) are distinct.
* The given input is **guaranteed** to be a tree and there will be **no repeated** edges.

Hide Hint #1

How many MHTs can a graph have at most?

## Solution

#### Overview

As the hints suggest, this problem is related to the [graph](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)) data structure. Moreover, it is closely related to the problems of [Course Schedule](https://leetcode.com/problems/course-schedule/) and [Course Schedule II](https://leetcode.com/problems/course-schedule-ii/). This relationship is not evident, yet it is the key to solve the problem, as one will see later.

First of all, as a **straight-forward** way to solve the problem, we can simply follow the requirements of the problem, as follows:

* Starting from each node in the graph, we treat it as a **root** to build a tree. Furthermore, we would like to know the distance between this root node and the rest of the nodes. The maximum of the distance would be the **height** of this tree.
* Then according to the definition of **Minimum Height Tree** (MHT), we simply filter out the roots that have the minimal height among all the trees.

The first step we describe above is actually the problem of [Maximum Depth of N-ary Tree](https://leetcode.com/problems/maximum-depth-of-n-ary-tree/), which is to find the maximum distance from the root to the leaf nodes. For this, we can either apply the [Depth-First Search](https://leetcode.com/explore/learn/card/queue-stack/232/practical-application-stack/) (**DFS**) or [Breadth-First Search](https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/) (**BFS**) algorithms.

Without a rigid proof, we can see that the above straight-forward solution is correct, and it would work for most of the test cases.

However, this solution is not efficient, whose time complexity would be \mathcal{O}(N^2)O(*N*2) where N*N* is the number of nodes in the tree. As one can imagine, it will result in **Time Limit Exceeded** exception in the online judge.

As a spoiler alert, in this article, we will present a [***topological sorting***](https://en.wikipedia.org/wiki/Topological_sorting) alike algorithm with time complexity of \mathcal{O}(N)O(*N*), which is also the algorithm to solve the well-known course schedule problems.

#### Approach 1: Topological Sorting

**Intuition**

First of all, let us clarify some concepts.

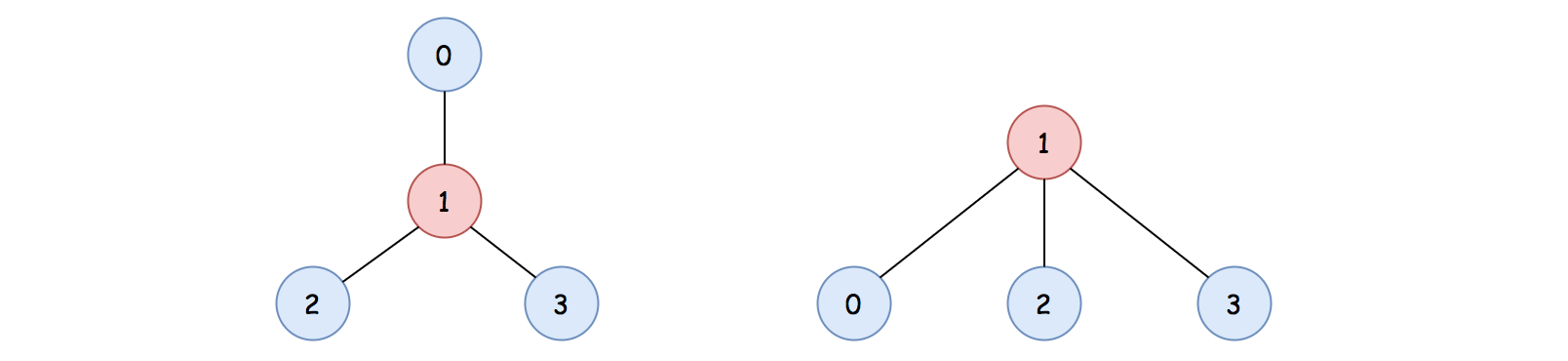
The **distance** between two nodes is the number of edges that connect the two nodes.

Note, normally there could be multiple paths to connect nodes in a graph. In our case though, since the input graph can form a tree from any node, as specified in the problem, there could only be **one path** between any two nodes. In addition, there would be no cycle in the graph. As a result, there would be no ambiguity in the above definition of distance.

The **height** of a tree can be defined as the maximum distance between the root and all its leaf nodes.

With the above definitions, we can rephrase the problem as finding out the nodes that are overall close to all other nodes, especially the leaf nodes.

If we view the graph as an area of circle, and the leaf nodes as the peripheral of the circle, then what we are looking for are actually the [***centroids***](https://en.wikipedia.org/wiki/Centroid) of the circle, i.e. nodes that is close to all the peripheral nodes (leaf nodes).



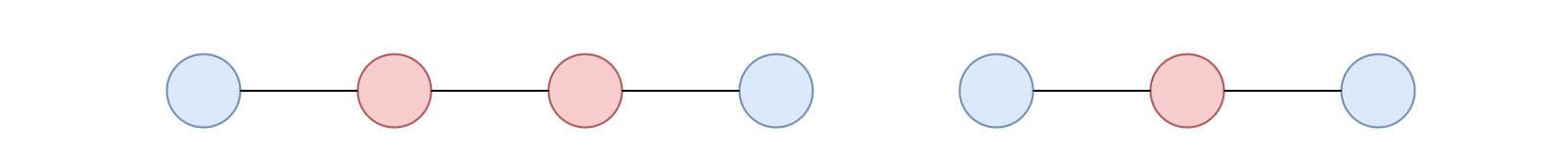
For instance, in the above graph, it is clear that the node with the value 1 is the centroid of the graph. If we pick the node 1 as the root to form a tree, we would obtain a tree with the minimum height, compared to other trees that are formed with any other nodes.

Before we proceed, here we make one assertion which is essential to the algorithm.

For the tree-alike graph, the number of centroids is no more than 2.

If the nodes form a chain, it is intuitive to see that the above statement holds, which can be broken into the following two cases:

* If the number of nodes is even, then there would be two centroids.
* If the number of nodes is odd, then there would be only one centroid.



For the rest of cases, we could prove by **contradiction**. Suppose that we have 3 centroids in the graph, if we remove all the non-centroid nodes in the graph, then the 3 centroids nodes must form a triangle shape, as follows:



Because these centroids are equally important to each other, and they should equally close to each other as well. If any of the edges that is missing from the triangle, then the 3 centroids would be reduced down to a single centroid.

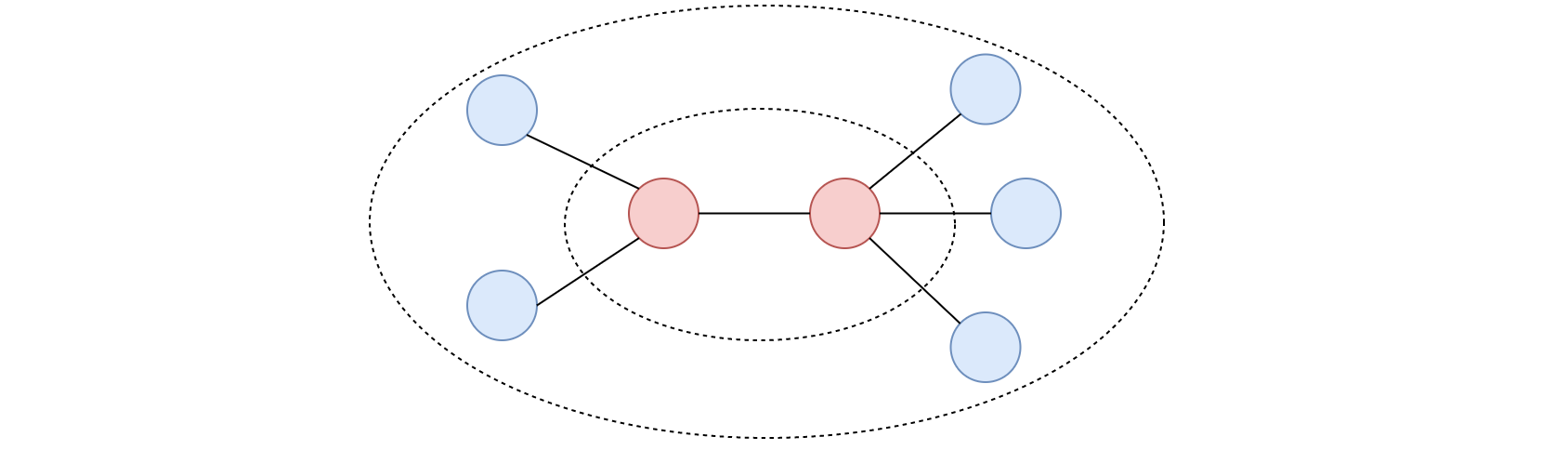
However, the triangle shape forms a cycle which is **contradicted** to the condition that there is no cycle in our tree-alike graph. Similarly, for any of the cases that have more than 2 centroids, they must form a cycle among the centroids, which is contradicted to our condition.

Therefore, there cannot be more than 2 centroids in a tree-alike graph.

**Algorithm**

Given the above intuition, the problem is now reduced down to looking for all the ***centroid*** nodes in a tree-alike graph, which in addition are no more than two.

The idea is that we trim out the leaf nodes layer by layer, until we reach the core of the graph, which are the centroids nodes.



Once we trim out the first layer of the leaf nodes (nodes that have only one connection), some of the non-leaf nodes would become leaf nodes.

The trimming process continues until there are only two nodes left in the graph, which are the centroids that we are looking for.

The above algorithm resembles the topological sorting algorithm which generates the order of objects based on their dependencies. For instance, in the scenario of course scheduling, the courses that have the least dependency would appear first in the order.

In our case, we trim out the leaf nodes first, which are the **farther** away from the centroids. At each step, the nodes we trim out are closer to the centroids than the nodes in the previous step. At the end, the trimming process terminates at the **centroids** nodes.

**Implementation**

Given the above algorithm, we could implement it via the Breadth First Search (BFS) strategy, to trim the leaf nodes layer by layer (i.e. level by level).

* Initially, we would build a graph with the [*adjacency list*](https://en.wikipedia.org/wiki/Adjacency_list) from the input.
* We then create a queue which would be used to hold the leaf nodes.
* At the beginning, we put all the current leaf nodes into the queue.
* We then run a loop until there is only two nodes left in the graph.
* At each iteration, we remove the current leaf nodes from the queue. While removing the nodes, we also remove the edges that are linked to the nodes. As a consequence, some of the non-leaf nodes would become leaf nodes. And these are the nodes that would be trimmed out in the next iteration.
* The iteration terminates when there are no more than two nodes left in the graph, which are the desired centroids nodes.

Here are some sample implementations that are inspired from the post of [dietpepsi](https://leetcode.com/problems/minimum-height-trees/discuss/76055/Share-some-thoughts) in the discussion forum.

|  |
| --- |
| class Solution {  public List<Integer> findMinHeightTrees(int n, int[][] edges) {  // base cases  if (n < 2) {  ArrayList<Integer> centroids = new ArrayList<>();  for (int i = 0; i < n; i++)  centroids.add(i);  return centroids;  }  // Build the graph with the adjacency list  ArrayList<Set<Integer>> neighbors = new ArrayList<>();  for (int i = 0; i < n; i++)  neighbors.add(new HashSet<Integer>());  for (int[] edge : edges) {  Integer start = edge[0], end = edge[1];  neighbors.get(start).add(end);  neighbors.get(end).add(start);  }  // Initialize the first layer of leaves  ArrayList<Integer> leaves = new ArrayList<>();  for (int i = 0; i < n; i++)  if (neighbors.get(i).size() == 1)  leaves.add(i);  // Trim the leaves until reaching the centroids  int remainingNodes = n;  while (remainingNodes > 2) {  remainingNodes -= leaves.size();  ArrayList<Integer> newLeaves = new ArrayList<>();  // remove the current leaves along with the edges  for (Integer leaf : leaves) {  // the only neighbor left for the leaf node  Integer neighbor = neighbors.get(leaf).iterator().next();  // remove the edge along with the leaf node  neighbors.get(neighbor).remove(leaf);  if (neighbors.get(neighbor).size() == 1)  newLeaves.add(neighbor);  }  // prepare for the next round  leaves = newLeaves;  }  // The remaining nodes are the centroids of the graph  return leaves;  }  } |

**Complexity Analysis**

Let |V|∣*V*∣ be the number of nodes in the graph, then the number of edges would be |V| - 1∣*V*∣−1 as specified in the problem.

* Time Complexity: \mathcal{O}(|V|)O(∣*V*∣)
  + First, it takes |V|-1∣*V*∣−1 iterations for us to construct a graph, given the edges.
  + With the constructed graph, we retrieve the initial leaf nodes, which takes |V|∣*V*∣ steps.
  + During the BFS trimming process, we will trim out almost all the nodes (|V|∣*V*∣) and edges (|V|-1∣*V*∣−1) from the edges. Therefore, it would take us around |V| + |V| - 1∣*V*∣+∣*V*∣−1 operations to reach the centroids.
  + To sum up, the overall time complexity of the algorithm is \mathcal{O}(|V|)O(∣*V*∣).
* Space Complexity: \mathcal{O}(|V|)O(∣*V*∣)
  + We construct the graph with adjacency list, which has |V|∣*V*∣ nodes and |V|-1∣*V*∣−1 edges. Therefore, we would need |V| + |V|-1∣*V*∣+∣*V*∣−1 space for the representation of the graph.
  + In addition, we use a queue to keep track of the leaf nodes. In the worst case, the nodes form a star shape, with one centroid and the rest of the nodes as leaf nodes. In this case, we would need |V|-1∣*V*∣−1 space for the queue.
  + To sum up, the overall space complexity of the algorithm is also \mathcal{O}(|V|)O(∣*V*∣).

**Minimum Cost to Move Chips to The Same Position**

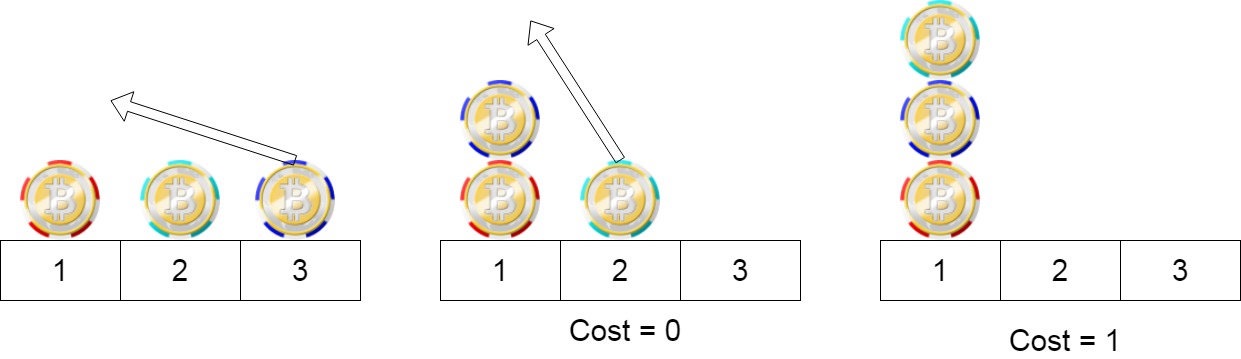
We have n chips, where the position of the ith chip is position[i].

We need to move all the chips to **the same position**. In one step, we can change the position of the ith chip from position[i] to:

* position[i] + 2 or position[i] - 2 with cost = 0.
* position[i] + 1 or position[i] - 1 with cost = 1.

Return the minimum cost needed to move all the chips to the same position.

**Example 1:**



**Input:** position = [1,2,3]

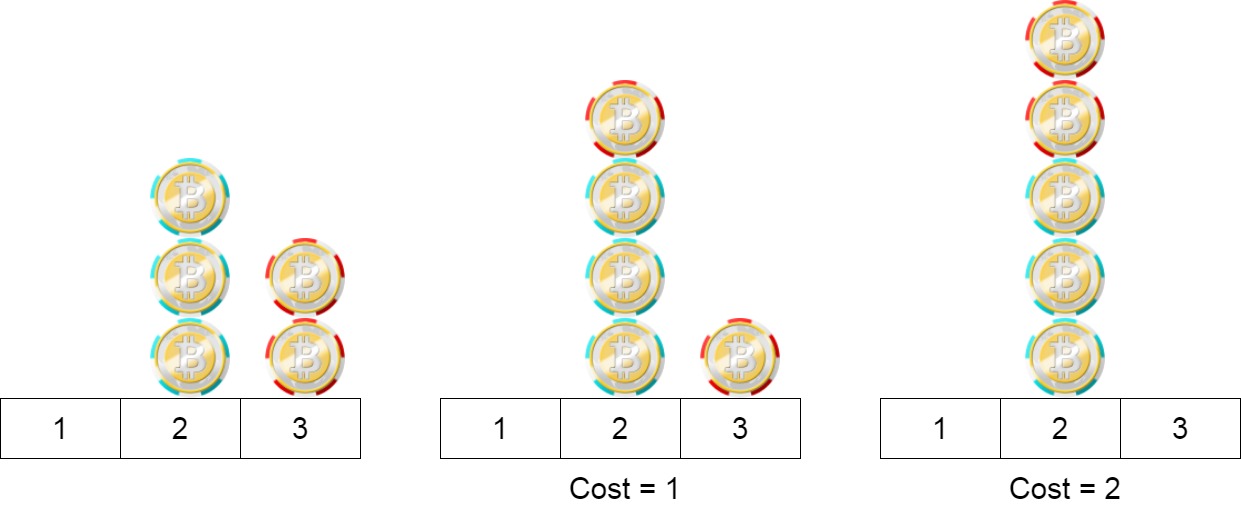
**Output:** 1

**Explanation:** First step: Move the chip at position 3 to position 1 with cost = 0.

Second step: Move the chip at position 2 to position 1 with cost = 1.

Total cost is 1.

**Example 2:**



**Input:** position = [2,2,2,3,3]

**Output:** 2

**Explanation:** We can move the two chips at position 3 to position 2. Each move has cost = 1. The total cost = 2.

**Example 3:**

**Input:** position = [1,1000000000]

**Output:** 1

**Constraints:**

* 1 <= position.length <= 100
* 1 <= position[i] <= 10^9

 Hide Hint #1

The first move keeps the parity of the element as it is.

   Hide Hint #2

The second move changes the parity of the element.

   Hide Hint #3

Since the first move is free, if all the numbers have the same parity, the answer would be zero.

   Hide Hint #4

Find the minimum cost to make all the numbers have the same parity.

## Solution

#### Overview

Though marked as "Easy", this problem is a little tricky and requires some observations and insights. It's recommended to try a few examples to find out some regular patterns.

Below, we will discuss a simple approach to solve this problem.

#### Approach 1: Moving Chips Cleverly

**Intuition**

Notice that we have two types of costs:

1. Costs 0 when moving to position[i] + 2 or position[i] - 2.
2. Costs 1 when moving to position[i] + 1 or position[i] - 1.

Since move to position[i] + 2 or position[i] - 2 is free, it is natural to think that firstly moving chips as close as possible, with 0 cost.

**In fact, we can move all chips at even positions to position 0, and move all chips at the odd positions to position 1.**

Then, we only have many chips at position 0 and other chips at position 1. Next, we only need to move those two piles together.

Given two piles of chips located at 0 and 1 respectively, intuitively it would be less effort-taking (i.e. less cost) to move the smaller pile to the larger one, which makes the total cost to:

Cost = min(even\\_cnt, odd\\_cnt)*Cost*=*min*(*even*\_*cnt*,*odd*\_*cnt*)

where even\\_cnt*even*\_*cnt* represents the number of chips at the even positions, and odd\\_cnt*odd*\_*cnt* represents the number of chips at the odd positions.

Good, now we have a not bad cost. Can we do better?

Well, now we will prove that this cost is the smallest possible one.

As for the final position of those chips pile, there are only two possibilities:

1. The final position is at the even position, or
2. The final position is at the odd position.

In the first case, we at least need to cost odd\_cnt to move all the chips at the odd positions to the even positions. Similarly, in the second case, we at least need to cost even\_cnt.

Therefore, min(even\_cnt, odd\_cnt) is the smallest possible cost.

In conclusion, the policy we gave above will achieve the smallest possible cost. What we need to return is just min(even\_cnt, odd\_cnt).

**Algorithm**

We just need to count the number of chips at the even positions and the number of chips at the odd positions and return the smaller one.

|  |
| --- |
| class Solution {  public int minCostToMoveChips(int[] position) {  int even\_cnt = 0;  int odd\_cnt = 0;  for (int i : position) {  if (i % 2 == 0) {  even\_cnt++;  } else {  odd\_cnt++;  }  }  return Math.min(odd\_cnt, even\_cnt);  }  } |

You can also count either the odd or the even numbers, and then deduct that number from the length to obtain the other one.

**Complexity Analysis**

Let N*N* be the length of position.

* Time Complexity : \mathcal{O}(N)O(*N*) since we need to iterate position once.
* Space Complexity : \mathcal{O}(1)O(1) since we only use two ints: even\_cnt and odd\_cnt.

**Find the Smallest Divisor Given a Threshold**

Given an array of integers nums and an integer threshold, we will choose a positive integer divisor, divide all the array by it, and sum the division's result. Find the **smallest** divisor such that the result mentioned above is less than or equal to threshold.

Each result of the division is rounded to the nearest integer greater than or equal to that element. (For example: 7/3 = 3 and 10/2 = 5).

It is guaranteed that there will be an answer.

**Example 1:**

**Input:** nums = [1,2,5,9], threshold = 6

**Output:** 5

**Explanation:** We can get a sum to 17 (1+2+5+9) if the divisor is 1.

If the divisor is 4 we can get a sum of 7 (1+1+2+3) and if the divisor is 5 the sum will be 5 (1+1+1+2).

**Example 2:**

**Input:** nums = [44,22,33,11,1], threshold = 5

**Output:** 44

**Example 3:**

**Input:** nums = [21212,10101,12121], threshold = 1000000

**Output:** 1

**Example 4:**

**Input:** nums = [2,3,5,7,11], threshold = 11

**Output:** 3

**Constraints:**

* 1 <= nums.length <= 5 \* 104
* 1 <= nums[i] <= 106
* nums.length <= threshold <= 106

Hide Hint #1

Examine every possible number for solution. Choose the largest of them.

   Hide Hint #2

Use binary search to reduce the time complexity.

## Solution

#### Overview

Time limits for this task are adjusted to cut off linear-time brute-force solution by TLE (Time Limit Exceeded).

|  |
| --- |
| class Solution {  public int computeSum(int[] nums, int x) {  int s = 0;  for (int n : nums) {  s += n / x + (n % x == 0 ? 0 : 1);  }  return s;  }    public int smallestDivisor(int[] nums, int threshold) {  int d = 1;  while(computeSum(nums, d) > threshold) {  ++d;  }  return d;  }  } |

That means that one should figure out a binary search logarithmic-time solution. To perform a binary search, one should somehow define the search boundaries for the divisor. There are two possible approaches here:

* Approach 1, Brute-force. First, use binary search to define search limits, i.e., left and right boundaries for the search. Then use binary search one more time to find the smallest divisor in the defined boundaries.
* Approach 2, Math. First, figure out that the maximum possible divisor is the maximum number in the array. Then use binary search in the limits from 1 to max(nums).

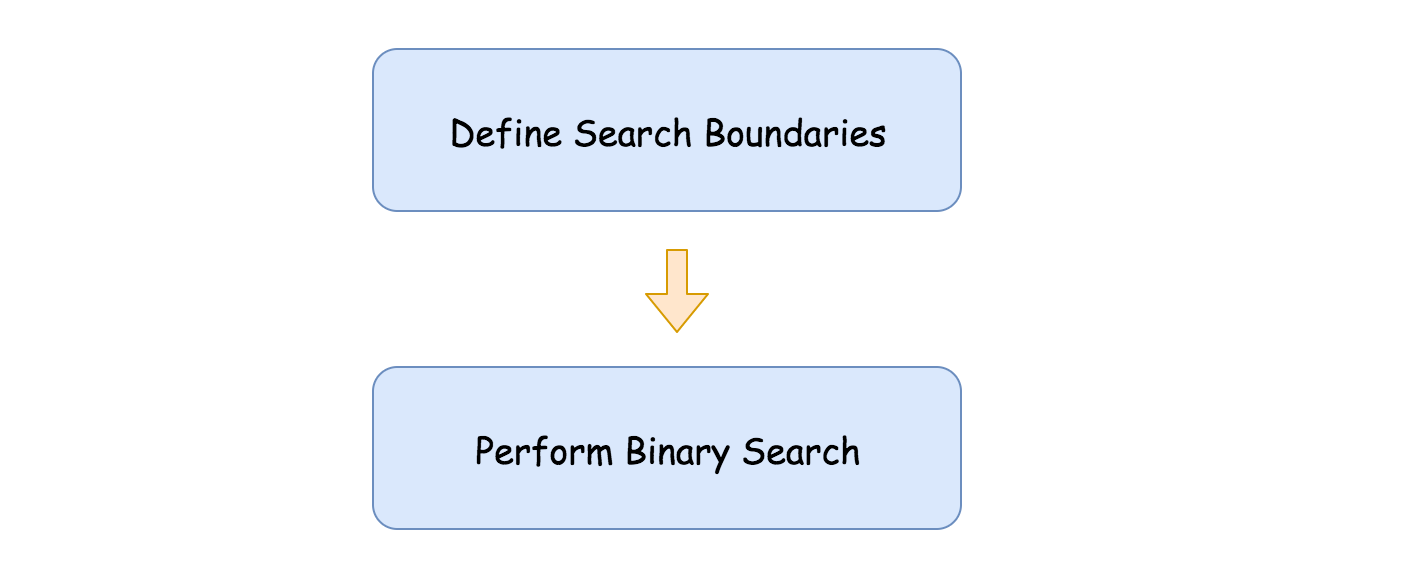
#### Approach 1: Brute-Force + Binary Search

The idea is to define search limits first and then find the divisor.

