**October 2020 Challenge**

Week 1: October 1st - October 7th

~~+35~~

**Maximum Distance in Arrays**

~~+10~~

**Number of Recent Calls**

~~+10~~

Combination Sum

~~+10~~

**K-diff Pairs in an Array**

~~+10~~

**Remove Covered Intervals**

~~+10~~

**Complement of Base 10 Integer**

~~+10~~

Insert into a Binary Search Tree

~~+10~~

Rotate List

Week 2: October 8th - October 14th

~~+35~~

Two Sum III - Data structure design

~~+10~~

 Binary Search

~~+10~~

 Serialize and Deserialize BST

~~+10~~

**Minimum Number of Arrows to Burst Balloons**

~~+10~~

**Remove Duplicate Letters**

~~+10~~

**Buddy Strings**

~~+10~~

 Sort List

~~+10~~

**House Robber II**

Week 3: October 15th - October 21st

~~+35~~

Meeting Rooms II

~~+10~~

Rotate Array

~~+10~~

Search a 2D Matrix

~~+10~~

**Repeated DNA Sequences**

~~+10~~

**Best Time to Buy and Sell Stock IV**

~~+10~~

**Minimum Domino Rotations For Equal Row**

~~+10~~

Clone Graph

~~+10~~

**Asteroid Collision**

Week 4: October 22nd - October 28th

~~+35~~

**Search in a Sorted Array of Unknown Size**

~~+10~~

**Minimum Depth of Binary Tree**

~~+10~~

**132 Pattern**

~~+10~~

**Bag of Tokens**

~~+10~~

**Stone Game IV**

~~+10~~

**Champagne Tower**

~~+10~~

Linked List Cycle II

~~+10~~

**Summary Ranges**

Week 5: October 29th - October 31st

~~+35~~

Encode N-ary Tree to Binary Tree

~~+10~~

Maximize Distance to Closest Person

~~+10~~

**Number of Longest Increasing Subsequence**

~~+10~~

**Recover Binary Search Tree**

**Maximum Distance in Arrays**

You are given m arrays, where each array is sorted in **ascending order**.

You can pick up two integers from two different arrays (each array picks one) and calculate the distance. We define the distance between two integers a and b to be their absolute difference |a - b|.

Return *the maximum distance*.

**Example 1:**

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

**Output:** 4

**Explanation:** One way to reach the maximum distance 4 is to pick 1 in the first or third array and pick 5 in the second array.

**Example 2:**

**Input:** arrays = [[1],[1]]

**Output:** 0

**Example 3:**

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

**Output:** 1

**Example 4:**

**Input:** arrays = [[1,4],[0,5]]

**Output:** 4

**Constraints:**

* m == arrays.length
* 2 <= m <= 105
* 1 <= arrays[i].length <= 500
* -104 <= arrays[i][j] <= 104
* arrays[i] is sorted in **ascending order**.
* There will be at most 105 integers in all the arrays.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

The simplest solution is to pick up every element of every array from the list*list* and find its distance from every element in all the other arrays except itself and find the largest distance from out of those.

|  |
| --- |
| public class Solution {  public int maxDistance(int[][] list) {  int res = 0;  for (int i = 0; i < list.length - 1; i++) {  for (int j = 0; j < list[i].length; j++) {  for (int k = i + 1; k < list.length; k++) {  for (int l = 0; l < list[k].length; l++) {  res = Math.max(res, Math.abs(list[i][j] - list[k][l]));  }  }  }  }  return res;  }  } |

**Complexity Analysis**

* Time complexity : O((n\*x)^2)*O*((*n*∗*x*)2). We traverse over all the arrays in list*list* for every element of every array considered. Here, n*n* refers to the number of arrays in the list*list* and x*x* refers to the average number of elements in each array in the list*list*.
* Space complexity : O(1)*O*(1). Constant extra space is used.

#### Approach #2 Better Brute Force [Time Limit Exceeded]

**Algorithm**

In the last approach, we didn't make use of the fact that every array in the list*list* is sorted. Thus, instead of considering the distances among all the elements of all the arrays(except intra-array elements), we can consider only the distances between the first(minimum element) element of an array and the last(maximum element) element of the other arrays and find out the maximum distance from among all such distances.

|  |
| --- |
| public class Solution {  public int maxDistance(int[][] list) {  int res = 0;  for (int i = 0; i < list.length - 1; i++) {  for (int j = i + 1; j < list.length; j++) {  res = Math.max(res, Math.abs(list[i][0] - list[j][list[j].length - 1]));  res = Math.max(res, Math.abs(list[j][0] - list[i][list[i].length - 1]));  }  }  return res;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). We consider only max and min values directly for every array currenty considered. Here, n*n* refers to the number of arrays in the list*list*.
* Space complexity : O(1)*O*(1). Constant extra space is used.

Approach #3 Single Scan [Accepted]

**Algorithm**

As discussed already, in order to find out the maximum distance between any two arrays, we need not compare every element of the arrays, since the arrays are already sorted. Thus, we can consider only the extreme points in the arrays to do the distance calculations.

Further, the two points being considered for the distance calculation should not both belong to the same array. Thus, for arrays a*a* and b*b* currently chosen, we can just find the maximum out of a[n-1]-b[0]*a*[*n*−1]−*b*[0] and b[m-1]-a[0]*b*[*m*−1]−*a*[0] to find the larger distance. Here, n*n* and m*m* refer to the lengths of arrays a*a* and b*b* respectively.

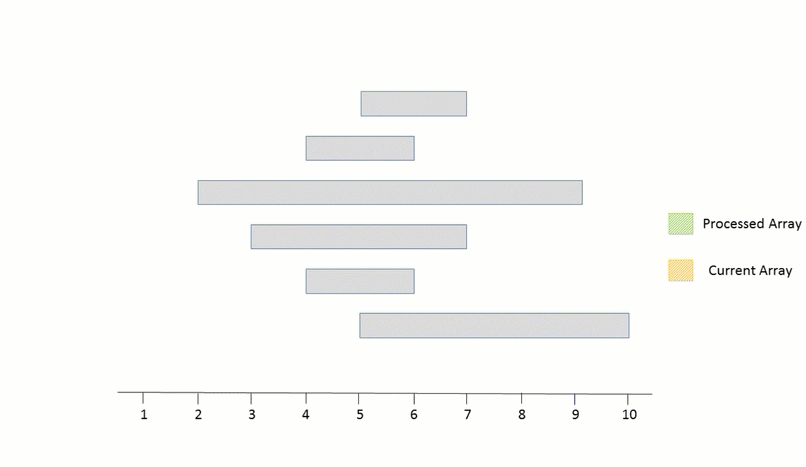
But, we need not compare all the array pairs possible to find the maximum distance. Instead, we can keep on traversing over the arrays in the list*list* and keep a track of the maximum distance found so far.

To do so, we keep a track of the element with minimum value(min\\_val*min*\_*val*) and the one with maximum value(max\\_val*max*\_*val*) found so far. Thus, now these extreme values can be treated as if they represent the extreme points of a cumulative array of all the arrays that have been considered till now.

For every new array, a*a* considered, we find the distance a[n-1]-min\\_val*a*[*n*−1]−*min*\_*val* and max\\_val - a[0]*max*\_*val*−*a*[0] to compete with the maximum distance found so far. Here, n*n* refers to the number of elements in the current array, a*a*. Further, we need to note that the maximum distance found till now needs not always be contributed by the end points of the distance being max\\_val*max*\_*val* and min\\_val*min*\_*val*.

But, such points could help in maximizing the distance in the future. Thus, we need to keep track of these maximum and minimum values along with the maximum distance found so far for future calculations. But, in general, the final maximum distance found will always be determined by one of these extreme values, max\\_val*max*\_*val* and min\\_val*min*\_*val*, or in some cases, by both of them.

The following animation illustrates the process.



From the above illustration, we can clearly see that although the max\\_val*max*\_*val* or min\\_val*min*\_*val* could not contribute to the local maximum distance values, they could later on contribute to the maximum distance.

|  |
| --- |
| public class Solution {  public int maxDistance(int[][] list) {  int res = 0, min\_val = list[0][0], max\_val = list[0][list[0].length - 1];  for (int i = 1; i < list.length; i++) {  res = Math.max(res, Math.max(Math.abs(list[i][list[i].length - 1] - min\_val), Math.abs(max\_val - list[i][0])));  min\_val = Math.min(min\_val, list[i][0]);  max\_val = Math.max(max\_val, list[i][list[i].length - 1]);  }  return res;  }  } |

\*\*Complexity Analysis\*\*

* Time complexity : O(n)*O*(*n*). We traverse over the list*list* of length n*n* once only.
* Space complexity : O(1)*O*(1). Constant extra space is used.

**Number of Recent Calls**

You have a RecentCounter class which counts the number of recent requests within a certain time frame.

Implement the RecentCounter class:

* RecentCounter() Initializes the counter with zero recent requests.
* int ping(int t) Adds a new request at time t, where t represents some time in milliseconds, and returns the number of requests that has happened in the past 3000 milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range [t - 3000, t].

It is **guaranteed** that every call to ping uses a strictly larger value of t than the previous call.

**Example 1:**

**Input**

["RecentCounter", "ping", "ping", "ping", "ping"]

[[], [1], [100], [3001], [3002]]

**Output**

[null, 1, 2, 3, 3]

**Explanation**

RecentCounter recentCounter = new RecentCounter();

recentCounter.ping(1); // requests = [1], range is [-2999,1], return 1

recentCounter.ping(100); // requests = [1, 100], range is [-2900,100], return 2

recentCounter.ping(3001); // requests = [1, 100, 3001], range is [1,3001], return 3

recentCounter.ping(3002); // requests = [1, 100, 3001, 3002], range is [2,3002], return 3

**Constraints:**

* 1 <= t <= 109
* Each test case will call ping with **strictly increasing** values of t.
* At most 104 calls will be made to ping.

## Solution

#### Overview

This problem is practical, which can test one's basic knowledge about the data structure and algorithm.

First of all, let us clarify the problem a bit. We are given a sequence of ping calls, i.e. [t\_1, t\_2, t\_3, ... t\_n][*t*1​,*t*2​,*t*3​,...*tn*​], ordered by the chronological order of their arrival time.

Given the current ping call t\_i*ti*​, we are asked to count the number of previous calls that fall in the range of [t\_i - 3000, \space t\_i][*ti*​−3000, *ti*​].

Voila. This is how we can reformulate the problem with the basic data structure such as Array. Note, the sequence of calls is ever-increasing, and we are given the call **one at a time**.

By the way, we also have a dedicated Explore card called [Array 101](https://leetcode.com/explore/learn/card/fun-with-arrays/) where one can review the characteristics and operations about Array. In addition, one will also discover many interesting problems that can be solved with Array.

#### Approach 1: Iteration over Sliding Window

**Intuition**

Now that we've clarified the nature of the problem, it shall not be difficult to come up with a solution. In fact, the solution is as simple as iterating through an array or a list.

The idea is that we can use a container such as **array** or **list** to keep track of all the incoming ping calls. At each occasion of ping(t) call, first we append the call to the container, and then starting from the current call, we **iterate backwards** to count the calls that fall into the time range of [t-3000, t].

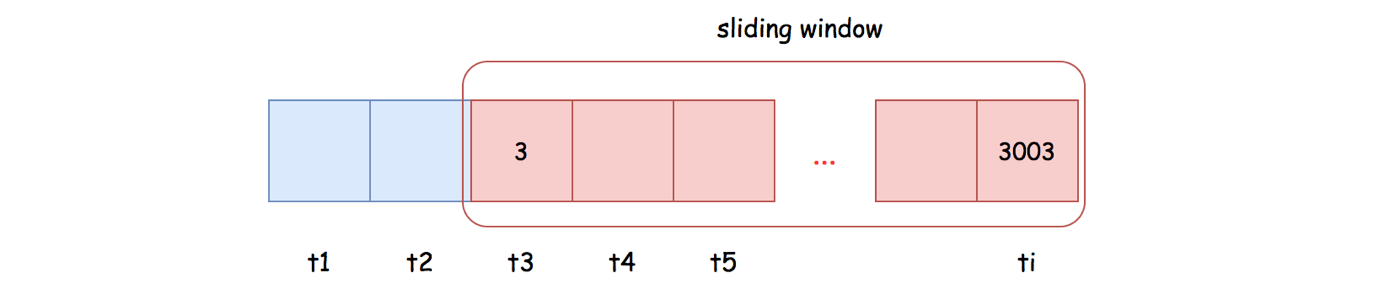
Before rushing to the implementation, let us dwell on the problem a bit more, since there are still plenty of things we could optimize.

One observation is that the sequence is ever-growing, so as our container.

On the other hand, once the ping calls become **outdated**, i.e. out of the scope of [t-3000, t], we do not need to keep them any longer in the container, since they will not contribute to the solution later.

As a result, one optimization that we could do is that rather than keeping all the **historical** ping calls in the container, we could **remove** the outdated calls on the go, which can avoid the overflow of the container and reduce the memory consumption to the least.

In summary, our container will function like a **sliding window** over the ever-growing sequence of ping calls.



Based on the above description, the **list** data structure seems to be more fit as the container for our tasks, than the **array**. Because the list is more adapted for the following two operations:

* **Appending**: we will append each incoming call to the tail of the sliding window.
* **Popping**: we need to pop out all the outdated calls from the head of the sliding window.

**Algorithm**

To implement the sliding window, we could use the LinkedList in Java or deque in Python.

Then the ping(t) function can be implemented in two steps:

* Step 1): we append the current ping call to the tail of the sliding window.
* Step 2): starting from the head of the sliding window, we remove the outdated calls, until we come across a still valid ping call.

As a result, the remaining calls in the sliding window are the ones that fall into the range of [t - 3000, t].

|  |
| --- |
| class RecentCounter {  LinkedList<Integer> slideWindow;  public RecentCounter() {  this.slideWindow = new LinkedList<Integer>();  }  public int ping(int t) {  // step 1). append the current call  this.slideWindow.addLast(t);  // step 2). invalidate the outdated pings  while (this.slideWindow.getFirst() < t - 3000)  this.slideWindow.removeFirst();  return this.slideWindow.size();  }  } |

**Complexity Analysis**

First of all, let us estimate the upper-bound on the size of our sliding window. Here we quote an important condition from the problem description: "It is guaranteed that every call to ping uses a strictly larger value of t than before." Based on the above condition, the maximal number of elements in our sliding window would be 30003000, which is also the maximal time difference between the head and the tail elements.