**Split into Two Subproblems**

There are two subproblems, and both should be done in a logarithmic time:

* Define search limits, i.e. left, and right boundaries for the search.
* Perform a binary search in the defined boundaries.

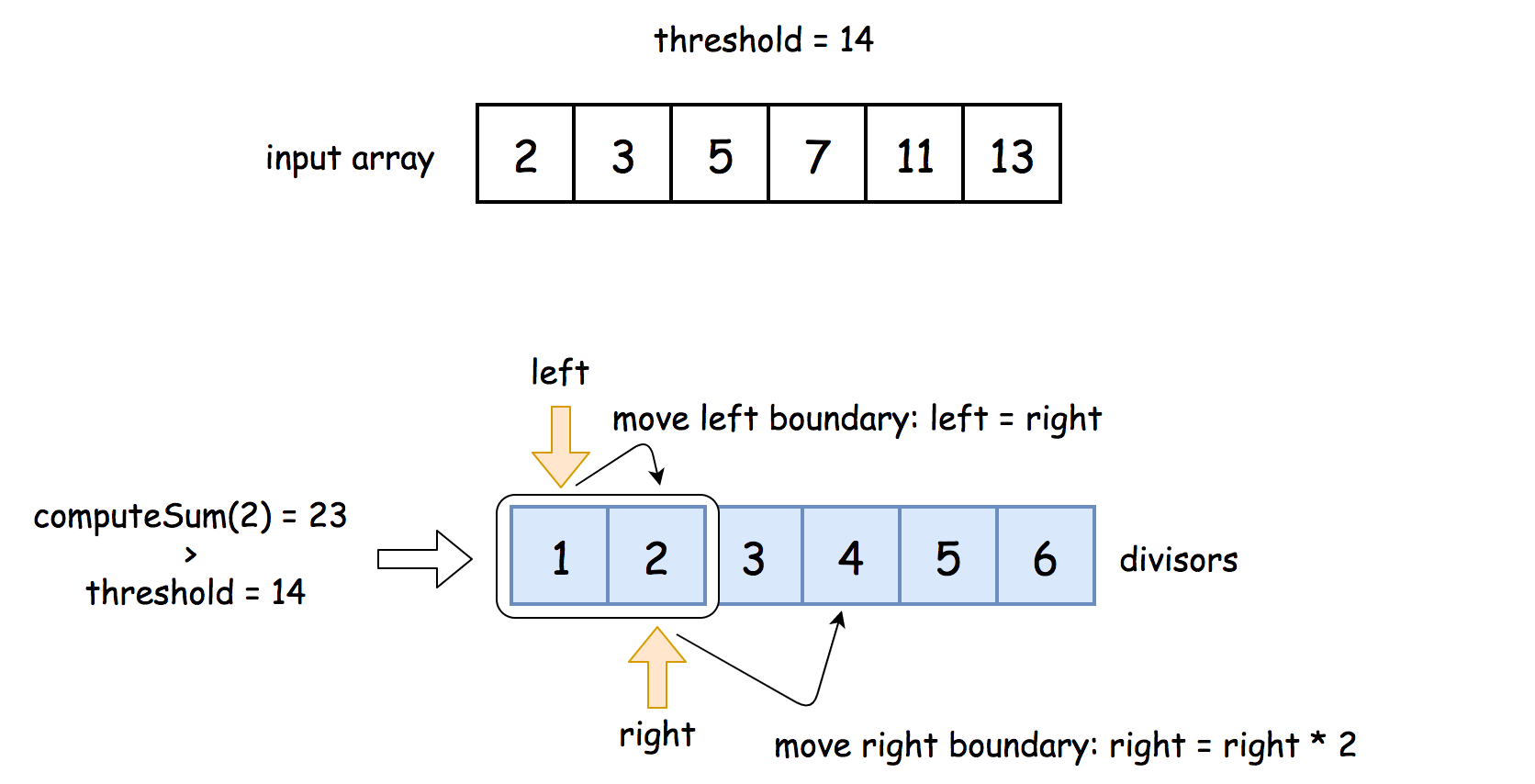
 Figure 1. Solution schema.

**Define Search Boundaries**

This is a key subproblem here.

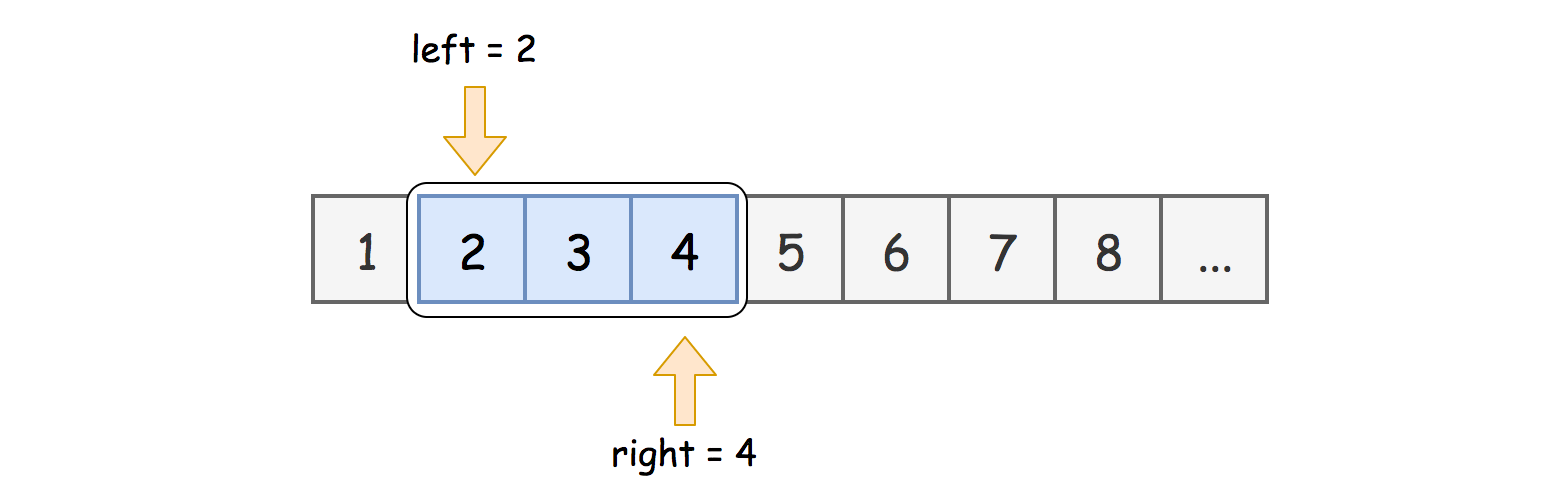
The idea is quite simple. Let's take 1 and 2, as left and right boundaries. If the required sum from the right limit is greater than the threshold, then it's outside the boundaries, on the right.

That means that the left boundary could be moved to the right, and the right boundary should be extended. To keep logarithmic time complexity, let's extend it twice as far: right = right \* 2.

 Figure 2. Define the search boundaries.

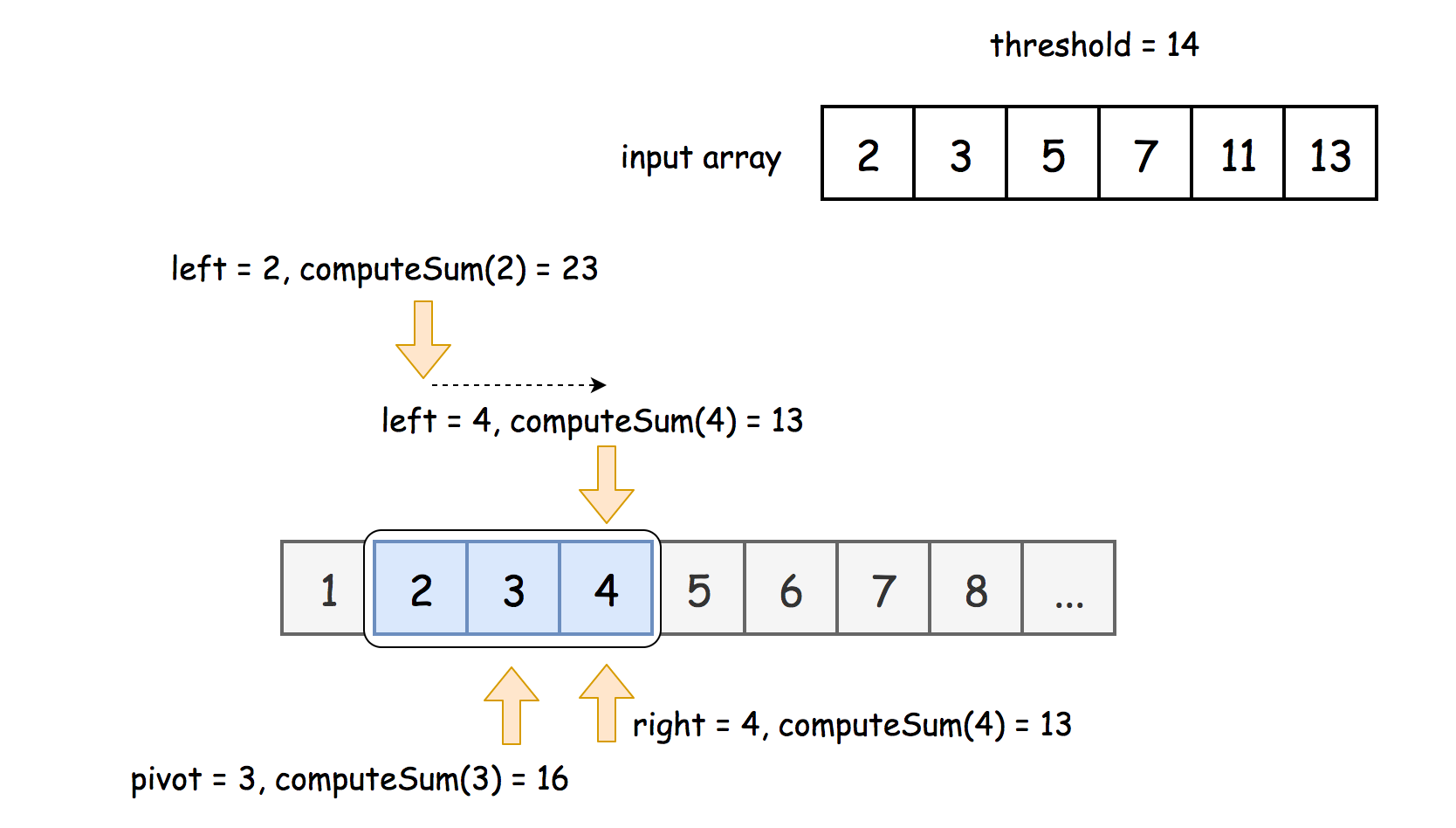
If the required sum from the right boundary is now less than the threshold, we're done, the boundaries are set. If not, repeat these two steps until the boundaries are established:

* Move the left boundary to the right: left = right.
* Extend the right boundary: right = right \* 2.

 Figure 3. Search boundaries: the range of divisors to check.

**Binary Search**

[Binary search](https://leetcode.com/explore/learn/card/binary-search/) is a textbook algorithm with a logarithmic time complexity. At each iteration, one has to choose one side of the array to continue.

 Figure 4. Binary Search.

**Prerequisites: left and right shifts**

To speed up, one could use here [bitwise shifts](https://wiki.python.org/moin/BitwiseOperators):

* Left shift: x << 1. The same as multiplying by 2: x \* 2.
* Right shift: x >> 1. The same as dividing by 2: x / 2.

**Algorithm**

* Implement function computeSum(x) which divides nums elements by x and then returns their sum.
* Define boundaries:
  + Initialize left = 1 and right = 2.
  + While desired divisor is still greater than the upper limit: compute\_sum(right) > threshold:
    - Set left boundary equal to the right one: left = right.
    - Extend right boundary: right \*= 2. To speed up, use right shift instead of multiplication: right <<= 1.
  + Now, the target is between left and right boundaries.
* Binary Search:
  + While left <= right:
    - Pick a pivot divisor in the middle: pivot = (left + right) / 2. To avoid overflow in Java, use the form pivot = left + ((right - left) >> 1) instead of straightforward expression above. Note, that there is no overflow in Python.
    - Compute the requested sum for that pivot divisor: num = compute\_sum(pivot).
    - Compare num and the threshold:
      * If num > threshold, continue to search on the right left = pivot + 1.
      * Else continue to search on the left right = pivot - 1.
  + At the end of loop, left > right, computeSum(right) > threshold and computeSum(left) <= threshold. Hence left is the smallest divisor for which the requested sum is less or equal to threshold. Return left.

**Implementation**

|  |
| --- |
| class Solution {  public long computeSum(int[] nums, int x) {  long s = 0;  for (int n : nums) {  s += n / x + (n % x == 0 ? 0 : 1);  }  return s;  }    public int smallestDivisor(int[] nums, int threshold) {  // search boundaries for the divisor  int left = 1, right = 2;  while (computeSum(nums, right) > threshold) {  left = right;  right <<= 1;  }    // binary search  while (left <= right) {  int pivot = left + ((right - left) >> 1);  long num = computeSum(nums, pivot);  if (num > threshold) {  left = pivot + 1;  } else {  right = pivot - 1;  }  }    // at the end of loop, left > right,  // computeSum(right) > threshold  // computeSum(left) <= threshold  // --> return left  return left;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N \log N\_{max})O(*N*log*Nmax*​), where N*N* is number of the elements in the input array, and N\_{max}*Nmax*​ is a maximum number in that array. There are two operations here: to define search boundaries and to perform binary search, both take \log N\_{max}log*Nmax*​ iterations, and at each iteration one computes a sum in \mathcal{O}(N)O(*N*) time.

Please check the time complexity discussion after Approach 2 to have a detailed explanation of how to derive binary search algorithm complexity using Master Theorem.

* Space complexity : \mathcal{O}(1)O(1), we don't allocate any additional data structures.

#### Approach 2: Math + Binary Search

The idea is to perform binary search in limits 1, max(nums). This approach uses the idea that the maximum divisor to consider is max(nums). For this divisor and all greater numbers, the requested sum is equal to 2N2*N*, and hence there is no sense to consider greater divisors.

**Algorithm**

* Implement function computeSum(x) which divides nums elements by x and then returns their sum.
* Initialize search limits: left = 1, right = max(nums) = nums[nums.length - 1].
* Perform binary search in boundaries from 1 to max(nums):
  + While left <= right:
    - Pick a pivot divisor in the middle: pivot = (left + right) / 2. To avoid overflow in Java, use the form pivot = left + ((right - left) >> 1) instead of straightforward expression above. Note, that there is no overflow in Python.
    - Compute the requested sum for that pivot divisor: num = compute\_sum(pivot).
    - Compare num and the threshold:
      * If num > threshold, continue to search on the right left = pivot + 1.
      * Else continue to search on the left right = pivot - 1.
  + At the end of loop, left > right, computeSum(right) > threshold and computeSum(left) <= threshold. Hence left is the smallest divisor for which the requested sum is less or equal to threshold. Return left.

**Implementation**

|  |
| --- |
| class Solution {  public int computeSum(int[] nums, int x) {  int s = 0;  for (int n : nums) {  s += n / x + (n % x == 0 ? 0 : 1);  }  return s;  }    public int smallestDivisor(int[] nums, int threshold) {  // binary search  int left = 1;  int right = nums[nums.length - 1];  while (left <= right) {  int pivot = left + ((right - left) >> 1);  int num = computeSum(nums, pivot);  if (num > threshold) {  left = pivot + 1;  } else {  right = pivot - 1;  }  }    // at the end of loop, left > right,  // computeSum(right) > threshold  // computeSum(left) <= threshold  // --> return left  return left;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N \log N\_{max})O(*N*log*Nmax*​), where N*N* is number of the elements in the input array, and N\_{max}*Nmax*​ is a maximum number in that array.

Binary search takes \log N\_{max}log*Nmax*​ iterations and at each iteration one computes a sum in \mathcal{O}(N)O(*N*) time.

Let's compute time complexity of the binary search with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) T(N\_{max}) = aT\left(\frac{N\_{max}}{b}\right) + \Theta(N\_{max}^d)*T*(*Nmax*​)=*aT*(*bNmax*​​)+Θ(*Nmaxd*​). The equation represents dividing the problem up into a*a* subproblems of size \frac{N\_{max}}{b}*bNmax*​​ in \Theta(N\_{max}^d)Θ(*Nmaxd*​) time. Here, at step, there is only one subproblem a = 1, its size is a half of the initial problem b = 2, and all this happens in a constant time d = 0. That means that \log\_b{a} = dlog*b*​*a*=*d* and hence we're dealing with [case 2](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in \mathcal{O}(N\_{max}^{\log\_b{a}} \log^{d + 1} N\_{max})O(*Nmax*log*b*​*a*​log*d*+1*Nmax*​) = \mathcal{O}(\log N\_{max})O(log*Nmax*​) time complexity.

* Space complexity : \mathcal{O}(1)O(1), it's a constant space solution.

**Two Sum Less Than K**

Given an array nums of integers and integer k, return the maximum sum such that there exists i < j with nums[i] + nums[j] = sum and sum < k. If no i, j exist satisfying this equation, return -1.

**Example 1:**

**Input:** nums = [34,23,1,24,75,33,54,8], k = 60

**Output:** 58

**Explanation:** We can use 34 and 24 to sum 58 which is less than 60.

**Example 2:**

**Input:** nums = [10,20,30], k = 15

**Output:** -1

**Explanation:** In this case it is not possible to get a pair sum less that 15.

**Constraints:**

* 1 <= nums.length <= 100
* 1 <= nums[i] <= 1000
* 1 <= k <= 2000

   Hide Hint #1

What if we have the array sorted?

   Hide Hint #2

Loop the array and get the value A[i] then we need to find a value A[j] such that A[i] + A[j] < K which means A[j] < K - A[i]. In order to do that we can find that value with a binary search.

## Solution Article

This problem is a variation of [Two Sum](https://leetcode.com/articles/two-sum/). The main difference is that we are not searching for the exact target here. Instead, our sum is in some relation with the target. For this problem, we are looking for a maximum sum that is smaller than the target.

First, let's check solutions for the similar problems:

1. [Two Sum](https://leetcode.com/articles/two-sum/) uses a hashmap to find complement values, and therefore achieves \mathcal{O}(N)O(*N*) time complexity.
2. [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/) uses the two pointers pattern and also has \mathcal{O}(N)O(*N*) time complexity for a sorted array. We can use this approach for any array if we sort it first, which bumps the time complexity to \mathcal{O}(n\log{n})O(*n*log*n*).

Since our sum can be any value smaller than the target, we cannot use a hashmap. We do not know which value to look up! Instead, we need to sort the array and use a binary search or the two pointers pattern, like in [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/). In a sorted array, it is easy to find elements that are close to a given value.

#### Approach 1: Brute Force

It is important to understand the input constraints to choose the most appropriate approach. For this problem, the size of our array is limited to 100. So, a brute force solution could be a reasonable option. It's simple and does not require any additional memory.

**Algorithm**

1. For each index i in nums:
   * For each index j > i in nums:
     + If nums[i] + nums[j] is less than k:
       - Track maximum nums[i] + nums[j] in the result answer.
2. Return the result answer.

|  |
| --- |
| class Solution {  public int twoSumLessThanK(int[] nums, int k) {  int answer = -1;  for (int i = 0; i < nums.length; i++) {  for (int j = i + 1; j < nums.length; j++) {  int sum = nums[i] + nums[j];  if (sum < k) {  answer = Math.max(answer, sum);  }  }  }  return answer;  }  } |

**Complexity Analysis**

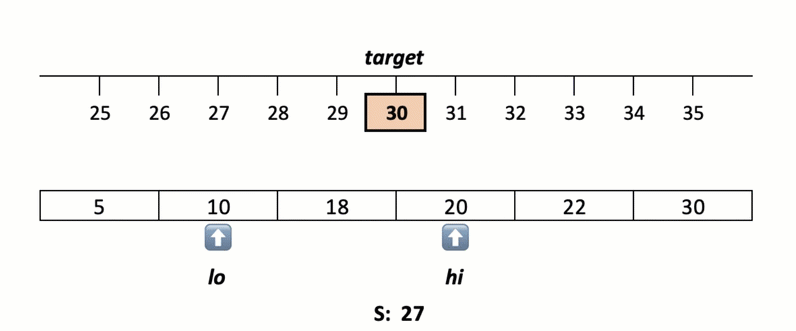
* Time Complexity: \mathcal{O}(n^2)O(*n*2). We have 2 nested loops.
* Space Complexity: \mathcal{O}(1)O(1).

#### Approach 2: Two Pointers

We will follow the same two pointers approach as in [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/). It requires the array to be sorted, so we'll do that first.

As a quick refresher, the pointers are initially set to the first and the last element respectively. We compare the sum of these two elements with the target. If it is smaller than the target, we increment the lower pointer left. Otherwise, we decrement the higher pointer right. Thus, the sum always moves toward the target, and we "prune" pairs that would move it further away. Again, this works only if the array is sorted. Head to the [Two Sum II](https://leetcode.com/articles/two-sum-ii-input-array-is-sorted/) solution for the detailed explanation.

Since the sum here must be smaller than the target, we don't stop when we find a pair that sums exactly to the target. We decrement the higher pointer and continue until our pointers collide. For each iteration, we track the maximum sum - if it's smaller than the target.



**Algorithm**

1. Sort the array.
2. Set the left pointer to zero, and right - to the last index.
3. While left is smaller than right:
   * If nums[left] + nums[right] is less than k:
     + Track maximum nums[left] + nums[right] in the result answer.
     + Increment left.
   * Else:
     + Decrement right.
4. Return the result answer.

|  |
| --- |
| class Solution {  public int twoSumLessThanK(int[] nums, int k) {  Arrays.sort(nums);  int answer = -1;  int left = 0;  int right = nums.length - 1;  while (left < right) {  int sum = nums[left] + nums[right];  if (sum < k) {  answer = Math.max(answer, sum);  left++;  } else {  right--;  }  }  return answer;  }  } |

**Optimizations**

We can break from the loop as soon as nums[left] > k / 2. In the sorted array, nums[left] is the smallest of the remaining elements, so nums[right] > k / 2 for any right. Therefore, nums[left] + nums[right] will be equal or greater than k for the remaining elements.

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n\log{n})O(*n*log*n*) to sort the array. The two pointers approach itself is \mathcal{O}(n)O(*n*), so the time complexity would be linear if the input is sorted.
* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm.

#### Approach 3: Binary Search

Instead of moving two pointers towards the target, we can iterate through each element nums[i], and binary-search for a complement value k - nums[i]. This approach is less efficient than the two pointers one, however, it can be more intuitive to come up with. Same as above, we need to sort the array first for this to work.

Note that the binary search returns the "insertion point" for the searched value, i.e. the position where that value would be inserted to keep the array sorted. So, the binary search result points to the first element that is equal or greater than the complement value. Since our sum must be smaller than k, we consider the element immediately before the found element.

**Algorithm**

1. Sort the array.
2. For each index i in nums:
   * Binary search for k - nums[i] starting from i + 1.
   * Set j to the position before the found element.
   * If j is less than i:
     + Track maximum nums[i] + nums[j] in the result answer.
3. Return the result answer.

Note that the binary search function in Java works a bit differently. If there are multiple elements that match the search value, it does not guarantee to point to the first one. That's why in the Java solution below we search for k - nums[i] - 1. Note that we decrement the pointer only if the value we found is greater than k - nums[i] - 1.

|  |
| --- |
| class Solution {  public int twoSumLessThanK(int[] nums, int k) {  int answer = -1;  Arrays.sort(nums);  for (int i = 0; i < nums.length; ++i) {  int idx = Arrays.binarySearch(nums, i + 1, nums.length, k - nums[i] - 1);  int j = (idx >= 0 ? idx : ~idx);  if (j == nums.length || nums[j] > k - nums[i] - 1) {  j--;  }  if (j > i) {  answer = Math.max(answer, nums[i] + nums[j]);  }  }  return answer;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n\log{n})O(*n*log*n*) to sort the array and do the binary search for each element.
* Space Complexity: from \mathcal{O}(\log{n})O(log*n*) to \mathcal{O}(n)O(*n*), depending on the implementation of the sorting algorithm.

#### Approach 4: Counting Sort

We can leverage the fact that the input number range is limited to [1..1000] and use a counting sort. Then, we can use the two pointers pattern to enumerate pairs in the [1..1000] range.

Note that the result can be a sum of two identical numbers, and that means that lo can be equal to hi. In this case, we need to check if the count for that number is greater than one.

**Algorithm**

1. Count each element using the array count.
2. Set the lo number to zero, and hi - to 1000.
3. While lo is smaller than, or **equals** hi:
   * If lo + hi is greater than k, or count[hi] == 0:
     + Decrement hi.
   * Else:
     + If count[lo] is greater than 0 (when lo < hi), or 1 (when lo == hi):
       - Track maximum lo + hi in the result answer.
     + Increment lo.
4. Return the result answer.

|  |
| --- |
| class Solution {  public int twoSumLessThanK(int[] nums, int k) {  int answer = -1;  int[] count = new int[1001];  for (int num : nums) {  count[num]++;  }  int lo = 1;  int hi = 1000;  while (lo <= hi) {  if (lo + hi >= k || count[hi] == 0) {  hi--;  } else {  if (count[lo] > (lo < hi ? 0 : 1)) {  answer = Math.max(answer, lo + hi);  }  lo++;  }  }  return answer;  }  } |

**Optimizations**

1. We can set hi to either maximum number, or k - 1, whichever is smaller.
2. We can ignore numbers greater than k - 1.
3. We can use a boolean array (e.g. seen) instead of count. In the first loop, we will check if i is a duplicate (seen[i] is already true) and set answer to the highest i + i < k. Note that the two pointers loop will run while lo < hi, not while lo <= hi.
4. We can break from the two pointers loop as soon as nums[lo] > k / 2.