* Time Complexity: \mathcal{O}(1)O(1)
  + The main time complexity of our ping() function lies in the loop, which in the worst case would run 3000 iterations to pop out all outdated elements, and in the best case a single iteration.
  + Therefore, for a single invocation of ping() function, its time complexity is \mathcal{O}(3000) = \mathcal{O}(1)O(3000)=O(1).
  + If we assume that there is a ping call at each timestamp, then the cost of ping() is further amortized, where at each invocation, we would only need to pop out a single element, once the sliding window reaches its upper bound.
* Space Complexity: \mathcal{O}(1)O(1)
  + As we estimated before, the maximal size of our sliding window is 3000, which is a constant.

#### Discussion

Since the elements in our sliding window are strictly ordered, due to the condition of the problem, one might argue that it might be more efficient to use [***binary search***](https://leetcode.com/explore/learn/card/binary-search/) to locate the most recent outdated calls and then starting from that point truncate all the previous calls.

In terms of search, binary search is seemingly more efficient than our linear search. When the elements are held in the array data structure, it is true that binary search is more efficient.

However, it is not the case for the linked list, since there is no way to locate an element in the middle of a linked list instantly, which is a critical condition for binary search algorithm.

As a result, in order to apply binary search, we might have to opt for the **Array** data structure. On the other hand, once we use the array as the container, we might have to keep all the historical elements, which in the long run is not space-efficient neither time-efficient later. Or we have to find a way to efficiently remove the elements from array without frequently reallocating memory.

To conclude, it is doable to have a ***binary search*** solution. Yet, it would complicate the design, and at the end the final solution is not necessarily more efficient than the above simple LinkedList-based sliding window.

Finally, if one is interested in such a problem, there is another rather similar problem called [logger rate limiter](https://leetcode.com/problems/logger-rate-limiter/).

**K-diff Pairs in an Array**

Given an array of integers nums and an integer k, return the number of **unique** k-diff pairs in the array.

A **k-diff** pair is an integer pair (nums[i], nums[j]), where the following are true:

* 0 <= i, j < nums.length
* i != j
* |nums[i] - nums[j]| == k

**Notice** that |val| denotes the absolute value of val.

**Example 1:**

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

**Output:** 2

**Explanation:** There are two 2-diff pairs in the array, (1, 3) and (3, 5).

Although we have two 1s in the input, we should only return the number of **unique** pairs.

**Example 2:**

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

**Output:** 4

**Explanation:** There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (4, 5).

**Example 3:**

**Input:** nums = [1,3,1,5,4], k = 0

**Output:** 1

**Explanation:** There is one 0-diff pair in the array, (1, 1).

**Example 4:**

**Input:** nums = [1,2,4,4,3,3,0,9,2,3], k = 3

**Output:** 2

**Example 5:**

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

**Output:** 2

**Constraints:**

* 1 <= nums.length <= 104
* -107 <= nums[i] <= 107
* 0 <= k <= 107

## Solution

Overview: Approach 1 exhibits a naive way to tackle this problem by checking all possible pairs. Approach 2 improves the time complexity of approach 1 by using left and right pointers. Approach 3 uses Hashmap and is the fastest of all three approaches.

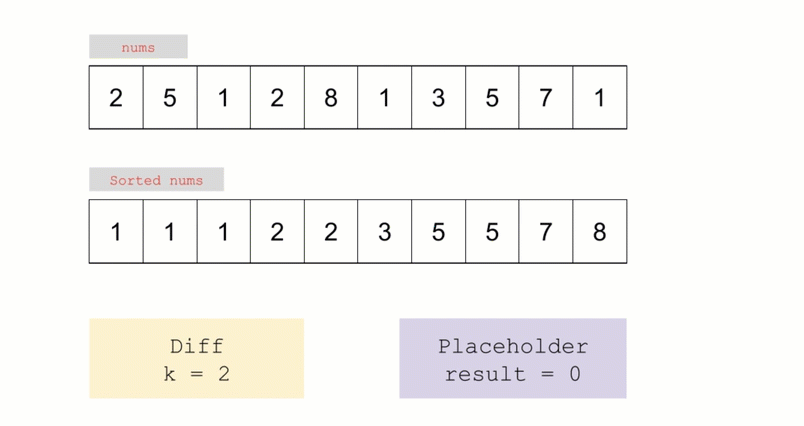
#### Approach 1: Brute Force

**Intuition**

The most naive way to tackle this problem is to sort the array and check every possible pair. We can have two loops, one loop fixing at one number while the other looping going over every number after that fixed number, to check every possible pair. One thing that we have to be aware of is to make sure that we don't repeatedly count the duplicate pairs. To do so, we will have to check whether the current number that we are looking at is the same as the previous number. If the current number is the same as the previous number, whether we are in the outer loop or the inner loop, we can just continue to the next number.

If the difference between the pair that we are looking is the same as k, we increment our placeholder variable, result.

For nums = [2,5,1,2,8,1,3,5,7,1] and k = 2:



|  |
| --- |
| import java.util.Arrays;  public class Solution {  public int findPairs(int[] nums, int k) {  Arrays.sort(nums);  int result = 0;  for (int i = 0; i < nums.length; i++) {  if (i > 0 && nums[i] == nums[i - 1])  continue;  for (int j = i + 1; j < nums.length; j++) {  if (j > i + 1 && nums[j] == nums[j - 1])  continue;  if (Math.abs(nums[j] - nums[i]) == k)  result++;  }  }  return result;  }  } |

**Complexity Analysis**

* Time complexity : O(N^2)*O*(*N*2) where N*N* is the size of nums. The time complexity for sorting is O(N \log N)*O*(*N*log*N*) while the time complexity for going through ever pair in the nums is O(N^2)*O*(*N*2). Therefore, the final time complexity is O(N \log N) + O(N^2) \approx O(N^2)*O*(*N*log*N*)+*O*(*N*2)≈*O*(*N*2).
* Space complexity : O(N)*O*(*N*) where N*N* is the size of nums. This space complexity is incurred by the sorting algorithm. Space complexity is bound to change depending on the sorting algorithm you use. There is no additional space required for the part with two for loops, apart from a single variable result. Therefore, the final space complexity is O(N) + O(1) \approx O(N)*O*(*N*)+*O*(1)≈*O*(*N*).

Addendum: We can also approach this problem using brute force without sorting nums. First, we have to create a hash set which will record pairs of numbers whose difference is k. Then, we look for every possible pair. As soon as we find a pair (say i and j) whose difference is k, we add (i, j) and (j, i) to the hash set and increment our placeholder result variable. Whenever we encounter another pair which is already in the hash set, we simply ignore that pair. By doing so we have a better practical runtime (since we are eliminating the sorting step) even though the time complexity is still O(N^2)*O*(*N*2) where N*N* is the size of nums.

#### Approach 2: Two Pointers

**Intuition**

We can do better than quadratic runtime in Approach 1. Rather than checking for every possible pair, we can have two pointers to point the left number and right number that should be checked in a sorted array.

First, we have to initialize the left pointer to point the first element and the right pointer to point the second element of nums array. The way we are going to move the pointers is as follows:

Take the difference between the numbers which left and right pointers point.