**Complexity Analysis**

* Time Complexity: \mathcal{O}(n + m)O(*n*+*m*), where m*m* corresponds to the range of values in the input array.
* Space Complexity: \mathcal{O}(m)O(*m*) to count each value.

#### Further Thoughts

Always clarify the problem constraints and inputs during an interview. This would help you choose the right approach.

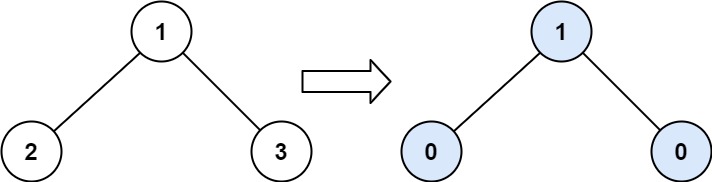
The Two Pointers approach is a good choice when the number of elements is large, and the range of possible values is not constrained. Also, if the input array is already sorted, this approach provides a linear time complexity and does not require additional memory.

**Binary Tree Tilt**

Given the root of a binary tree, return *the sum of every tree node's****tilt****.*

The **tilt** of a tree node is the **absolute difference** between the sum of all left subtree node **values** and all right subtree node **values**. If a node does not have a left child, then the sum of the left subtree node **values** is treated as 0. The rule is similar if there the node does not have a right child.

**Example 1:**



**Input:** root = [1,2,3]

**Output:** 1

**Explanation:**

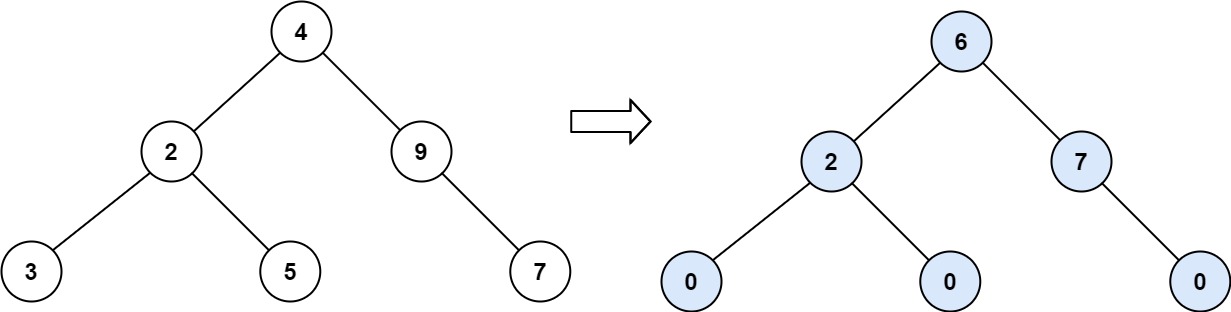
Tilt of node 2 : |0-0| = 0 (no children)

Tilt of node 3 : |0-0| = 0 (no children)

Tilt of node 1 : |2-3| = 1 (left subtree is just left child, so sum is 2; right subtree is just right child, so sum is 3)

Sum of every tilt : 0 + 0 + 1 = 1

**Example 2:**



**Input:** root = [4,2,9,3,5,null,7]

**Output:** 15

**Explanation:**

Tilt of node 3 : |0-0| = 0 (no children)

Tilt of node 5 : |0-0| = 0 (no children)

Tilt of node 7 : |0-0| = 0 (no children)

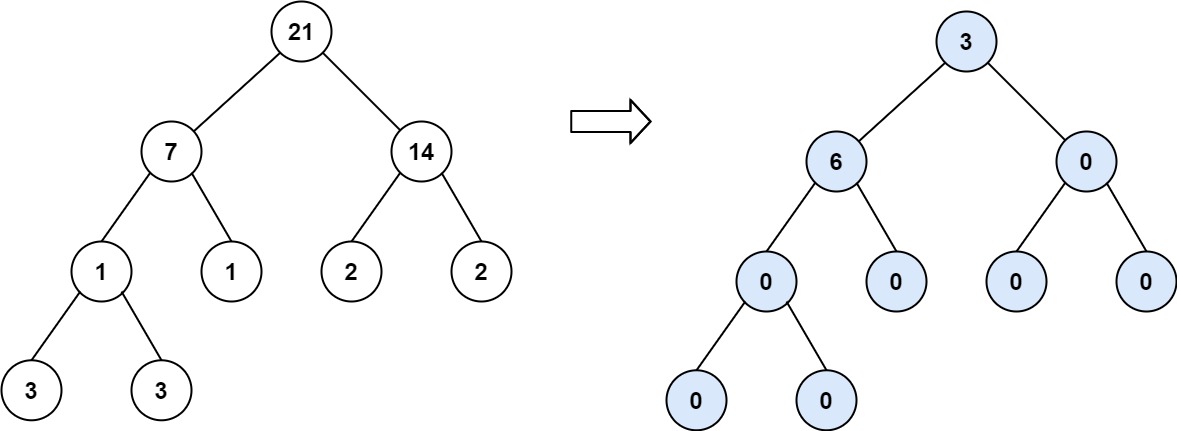
Tilt of node 2 : |3-5| = 2 (left subtree is just left child, so sum is 3; right subtree is just right child, so sum is 5)

Tilt of node 9 : |0-7| = 7 (no left child, so sum is 0; right subtree is just right child, so sum is 7)

Tilt of node 4 : |(3+5+2)-(9+7)| = |10-16| = 6 (left subtree values are 3, 5, and 2, which sums to 10; right subtree values are 9 and 7, which sums to 16)

Sum of every tilt : 0 + 0 + 0 + 2 + 7 + 6 = 15

**Example 3:**



**Input:** root = [21,7,14,1,1,2,2,3,3]

**Output:** 9

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -1000 <= Node.val <= 1000

Hide Hint #1

Don't think too much, this is an easy problem. Take some small tree as an example.

   Hide Hint #2

Can a parent node use the values of its child nodes? How will you implement it?

   Hide Hint #3

May be recursion and tree traversal can help you in implementing.

   Hide Hint #4

What about postorder traversal, using values of left and right childs?

## Solution

#### Overview

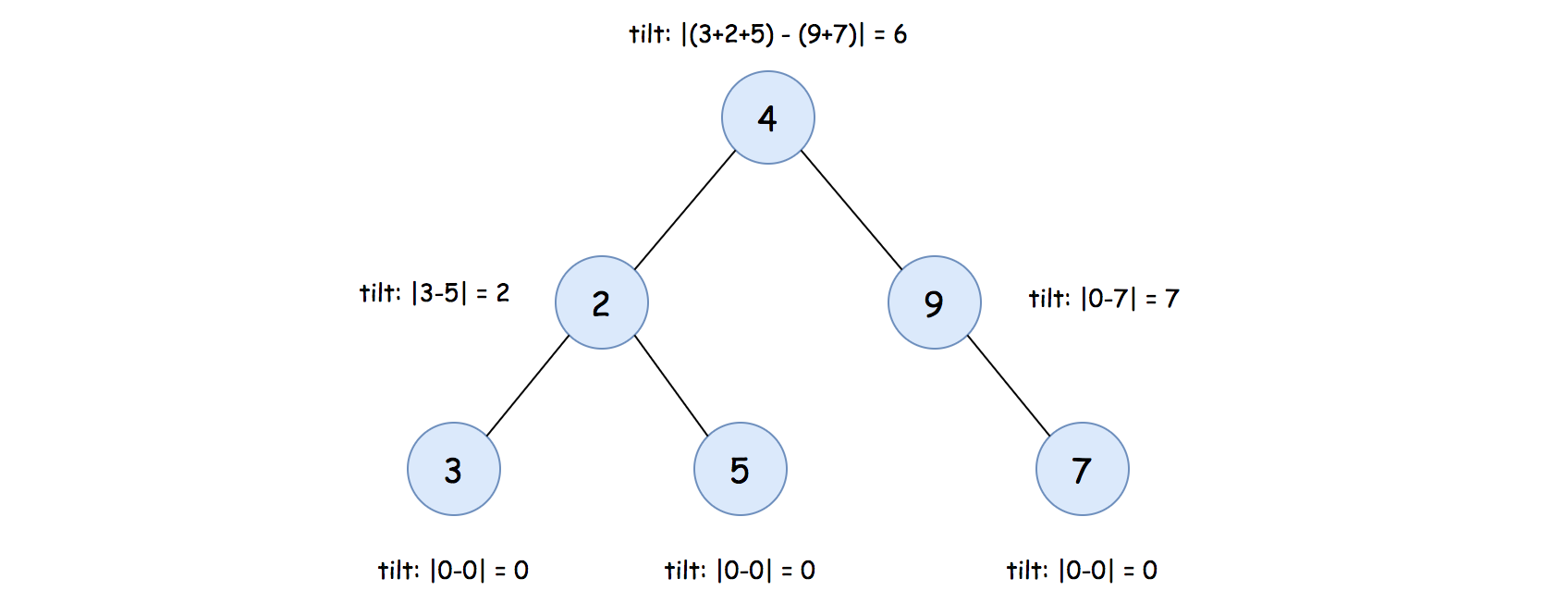
First of all, let us clarify the concept of **tilt** for a given node in a tree.

In order to calculate the tilt value for a node, we need to know the sum of nodes in its left and right subtrees respectively.

Assume that we have a function valueSum(node) which gives the sum of all nodes, starting from the input node, then the sum of the node's left subtree would be valueSum(node.left). Similarly, the sum of its right subtree would be valueSum(node.right).

With the above functions, we can then define the tilt value of a node as follows: \text{tilt(node)} = |\text{valueSum(node.left)} - \text{valueSum(node.right)}|tilt(node)=∣valueSum(node.left)−valueSum(node.right)∣

Given the above formula, we show an example on how the tilt value of each node looks like, in the following graph:



Note: when a subtree is empty, its value sum is zero. As a result, the tilt value for a leaf node would be zero, since both the left and right subtree of a leaf node are empty.

#### Approach 1: Post-Order DFS Traversal

**Intuition**

The overall idea is that we traverse each node, and calculate the tilt value for each node. At the end, we sum up all the tilt values, which is the desired result of the problem.

There are in general two strategies to traverse a tree data structure, namely [Breadth-First Search](https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/) (**BFS**) and [Depth-First Search](https://leetcode.com/explore/learn/card/queue-stack/232/practical-application-stack/) (**DFS**).

Concerning the DFS strategy, it can further be divided into three categories: Pre-Order, In-Order and Post-Order, depending on the relative order of visit among the node and its children nodes.

Sometimes, both strategies could work for a specific problem. In other cases, one of them might be more adapted to the problem. In our case here, the DFS is a more optimized choice, as one will see later. More specifically, we could apply the **Post-Order DFS** traversal here.

**Algorithm**

As we discussed before, in order to calculate the tilt value for a node, we need to calculate the sum of its left and right subtrees respectively.

Let us first implement the function valueSum(node) which returns the sum of values for all nodes starting from the given node, which can be summarized with the following recursive formula:

\text{valueSum(node)} = \text{node.val} + \text{valueSum(node.left)} + \text{valueSum(node.right)}valueSum(node)=node.val+valueSum(node.left)+valueSum(node.right)

Furthermore, the tilt value of a node also depends on the value sum of its left and right subtrees, as follows:

\text{tilt(node)} = |\text{valueSum(node.left)} - \text{valueSum(node.right)}|tilt(node)=∣valueSum(node.left)−valueSum(node.right)∣

Intuitively, we could combine the above calculations within a single recursive function. In this way, we only need to traverse each node once and only once.

More specifically, we will traverse the tree in the **post-order DFS**, i.e. we visit a node's left and right subtrees before processing the value of the current node.

Here are some sample implementations.

|  |
| --- |
| class Solution {  private int totalTilt = 0;  protected int valueSum(TreeNode node) {  if (node == null)  return 0;  int leftSum = this.valueSum(node.left);  int rightSum = this.valueSum(node.right);  int tilt = Math.abs(leftSum - rightSum);  this.totalTilt += tilt;  // return the sum of values starting from this node.  return node.val + leftSum + rightSum;  }  public int findTilt(TreeNode root) {  this.totalTilt = 0;  this.valueSum(root);  return this.totalTilt;  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the input tree.

* Time Complexity: \mathcal{O}(N)O(*N*)
  + We traverse each node once and only once. During the traversal, we calculate the tilt value for each node.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + Although the variables that we used in the algorithm are of constant-size, we applied recursion in the algorithm which incurs additional memory consumption in function call stack.
  + In the worst case where the tree is not well balanced, the recursion could pile up N*N* times. As a result, the space complexity of the algorithm is \mathcal{O}(N)O(*N*).

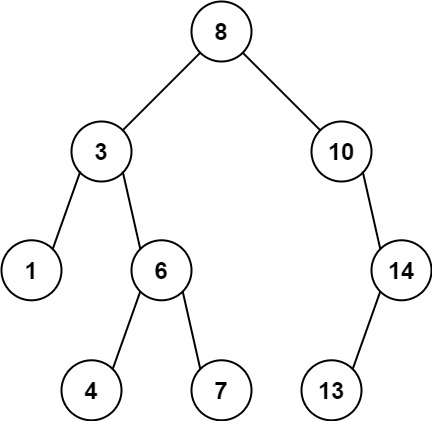
**Maximum Difference Between Node and Ancestor**

**Solution**

Given the root of a binary tree, find the maximum value V for which there exist **different** nodes A and B where V = |A.val - B.val| and A is an ancestor of B.

A node A is an ancestor of B if either: any child of A is equal to B, or any child of A is an ancestor of B.

**Example 1:**



**Input:** root = [8,3,10,1,6,null,14,null,null,4,7,13]

**Output:** 7

**Explanation:** We have various ancestor-node differences, some of which are given below :

|8 - 3| = 5

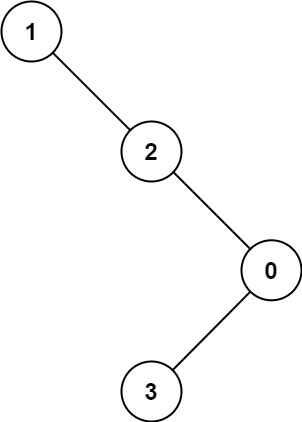
|3 - 7| = 4

|8 - 1| = 7

|10 - 13| = 3

Among all possible differences, the maximum value of 7 is obtained by |8 - 1| = 7.

**Example 2:**



**Input:** root = [1,null,2,null,0,3]

**Output:** 3

**Constraints:**

* The number of nodes in the tree is in the range [2, 5000].
* 0 <= Node.val <= 105

 Hide Hint #1

For each subtree, find the minimum value and maximum value of its descendants.

#### Overview

The problem is one of the typical tree problems that can be implemented using recursion. We could simply perform recursive tree traversal. Also, with a little insight, we can implement a slightly different approach.

Below, we will discuss two similar approaches: Recursion and Maximum Minus Minimum. Both are recursive approaches, but the later one uses a little insight.

#### Approach #1: Recursion

**Intuition**

Let's start from **Brute Force**.

A typical Brute Force approach is to compare every node with its ancestors. In the worst case, we have to compare every node-pair (when the tree is a single line).

The time complexity would be \mathcal{O}(N^2)O(*N*2), given N*N* is the number of nodes in the binary tree.

Can we simplify it?

Since the problem asks us the **Maximum Difference**, maybe we do not need to compare all ancestor for a given node and we only need to compare the ancestors with **Maximum** value and **Minimum** value.

Therefore, for a given node, we only need the maximum value and the minimum value from the root to this node.

To achieve this, we can define a function helper to start recursion, which receives a node and two integers, the maximum and minimum value of its ancestors, as input.

In the function helper, we need to update the maximum difference, the current maximum value, and the current minimum value.

**Algorithm**

Step 1: Initialize a variable result to record the required maximum difference.

Step 2: Define a function helper, which takes three arguments as input.

* The first argument is the current node, and the second and third arguments are the maximum and minimum values along the root to the current node, respectively.
* In the function helper, update result and call helper on the left and right subtrees.

Step 3: Run helper on the root. It will automatically do recursion on every node.

Step 4: Finally, return result.

**Implementation**

|  |
| --- |
| class Solution {  // record the required maximum difference  int result = 0;  public int maxAncestorDiff(TreeNode root) {  if (root == null) {  return 0;  }  result = 0;  helper(root, root.val, root.val);  return result;  }  void helper(TreeNode node, int curMax, int curMin) {  if (node == null) {  return;  }  // update `result`  int possibleResult = Math.max(Math.abs(curMax - node.val), Math.abs(curMin - node.val));  result = Math.max(result, possibleResult);  // update the max and min  curMax = Math.max(curMax, node.val);  curMin = Math.min(curMin, node.val);  helper(node.left, curMax, curMin);  helper(node.right, curMax, curMin);  return;  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the binary tree.

* Time complexity: \mathcal{O}(N)O(*N*) since we visit all nodes once.
* Space complexity: \mathcal{O}(N)O(*N*) since we need stacks to do recursion, and the maximum depth of the recursion is the height of the tree, which is \mathcal{O}(N)O(*N*) in the worst case and \mathcal{O}(\log(N))O(log(*N*)) in the best case.

#### Approach #2: Maximum Minus Minimum

**Intuition**

An insight is that:

* Given any two nodes on the same root-to-leaf path, they must have the required **ancestor** relationship.

Therefore, we just need to record the maximum and minimum values of all root-to-leaf paths and return the maximum difference.

To achieve this, we can record the maximum and minimum values during the recursion and return the difference when encountering leaves.

**Algorithm**

Step 1: Define a function helper, which takes three arguments as input and returns an integer.

* The first argument node is the current node, and the second argument cur\_max and third argument cur\_min are the maximum and minimum values along the root to the current node, respectively.
* Function helper returns cur\_max - cur\_min when encountering leaves. Otherwise, it calls helper on the left and right subtrees and returns their maximum.

Step 2: Run helper on the root and return the result.

**Implementation**

|  |
| --- |
| class Solution {  public int maxAncestorDiff(TreeNode root) {  if (root == null) {  return 0;  }  return helper(root, root.val, root.val);  }  public int helper(TreeNode node, int curMax, int curMin) {  // if encounter leaves, return the max-min along the path  if (node == null) {  return curMax - curMin;  }  // else, update max and min  // and return the max of left and right subtrees  curMax = Math.max(curMax, node.val);  curMin = Math.min(curMin, node.val);  int left = helper(node.left, curMax, curMin);  int right = helper(node.right, curMax, curMin);  return Math.max(left, right);  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the binary tree.

* Time complexity: \mathcal{O}(N)O(*N*) since we visit all nodes once.
* Space complexity: \mathcal{O}(N)O(*N*) since we need stacks to do recursion, and the maximum depth of the recursion is the height of the tree, which is \mathcal{O}(N)O(*N*) in the worst case and \mathcal{O}(\log(N))O(log(*N*)) in the best case.

**Flipping an Image**

Given an n x n binary matrix image, flip the image **horizontally**, then invert it, and return *the resulting image*.

To flip an image horizontally means that each row of the image is reversed.

* For example, flipping [1,1,0] horizontally results in [0,1,1].

To invert an image means that each 0 is replaced by 1, and each 1 is replaced by 0.

* For example, inverting [0,1,1] results in [1,0,0].

**Example 1:**

**Input:** image = [[1,1,0],[1,0,1],[0,0,0]]

**Output:** [[1,0,0],[0,1,0],[1,1,1]]

**Explanation:** First reverse each row: [[0,1,1],[1,0,1],[0,0,0]].

Then, invert the image: [[1,0,0],[0,1,0],[1,1,1]]

**Example 2:**

**Input:** image = [[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]]

**Output:** [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]

**Explanation:** First reverse each row: [[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]].

Then invert the image: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]

**Constraints:**

* n == image.length
* n == image[i].length
* 1 <= n <= 20
* images[i][j] is either 0 or 1.

#### Approach #1: Direct [Accepted]

**Intuition and Algorithm**

We can do this in place. In each row, the ith value from the left is equal to the inverse of the ith value from the right.

We use (C+1) / 2 (with floor division) to iterate over all indexes i in the first half of the row, including the center.

|  |
| --- |
| class Solution {  public int[][] flipAndInvertImage(int[][] A) {  int C = A[0].length;  for (int[] row: A)  for (int i = 0; i < (C + 1) / 2; ++i) {  int tmp = row[i] ^ 1;  row[i] = row[C - 1 - i] ^ 1;  row[C - 1 - i] = tmp;  }  return A;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N is the total number of elements in A.
* Space Complexity: O(1)*O*(1) in *additional* space complexity.

**Poor Pigs**

There are buckets buckets of liquid, where **exactly one** of the buckets is poisonous. To figure out which one is poisonous, you feed some number of (poor) pigs the liquid to see whether they will die or not. Unfortunately, you only have minutesToTest minutes to determine which bucket is poisonous.

You can feed the pigs according to these steps:

1. Choose some live pigs to feed.
2. For each pig, choose which buckets to feed it. The pig will consume all the chosen buckets simultaneously and will take no time.
3. Wait for minutesToDie minutes. You may **not** feed any other pigs during this time.
4. After minutesToDie minutes have passed, any pigs that have been fed the poisonous bucket will die, and all others will survive.
5. Repeat this process until you run out of time.

Given buckets, minutesToDie, and minutesToTest, return *the****minimum****number of pigs needed to figure out which bucket is poisonous within the allotted time*.

**Example 1:**

**Input:** buckets = 1000, minutesToDie = 15, minutesToTest = 60

**Output:** 5

**Example 2:**

**Input:** buckets = 4, minutesToDie = 15, minutesToTest = 15

**Output:** 2

**Example 3:**

**Input:** buckets = 4, minutesToDie = 15, minutesToTest = 30

**Output:** 2

**Constraints:**

* 1 <= buckets <= 1000
* 1 <= minutesToDie <= minutesToTest <= 100

Hide Hint #1

What if you only have one shot? Eg. 4 buckets, 15 mins to die, and 15 mins to test.

   Hide Hint #2

How many states can we generate with x pigs and T tests?

   Hide Hint #3

Find minimum x such that (T+1)^x >= N

## Solution

#### Approach 1: Pig as a [qubit](https://en.wikipedia.org/wiki/Qubit)

**Intuition**

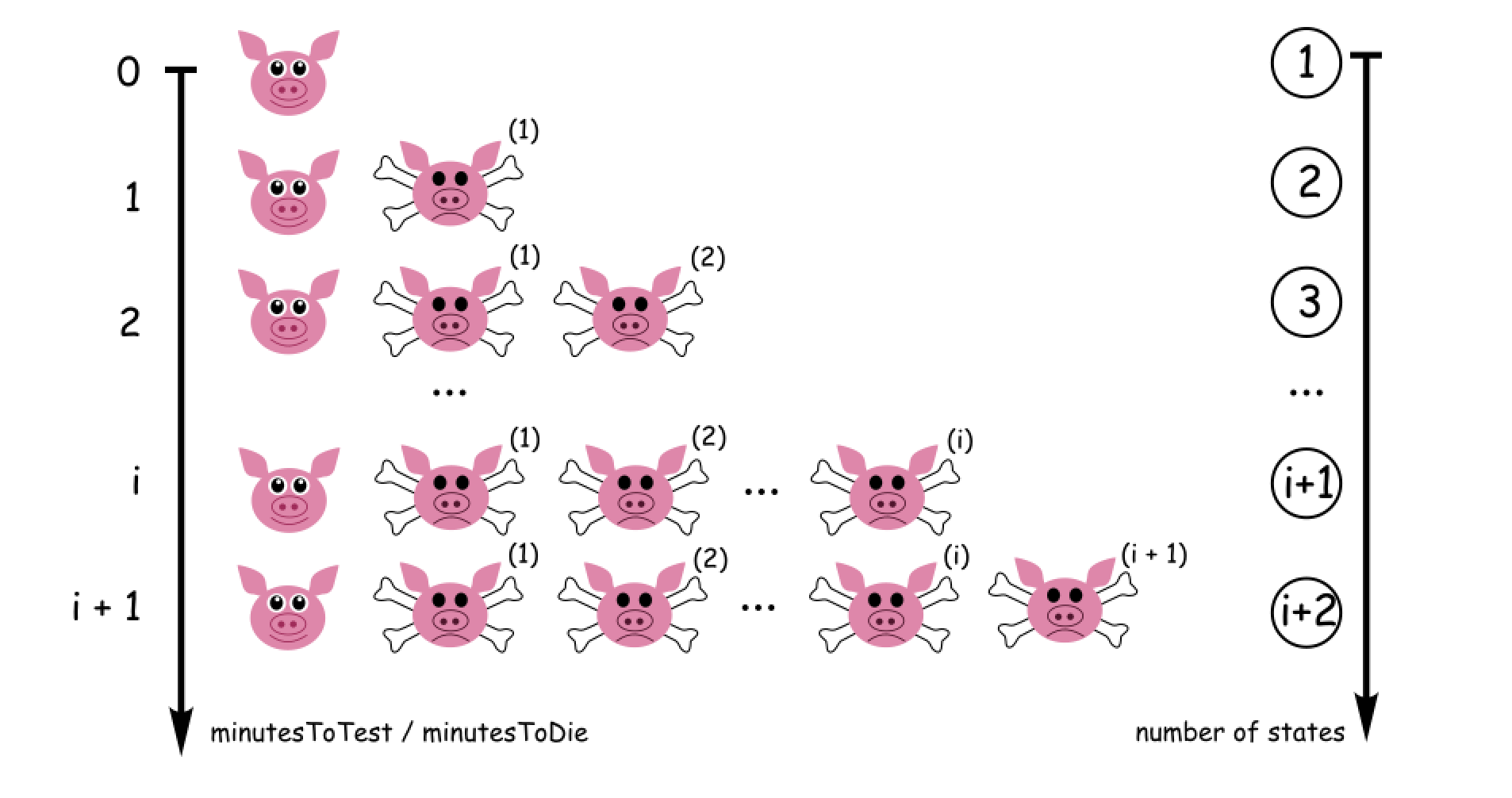
These "strange" questions are now asked by Google, Baidu and IBM because of their interest in quantum computing. [Quantum bit](https://en.wikipedia.org/wiki/Qubit) (or qubit) is the basic unit of quantum information, it's the quantum version of the classical binary bit. Binary bit has only two states : 0 and 1, and on a very basic level the qubit has more. In such questions we deal with an object (here is a pig) which has more than two states.

**How many states does a pig have**

If there is no time to test, i.e. minutesToTest / minutesToDie = 0, the pig has only one state - alive.

If minutesToTest / minutesToDie = 1 then the pig has a time to die from the poison, that means that now there are two states available for the pig : alive or dead.

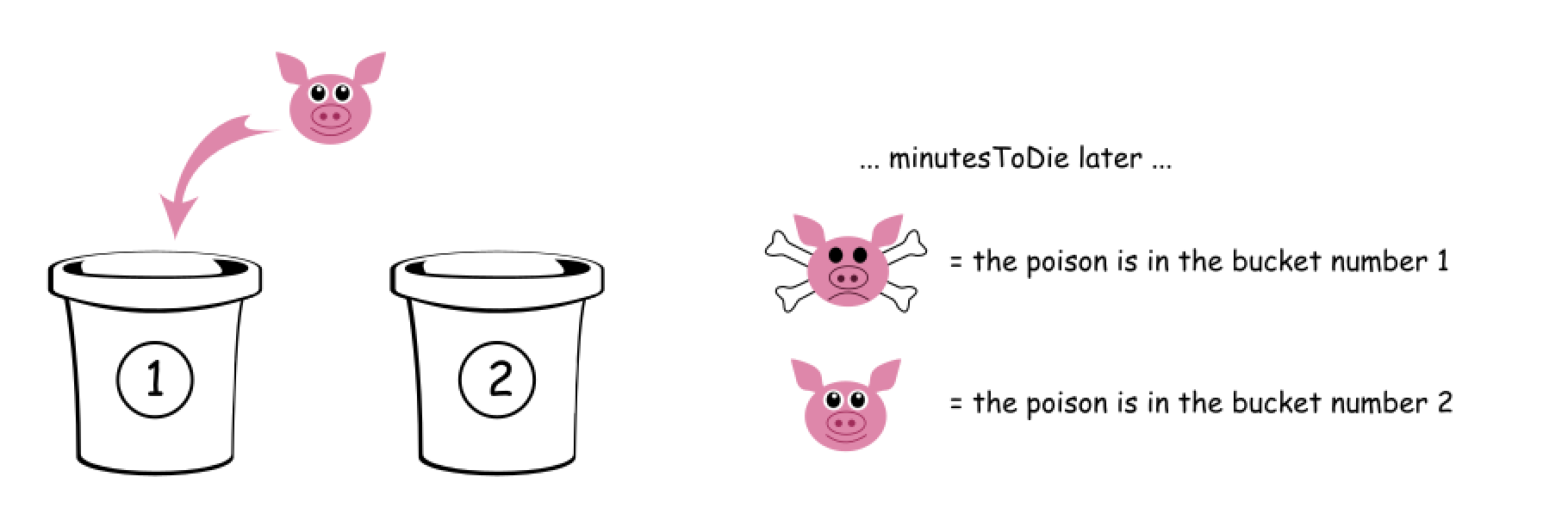
One more step. If minutesToTest / minutesToDie = 2 then there are three available states for the pig : alive / dead after the first test / dead after the second test.



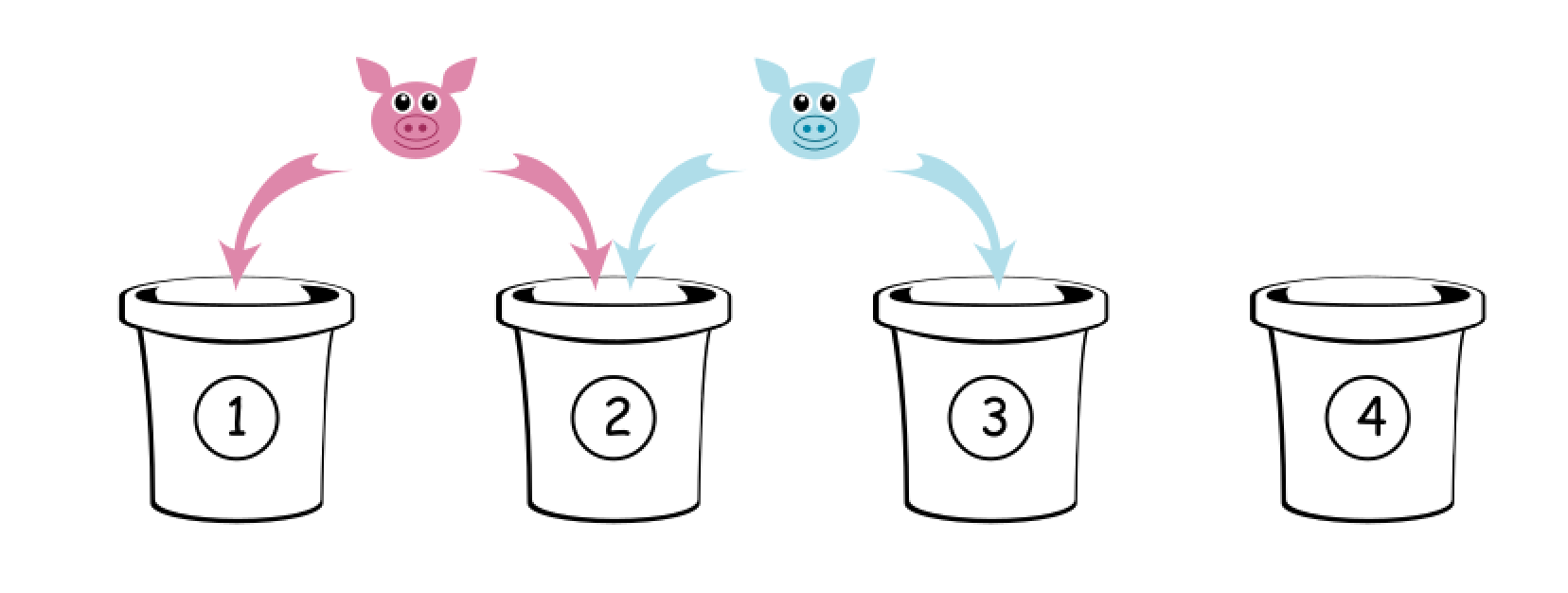
The number of available states for the pig is states = minutesToTest / minutesToDie + 1.

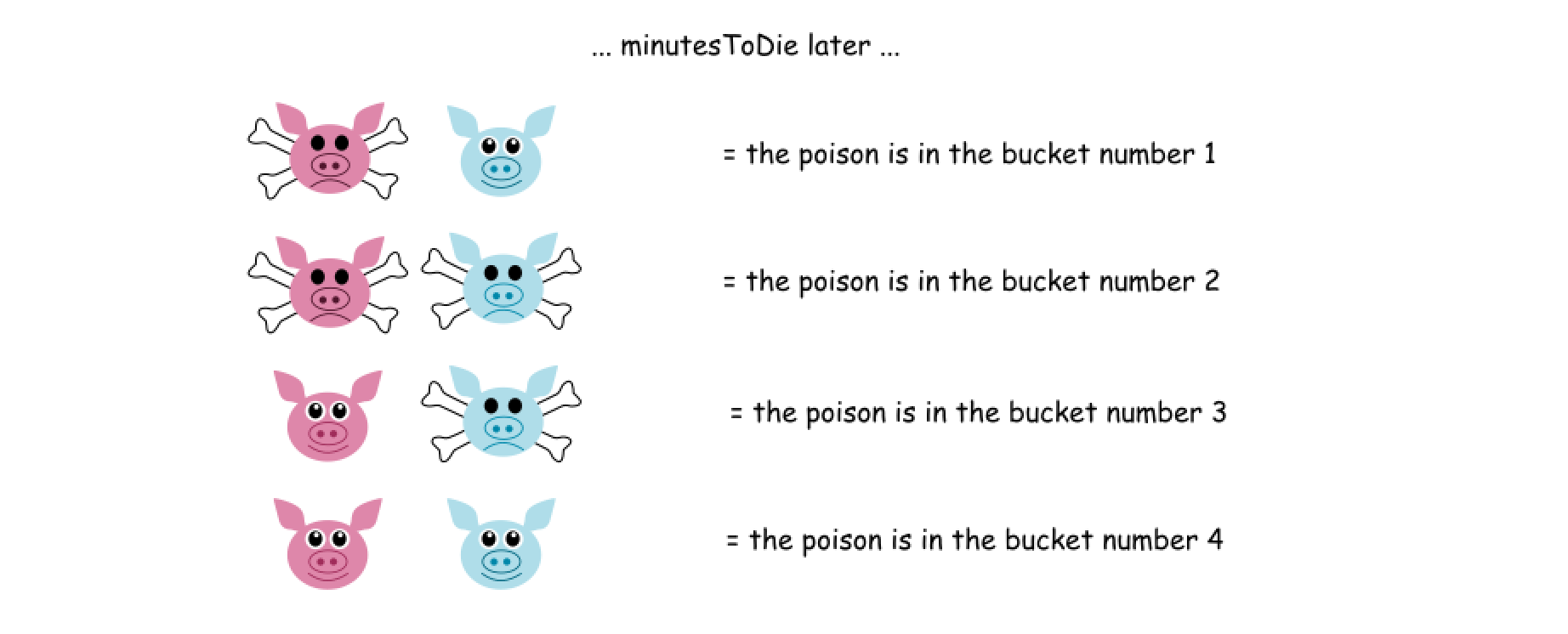
**How many buckets could test x pigs with 2 available states**

One pig could test 2 buckets - let's make him drink from the bucket number 1 and then wait minutesToDie time. If he is alive - the poison is in the bucket number 2. If he is dead - the poison is in the bucket number 1.



The same way two pigs could test 2^2 = 422=4 buckets



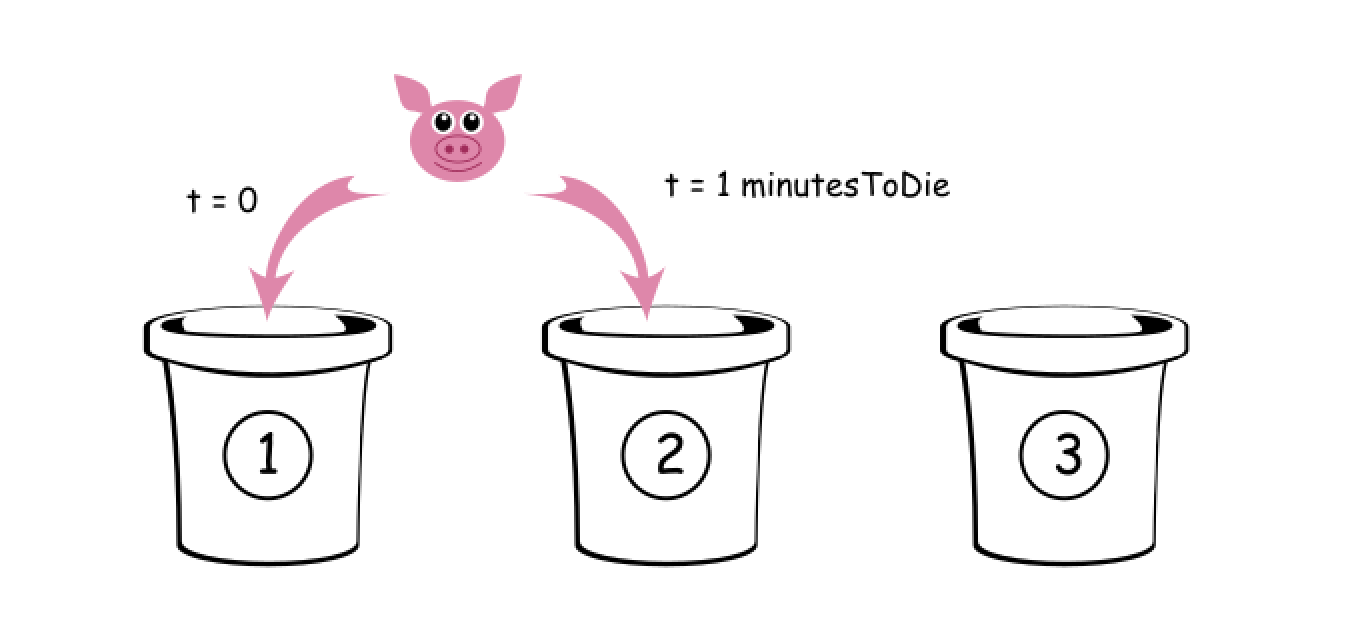


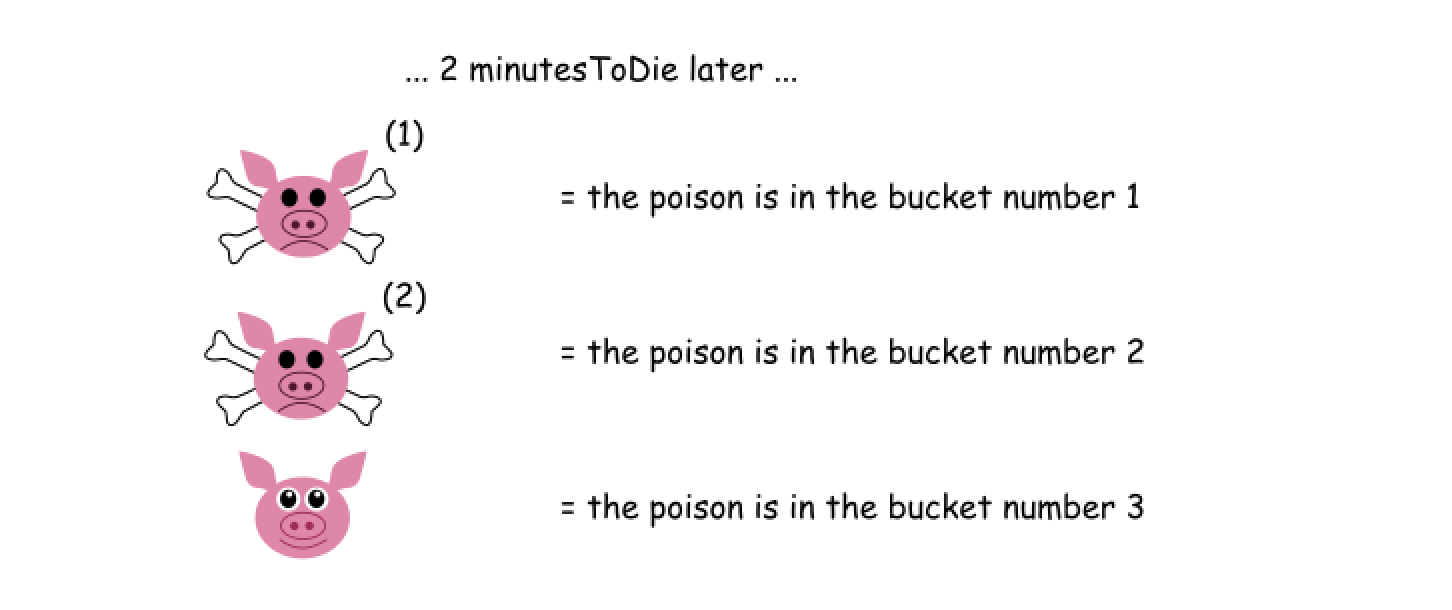
Hence if one pig has two available states, x pigs could test 2^x2*x* buckets.

**How many buckets could test x pigs with s available states**

After the discussion above, the answer is quite obvious : s^x*sx* buckets.

Let's consider as an example one pig with 3 states, i.e. s = minutesToTest / minutesToDie + 1 = 2 + 1 = 3, and show that he could test 3 buckets.





**Solution**

Hence the problem is to find x such that \textrm{states}^x \ge \textrm{buckets}states*x*≥buckets, where x is a number of pigs, states = minutesToTest / minutesToDie + 1 is a number of states available for each pig, and \textrm{buckets}buckets is number of buckets.

[The solution is well known](https://en.wikipedia.org/wiki/Exponential_function) : x \ge \log\_{\textrm{states}}{\textrm{buckets}}*x*≥logstates​buckets. To simplify the code let's rewrite the equation with the help of [natural logarithms](https://en.wikipedia.org/wiki/List_of_logarithmic_identities#Changing_the_base) :

x \ge \frac{\log \textrm{buckets}}{\log \textrm{states}}*x*≥logstateslogbuckets​

**Implementation**

|  |
| --- |
| class Solution {  public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {  int states = minutesToTest / minutesToDie + 1;  return (int) Math.ceil(Math.log(buckets) / Math.log(states));  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(1)O(1) since it's a [constant time solution](https://stackoverflow.com/a/7317571/7775414).
* Space complexity : \mathcal{O}(1)O(1).

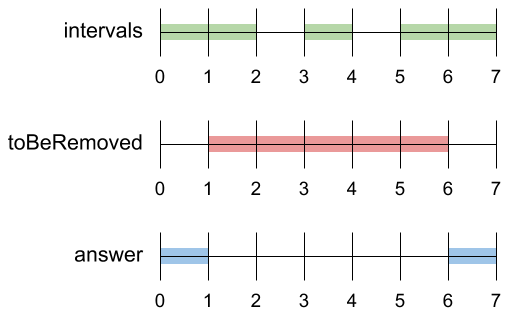
**Remove Interval**

A set of real numbers can be represented as the union of several disjoint intervals, where each interval is in the form [a, b). A real number x is in the set if one of its intervals [a, b) contains x (i.e. a <= x < b).

You are given a **sorted** list of disjoint intervals intervals representing a set of real numbers as described above, where intervals[i] = [ai, bi] represents the interval [ai, bi). You are also given another interval toBeRemoved.

Return *the set of real numbers with the interval*toBeRemoved***removed****from*intervals*. In other words, return the set of real numbers such that every*x*in the set is in*intervals*but****not****in*toBeRemoved*. Your answer should be a****sorted****list of disjoint intervals as described above.*

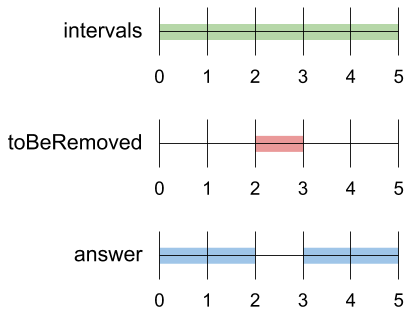
**Example 1:**



**Input:** intervals = [[0,2],[3,4],[5,7]], toBeRemoved = [1,6]

**Output:** [[0,1],[6,7]]

**Example 2:**



**Input:** intervals = [[0,5]], toBeRemoved = [2,3]

**Output:** [[0,2],[3,5]]

**Example 3:**

**Input:** intervals = [[-5,-4],[-3,-2],[1,2],[3,5],[8,9]], toBeRemoved = [-1,4]

**Output:** [[-5,-4],[-3,-2],[4,5],[8,9]]

**Constraints:**

* 1 <= intervals.length <= 104
* -109 <= ai < bi <= 109

Hide Hint #1

Solve the problem for every interval alone.

   Hide Hint #2

Divide the problem into cases according to the position of the two intervals.

## Solution Article

#### Approach 1: Sweep Line, One Pass.

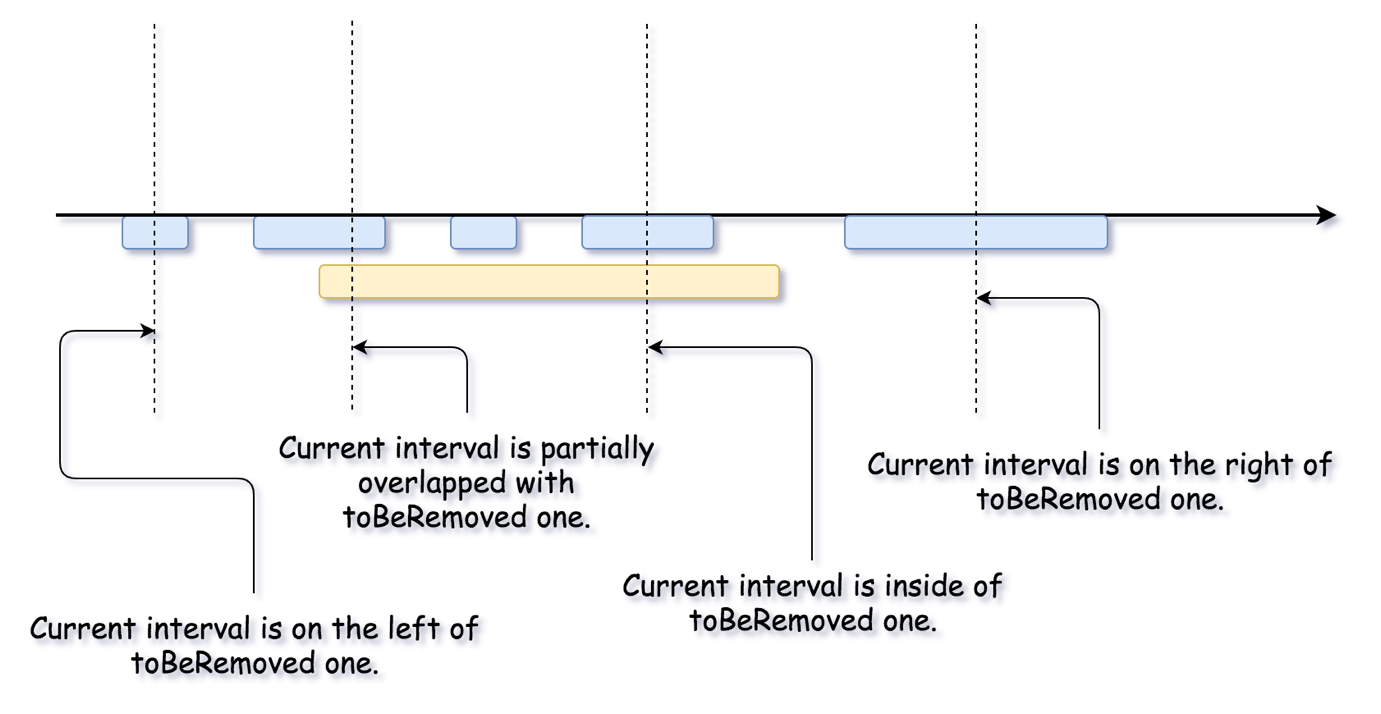
**Best Possible Time Complexity**

What is the best possible time complexity here?

The input is sorted, that usually means at least linear time complexity. Is it possible to do \mathcal{O}(\log N)O(log*N*)? No, because to copy input elements into output still requires \mathcal{O}(N)O(*N*) time.

**Sweep Line**

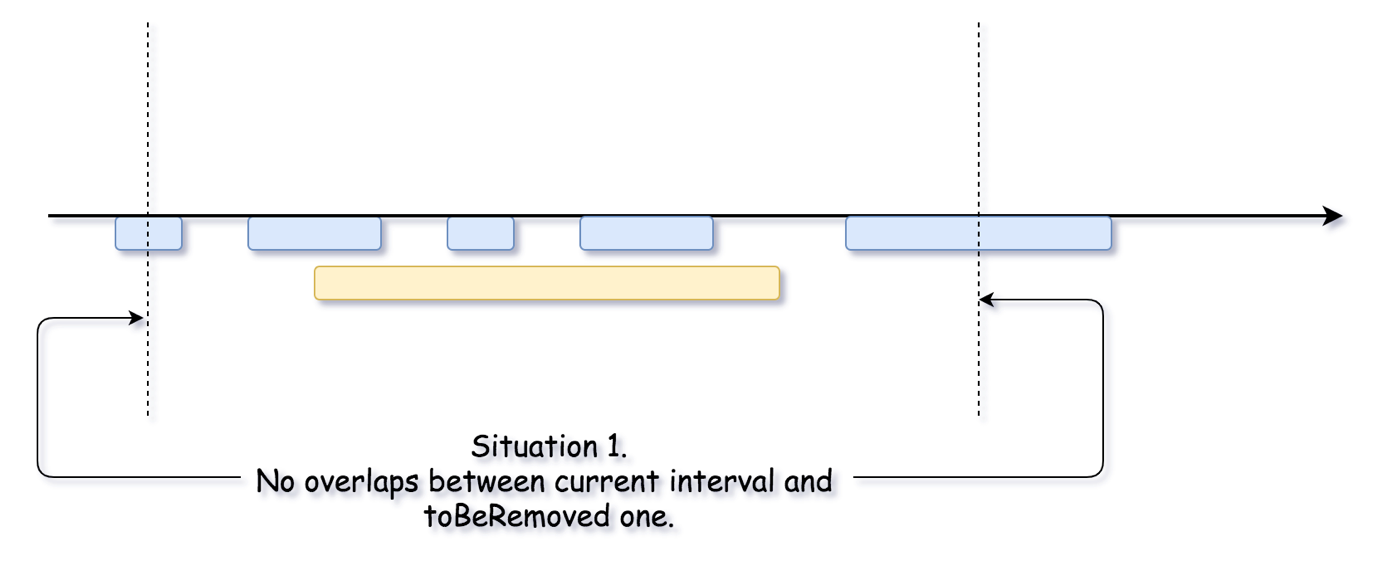
[Sweep Line algorithm](https://en.wikipedia.org/wiki/Sweep_line_algorithm) is a sort of geometrical visualisation. Let's imagine a vertical line which is swept across the plane, stopping at some points. That could create various situations, and the decision to make depends on the stop point.