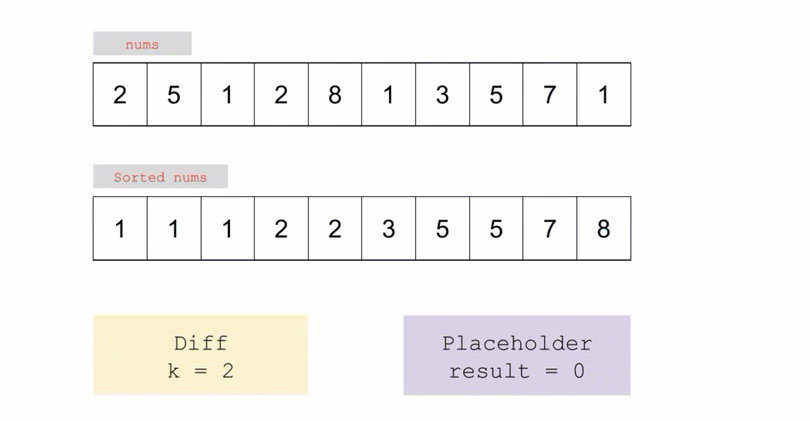
1. If it is less than k, we increment the right pointer.
   * If left and right pointers are pointing to the same number, we increment the right pointer too.
2. If it is greater than k, we increment the left pointer.
3. If it is exactly k, we have found our pair, we increment our placeholder result and increment left pointer.

The idea behind the behavior of incrementing left and right pointers in the manner above is similar to:

* Extending the range between left and right pointers when the difference between left and right pointers is less than k (i.e. the range is too small).
  + Therefore, we extend the range (by incrementing the right pointer) when left and right pointer are pointing to the same number.
* Contracting the range between left and right pointers when the difference between left and right pointers is more than k (i.e. the range is too large).

This is the core of the idea but there is another issue which we have to take care of to make everything work correctly. We have to make sure duplicate pairs are not counted repeatedly. In order to do so, whenever we have a pair whose difference matches with k, we keep incrementing the left pointer as long as the incremented left pointer points to the number which is equal to the previous number.

For nums = [2,5,1,2,8,1,3,5,7,1] and k = 2:



|  |
| --- |
| import java.util.Arrays;  public class Solution {  public int findPairs(int[] nums, int k) {  Arrays.sort(nums);  int left = 0, right = 1;  int result = 0;  while (left < nums.length && right < nums.length) {  if (left == right || nums[right] - nums[left] < k) {  // List item 1 in the text  right++;  } else if (nums[right] - nums[left] > k) {  // List item 2 in the text  left++;  } else {  // List item 3 in the text  left++;  result++;  while (left < nums.length && nums[left] == nums[left - 1])  left++;  }  }  return result;  }  } |

**Complexity Analysis**

* Time complexity : O(N \log N)*O*(*N*log*N*) where N*N* is the size of nums. The time complexity for sorting is O(N \log N)*O*(*N*log*N*) while the time complexity for going through nums is O(N)*O*(*N*). One might mistakenly think that it should be O(N^2)*O*(*N*2) since there is another while loop inside the first while loop. The while loop inside is just incrementing the pointer to skip numbers which are the same as the previous number. The animation should explain this behavior clearer. Therefore, the final time complexity is O(N \log N) + O(N) \approx O(N \log N)*O*(*N*log*N*)+*O*(*N*)≈*O*(*N*log*N*).
* Space complexity : O(N)*O*(*N*) where N*N* is the size of nums. Similar to approach 1, this space complexity is incurred by the sorting algorithm. Space complexity is bound to change depending on the sorting algorithm you use. There is no additional space required for the part where two pointers are being incremented, apart from a single variable result. Therefore, the final space complexity is O(N) + O(1) \approx O(N)*O*(*N*)+*O*(1)≈*O*(*N*).

#### Approach 3: Hashmap

**Intuition**

This method removes the need to sort the nums array. Rather than that, we will be building a frequency hash map. This hash map will have every unique number in nums as keys and the number of times each number shows up in nums as values.

For example:

nums = [2,4,1,3,5,3,1], k = 3

hash\_map = {1: 2,

2: 1,

3: 2,

4: 1,

5: 1}

Next, we look at a key (let's call x) in the hash map and ask whether:

* There is a key in the hash map which is equal to x+k **IF** k > 0.
  + For example, if a number in nums is 1 (x=1) and k is 3, you would need to have 4 to satisfy this condition (thus, we need to look for 1+3 = 4 in the hash map). Using addition to look for a complement pair has the advantage of not double-counting the same pair, but in reverse order (i.e. if we have found a pair (1,4), we won't be counting (4,1)).
* There is more than one occurrence of x **IF** k = 0.
  + For example, if we have nums = [1,1,1,1] and k = 0, we have one unique (1,1) pair. In this case, our hash map will be {1: 4}, and this condition is satisfied since we have more than one occurrence of number 1.

If we can satisfy either of the above conditions, we can increment our placeholder result variable.

Then we look at the next key in the hash map.

**Implementation**

|  |
| --- |
| public class Solution {  public int findPairs(int[] nums, int k) {  int result = 0;  HashMap <Integer,Integer> counter = new HashMap<>();  for (int n: nums) {  counter.put(n, counter.getOrDefault(n, 0)+1);  }  for (Map.Entry <Integer, Integer> entry: counter.entrySet()) {  int x = entry.getKey();  int val = entry.getValue();  if (k > 0 && counter.containsKey(x + k)) {  result++;  } else if (k == 0 && val > 1) {  result++;  }  }  return result;  }  } |

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

* Time complexity : O(N)*O*(*N*).
  + It takes O(N)*O*(*N*) to create an initial frequency hash map and another O(N)*O*(*N*) to traverse the keys of that hash map. One thing to note about is the hash key lookup. The time complexity for hash key lookup is O(1)*O*(1) but if there are hash key collisions, the time complexity will become O(N)*O*(*N*). However those cases are rare and thus, the amortized time complexity is O(2N) \approx O(N)*O*(2*N*)≈*O*(*N*).
* Space complexity : O(N)*O*(*N*)
  + We keep a table to count the frequency of each unique number in the input. In the worst case, all numbers are unique in the array. As a result, the maximum size of our table would be O(N)*O*(*N*).

**Remove Covered Intervals**

**Solution**

Given a list of intervals, remove all intervals that are covered by another interval in the list.

Interval [a,b) is covered by interval [c,d) if and only if c <= a and b <= d.

After doing so, return the number of remaining intervals.

**Example 1:**

**Input:** intervals = [[1,4],[3,6],[2,8]]

**Output:** 2

**Explanation:** Interval [3,6] is covered by [2,8], therefore it is removed.

**Example 2:**

**Input:** intervals = [[1,4],[2,3]]

**Output:** 1

**Example 3:**

**Input:** intervals = [[0,10],[5,12]]

**Output:** 2

**Example 4:**

**Input:** intervals = [[3,10],[4,10],[5,11]]

**Output:** 2

**Example 5:**

**Input:** intervals = [[1,2],[1,4],[3,4]]

**Output:** 1

**Constraints:**

* 1 <= intervals.length <= 1000
* intervals[i].length == 2
* 0 <= intervals[i][0] < intervals[i][1] <= 10^5
* All the intervals are **unique**.

 Hide Hint #1

How to check if an interval is covered by another?

   Hide Hint #2

Compare each interval to all others and check if it is covered by any interval.

## Solution

#### Approach 1: Greedy Algorithm

**Solution Pattern**

The idea of greedy algorithm is to pick the locally optimal move at each step, which would lead to the globally optimal solution.

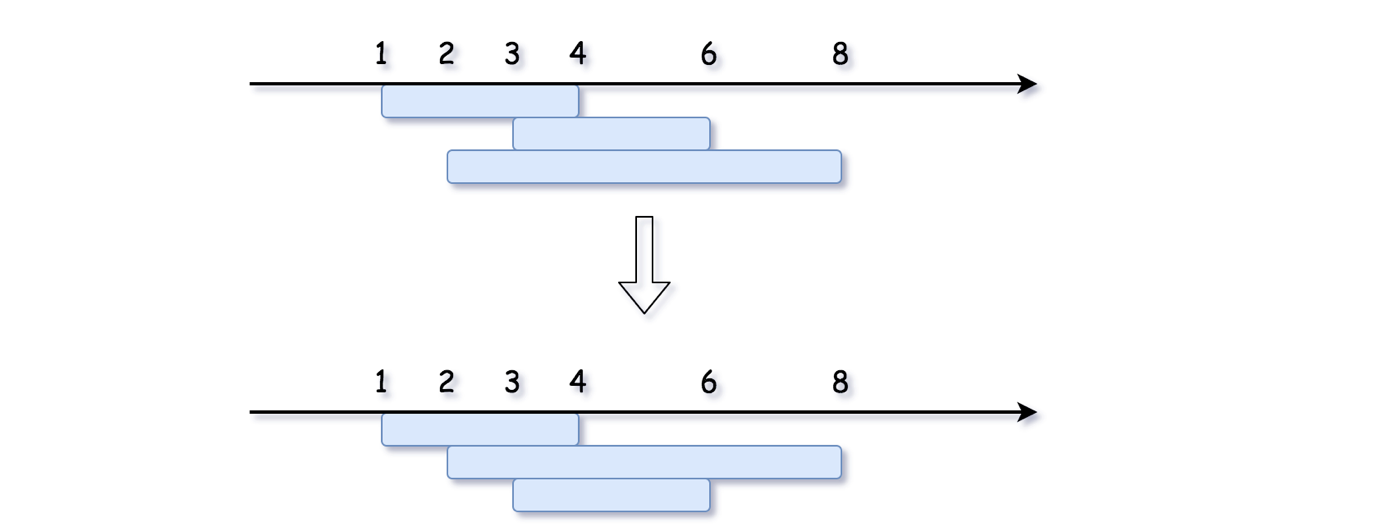
Typical greedy solution has \mathcal{O}(N \log N)O(*N*log*N*) time complexity and consists of two steps:

* Figure out how to sort the input data. That would take \mathcal{O}(N \log N)O(*N*log*N*) time, and could be done directly by sorting or indirectly by using heap data structure. Usually sorting is better than heap usage because of gain in space.
* Parse the sorted input in \mathcal{O}(N)O(*N*) time to construct a solution.

In the case of already sorted input, the greedy solution could have \mathcal{O}(N)O(*N*) time complexity, [here is an example](https://leetcode.com/articles/gas-station/).

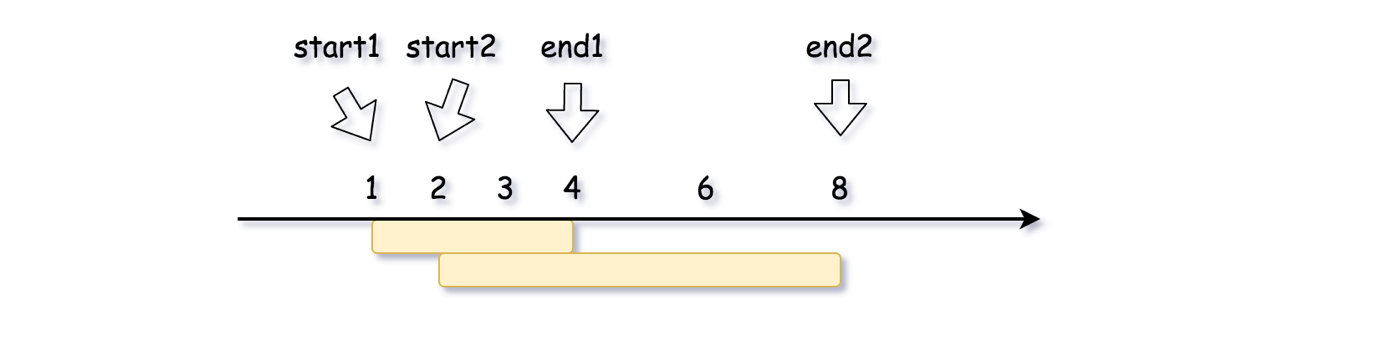
**Intuition**

Let us figure out how to sort the input. The idea to sort by start point is pretty obvious, because it simplifies further parsing:

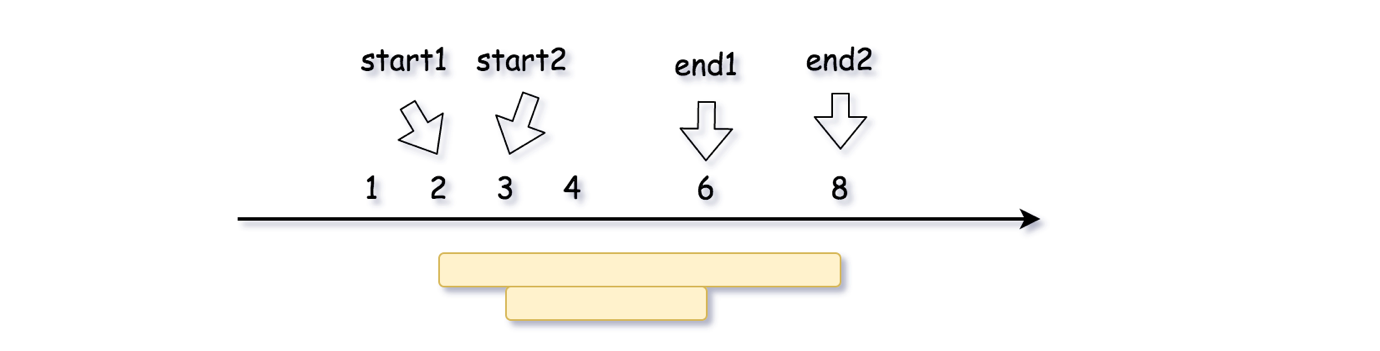


Let us consider two subsequent intervals after sorting. Since sorting ensures that start1 < start2, it's sufficient to compare the end boundaries:

* If end1 < end2, the intervals won't completely cover one another, though they have some overlapping.