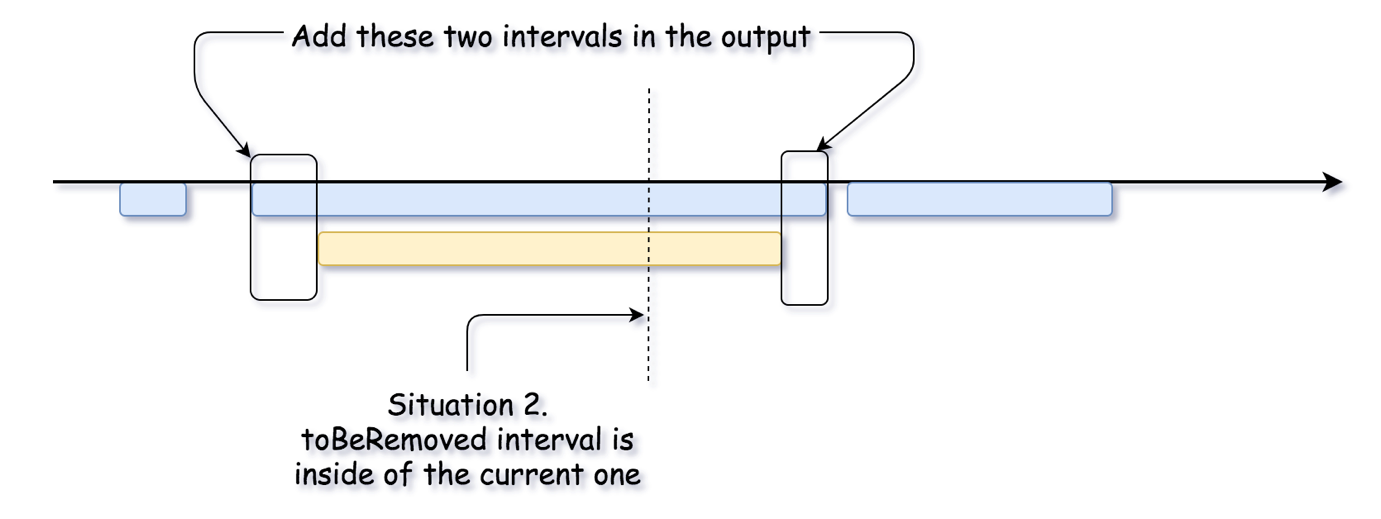
**Algorithm**

Let's sweep the line by iterating over input intervals and consider what it could bring to us.

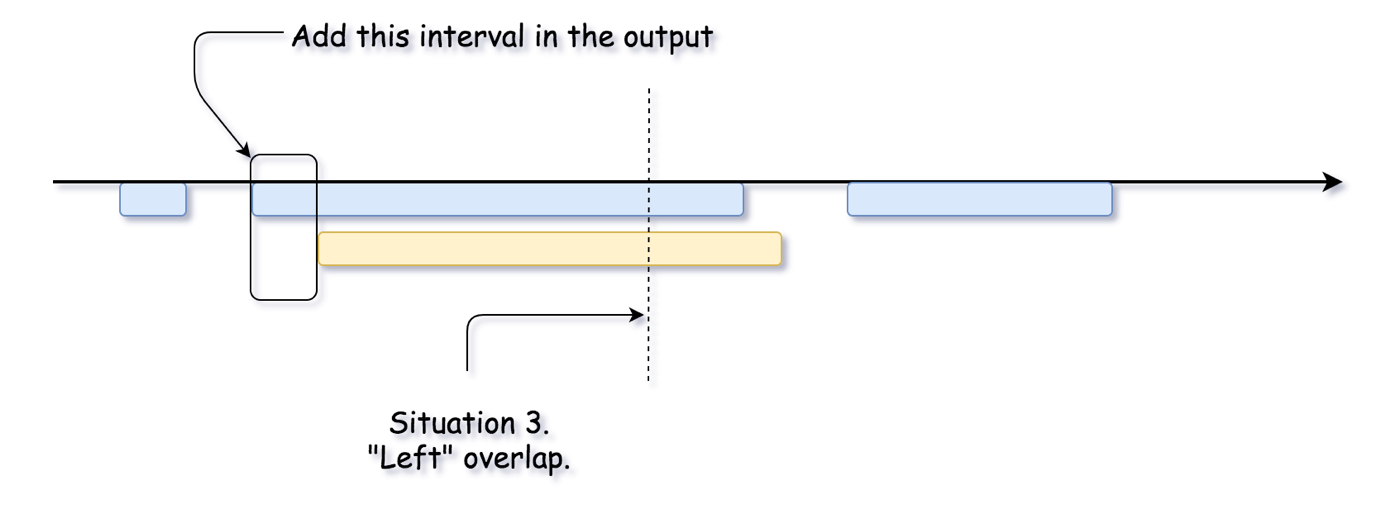
* Current interval has no overlaps with toBeRemoved one. That means there is nothing to take care about, just update the output.



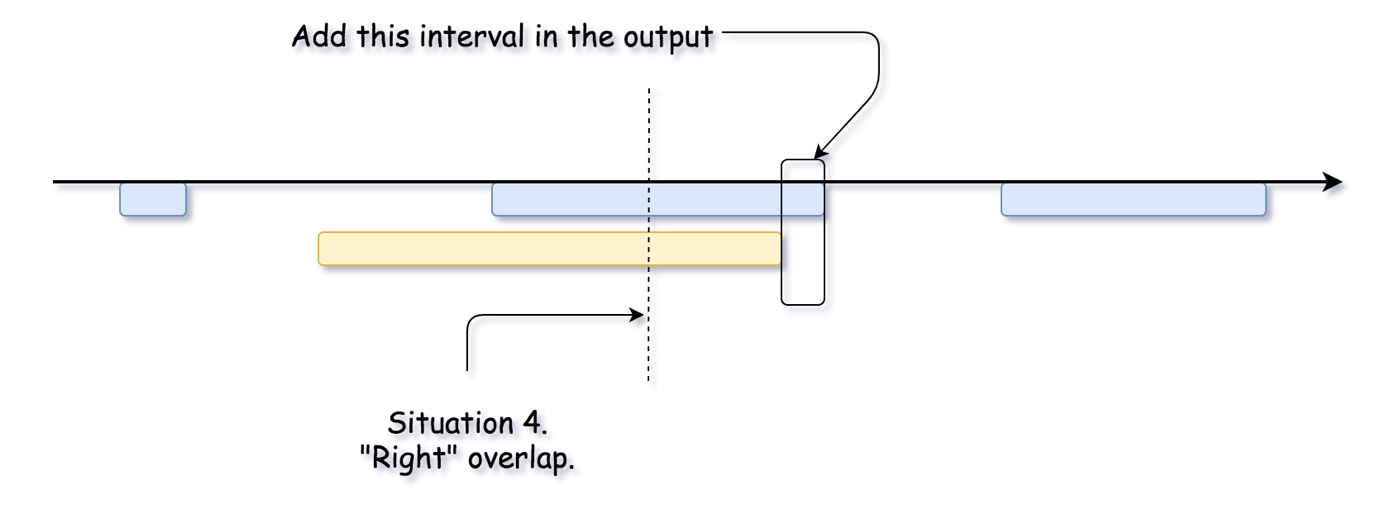
* Second situation is when toBeRemoved interval is inside of the current interval. Then one has to add two non-overlapping parts of the current interval in the output.



* "Left" overlap.



* "Right" overlap.



And here we are, all situations are covered, the job is done.

**Implementation**

One way of converting the above into code would be to check for each of the four situations described above. A better way though is to recognize that if there is any overlap, then the overlapped interval will be broken into up to two new intervals; a left interval and a right interval. We can, therefore, treat situation 2 as being both situation 3 and situation 4.

|  |
| --- |
| class Solution {  public List<List<Integer>> removeInterval(int[][] intervals, int[] toBeRemoved) {  List<List<Integer>> result = new ArrayList<>();  for (int[] interval : intervals) {  // If there are no overlaps, add the interval to the list as is.  if (interval[0] > toBeRemoved[1] || interval[1] < toBeRemoved[0]) {  result.add(Arrays.asList(interval[0], interval[1]));  } else {  // Is there a left interval we need to keep?  if (interval[0] < toBeRemoved[0]) {  result.add(Arrays.asList(interval[0], toBeRemoved[0]));  }  // Is there a right interval we need to keep?  if (interval[1] > toBeRemoved[1]) {  result.add(Arrays.asList(toBeRemoved[1], interval[1]));  }  }  }  return result;  }  } |

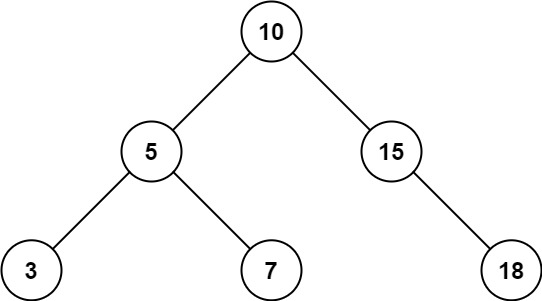
**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since it's one pass along the input array.
* Space complexity : \mathcal{O}(1)O(1) without considering \mathcal{O}(N)O(*N*) space for the output list.

**Range Sum of BST**

Given the root node of a binary search tree, return *the sum of values of all nodes with a value in the range [low, high]*.

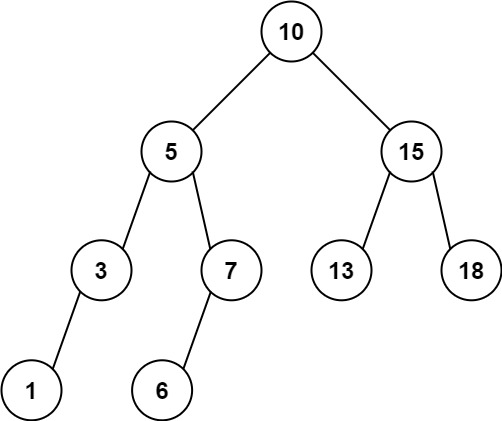
**Example 1:**



**Input:** root = [10,5,15,3,7,null,18], low = 7, high = 15

**Output:** 32

**Example 2:**



**Input:** root = [10,5,15,3,7,13,18,1,null,6], low = 6, high = 10

**Output:** 23

**Constraints:**

* The number of nodes in the tree is in the range [1, 2 \* 104].
* 1 <= Node.val <= 105
* 1 <= low <= high <= 105
* All Node.val are **unique**.

## Solution Article

#### Approach 1: Depth First Search

**Intuition and Algorithm**

We traverse the tree using a depth first search. If node.val falls outside the range [L, R], (for example node.val < L), then we know that only the right branch could have nodes with value inside [L, R].

We showcase two implementations - one using a recursive algorithm, and one using an iterative one.

**Recursive Implementation**

|  |
| --- |
| class Solution {  int ans;  public int rangeSumBST(TreeNode root, int L, int R) {  ans = 0;  dfs(root, L, R);  return ans;  }  public void dfs(TreeNode node, int L, int R) {  if (node != null) {  if (L <= node.val && node.val <= R)  ans += node.val;  if (L < node.val)  dfs(node.left, L, R);  if (node.val < R)  dfs(node.right, L, R);  }  }  } |

**Iterative Implementation**

|  |
| --- |
| class Solution {  public int rangeSumBST(TreeNode root, int L, int R) {  int ans = 0;  Stack<TreeNode> stack = new Stack();  stack.push(root);  while (!stack.isEmpty()) {  TreeNode node = stack.pop();  if (node != null) {  if (L <= node.val && node.val <= R)  ans += node.val;  if (L < node.val)  stack.push(node.left);  if (node.val < R)  stack.push(node.right);  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the tree.
* Space Complexity: O(N)*O*(*N*)
  + For the recursive implementation, the recursion will consume additional space in the function call stack. In the worst case, the tree is of chain shape, and we will reach all the way down to the leaf node.
  + For the iterative implementation, essentially we are doing a **BFS** (Breadth-First Search) traversal, where the stack will contain no more than two levels of the nodes. The maximal number of nodes in a binary tree is \frac{N}{2}2*N*​. Therefore, the maximal space needed for the stack would be O(N)*O*(*N*).

**Longest Mountain in Array**

You may recall that an array arr is a **mountain array** if and only if:

* arr.length >= 3
* There exists some index i (**0-indexed**) with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
  + arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

Given an integer array arr, return *the length of the longest subarray, which is a mountain*. Return 0 if there is no mountain subarray.

**Example 1:**

**Input:** arr = [2,1,4,7,3,2,5]

**Output:** 5

**Explanation:** The largest mountain is [1,4,7,3,2] which has length 5.

**Example 2:**

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

**Output:** 0

**Explanation:** There is no mountain.

**Constraints:**

* 1 <= arr.length <= 104
* 0 <= arr[i] <= 104

**Follow up:**

* Can you solve it using only one pass?
* Can you solve it in O(1) space?

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

**Intuition**

Without loss of generality, a mountain can only start after the previous one ends.

This is because if it starts before the peak, it will be smaller than a mountain starting previous; and it is impossible to start after the peak.

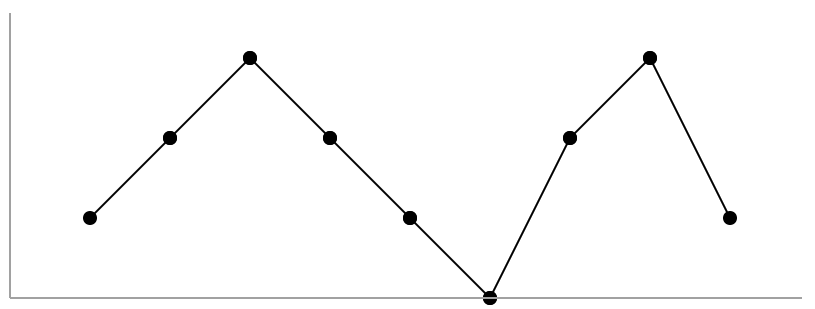
**Algorithm**

For a starting index base, let's calculate the length of the longest mountain A[base], A[base+1], ..., A[end].

If such a mountain existed, the next possible mountain will start at base = end; if it didn't, then either we reached the end, or we have A[base] >= A[base+1] and we can start at base + 1.

**Example**

Here is a worked example on the array A = [1, 2, 3, 2, 1, 0, 2, 3, 1]:



base starts at 0, and end travels using the first while loop to end = 2 (A[end] = 3), the potential peak of this mountain. After, it travels to end = 5 (A[end] = 0) during the second while loop, and a candidate answer of 6 (base = 0, end = 5) is recorded.

Afterwards, base is set to 5 and the process starts over again, with end = 7 the peak of the mountain, and end = 8 the right boundary, and the candidate answer of 4 (base = 5, end = 8) being recorded.

|  |
| --- |
| class Solution {  public int longestMountain(int[] A) {  int N = A.length;  int ans = 0, base = 0;  while (base < N) {  int end = base;  // if base is a left-boundary  if (end + 1 < N && A[end] < A[end + 1]) {  // set end to the peak of this potential mountain  while (end + 1 < N && A[end] < A[end + 1]) end++;  // if end is really a peak..  if (end + 1 < N && A[end] > A[end + 1]) {  // set end to the right-boundary of mountain  while (end + 1 < N && A[end] > A[end + 1]) end++;  // record candidate answer  ans = Math.max(ans, end - base + 1);  }  }  base = Math.max(end, base + 1);  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of A.
* Space Complexity: O(1)*O*(1).

**Mirror Reflection**

There is a special square room with mirrors on each of the four walls.  Except for the southwest corner, there are receptors on each of the remaining corners, numbered 0, 1, and 2.

The square room has walls of length p, and a laser ray from the southwest corner first meets the east wall at a distance q from the 0th receptor.

Return the number of the receptor that the ray meets first.  (It is guaranteed that the ray will meet a receptor eventually.)

**Example 1:**

**Input:** p = 2, q = 1

**Output:** 2

**Explanation:** The ray meets receptor 2 the first time it gets reflected back to the left wall.



**Note:**

1. 1 <= p <= 1000
2. 0 <= q <= p

## Solution

#### Approach 1: Simulation

**Intuition**

The initial ray can be described as going from an origin (x, y) = (0, 0) in the direction (rx, ry) = (p, q). From this, we can figure out which wall it will meet and where, and what the appropriate new ray will be (based on reflection.) We keep simulating the ray until it finds it's destination.

**Algorithm**

The parameterized position of the laser after time t will be (x + rx \* t, y + ry \* t). From there, we know when it will meet the east wall (if x + rx \* t == p), and so on. For a positive (and nonnegligible) time t, it meets the next wall.

We can then calculate how the ray reflects. If it hits an east or west wall, then rx \*= -1, else ry \*= -1.

In Java, care must be taken with floating point operations.

|  |
| --- |
| class Solution {  double EPS = 1e-6;  public int mirrorReflection(int p, int q) {  double x = 0, y = 0;  double rx = p, ry = q;  // While it hasn't reached a receptor,...  while (!( close(x, p) && (close(y, 0) || close(y, p))  || close(x, 0) && close (y, p) )) {  // Want smallest t so that some x + rx, y + ry is 0 or p  // x + rxt = 0, then t = -x/rx etc.  double t = 1e9;  if ((-x / rx) > EPS) t = Math.min(t, -x / rx);  if ((-y / ry) > EPS) t = Math.min(t, -y / ry);  if (((p-x) / rx) > EPS) t = Math.min(t, (p-x) / rx);  if (((p-y) / ry) > EPS) t = Math.min(t, (p-y) / ry);  x += rx \* t;  y += ry \* t;  if (close(x, p) || close(x, 0)) rx \*= -1;  if (close(y, p) || close(y, 0)) ry \*= -1;  }  if (close(x, p) && close(y, p)) return 1;  return close(x, p) ? 0 : 2;  }  public boolean close(double x, double y) {  return Math.abs(x - y) < EPS;  }  } |

**Complexity Analysis**

* Time Complexity: O(p)*O*(*p*). We can prove (using Approach #2) that the number of bounces is bounded by this.
* Space Complexity: O(1)*O*(1).

#### Approach 2: Mathematical

**Intuition and Algorithm**

Instead of modelling the ray as a bouncing line, model it as a straight line through reflections of the room.

For example, if p = 2, q = 1, then we can reflect the room horizontally, and draw a straight line from (0, 0) to (4, 2). The ray meets the receptor 2, which was reflected from (0, 2) to (4, 2).

In general, the ray goes to the first integer point (kp, kq) where k is an integer, and kp and kq are multiples of p. Thus, the goal is just to find the smallest k for which kq is a multiple of p.

The mathematical answer is k = p / gcd(p, q).

|  |
| --- |
| class Solution {  public int mirrorReflection(int p, int q) {  int g = gcd(p, q);  p /= g; p %= 2;  q /= g; q %= 2;  if (p == 1 && q == 1) return 1;  return p == 1 ? 0 : 2;  }  public int gcd(int a, int b) {  if (a == 0) return b;  return gcd(b % a, a);  }  } |

**Complexity Analysis**

* Time Complexity: O(\log P)*O*(log*P*), the complexity of the gcd operation.
* Space Complexity: O(1)*O*(1).

**Search in Rotated Sorted Array II**

There is an integer array nums sorted in non-decreasing order (not necessarily with **distinct** values).

Before being passed to your function, nums is **rotated** at an unknown pivot index k (0 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,4,4,5,6,6,7] might be rotated at pivot index 5 and become [4,5,6,6,7,0,1,2,4,4].

Given the array nums **after** the rotation and an integer target, return true*if*target*is in*nums*, or*false*if it is not in*nums*.*

**Example 1:**

**Input:** nums = [2,5,6,0,0,1,2], target = 0

**Output:** true

**Example 2:**

**Input:** nums = [2,5,6,0,0,1,2], target = 3

**Output:** false

**Constraints:**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* nums is guaranteed to be rotated at some pivot.
* -104 <= target <= 104

**Follow up:** This problem is the same as [Search in Rotated Sorted Array](https://leetcode.com/problems/search-in-rotated-sorted-array/description/), where nums may contain **duplicates**. Would this affect the runtime complexity? How and why?

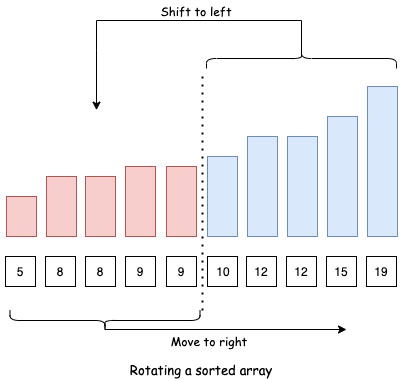
## Solution

#### Approach 1: Binary Search

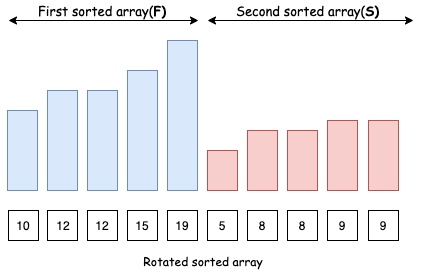
This problem is an extension to [33. Search in Rotated Sorted Array](https://leetcode.com/problems/search-in-rotated-sorted-array/). The only difference is that this problem allows duplicate elements.

**Intuition**

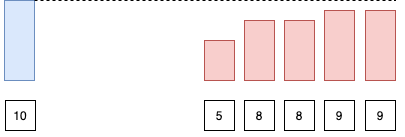
Recall that after rotating a sorted array, what we get is two sorted arrays appended to each other.



Let's refer to the first sorted array as F and second as S.



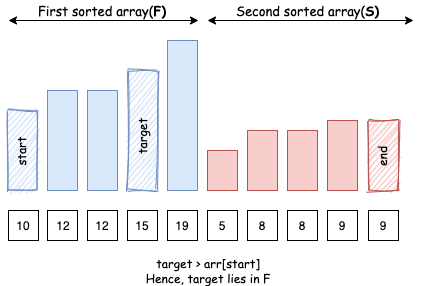
Also, we can observe that all the elements of the second array S will be smaller or equal to the first element start of F.



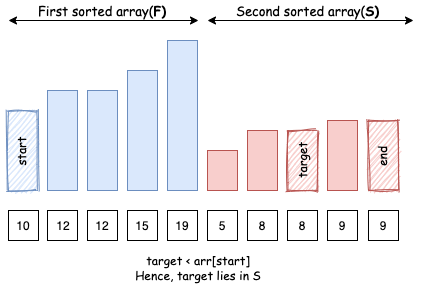
With this observation in mind, we can easily tell which of the 2 arrays (F or S) does a target element lie in by just comparing it with the first element of the array.

Let's say we are looking for element target in array arr:

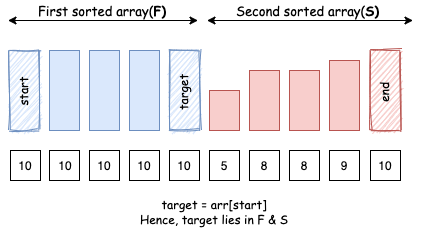
* Case 1: If target > arr[start]: target exists in the first array F.



* Case 2: If target < arr[start]: target exists in the second array S.



* Case 3: If target == arr[start]: target obviously exists in the first array F, but it might also be present in the second array S.



Let's define a helper function that tells us which array a target element might be present in:

|  |
| --- |
| // returns true if element `target` exists in the first sorted array.  private boolean existsInFirst(int[] arr, int start, int element) {  return arr[start] <= element;  } |

**Algorithm**

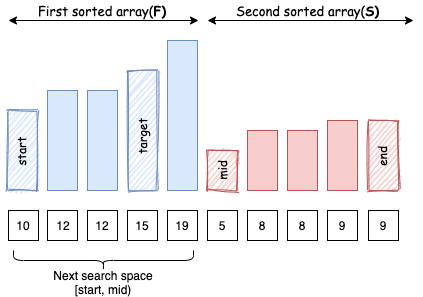
Recall that in standard binary search, we keep two pointers (i.e. start and end) to track the search scope in an arr array. We then divide the search space in three parts [start, mid), [mid, mid], (mid, end]. Now, we continue to look for our target element in one of these search spaces.

By identifying the positions of both arr[mid] and target in F and S, we can reduce search space in the very same way as in standard binary search:

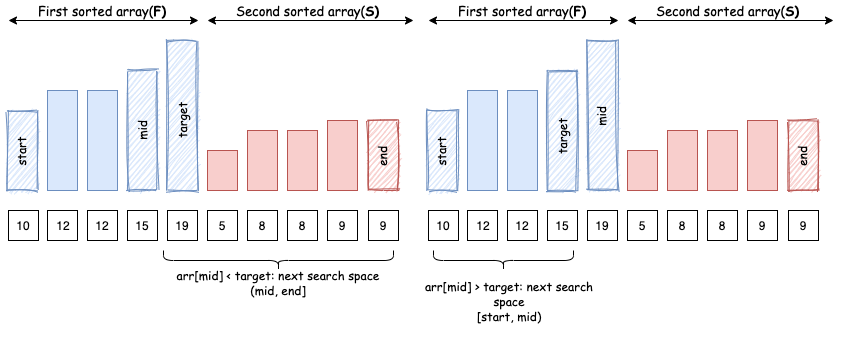
* Case 1: arr[mid] lies in F, target lies in S: Since S starts after F ends, we know that element lies here:(mid, end].



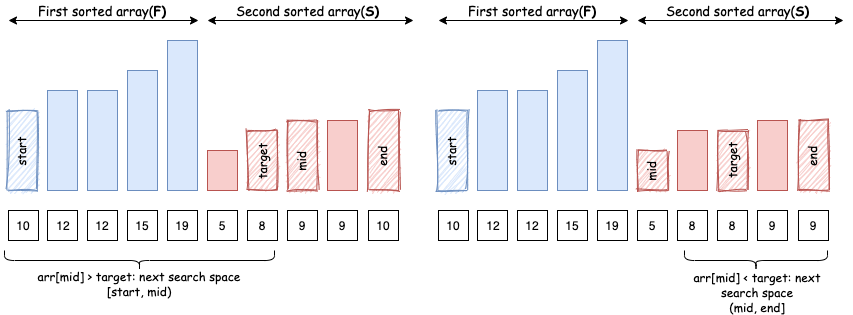
* Case 2: arr[mid] lies in the S, target lies in F: Similarly, we know that element lies here: [start, mid).