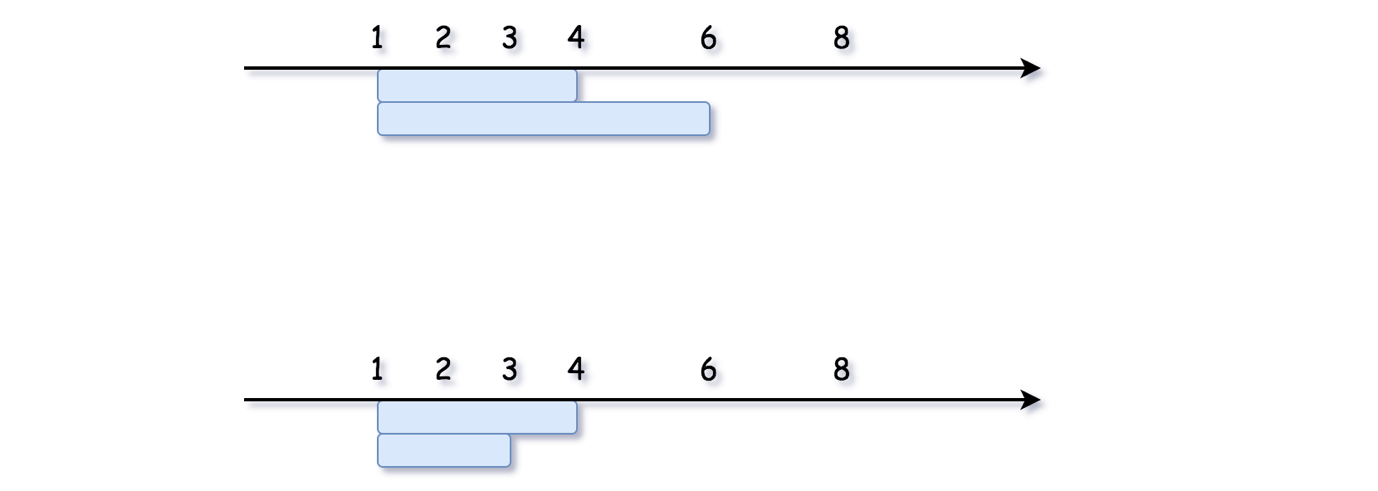
* If end1 >= end2, the interval 2 is covered by the interval 1.



**Edge case: How to treat intervals which share start point**

We've missed an important edge case in the previous discussion: what if two intervals share the start point, i.e. start1 == start2?

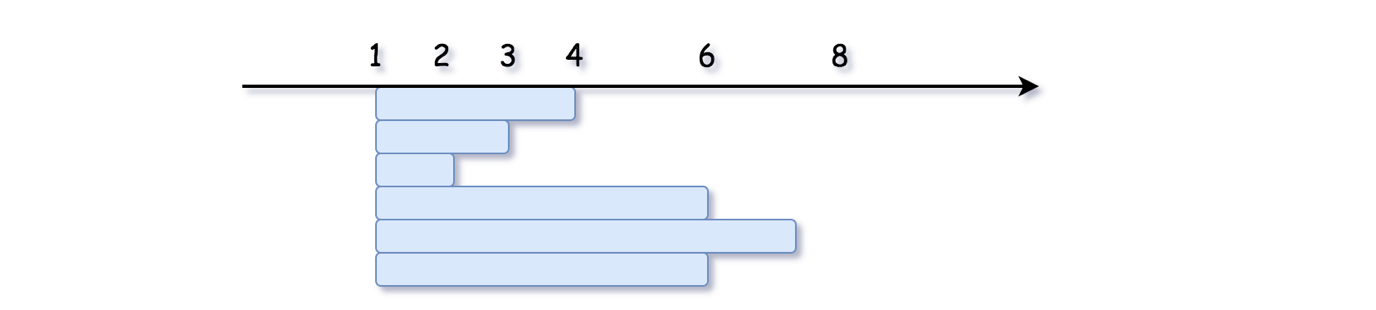
The above algorithm will fail because it cannot distinguish these two situations as follows:



One of the intervals is covered by another, but if we sort only by the start point, we would not know which one. Hence, we need to sort by the end point as well.

If two intervals share the same start point, one has to put the longer interval in front.

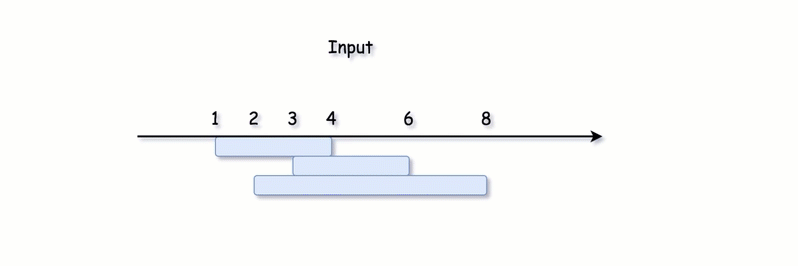
This way the above algorithm would work fine here as well. Moreover, it can deal with more complex cases, like the one below:



**Algorithm**

* Sort in the ascending order by the start point. If two intervals share the same start point, put the longer one to be the first.
* Initiate the number of non-covered intervals: count = 0.
* Iterate over sorted intervals and compare end points.
  + If the current interval is not covered by the previous one end > prev\_end, increase the number of non-covered intervals. Assign the current interval to be previous for the next step.
  + Otherwise, the current interval is covered by the previous one. Do nothing.
* Return count.

**Implementation**



|  |
| --- |
| class Solution {  public int removeCoveredIntervals(int[][] intervals) {  Arrays.sort(intervals, new Comparator<int[]>() {  @Override  public int compare(int[] o1, int[] o2) {  // Sort by start point.  // If two intervals share the same start point,  // put the longer one to be the first.  return o1[0] == o2[0] ? o2[1] - o1[1]: o1[0] - o2[0];  }  });  int count = 0;  int end, prev\_end = 0;  for (int[] curr : intervals) {  end = curr[1];  // if current interval is not covered  // by the previous one  if (prev\_end < end) {  ++count;  prev\_end = end;  }  }  return count;  }  } |

**Complexity Analysis**

* Time complexity :O(*N*log*N*) since the sorting dominates the complexity of the algorithm.
* Space complexity : \mathcal{O}(N)O(*N*) or \mathcal{O}(\log{N})O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the sorted() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is \mathcal{O}(N)O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is \mathcal{O}(\log{N})O(log*N*).

**Complement of Base 10 Integer**

Every non-negative integer N has a binary representation.  For example, 5 can be represented as "101" in binary, 11 as "1011" in binary, and so on.  Note that except for N = 0, there are no leading zeroes in any binary representation.

The complement of a binary representation is the number in binary you get when changing every 1 to a 0 and 0 to a 1.  For example, the complement of "101" in binary is "010" in binary.

For a given number N in base-10, return the complement of it's binary representation as a base-10 integer.

**Example 1:**

**Input:** 5

**Output:** 2

**Explanation:** 5 is "101" in binary, with complement "010" in binary, which is 2 in base-10.

**Example 2:**

**Input:** 7

**Output:** 0

**Explanation:** 7 is "111" in binary, with complement "000" in binary, which is 0 in base-10.

**Example 3:**

**Input:** 10

**Output:** 5

**Explanation:** 10 is "1010" in binary, with complement "0101" in binary, which is 5 in base-10.

**Note:**

1. 0 <= N < 10^9
2. This question is the same as 476: <https://leetcode.com/problems/number-complement/>

   Hide Hint #1

A binary number plus its complement will equal 111....111 in binary. Also, N = 0 is a corner case.

## Solution

#### Prerequisites

**XOR**

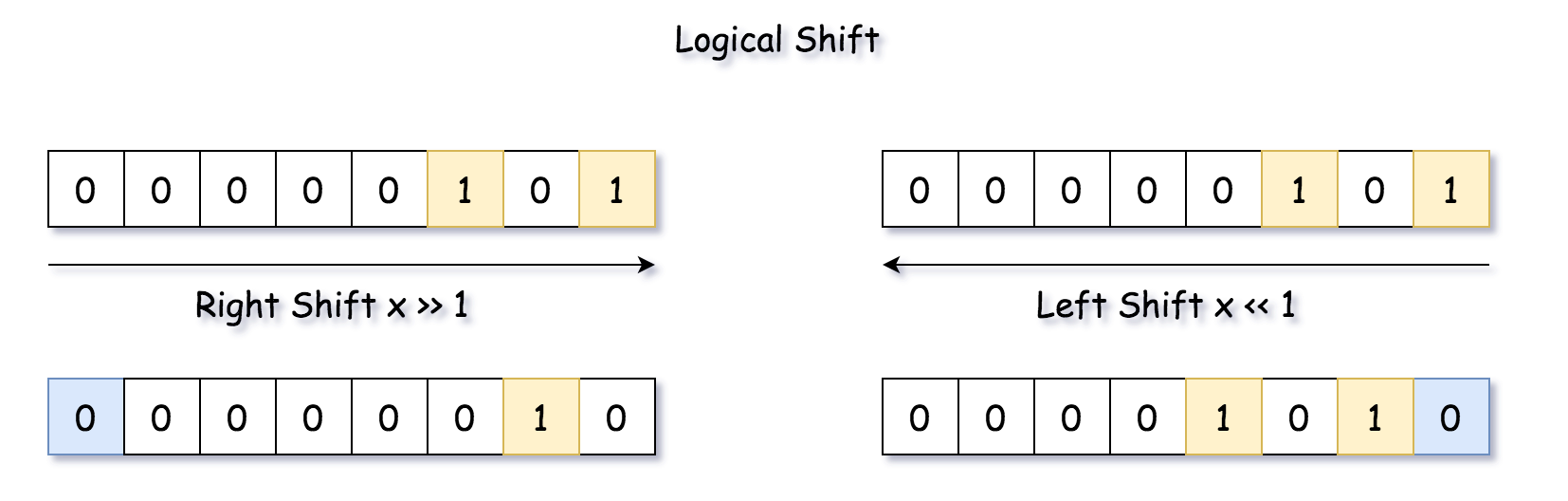
[XOR](https://en.wikipedia.org/wiki/Exclusive_or) of zero and a bit results in that bit

0 \oplus x = x0⊕*x*=*x*

XOR of one and a bit flips that bit

1 \oplus x = 1 - x1⊕*x*=1−*x*

**Right Shift and Left Shift**



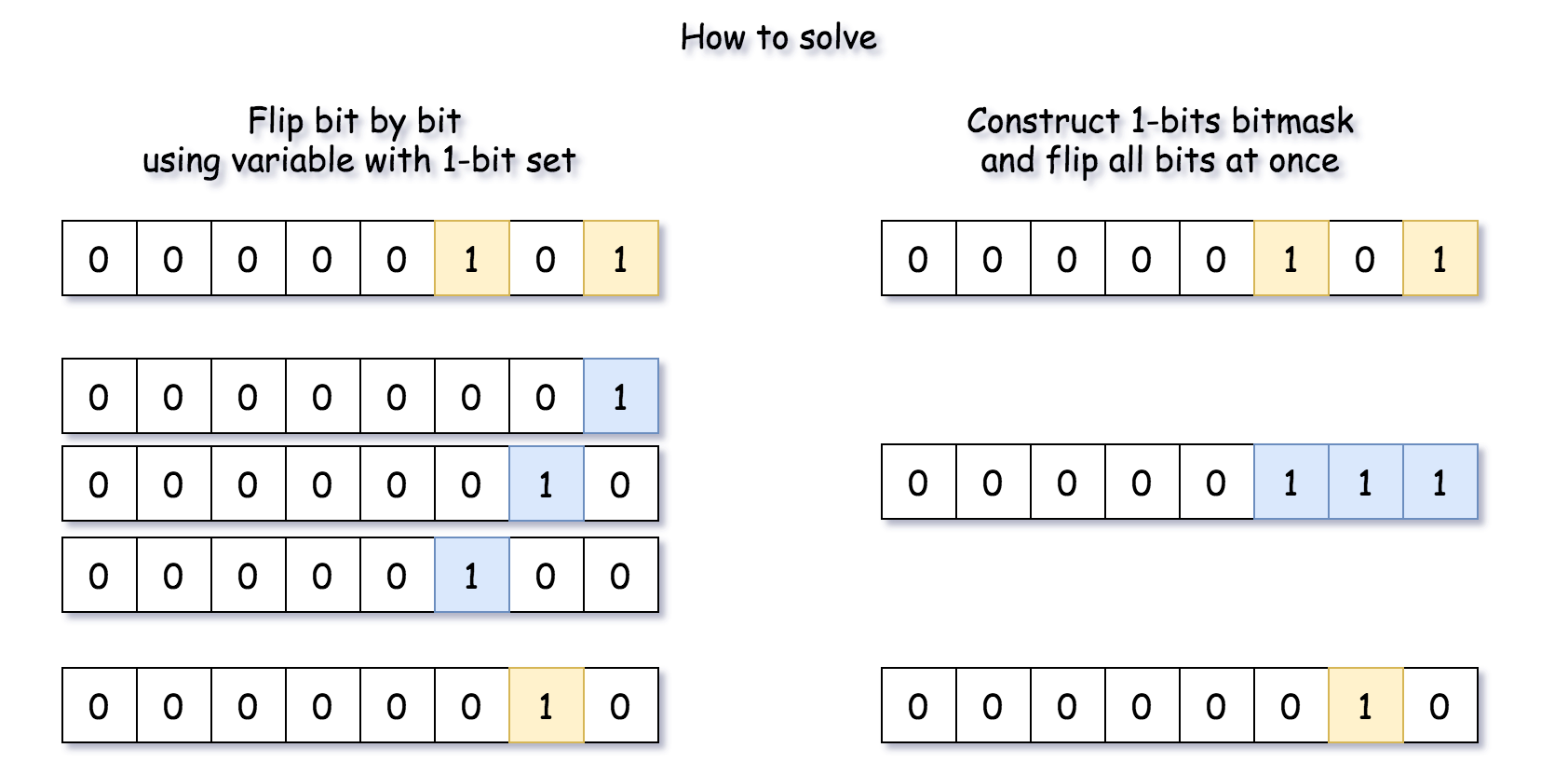
#### Overview

The article is long, and the best approach is the one number 4. In the case of limited time, you could jump to it directly.

There are two standard ways to solve the problem:

* To move along the number and flip bit by bit.
* To construct 1-bits bitmask which has the same length as the input number, and to get the answer as bitmask - num or bitmask ^ num.

For example, for \textrm{num} = 5 = (101)\_2num=5=(101)2​ the bitmask is \textrm{bitmask} = (111)\_2bitmask=(111)2​, and the complement number is \textrm{bitmask} \oplus \textrm{num} = (010)\_2 = 2bitmask⊕num=(010)2​=2.



#### Approach 1: Flip Bit by Bit

**Algorithm**

* Initiate 1-bit variable which will be used to flip bits one by one. Set it to the smallest register bit = 1.
* Initiate the marker variable which will be used to stop the loop over the bits todo = num.
* Loop over the bits. While todo != 0:
  + Flip the current bit: num = num ^ bit.
  + Prepare for the next run. Shift flip variable to the left and todo variable to the right.
* Return num.