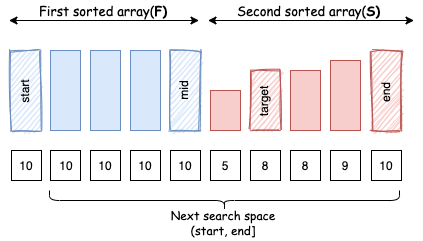
* Case 3: Both arr[mid] and target lie in F: since both of them are in same sorted array, we can compare arr[mid] and target in order to decide how to reduce search space.



* Case 4: Both arr[mid] and target lie in S: Again, since both of them are in same sorted array, we can compare arr[mid] and target in order to decide how to reduce search space.



But there is a catch, if arr[mid] equals arr[start], then we know that arr[mid] might belong to both F and S and hence we cannot find the relative position of target from it.



|  |
| --- |
| // returns true if we can reduce the search space in current binary search space  private boolean isBinarySearchHelpful(int[] arr, int left, int element) {  return arr[left] != element;  } |

In this case, we have no option but to move to next search space iteratively. Hence, there are certain search spaces that allow a binary search, and some search spaces that don't.

|  |
| --- |
| class Solution {  public boolean search(int[] nums, int target) {  int n = nums.length;  if (n == 0) return false;  int end = n - 1;  int start = 0;  while (start <= end) {  int mid = start + (end - start) / 2;  if (nums[mid] == target) {  return true;  }  if (!isBinarySearchHelpful(nums, start, nums[mid])) {  start++;  continue;  }  // which array does pivot belong to.  boolean pivotArray = existsInFirst(nums, start, nums[mid]);  // which array does target belong to.  boolean targetArray = existsInFirst(nums, start, target);  if (pivotArray ^ targetArray) { // If pivot and target exist in different sorted arrays, recall that xor is true when both operands are distinct  if (pivotArray) {  start = mid + 1; // pivot in the first, target in the second  } else {  end = mid - 1; // target in the first, pivot in the second  }  } else { // If pivot and target exist in same sorted array  if (nums[mid] < target) {  start = mid + 1;  } else {  end = mid - 1;  }  }  }  return false;  }  // returns true if we can reduce the search space in current binary search space  private boolean isBinarySearchHelpful(int[] arr, int start, int element) {  return arr[start] != element;  }  // returns true if element exists in first array, false if it exists in second  private boolean existsInFirst(int[] arr, int start, int element) {  return arr[start] <= element;  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*) worst case, O(\log N)*O*(log*N*) best case, where N*N* is the length of the input array.

Worst case: This happens when all the elements are the same and we search for some different element. At each step, we will only be able to reduce the search space by 1 since arr[mid] equals arr[start] and it's not possible to decide the relative position of target from arr[mid]. Example: [1, 1, 1, 1, 1, 1, 1], target = 2.

Best case: This happens when all the elements are distinct. At each step, we will be able to divide our search space into half just like a normal binary search.

This also answers the following follow-up question:

1. Would this (having duplicate elements) affect the run-time complexity? How and why?

As we can see, by having duplicate elements in the array, we often miss the opportunity to apply binary search in certain search spaces. Hence, we get O(N)*O*(*N*) worst case (with duplicates) vs O(\log N)*O*(log*N*) best case complexity (without duplicates).

* Space complexity : O(1)*O*(1).

**Numbers At Most N Given Digit Set**

Given an array of digits which is sorted in **non-decreasing** order. You can write numbers using each digits[i] as many times as we want. For example, if digits = ['1','3','5'], we may write numbers such as '13', '551', and '1351315'.

Return *the number of positive integers that can be generated*that are less than or equal to a given integer n.

**Example 1:**

**Input:** digits = ["1","3","5","7"], n = 100

**Output:** 20

**Explanation:**

The 20 numbers that can be written are:

1, 3, 5, 7, 11, 13, 15, 17, 31, 33, 35, 37, 51, 53, 55, 57, 71, 73, 75, 77.

**Example 2:**

**Input:** digits = ["1","4","9"], n = 1000000000

**Output:** 29523

**Explanation:**

We can write 3 one digit numbers, 9 two digit numbers, 27 three digit numbers,

81 four digit numbers, 243 five digit numbers, 729 six digit numbers,

2187 seven digit numbers, 6561 eight digit numbers, and 19683 nine digit numbers.

In total, this is 29523 integers that can be written using the digits array.

**Example 3:**

**Input:** digits = ["7"], n = 8

**Output:** 1

**Constraints:**

* 1 <= digits.length <= 9
* digits[i].length == 1
* digits[i] is a digit from '1' to '9'.
* All the values in digits are **unique**.
* digits is sorted in **non-decreasing** order.
* 1 <= n <= 109

## Solution

#### Approach 1: Dynamic Programming + Counting

**Intuition**

First, call a positive integer X valid if X <= N and X only consists of digits from D. Our goal is to find the number of valid integers.

Say N has K digits. If we write a valid number with k digits (k < K), then there are (D\text{.length})^k(*D*.length)*k* possible numbers we could write, since all of them will definitely be less than N.

Now, say we are to write a valid K digit number from left to right. For example, N = 2345, K = 4, and D = '1', '2', ..., '9'. Let's consider what happens when we write the first digit.

* If the first digit we write is less than the first digit of N, then we could write any numbers after, for a total of (D\text{.length})^{K-1}(*D*.length)*K*−1 valid numbers from this one-digit prefix. In our example, if we start with 1, we could write any of the numbers 1111 to 1999 from this prefix.
* If the first digit we write is the same, then we require that the next digit we write is equal to or lower than the next digit in N. In our example (with N = 2345), if we start with 2, the next digit we write must be 3 or less.
* We can't write a larger digit, because if we started with eg. 3, then even a number of 3000 is definitely larger than N.

**Algorithm**

Let dp[i] be the number of ways to write a valid number if N became N[i], N[i+1], .... For example, if N = 2345, then dp[0] would be the number of valid numbers at most 2345, dp[1] would be the ones at most 345, dp[2] would be the ones at most 45, and dp[3] would be the ones at most 5.

Then, by our reasoning above, dp[i] = (number of d in D with d < S[i]) \* ((D.length) \*\* (K-i-1)), plus dp[i+1] if S[i] is in D.

|  |
| --- |
| class Solution {  public int atMostNGivenDigitSet(String[] D, int N) {  String S = String.valueOf(N);  int K = S.length();  int[] dp = new int[K+1];  dp[K] = 1;  for (int i = K-1; i >= 0; --i) {  // compute dp[i]  int Si = S.charAt(i) - '0';  for (String d: D) {  if (Integer.valueOf(d) < Si)  dp[i] += Math.pow(D.length, K-i-1);  else if (Integer.valueOf(d) == Si)  dp[i] += dp[i+1];  }  }  for (int i = 1; i < K; ++i)  dp[0] += Math.pow(D.length, i);  return dp[0];  }  } |

**Complexity Analysis**

* Time Complexity: O(\log N)*O*(log*N*), and assuming D\text{.length}*D*.length is constant. (We could make this better by pre-calculating the number of d < S[i] for all possible digits S[i], but this isn't necessary.)
* Space Complexity: O(\log N)*O*(log*N*), the space used by S and dp. (Actually, we could store only the last 2 entries of dp, but this isn't necessary.)

#### Approach 2: Mathematical

**Intuition**

As in Approach #1, call a positive integer X valid if X <= N and X only consists of digits from D.

Now let B = D.length. There is a bijection between valid integers and so called "bijective-base-B" numbers. For example, if D = ['1', '3', '5', '7'], then we could write the numbers '1', '3', '5', '7', '11', '13', '15', '17', '31', ... as (bijective-base-B) numbers '1', '2', '3', '4', '11', '12', '13', '14', '21', ....

It is clear that both of these sequences are increasing, which means that the first sequence is a contiguous block of valid numbers, followed by invalid numbers.

Our approach is to find the largest valid integer, and convert it into bijective-base-B from which it is easy to find its rank (position in the sequence.) Because of the bijection, the rank of this element must be the number of valid integers.

Continuing our example, if N = 64, then the valid numbers are '1', '3', ..., '55', '57', which can be written as bijective-base-4 numbers '1', '2', ..., '33', '34'. Converting this last entry '34' to decimal, the answer is 16 (3 \* 4 + 4).

**Algorithm**

Let's convert N into the largest possible valid integer X, convert X to bijective-base-B, then convert that result to a decimal answer. The last two conversions are relatively straightforward, so let's focus on the first part of the task.

Let's try to write X one digit at a time. Let's walk through an example where D = ['2', '4', '6', '8']. There are some cases:

* If the first digit of N is in D, we write that digit and continue. For example, if N = 25123, then we will write 2 and continue.
* If the first digit of N is larger than min(D), then we write the largest possible number from D less than that digit, and the rest of the numbers will be big. For example, if N = 5123, then we will write 4888 (4 then 888).
* If the first digit of N is smaller than min(D), then we must "subtract 1" (in terms of X's bijective-base-B representation), and the rest of the numbers will be big.

For example, if N = 123, we will write 88. If N = 4123, we will write 2888. And if N = 22123, we will write 8888. This is because "subtracting 1" from '', '4', '22' yields '', '2', '8' (can't go below 0).

Actually, in our solution, it is easier to write in bijective-base-B, so instead of writing digits of D, we'll write the index of those digits (1-indexed). For example, X = 24888 will be A = [1, 2, 4, 4, 4]. Afterwards, we convert this to decimal.

|  |
| --- |
| class Solution {  public int atMostNGivenDigitSet(String[] D, int N) {  int B = D.length; // bijective-base B  char[] ca = String.valueOf(N).toCharArray();  int K = ca.length;  int[] A = new int[K];  int t = 0;  for (char c: ca) {  int c\_index = 0; // Largest such that c >= D[c\_index - 1]  boolean match = false;  for (int i = 0; i < B; ++i) {  if (c >= D[i].charAt(0))  c\_index = i+1;  if (c == D[i].charAt(0))  match = true;  }  A[t++] = c\_index;  if (match) continue;  if (c\_index == 0) { // subtract 1  for (int j = t-1; j > 0; --j) {  if (A[j] > 0) break;  A[j] += B;  A[j-1]--;  }  }  while (t < K)  A[t++] = B;  break;  }  int ans = 0;  for (int x: A)  ans = ans \* B + x;  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(\log N)*O*(log*N*), and assuming D\text{.length}*D*.length is constant.
* Space Complexity: O(\log N)*O*(log*N*), the space used by A.

**Unique Morse Code Words**

International Morse Code defines a standard encoding where each letter is mapped to a series of dots and dashes, as follows: "a" maps to ".-", "b" maps to "-...", "c" maps to "-.-.", and so on.

For convenience, the full table for the 26 letters of the English alphabet is given below:

[".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-",".-..","--","-.","---",".--.","--.-",".-.","...","-","..-","...-",".--","-..-","-.--","--.."]

Now, given a list of words, each word can be written as a concatenation of the Morse code of each letter. For example, "cab" can be written as "-.-..--...", (which is the concatenation "-.-." + ".-" + "-..."). We'll call such a concatenation, the transformation of a word.

Return the number of different transformations among all words we have.

**Example:**

**Input:** words = ["gin", "zen", "gig", "msg"]

**Output:** 2

**Explanation:**

The transformation of each word is:

"gin" -> "--...-."

"zen" -> "--...-."

"gig" -> "--...--."

"msg" -> "--...--."

There are 2 different transformations, "--...-." and "--...--.".

**Note:**

* The length of words will be at most 100.
* Each words[i] will have length in range [1, 12].
* words[i] will only consist of lowercase letters.

#### Approach 1: Hash Set

**Intuition and Algorithm**

We can transform each word into it's Morse Code representation.

After, we put all transformations into a set seen, and return the size of the set.

|  |
| --- |
| class Solution {  public int uniqueMorseRepresentations(String[] words) {  String[] MORSE = new String[]{".-","-...","-.-.","-..",".","..-.","--.",  "....","..",".---","-.-",".-..","--","-.",  "---",".--.","--.-",".-.","...","-","..-",  "...-",".--","-..-","-.--","--.."};  Set<String> seen = new HashSet();  for (String word: words) {  StringBuilder code = new StringBuilder();  for (char c: word.toCharArray())  code.append(MORSE[c - 'a']);  seen.add(code.toString());  }  return seen.size();  }  } |

**Complexity Analysis**

* Time Complexity: O(S)*O*(*S*), where S*S* is the sum of the lengths of words in words. We iterate through each character of each word in words.
* Space Complexity: O(S)*O*(*S*).

**House Robber III**

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

**Example 1:**

**Input:** [3,2,3,null,3,null,1]

3

/ \

2 3

\ \

3 1

**Output:** 7

**Explanation:** Maximum amount of money the thief can rob = 3 + 3 + 1 = **7**.

**Example 2:**

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

  3

/ \

4 5

/ \ \

1 3 1

**Output:** 9

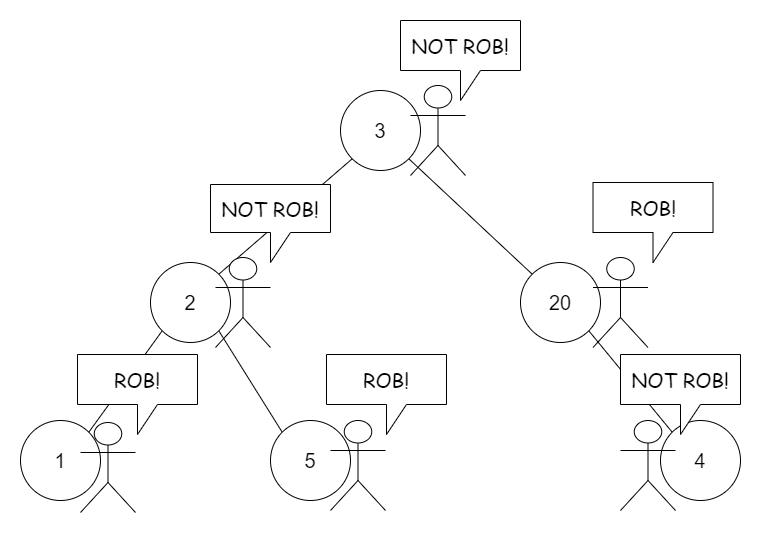
**Explanation:** Maximum amount of money the thief can rob = 4 + 5 = **9**.

## Solution

### **Overview**

This problem is an extension of the original [House Robber Problem](https://leetcode.com/problems/house-robber/). In this problem, our poor thief does not have a straight street, but have a binary-tree-like area instead.

To help better understand the problem, below is an example robbing plan on a binary tree:



Well, it looks a bit more complicated. For problems related to the tree data structure, often we could apply the recursion. Also, due to the characteristic of this problem, the [memoization](https://en.wikipedia.org/wiki/Memoization) approach and the DP approach are available.

Below, we will discuss three approaches: Recursion , Recursion with Memoization, and Dynamic Programming. They are similar but have some differences. Generally, we recommend the first and the second approaches, and provide the third approach as an extension reading for interested readers.

#### Approach 1: Recursion

**Intuition**

In this part, we explain how to think of this approach step by step. If you are only interested in the pure algorithm, you can jump to the algorithm part.

Since the tree itself is a recursive data structure, usually recursion is used to tackle these tree-relative problems.

Now we need a recursive function, let's call it helper (or whatever you want to call it).

Usually, we use a node as the input to the helper and add other arguments if we need more information.

The pseudo code of the common structure to solve recursive problems is as below:

function helper(node, other\_information) {

// basic case, such as node is null

if node is null:

return things like 0 or null

else:

do something relates to helper(node.left) and helper(node.right)

}

function answerToProblem(root) {

return helper(root, other\_information)

}

In some cases, we can use answerToProblem itself as the helper function.

OK, back to our problem. The next question is what should our helper function return.

Since the problem asks us to find out the maximum amount of money the thief can get, we could try using this maximum value as the return value of the helper function.

So helper receives a node as input, and returns the maximum amount the thief can get starting from this node.

Let's try writing the actual code. Well, it's a bit of trouble...

function helper(node) { // return the maximum by starting from this node

if node is null: // basic case

return 0

else:

two choices: rob this node or not?

if not rob... we have: helper(node.left) + helper(node.right)

what if rob? we get node.val!

what about node.left and node.right? we can not rob them.

Hmm... maybe we need to touch node.left.left and its other siblings... troublesome!

}

If we need to touch the grandchildren of this node, the case becomes complicated. Well, it is not infeasible but requires extra effort and code. Often, the best practice is to only touch its children, not its grandchildren.

The ideal situation is to make node.left and node.right automatically handle the grandchildren for us.

How to do it? Well, we can let them know whether we robbed this node or not by passing this information as input, like this:

two choices: rob this node or not?

rob = node.val + helper(node.left, parent\_robbed=True)

+ helper(node.right, parent\_robbed=True)

not\_rob = helper(node.left, parent\_robbed=False)

+ helper(node.right, parent\_robbed=False)

return max(rob, not\_rob)

Cool, we also need to change the input correspondingly:

function helper(node, the parent is robbed or not?) {

// return the maximum by starting from this node

tackle basic case...

if the parent is robbed:

we can not rob this node.

return helper(node.left, False) + helper(node.right, False)

if the parent is not robbed:

two choices: rob this node or not?

calculate `rob` and `not\_rob`...

return max(rob, not\_rob)

}

Good, now we have a functioning code. But the code is still not perfect.

An obvious problem is that the helper is called too many times. Ideally, it should only be called as least as possible to reduce redundant calculations.

For example, when calculating rob and not\_rob:

rob = node.val + helper(node.left, True) + helper(node.right, True)

not\_rob = helper(node.left, False) + helper(node.right, False)

The helper is called four times. Moreover, when we call helper(node.left, True) and helper(node.left, False), those two involve same calculations internally, such as helper(node.left.left, False).

In other words, helper(node.left.left, False) is called inside helper(node.left, True), and also is called inside helper(node.left, False). It is calculated twice! We do not want that.

What if... we can combine them together?

We return the results of helper(node.left, True) and helper(node.left, False) in a single function: helper(node.left). Those two results can be stored in a two-element array.

function helper(node) {

// return original [`helper(node.left, True)`, `helper(node.left, False)`]

tackle basic case...

left = helper(node.left)

right = helper(node.right)

some calculation...

return [max\_if\_rob, max\_if\_not\_rob]

}

In this case, we fully use the calculation results without redundant calculation.

Also, you can reduce extra calculations by [memoization](https://en.wikipedia.org/wiki/Memoization) or by DP, which we will discuss in the following approaches.

**Algorithm**

Use a helper function which receives a node as input and returns a two-element array, where the first element represents the maximum amount of money the thief can rob if starting from this node without robbing this node, and the second element represents the maximum amount of money the thief can rob if starting from this node and robbing this node.

The basic case of the helper function should be null node, and in this case, it returns two zeros.

Finally, call the helper(root) in the main function, and return its maximum value.

**Implementation**

|  |
| --- |
| class Solution {  public int[] helper(TreeNode node) {  // return [rob this node, not rob this node]  if (node == null) {  return new int[] { 0, 0 };  }  int left[] = helper(node.left);  int right[] = helper(node.right);  // if we rob this node, we cannot rob its children  int rob = node.val + left[1] + right[1];  // else, we free to choose rob its children or not  int notRob = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);  return new int[] { rob, notRob };  }  public int rob(TreeNode root) {  int[] answer = helper(root);  return Math.max(answer[0], answer[1]);  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the binary tree.

* Time complexity: \mathcal{O}(N)O(*N*) since we visit all nodes once.
* Space complexity: \mathcal{O}(N)O(*N*) since we need stacks to do recursion, and the maximum depth of the recursion is the height of the tree, which is \mathcal{O}(N)O(*N*) in the worst case and \mathcal{O}(\log(N))O(log(*N*)) in the best case.

#### Approach 2: Recursion with Memoization

**Intuition**

Recall in the later part of approach 1, we have the following helper function:

function helper(node, the parent is robbed or not?) {

// return the maximum by starting from this node

tackle basic case...

if the parent is robbed:

return helper(node.left, True) + helper(node.right, True)

if the parent is not robbed:

two choices: rob this node or not?

calculate `rob` and `not\_rob`...

return max(rob, not\_rob)

}

However, this code is not good since it involves some redundant calculations. Improvements are needed.

To avoid those redundant calculations, using memoization is an option. i.e., save the results of the functions in hash maps, and return the stored results when are called next time.

You can also add memoization to the "grandchildren" method we mentioned in approach 1. That would be another functioning approach. There is a lot of possibilities.

**Algorithm**

Use a helper function which receives a node and a bool variable as input, and if that variable is true, it returns the maximum amount of money the thief can rob if starting from this node and robbing this node, else returns the maximum amount of money the thief can rob if starting from this node without robbing this node.

The result of this helper function would be saved in the maps, and return from the maps when are called next time.

The basic case of the helper function should be null node, and in this case, it returns zero.

Finally, call the helper(root) in the main function, and return its value.

**Implementation**

|  |
| --- |
| class Solution {  HashMap<TreeNode, Integer> robResult = new HashMap<>();  HashMap<TreeNode, Integer> notRobResult = new HashMap<>();  public int helper(TreeNode node, boolean parentRobbed) {  if (node == null) {  return 0;  }  if (parentRobbed) {  if (robResult.containsKey(node)) {  return robResult.get(node);  }  int result = helper(node.left, false) + helper(node.right, false);  robResult.put(node, result);  return result;  } else {  if (notRobResult.containsKey(node)) {  return notRobResult.get(node);  }  int rob = node.val + helper(node.left, true) + helper(node.right, true);  int notRob = helper(node.left, false) + helper(node.right, false);  int result = Math.max(rob, notRob);  notRobResult.put(node, result);  return result;  }  }  public int rob(TreeNode root) {  return helper(root, false);  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the binary tree.

* Time complexity: \mathcal{O}(N)O(*N*) since we run the helper function for all nodes once, and saved the results to prevent the second calculation.
* Space complexity: \mathcal{O}(N)O(*N*) since we need two maps with the size of \mathcal{O}(N)O(*N*) to store the results, and \mathcal{O}(N)O(*N*) space for stacks to start recursion.

#### Approach 3: Dynamic Programming

**Intuition**

As we mentioned before, we could also apply the Dynamic Programming (DP) algorithm to this problem. Tree DP is a good method to tackle some tree problems.

Technically, the memoization approach we mentioned above can also be viewed as a DP approach. However, in this approach, we use DP arrays to generalize it.

This approach is a little hard, and you can treat it as extra reading for interested readers.

To perform a tree DP, we still need DP arrays, and by filling these arrays to get the final result.

Here we create two arrays: dp\_rob and dp\_not\_rob.

dp\_rob[i] represents the max money we can rob if we start from node i and rob this node.

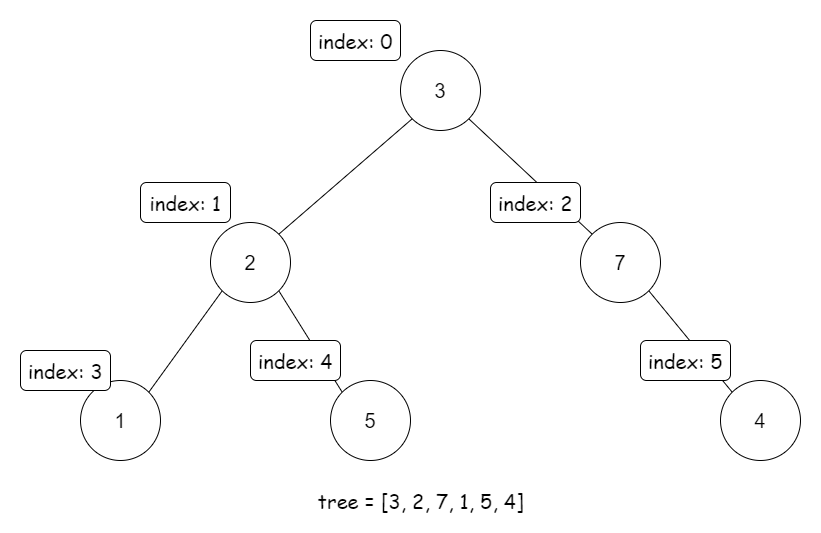
dp\_not\_rob[i] represents the max money we can rob if we start from node i and do not rob this node.

Here comes a question: what is node i? We do not have an index i for each node.

It seems like we need to index or map the nodes in the tree to some integers.

Well, there are many available mapping methods. Of course, you can choose whatever kind of mapping you want.

Below is the mapping we use in this approach: From left to right, from up to down:



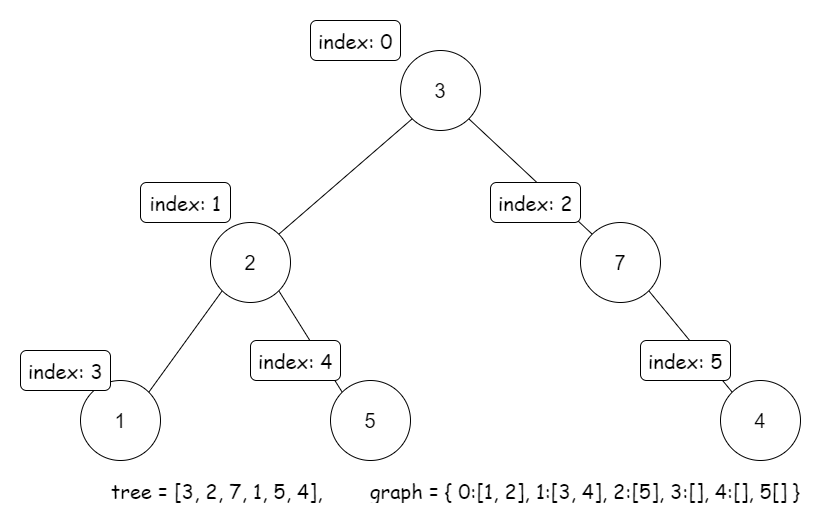
This mapping is used in many situations, such as heap and segment tree, where you store a tree in an array.

The Breadth-First Search (BFS) suffices to construct this mapping.

In some cases, we are provided with an array of node's values and a hashmap storing the relationship between nodes. If so, we can directly perform the dp since we already have the mapping.

However, in this problem, we have to use a lot of code to construct the mapping.

Also, we need to build a graph to store the relationships between those integers -- the parents and children's relationships:



OK. Now we have two questions remained: the basic cases of our DP, and the transition equations.

For the first question, the basic cases should be leaf nodes. For those leaf nodes, dp\_rob[i] is just node.val, and dp\_not\_rob[i] is 0.

For the the transition equations, let's say we need to calculate dp\_rob[i] and dp\_not\_rob[i].

If we want to rob node i, then we can not rob its children. Each child provides dp\_not\_rob[child]. Therefore:

\text{dp\\_rob}\_i = \text{tree}\_i + \sum\_{\text{child} \in \text{graph}\_i}\text{dp\\_not\\_rob}\_{\text{child}}dp\_rob*i*​=tree*i*​+∑child∈graph*i*​​dp\_not\_robchild​

If we do not want to rob node i, then we can choose to rob its children or not. In this case, each child provides the maximum of dp\_rob[child] and dp\_not\_rob[child].

Therefore:

\text{dp\\_not\\_rob}\_i = \sum\_{\text{child} \in \text{graph}\_i}\max(\text{dp\\_rob}\_{\text{child}}, \text{dp\\_not\\_rob}\_{\text{child}})dp\_not\_rob*i*​=∑child∈graph*i*​​max(dp\_robchild​,dp\_not\_robchild​)

Note that the child's index is always larger than the parent's index in our mapping, so we should iterate the dp arrays backward.

**Algorithm**

Transform the tree from node-based into an array-based tree and a map graph.

Then create two DP arrays, where dp\_rob[i] represents the maximum amount of money the thief can rob if starting from node i with robbing this node, and dp\_not\_rob[i] represents the maximum amount of money the thief can rob if starting from node i without robbing this node.

The transition equations is:

\text{dp\\_rob}\_i = \text{tree}\_i + \sum\_{\text{child} \in \text{graph}\_i}\text{dp\\_not\\_rob}\_{\text{child}}dp\_rob*i*​=tree*i*​+∑child∈graph*i*​​dp\_not\_robchild​

\text{dp\\_not\\_rob}\_i = \sum\_{\text{child} \in \text{graph}\_i}\max(\text{dp\\_rob}\_{\text{child}}, \text{dp\\_not\\_rob}\_{\text{child}})dp\_not\_rob*i*​=∑child∈graph*i*​​max(dp\_robchild​,dp\_not\_robchild​)

Finally, return the maximum of dp\_rob[0] and dp\_not\_rob[0].

**Implementation**

|  |
| --- |
| class Solution {  public int rob(TreeNode root) {  if (root == null) {  return 0;  }  // reform tree into array-based tree  ArrayList<Integer> tree = new ArrayList<>();  HashMap<Integer, ArrayList<Integer>> graph = new HashMap<>();  graph.put(-1, new ArrayList<>());  int index = -1;  // we use two Queue to store node and index  Queue<TreeNode> q\_node = new LinkedList<>();  q\_node.add(root);  Queue<Integer> q\_index = new LinkedList<>();  q\_index.add(index);  while (q\_node.size() > 0) {  TreeNode node = q\_node.poll();  int parentIndex = q\_index.poll();  if (node != null) {  index++;  tree.add(node.val);  graph.put(index, new ArrayList<>());  graph.get(parentIndex).add(index);  // push new node into Queue  q\_node.add(node.left);  q\_index.add(index);  q\_node.add(node.right);  q\_index.add(index);  }  }  // represent the maximum start by node i with robbing i  int[] dpRob = new int[index + 1];  // represent the maximum start by node i without robbing i  int[] dpNotRob = new int[index + 1];  for (int i = index; i >= 0; i--) {  ArrayList<Integer> children = graph.get(i);  if (children == null || children.size() == 0) {  // if is leaf  dpRob[i] = tree.get(i);  dpNotRob[i] = 0;  } else {  dpRob[i] = tree.get(i);  for (int child : children) {  dpRob[i] += dpNotRob[child];  dpNotRob[i] += Math.max(dpRob[child], dpNotRob[child]);  }  }  }  return Math.max(dpRob[0], dpNotRob[0]);  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the binary tree.

* Time complexity: \mathcal{O}(N)O(*N*) since we visit all nodes once to form the tree-array, and then iterate two DP array, which both have length \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*) since we need an array of length \mathcal{O}(N)O(*N*) to store the tree, and two DP arrays of length \mathcal{O}(N)O(*N*). Also, the sizes of other data structures in code do not exceed \mathcal{O}(N)O(*N*).

**Smallest Integer Divisible by K**

Given a positive integer K, you need to find the **length** of the **smallest** positive integer N such that N is divisible by K, and N only contains the digit 1.

Return *the****length****of*N. If there is no such N, return -1.

**Note:** N may not fit in a 64-bit signed integer.

**Example 1:**

**Input:** K = 1

**Output:** 1

**Explanation:** The smallest answer is N = 1, which has length 1.

**Example 2:**

**Input:** K = 2

**Output:** -1

**Explanation:** There is no such positive integer N divisible by 2.

**Example 3:**

**Input:** K = 3

**Output:** 3

**Explanation:** The smallest answer is N = 111, which has length 3.

**Constraints:**

* 1 <= K <= 105

 Hide Hint #1

11111 = 1111 \* 10 + 1 We only need to store remainders modulo K.

   Hide Hint #2

If we never get a remainder of 0, why would that happen, and how would we know that?

## Solution

#### Overview

It's an interesting problem that requires a little observation and insight. It's recommended to try a few numbers to find out some regular patterns. Below, we will discuss a simple approach to solve this problem.

#### Approach 1: Checking Loop

**Intuition**

We need to do two things:

1. check if the required number N exists.
2. find out length(N).

The second one is easy: we only need to keep multiplying N by 10 and adding 1 until N%K==0. However, since N might overflow, we need to use the remainder. The pseudo-code is as below:

remainder = 1

length\_N = 1

while remainder%K != 0

N = remainder\*10 + 1

remainder = N%K

length\_N += 1

return length\_N

Since the remainder and N have the same remainder of K, it OK to use remainder instead of N.

Now, the only problem is how to check whether the required number N exists.

Notice that if N does not exist, this while-loop will continue endlessly. However, the possible values of remainder are limited -- ranging from 0 to K-1. Therefore, if the while-loop continues forever, the remainder repeats. Also, if remainder repeats, then it gets into a loop. Hence, the while-loop is endless if and only if the remainder repeats.

In this case, we can check if the remainder repeats to check if the while-loop is endless:

remainder = 1

length\_N = 1

seen\_remainders = set()

while remainder%K != 0

N = remainder\*10 + 1

remainder = N%K

length\_N += 1

if remainder in seen\_remainders

return -1

else

seen\_remainders.add(remainder)

return length\_N

Now we have an algorithm that can solve the problem.

Furthermore, we can improve this algorithm with [Pigeonhole Principle](https://en.wikipedia.org/wiki/Pigeonhole_principle). Recall that the number of possible values of remainder (ranging from 0 to K-1) is limited, and in fact, the number is K. As a result, if the while-loop continues more than K times, and haven't stopped, then we can conclude that remainder repeats -- you can not have more than K different remainder.

Hence, if N exists, the while-loop must return length\_N in the first K loops. Otherwise, it goes into an infinite loop.

Therefore, we can just run the while-loop K times, and return -1 if not stopped.

**Algorithm**

We just run the while-loop K times, check if the remainder is 0, and return -1 if not stopped.

Note: After reading the Algorithm part, it is recommended to try writing the code on your own before reading the solution code.

|  |
| --- |
| class Solution {  public int smallestRepunitDivByK(int K) {  int remainder = 0;  for (int length\_N = 1; length\_N <= K; length\_N++) {  remainder = (remainder \* 10 + 1) % K;  if (remainder == 0) {  return length\_N;  }  }  return -1;  }  } |

There are a few interesting points worth pointing out in the code above:

1. We initialize remainder to 0, not 1, to keep code consistency because in the first loop the remainder changes to 1. You can also initialize it as 1, but it requires a little change in code.
2. We only run the loop K times at most, not K+1. This is because if it does not stop in the previous K loop, it will continue the K+1-th iteration, which must have repeated remainder. Therefore, it is not necessary to check the K+1-th iteration.

Also, note that 111...111 can never be divided by 2 or 5 because its last digit is never an even number or 5. You can just return -1 if you find that 2 or 5 is a factor of K.

**Complexity Analysis**

* Time Complexity : \mathcal{O}(K)O(*K*) since we at most run the loop \mathcal{O}(K)O(*K*) times.
* Space Complexity : \mathcal{O}(1)O(1) since we only use three ints: K, remainder, and length\_N.

**Longest Substring with At Least K Repeating Characters**

Given a string s and an integer k, return *the length of the longest substring of* s *such that the frequency of each character in this substring is greater than or equal to* k.

**Example 1:**

**Input:** s = "aaabb", k = 3

**Output:** 3

**Explanation:** The longest substring is "aaa", as 'a' is repeated 3 times.

**Example 2:**

**Input:** s = "ababbc", k = 2

**Output:** 5

**Explanation:** The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3 times.

**Constraints:**

* 1 <= s.length <= 104
* s consists of only lowercase English letters.
* 1 <= k <= 105

## Solution

#### Overview

We want to find the longest substring in a given string s where each character is repeated at least k times. This is an interesting problem that can be solved using different algorithm paradigms like Divide and Conquer and the Sliding Window Approach. We will start by discussing the brute force approach, moving towards more efficient implementations.

Let's discuss each approach in detail.

#### Approach 1: Brute Force

**Intuition**

The naive approach would be to generate all possible substrings for a given string s. For each substring, we must check if all the characters are repeated at least k times. Among all the substrings that satisfy the given condition, return the length of the longest substring.

**Algorithm**

* Generate substrings from string s starting at index start and ending at index end.
* Use the countMap array to store the frequency of each character in the substring.
* The isValid method uses countMap to check whether every character in substring has at least k frequency.
* Track the maximum substring length and return the result.

**Implementation**

|  |
| --- |
| class Solution {  public int longestSubstring(String s, int k) {  if (s == null || s.isEmpty() || k > s.length()) {  return 0;  }  int[] countMap = new int[26];  int n = s.length();  int result = 0;  for (int start = 0; start < n; start++) {  // reset the count map  Arrays.fill(countMap, 0);  for (int end = start; end < n; end++) {  countMap[s.charAt(end) - 'a']++;  if (isValid(s, start, end, k, countMap)) {  result = Math.max(result, end - start + 1);  }  }  }  return result;  }  private boolean isValid(String s, int start, int end, int k, int[] countMap) {  int countLetters = 0, countAtLeastK = 0;  for (int freq : countMap) {  if (freq > 0) countLetters++;  if (freq >= k) countAtLeastK++;  }  return countAtLeastK == countLetters;  }  } |

**Complexity Analysis**

* Time Complexity : \mathcal{O}(n^{2})O(*n*2), where n*n* is equal to length of string s*s*. The nested for loop that generates all substrings from string s*s* takes \mathcal{O}(n^{2})O(*n*2) time, and for each substring, we iterate over \text{countMap}countMap array of size 2626. This gives us time complexity as \mathcal{O}(26 \cdot n^{2})O(26⋅*n*2) = \mathcal{O}(n^{2})O(*n*2).

This approach is exhaustive and results in Time Limit Exceeded (TLE).

* Space Complexity: \mathcal{O}(1)O(1) We use constant extra space of size 26 for countMap array.

#### Approach 2: Divide And Conquer

**Intuition**

[Divide and Conquer](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm) is one of the popular strategies that work in 2 phases.

* Divide the problem into subproblems. (Divide Phase).
* Repeatedly solve each subproblem independently and combine the result to solve the original problem. (Conquer Phase).

We could apply this strategy by recursively splitting the string into substrings and combine the result to find the longest substring that satisfies the given condition. The longest substring for a string starting at index start and ending at index end can be given by,

longestSustring(start, end) = max(longestSubstring(start, mid), longestSubstring(mid+1, end))

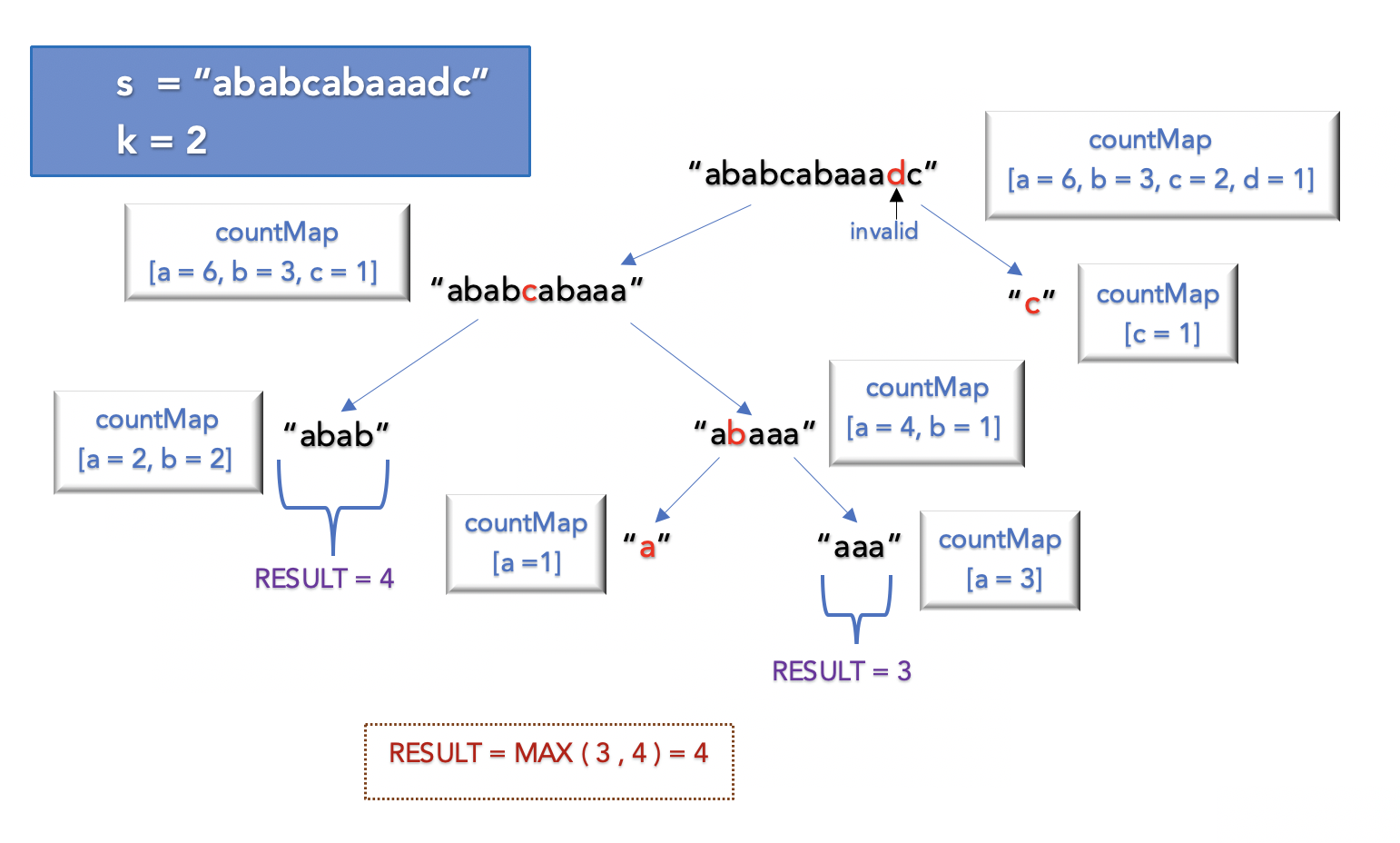
Finding the split position *(mid)*

The string would be split only when we find an invalid character. An invalid character is the one with a frequency of less than k. As we know, the invalid character cannot be part of the result, we split the string at the index where we find the invalid character, recursively check for each split, and combine the result.

**Algorithm**

* Build the countMap with the frequency of each character in the string s.
* Find the position for mid index by iterating over the string. The mid index would be the first invalid character in the string.
* Split the string into 2 substrings at the mid index and recursively find the result.

To make it more efficient, we ignore all the invalid characters after the mid index as well, thereby reducing the number of recursive calls.



**Implementation**

|  |
| --- |
| class Solution {  public int longestSubstring(String s, int k) {  return longestSubstringUtil(s, 0, s.length(), k);  }  int longestSubstringUtil(String s, int start, int end, int k) {  if (end < k) return 0;  int[] countMap = new int[26];  // update the countMap with the count of each character  for (int i = start; i < end; i++)  countMap[s.charAt(i) - 'a']++;  for (int mid = start; mid < end; mid++) {  if (countMap[s.charAt(mid) - 'a'] >= k) continue;  int midNext = mid + 1;  while (midNext < end && countMap[s.charAt(midNext) - 'a'] < k) midNext++;  return Math.max(longestSubstringUtil(s, start, mid, k),  longestSubstringUtil(s, midNext, end, k));  }  return (end - start);  }  } |

**Complexity Analysis**

* Time Complexity : \mathcal{O}(N ^ {2})O(*N*2), where N*N* is the length of string s*s*. Though the algorithm performs better in most cases, the worst case time complexity is still \mathcal{O}(N ^ {2})O(*N*2).

In cases where we perform split at every index, the maximum depth of recursive call could be \mathcal{O}(N)O(*N*). For each recursive call it takes \mathcal{O}(N)O(*N*) time to build the countMap resulting in \mathcal{O}(n ^ {2})O(*n*2) time complexity.

* Space Complexity: \mathcal{O}(N)O(*N*) This is the space used to store the recursive call stack. The maximum depth of recursive call stack would be \mathcal{O}(N)O(*N*).

#### Approach 3: Sliding Window

**Intuition**

There is another intuitive method to solve the problem by using the Sliding Window Approach. The sliding window slides over the string s and validates each character. Based on certain conditions, the sliding window either expands or shrinks.

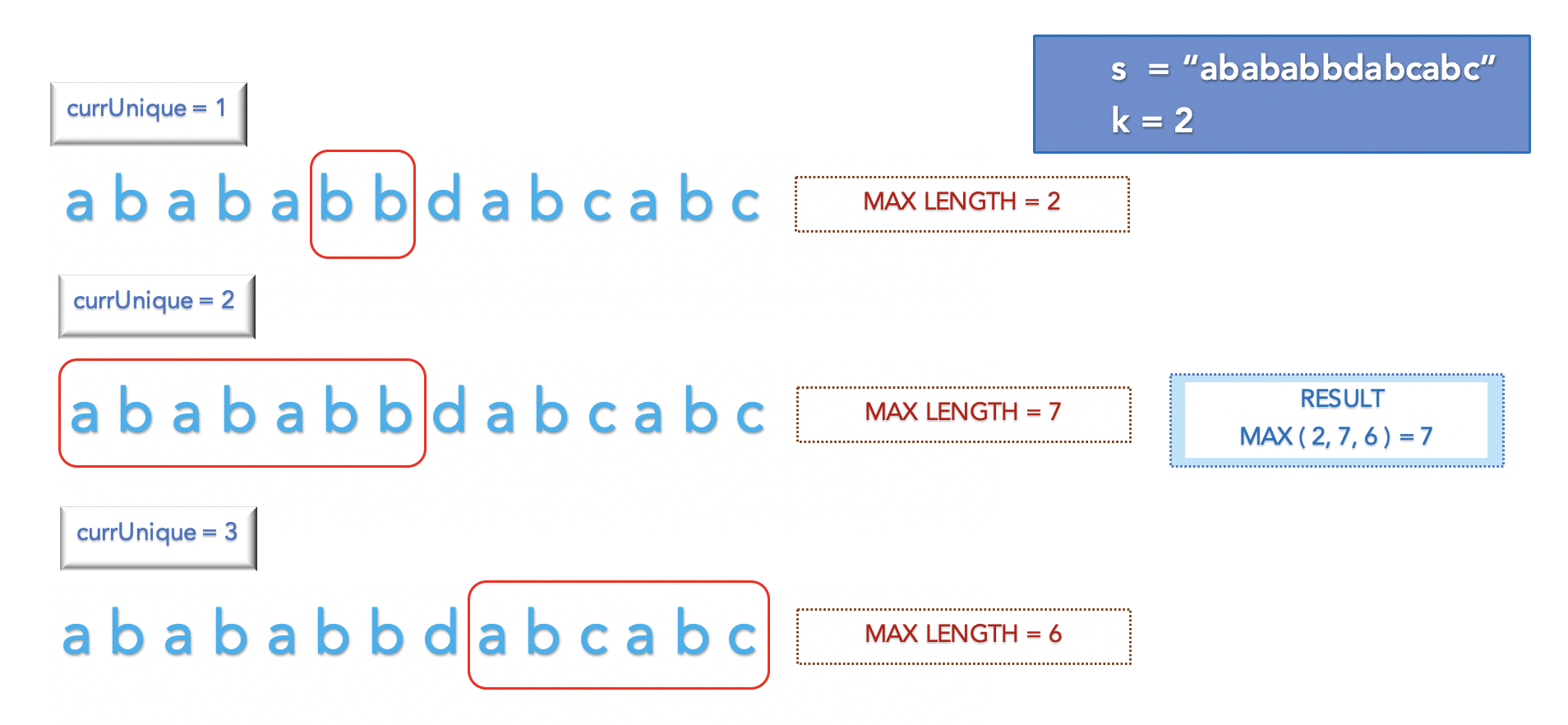
A substring is valid if each character has at least k frequency. The main idea is to find all the valid substrings with a different number of unique characters and track the maximum length. Let's look at the algorithm in detail.

**Algorithm**

1. Find the number of unique characters in the string s and store the count in variable maxUnique. For s = aabcbacad, the unique characters are a,b,c,d and maxUnique = 4.
2. Iterate over the string s with the value of currUnique ranging from 1 to maxUnique. In each iteration, currUnique is the maximum number of unique characters that must be present in the sliding window.
3. The sliding window starts at index windowStart and ends at index windowEnd and slides over string s until windowEnd reaches the end of string s. At any given point, we shrink or expand the window to ensure that the number of unique characters is not greater than currUnique.

* If the number of unique character in the sliding window is less than or equal to currUnique, expand the window from the right by adding a character to the end of the window given by windowEnd
* Otherwise, shrink the window from the left by removing a character from the start of the window given by windowStart.

1. Keep track of the number of unique characters in the current sliding window having at least k frequency given by countAtLeastK. Update the result if all the characters in the window have at least k frequency.



**Implementation**

|  |
| --- |
| public class Solution {  public int longestSubstring(String s, int k) {  char[] str = s.toCharArray();  int[] countMap = new int[26];  int maxUnique = getMaxUniqueLetters(s);  int result = 0;  for (int currUnique = 1; currUnique <= maxUnique; currUnique++) {  // reset countMap  Arrays.fill(countMap, 0);  int windowStart = 0, windowEnd = 0, idx = 0, unique = 0, countAtLeastK = 0;  while (windowEnd < str.length) {  // expand the sliding window  if (unique <= currUnique) {  idx = str[windowEnd] - 'a';  if (countMap[idx] == 0) unique++;  countMap[idx]++;  if (countMap[idx] == k) countAtLeastK++;  windowEnd++;  }  // shrink the sliding window  else {  idx = str[windowStart] - 'a';  if (countMap[idx] == k) countAtLeastK--;  countMap[idx]--;  if (countMap[idx] == 0) unique--;  windowStart++;  }  if (unique == currUnique && unique == countAtLeastK)  result = Math.max(windowEnd - windowStart, result);  }  }  return result;  }  // get the maximum number of unique letters in the string s  int getMaxUniqueLetters(String s) {  boolean map[] = new boolean[26];  int maxUnique = 0;  for (int i = 0; i < s.length(); i++) {  if (!map[s.charAt(i) - 'a']) {  maxUnique++;  map[s.charAt(i) - 'a'] = true;  }  }  return maxUnique;  }  } |

**Complexity Analysis**

* Time Complexity : \mathcal{O}(\text{maxUnique} \cdot N)O(maxUnique⋅*N*). We iterate over the string of length N*N*, \text{maxUnqiue}maxUnqiue times. Ideally, the number of unique characters in the string would not be more than 2626 (a to z). Hence, the time complexity is approximately \mathcal{O}( 26 \cdot N)O(26⋅*N*) = \mathcal{O}(N)O(*N*)
* Space Complexity: \mathcal{O}(1)O(1) We use constant extra space of size 26 to store the countMap.

**Partition Equal Subset Sum**

Given a **non-empty** array nums containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

**Example 1:**

**Input:** nums = [1,5,11,5]

**Output:** true

**Explanation:** The array can be partitioned as [1, 5, 5] and [11].

**Example 2:**

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

**Output:** false

**Explanation:** The array cannot be partitioned into equal sum subsets.

**Constraints:**

* 1 <= nums.length <= 200
* 1 <= nums[i] <= 100

## Solution Article

#### Overview

The problem is similar to the classic Knapsack problem. The basic idea is to understand that to partition an array into two subsets of equal sum say \text{subSetSum}subSetSum, the \text{totalSum}totalSum of given array must be twice the \text{subSetSum}subSetSum

\text{totalSum} = \text{subSetSum} \* 2totalSum=subSetSum∗2

This could also be written as, \text{subSetSum} = \text{totalSum}/2subSetSum=totalSum/2

Example Assume \text{totalSum}totalSum of an array is 2020 and if we want to partition it into 2 subsets of equal sum, then the \text{subSetSum}subSetSum must be (20/2) = 10(20/2)=10.

Now, the problem to find the subset with a sum equals a given target. Here target is \text{subSetSum}subSetSum.

It must be noted that the total sum of an array must be even, only then we can divide it into 2 equal subsets. For instance, we cannot have an equal \text{subSetSum}subSetSum for an array with total sum as 2121.

**Note:**

Finding a subset with a sum equal to a given target is different than [Subarray sum equals k](https://leetcode.com/problems/subarray-sum-equals-k/). Subarray is a contiguous sequence of array elements, whereas the subset could consist of any array elements regardless of the sequence. But each array element must belong to exactly one subset.

Let's discuss different algorithms to find the subset with a given sum.

#### Approach 1: Brute Force

**Intuition**

We have to find a subset in an array where the sum must be equal to \text{subSetSum}subSetSum that we calculated above. The brute force approach would be to generate all the possible subsets of an array and return true if we find a subset with the required sum.

**Algorithm**

Assume, there is an array \text{nums}nums of size n*n* and we have to find if there exists a subset with total \text{sum} = \text{subSetSum}sum=subSetSum. For a given array element x*x*, there could be either of 2 possibilities:

* Case 1) x*x* is included in subset sum. \text{subSetSum} = \text{subSetSum} - xsubSetSum=subSetSum−*x*
* Case 2) x*x* is not included in subset sum, so we must take previous sum without x*x*. \text{subSetSum} = \text{subSetSum}subSetSum=subSetSum

We can use depth first search and recursively calculate the \text{subSetSum}subSetSum for each case and check if either of them is true. This can be formulated as

isSum (subSetSum, n) = isSum(subSetSum- nums[n], n-1) || isSum(subSetSum, n-1)

Base Cases

* If \text{subSetSum}subSetSum is 00, return true ( Since we found a subset with sum subSetSum )
* If \text{subSetSum}subSetSum is less than 00, return false

|  |
| --- |
| class Solution {  public boolean canPartition(int[] nums) {  int totalSum = 0;  // find sum of all array elements  for (int num : nums) {  totalSum += num;  }  // if totalSum is odd,it cannot be partitioned into equal sum subset  if (totalSum % 2 != 0) return false;  int subSetSum = totalSum / 2;  int n = nums.length;  return dfs(nums, n - 1, subSetSum);  }  public boolean dfs(int[] nums, int n, int subSetSum) {  // Base Cases  if (subSetSum == 0)  return true;  if (n == 0 || subSetSum < 0)  return false;  return dfs(nums, n - 1, subSetSum - nums[n - 1]) || dfs(nums, n - 1, subSetSum);  }  } |

**Complexity Analysis**

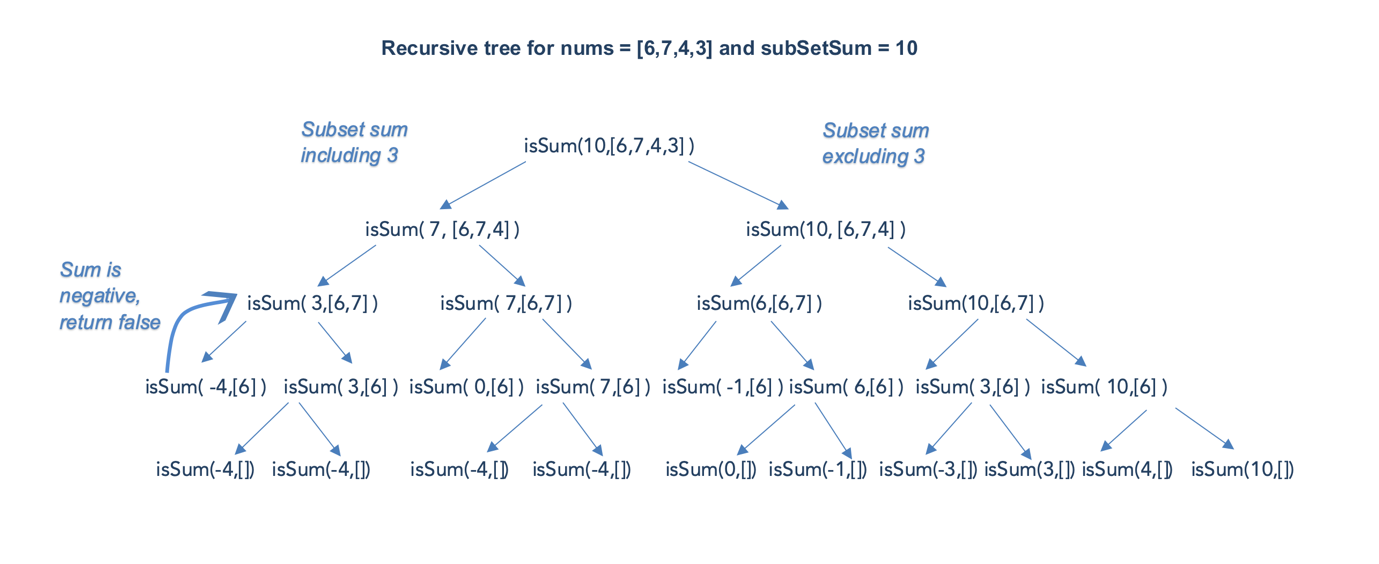
* Time Complexity : \mathcal{O}(2^{n})O(2*n*), where n*n* is equal to number of array elements. The recursive solution takes the form of a binary tree where there are 2 possibilities for every array element and the maximum depth of the tree could be n*n*. The time complexity is exponential, hence this approach is exhaustive and results in Time Limit Exceeded (TLE).
* Space Complexity: \mathcal{O}(N)O(*N*) This space will be used to store the recursion stack. We can’t have more than n*n* recursive calls on the call stack at any time.

#### Approach 2: Top Down Dynamic Programming - Memoization

**Intuition**

In the above approach, we observe that the same subproblem is computed and solved multiple times.

Example :



In the above recursion tree, we could see that \text{isSum}( 3,[6] )isSum(3,[6]) is computed twice and the result is always the same. Since the same subproblem is computed again and again, the problem has Overlapping Subproblem property and can be solved using Dynamic Programming.

**Algorithm**

We could have stored the results of our computation for the first time and used it later. This technique of computing once and returning the stored value is called memoization. We use a two dimensional array \text{memo}memo and follow the following steps for each recursive call :

* Check if subSetSum for a given n*n* exists in \text{memo}memo to see if we can avoid computing the answer and return the result stored in memo.
* Save the results of any calculations to \text{memo}memo.

|  |
| --- |
| class Solution {  public boolean canPartition(int[] nums) {  int totalSum = 0;  // find sum of all array elements  for (int num : nums) {  totalSum += num;  }  // if totalSum is odd, it cannot be partitioned into equal sum subset  if (totalSum % 2 != 0) return false;  int subSetSum = totalSum / 2;  int n = nums.length;  Boolean[][] memo = new Boolean[n + 1][subSetSum + 1];  return dfs(nums, n - 1, subSetSum, memo);  }  public boolean dfs(int[] nums, int n, int subSetSum, Boolean[][] memo) {  // Base Cases  if (subSetSum == 0)  return true;  if (n == 0 || subSetSum < 0)  return false;  // check if subSetSum for given n is already computed and stored in memo  if (memo[n][subSetSum] != null)  return memo[n][subSetSum];  boolean result = dfs(nums, n - 1, subSetSum - nums[n - 1], memo) ||  dfs(nums, n - 1, subSetSum, memo);  // store the result in memo  memo[n][subSetSum] = result;  return result;  }  } |

**Complexity Analysis**

Let n*n* be the number of array elements and m*m* be the \text{subSetSum}subSetSum.

* Time Complexity : \mathcal{O}(m \cdot n)O(*m*⋅*n*).
  + In the worst case where there is no overlapping calculation, the maximum number of entries in the memo would be m \cdot n*m*⋅*n*. For each entry, overall we could consider that it takes constant time, i.e. each invocation of dfs() at most emits one entry in the memo.
  + The overall computation is proportional to the number of entries in memo. Hence, the overall time complexity is \mathcal{O}(m \cdot n)O(*m*⋅*n*).
* Space Complexity: \mathcal{O}(m \cdot n)O(*m*⋅*n*). We are using a 2 dimensional array \text{memo}memo of size (m \cdot n)(*m*⋅*n*) and \mathcal{O}(n)O(*n*) space to store the recursive call stack. This gives us the space complexity as \mathcal{O}(n)O(*n*) + \mathcal{O}(m \cdot n)O(*m*⋅*n*) = \mathcal{O}(m \cdot n)O(*m*⋅*n*)

#### Approach 3: Bottom Up Dynamic Programming

**Intuition**

This is another approach to solving the Dynamic Programming problems. We use the iterative approach and store the result of subproblems in bottom-up fashion also known as Tabulation.

**Algorithm**

We maintain a 2D array , \text{dp}[n][\text{subSetSum}]dp[*n*][subSetSum] For an array element i*i* and sum j*j* in array \text{nums}nums,

\text{dp}[i][j] = \text{true}dp[*i*][*j*]=true if the sum j*j* can be formed by array elements in subset \text{nums[0]} .. \text{nums[i]}nums[0]..nums[i],otherwise \text{dp}[i][j] = \text{false}dp[*i*][*j*]=false

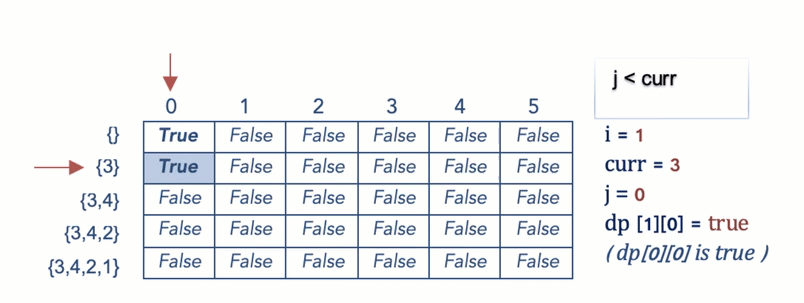
\text{dp}[i][j]dp[*i*][*j*] is \text{true}true it satisfies one of the following conditions :

* Case 1) sum j*j* can be formed without including i^{th}*ith* element,

\text{if } \text{dp}[i-1][j] == \text{true}if dp[*i*−1][*j*]==true

* Case 2) sum j*j* can be formed including i^{th}*ith* element,

\text{if } \text{dp}[i-1][j - \text{nums}[i]] == \text{true}if dp[*i*−1][*j*−nums[*i*]]==true



|  |
| --- |
| class Solution {  public boolean canPartition(int[] nums) {  int totalSum = 0;  // find sum of all array elements  for (int num : nums) {  totalSum += num;  }  // if totalSum is odd, it cannot be partitioned into equal sum subset  if (totalSum % 2 != 0) return false;  int subSetSum = totalSum / 2;  int n = nums.length;  boolean dp[][] = new boolean[n + 1][subSetSum + 1];  dp[0][0] = true;  for (int i = 1; i <= n; i++) {  int curr = nums[i - 1];  for (int j = 0; j <= subSetSum; j++) {  if (j < curr)  dp[i][j] = dp[i - 1][j];  else  dp[i][j] = dp[i - 1][j] || (dp[i - 1][j - curr]);  }  }  return dp[n][subSetSum];  }  } |

**Complexity Analysis**

* Time Complexity : \mathcal{O}(m \cdot n)O(*m*⋅*n*), where m*m* is the \text{subSetSum}subSetSum, and n*n* is the number of array elements. We iteratively fill the array of size m \cdot n*m*⋅*n*.
* Space Complexity : \mathcal{O}(m \cdot n)O(*m*⋅*n*) , where n*n* is the number of array elements and m*m* is the \text{subSetSum}subSetSum. We are using a 2 dimensional array \text{dp}dp of size m \cdot n*m*⋅*n*

#### Approach 4: Optimised Dynamic Programming - Using 1D Array

**Intuition**

We could further optimize Approach 3. We must understand that for any array element i*i*, we need results of the previous iteration (i-1) only. Hence, we could achieve the same using a one-dimensional array as well.

|  |
| --- |
| class Solution {  public boolean canPartition(int[] nums) {  if (nums.length == 0)  return false;  int totalSum = 0;  // find sum of all array elements  for (int num : nums) {  totalSum += num;  }  // if totalSum is odd, it cannot be partitioned into equal sum subset  if (totalSum % 2 != 0) return false;  int subSetSum = totalSum / 2;  boolean dp[] = new boolean[subSetSum + 1];  dp[0] = true;  for (int curr : nums) {  for (int j = subSetSum; j >= curr; j--) {  dp[j] |= dp[j - curr];  }  }  return dp[subSetSum];  }  } |

**Complexity Analysis**

* Time Complexity : \mathcal{O}(m \cdot n)O(*m*⋅*n*), where m*m* is the \text{subSetSum}subSetSum, and n*n* is the number of array elements. The time complexity is the same as *Approach 3*.
* Space Complexity: \mathcal{O}(m)O(*m*), As we use an array of size m*m* to store the result of subproblems.

**Note:**

The overall performance of *Approach 2* is better than all the approaches discussed above. This is because we terminate our search as soon as we find a subset with the required sum. Hence, it performs better in most cases except for the worst case.

**Maximum Average Subarray II**

You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose **length is greater than or equal to** k that has the maximum average value and return *this value*. Any answer with a calculation error less than 10-5 will be accepted.

**Example 1:**

**Input:** nums = [1,12,-5,-6,50,3], k = 4

**Output:** 12.75000

**Explanation:**

- When the length is 4, averages are [0.5, 12.75, 10.5] and the maximum average is 12.75

- When the length is 5, averages are [10.4, 10.8] and the maximum average is 10.8

- When the length is 6, averages are [9.16667] and the maximum average is 9.16667

The maximum average is when we choose a subarray of length 4 (i.e., the sub array [12, -5, -6, 50]) which has the max average 12.75, so we return 12.75

Note that we do not consider the subarrays of length < 4.

**Example 2:**

**Input:** nums = [5], k = 1

**Output:** 5.00000

**Constraints:**

* n == nums.length
* 1 <= k <= n <= 104
* -104 <= nums[i] <= 104

## Solution Article

#### Approach #1 Iterative method [Time Limit Exceeded]

One of the simplest solutions is to consider the sum of every possible subarray with length greater than or equal to k*k* and to determine the maximum average from out of those. But, instead of finding out this sum in a naive manner for every subarray with length greater than or equal to k*k* separately, we can do as follows.

For every starting point, s*s*, considered, we can iterate over the elements of nums*nums* starting from nums*nums*, and keep a track of the sum*sum* found till the current index(i*i*). Whenever the index reached is such that the number of elements lying between s*s* and i*i* is greater than or equal to k*k*, we can check if the average of the elements between s*s* and i*i* is greater than the average found till now or not.

|  |
| --- |
| public class Solution {  public double findMaxAverage(int[] nums, int k) {  double res = Integer.MIN\_VALUE;  for (int s = 0; s < nums.length - k + 1; s++) {  long sum = 0;  for (int i = s; i < nums.length; i++) {  sum += nums[i];  if (i - s + 1 >= k)  res = Math.max(res, sum \* 1.0 / (i - s + 1));  }  }  return res;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). Two for loops iterating over the whole length of nums*nums* with n*n* elements.
* Space complexity : O(1)*O*(1). Constant extra space is used.

#### Approach #2 Using Binary Search [Accepted]

**Algorithm**

To understand the idea behind this method, let's look at the following points.

Firstly, we know that the value of the average could lie between the range (min, max)(*min*,*max*). Here, min*min* and max*max* refer to the minimum and the maximum values out of the given nums*nums* array. This is because, the average can't be lesser than the minimum value and can't be larger than the maximum value.

But, in this case, we need to find the maximum average of a subarray with atleast k*k* elements. The idea in this method is to try to approximate(guess) the solution and to try to find if this solution really exists.

If it exists, we can continue trying to approximate the solution even to a further precise value, but choosing a larger number as the next approximation. But, if the initial guess is wrong, and the initial maximum average value(guessed) isn't possible, we need to try with a smaller number as the next approximate.

Now, instead of doing the guesses randomly, we can make use of Binary Search. With min*min* and max*max* as the initial numbers to begin with, we can find out the mid*mid* of these two numbers given by (min+max)/2(*min*+*max*)/2. Now, we need to find if a subarray with length greater than or equal to k*k* is possible with an average sum greater than this mid*mid* value.

To determine if this is possible in a single scan, let's look at an observation. Suppose, there exist j*j* elements, a\_1, a\_2, a\_3..., a\_j*a*1​,*a*2​,*a*3​...,*aj*​ in a subarray within nums*nums* such that their average is greater than mid*mid*. In this case, we can say that

(a\_1+a\_2+ a\_3...+a\_j)/j ≥ mid or

(a\_1+a\_2+ a\_3...+a\_j) ≥ j\*mid or

(a\_1-mid) +(a\_2-mid)+ (a\_3-mid) ...+(a\_j-mid) ≥ 0

Thus, we can see that if after subtracting the mid*mid* number from the elements of a subarray with more than k-1*k*−1 elements, within nums*nums*, if the sum of elements of this reduced subarray is greater than 0, we can achieve an average value greater than mid*mid*. Thus, in this case, we need to set the mid*mid* as the new minimum element and continue the process.

Otherwise, if this reduced sum is lesser than 0 for all subarrays with greater than or equal to k*k* elements, we can't achieve mid*mid* as the average. Thus, we need to set mid*mid* as the new maximum element and continue the process.

In order to determine if such a subarray exists in a linear manner, we keep on adding nums[i]-mid*nums*[*i*]−*mid* to the sum*sum* obtained till the i^{th}*ith* element while traversing over the nums*nums* array. If on traversing the first k*k* elements, the sum*sum* becomes greater than or equal to 0, we can directly determine that we can increase the average beyond mid*mid*. Otherwise, we continue making additions to sum*sum* for elements beyond the k^{th}*kth* element, making use of the following idea.

If we know the cumulative sum upto indices i*i* and j*j*, say sum\_i*sumi*​ and sum\_j*sumj*​ respectively, we can determine the sum of the subarray between these indices(including j*j*) as sum\_j - sum\_i*sumj*​−*sumi*​. In our case, we want this difference between the cumulative sums to be greater than or equal to 0 as discusssed above.

Further, for sum\_i*sumi*​ as the cumulative sum upto the current(i^{th}*ith*) index, all we need is sum\_j - sum\_i ≥ 0 such that j - i ≥ k.

To achive this, instead of checking with all possible values of sum\_i*sumi*​, we can just consider the minimum cumulative sum upto the index j - k*j*−*k*. This is because if the required condition can't be sastisfied with the minimum sum\_i*sumi*​, it can never be satisfied with a larger value.

To fulfil this, we make use of a prev*prev* variable which again stores the cumulative sums but, its current index(for cumulative sum) lies behind the current index for sum*sum* at an offset of k*k* units. Thus, by finding the minimum out of prev*prev* and the last minimum value, we can easily find out the required minimum sum value.

Every time after checking the possiblility with a new mid*mid* value, at the end, we need to settle at some value as the average. But, we can observe that eventually, we'll reach a point, where we'll keep moving near some same value with very small changes. In order to keep our precision in control, we limit this process to 10^-510−5 precision, by making use of error*error* and continuing the process till error*error* becomes lesser than 0.00001 .

|  |
| --- |
| public class Solution {  public double findMaxAverage(int[] nums, int k) {  double max\_val = Integer.MIN\_VALUE;  double min\_val = Integer.MAX\_VALUE;  for (int n: nums) {  max\_val = Math.max(max\_val, n);  min\_val = Math.min(min\_val, n);  }  double prev\_mid = max\_val, error = Integer.MAX\_VALUE;  while (error > 0.00001) {  double mid = (max\_val + min\_val) \* 0.5;  if (check(nums, mid, k))  min\_val = mid;  else  max\_val = mid;  error = Math.abs(prev\_mid - mid);  prev\_mid = mid;  }  return min\_val;  }  public boolean check(int[] nums, double mid, int k) {  double sum = 0, prev = 0, min\_sum = 0;  for (int i = 0; i < k; i++)  sum += nums[i] - mid;  if (sum >= 0)  return true;  for (int i = k; i < nums.length; i++) {  sum += nums[i] - mid;  prev += nums[i - k] - mid;  min\_sum = Math.min(prev, min\_sum);  if (sum >= min\_sum)  return true;  }  return false;  }  } |

**Complexity Analysis**

Let N be the number of element in the array, and range be the difference between the maximal and minimal values in the array, *i.e.* range = max\_val - min\_val, and finally the error be the precision required in the problem.

* Time complexity : O\big(N \cdot \log\_2{\frac{(\text{max\\_val} - \text{min\\_val})}{0.00001}} \big)*O*(*N*⋅log2​0.00001(max\_val−min\_val)​).
  + The algorithm consists of a binary search loop in the function of findMaxAverage().
  + At each iteration of the loop, the check() function dominates the time complexity, which is of O(N)*O*(*N*) for each invocation.
  + It now boils down to how many iterations the loop would run eventually. To calculate the number of iterations, let us break it down in the following steps.
  + After the first iteration, the \text{error}error would be \frac{\text{range}}{2}2range​, as one can see. Further on, at each iteration, the \text{error}error would be reduced into half. For example, after the second iteration, we would have the \text{error}error as \frac{\text{range}}{2}\cdot \frac{1}{2}2range​⋅21​.
  + As a result, after K*K* iterations, the error would become \text{error} = \text{range}\cdot 2^{-K}error=range⋅2−*K*. Given the condition of the loop, *i.e.* \text{error} < 0.00001error<0.00001, we can deduct that K > \log\_2{\frac{\text{range}}{0.00001}} = \log\_2{\frac{(\text{max\\_val} - \text{min\\_val})}{0.00001}}*K*>log2​0.00001range​=log2​0.00001(max\_val−min\_val)​
  + To sum up, the time complexity of the algorithm would be O\big( N \cdot K \big) = O\big(N \cdot \log\_2{\frac{(\text{max\\_val} - \text{min\\_val})}{0.00001}} \big)*O*(*N*⋅*K*)=*O*(*N*⋅log2​0.00001(max\_val−min\_val)​).
* Space complexity : O(1)*O*(1). Constant Space is used.

**Jump Game III**

**Solution**

Given an array of non-negative integers arr, you are initially positioned at start index of the array. When you are at index i, you can jump to i + arr[i] or i - arr[i], check if you can reach to **any** index with value 0.

Notice that you can not jump outside of the array at any time.

**Example 1:**

**Input:** arr = [4,2,3,0,3,1,2], start = 5

**Output:** true

**Explanation:**

All possible ways to reach at index 3 with value 0 are:

index 5 -> index 4 -> index 1 -> index 3

index 5 -> index 6 -> index 4 -> index 1 -> index 3

**Example 2:**

**Input:** arr = [4,2,3,0,3,1,2], start = 0

**Output:** true

**Explanation:**

One possible way to reach at index 3 with value 0 is:

index 0 -> index 4 -> index 1 -> index 3

**Example 3:**

**Input:** arr = [3,0,2,1,2], start = 2

**Output:** false

**Explanation:** There is no way to reach at index 1 with value 0.

**Constraints:**

* 1 <= arr.length <= 5 \* 104
* 0 <= arr[i] < arr.length
* 0 <= start < arr.length

Hide Hint #1

Think of BFS to solve the problem.

   Hide Hint #2

When you reach a position with a value = 0 then return true.

## Solution Article

You probably can guess from the problem title, this is the third problem in the series of [Jump Game](https://leetcode.com/problems/jump-game/) problems. Those problems are similar, but have considerable differences, making their solutions quite different.

Here, two approaches are introduced: Breadth-First Search approach and Depth-First Search approach.

#### Approach 1: Breadth-First Search

Most solutions start from a brute force approach and are optimized by removing unnecessary calculations. Same as this one.

A naive brute force approach is to iterate all the possible routes and check if there is one reaches zero. However, if we already checked one index, we do not need to check it again. We can mark the index as visited by make it negative.

|  |
| --- |
| class Solution {  public boolean canReach(int[] arr, int start) {  int n = arr.length;  Queue<Integer> q = new LinkedList<>();  q.add(start);  while (!q.isEmpty()) {  int node = q.poll();  // check if reach zero  if (arr[node] == 0) {  return true;  }  if (arr[node] < 0) {  continue;  }  // check available next steps  if (node + arr[node] < n) {  q.offer(node + arr[node]);  }  if (node - arr[node] >= 0) {  q.offer(node - arr[node]);  }  // mark as visited  arr[node] = -arr[node];  }  return false;  }  } |

**Complexity Analysis**

Assume N*N* is the length of arr.

* Time complexity: \mathcal{O}(N)O(*N*) since we will visit every index at most once.
* Space complexity : \mathcal{O}(N)O(*N*) since it needs q to store next index. In fact, q would keep at most two levels of nodes. Since we got two children for each node, the traversal of this solution is a binary tree. The maximum number of nodes within a single level for a binary tree would be \frac{N}{2}2*N*​, so the maximum length of q is \mathcal{O}(\frac{N}{2} + \frac{N}{2})=\mathcal{O}(N)O(2*N*​+2*N*​)=O(*N*).

#### Approach 2: Depth-First Search

DFS is similar to BFS but differs in the order of searching. You should consider DFS when you think of BFS.

Still, we make the value negative to mark as visited.

|  |
| --- |
| class Solution {  public boolean canReach(int[] arr, int start) {  if (start >= 0 && start < arr.length && arr[start] >= 0) {  if (arr[start] == 0) {  return true;  }  arr[start] = -arr[start];  return canReach(arr, start + arr[start]) || canReach(arr, start - arr[start]);  }  return false;  }  } |

**Complexity Analysis**

Assume N*N* is the length of arr.

* Time complexity: \mathcal{O}(N)O(*N*), since we will visit every index only once.
* Space complexity: \mathcal{O}(N)O(*N*) since it needs at most \mathcal{O}(N)O(*N*) stacks for recursions.