**Implementation**

|  |
| --- |
| class Solution {  public int bitwiseComplement(int N) {  if (N == 0) return 1;  int todo = N, bit = 1;  while (todo != 0) {  // flip current bit  N = N ^ bit;  // prepare for the next run  bit = bit << 1;  todo = todo >> 1;  }  return N;  }  } |

**Complexity**

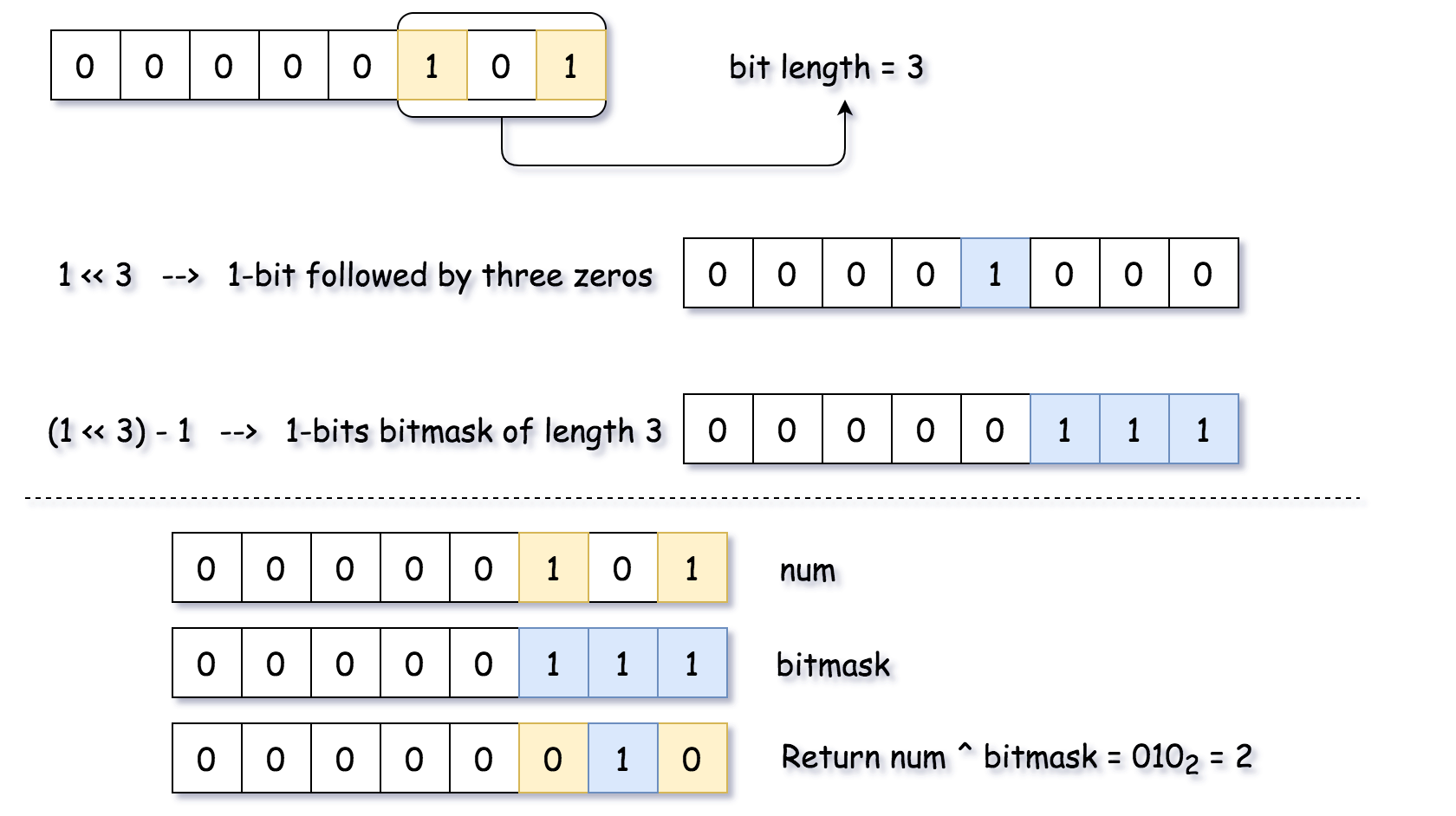
* Time Complexity: \mathcal{O}(1)O(1), since we're doing not more than 32 iterations here.
* Space Complexity: \mathcal{O}(1)O(1).

#### Approach 2: Compute Bit Length and Construct 1-bits Bitmask

Instead of flipping bits one by one, let's construct 1-bits bitmask and flip all the bits at once.

There are many ways to do it, let's start from the simplest one:

* Compute bit length of the input number l = [\log\_2 \textrm{num}] + 1*l*=[log2​num]+1.
* Compute 1-bits bitmask of length l*l*: \textrm{bitmask} = (1 << l) - 1bitmask=(1<<*l*)−1.
* Return num ^ bitmask.



**Implementation**

|  |
| --- |
| class Solution {  public int bitwiseComplement(int N) {  // l is a length of N in binary representation  int l = (int)( Math.log(N) / Math.log(2) ) + 1;  // bitmask has the same length as num and contains only ones 1...1  int bitmask = (1 << l) - 1;  // flip all bits  return bitmask ^ N;  }  } |

**Complexity**

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

#### Approach 3: Built-in Functions to Construct 1-bits Bitmask

Approach 2 could be rewritten with the help of built-in functions: bit\_length in Python and highestOneBit in Java. The first one is trivial, and Integer.highestOneBit(int x) method in Java returns int with leftmost bit set in x, i.e. Integer.highestOneBit(3) = 2.

**Implementation**

|  |
| --- |
| class Solution {  public int bitwiseComplement(int N) {  return N == 0 ? 1 : (Integer.highestOneBit(N) << 1) - N - 1;  }  } |

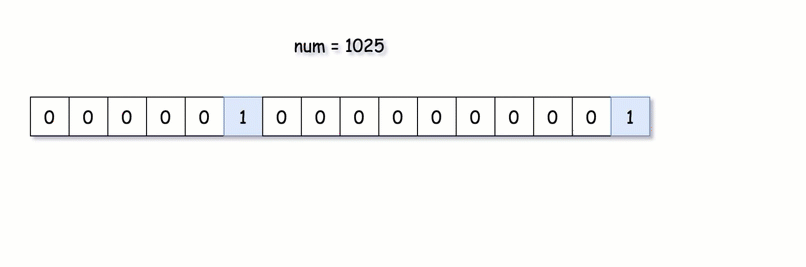
**Complexity**

* Time Complexity: \mathcal{O}(1)O(1) because one deals here with integers of not more than 32 bits.
* Space Complexity: \mathcal{O}(1)O(1).

#### Approach 4: highestOneBit OpenJDK algorithm from Hacker's Delight

The best algorithm for this task is an implementation of highestOneBit in OpenJDK. [This implementation is taken from "Hacker's Delight" book](http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/Integer.java#l40).

The idea is to create the same 1-bits bitmask by propagating the highest 1-bit into the lower ones.



|  |
| --- |
| class Solution {  public int bitwiseComplement(int N) {  if (N == 0) return 1;  // bitmask has the same length as N and contains only ones 1...1  int bitmask = N;  bitmask |= (bitmask >> 1);  bitmask |= (bitmask >> 2);  bitmask |= (bitmask >> 4);  bitmask |= (bitmask >> 8);  bitmask |= (bitmask >> 16);  // flip all bits  return bitmask ^ N;  }  } |

**Complexity**

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

**Minimum Number of Arrows to Burst Balloons**

There are some spherical balloons spread in two-dimensional space. For each balloon, provided input is the start and end coordinates of the horizontal diameter. Since it's horizontal, y-coordinates don't matter, and hence the x-coordinates of start and end of the diameter suffice. The start is always smaller than the end.

An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with xstart and xend bursts by an arrow shot at x if xstart ≤ x ≤ xend. There is no limit to the number of arrows that can be shot. An arrow once shot keeps traveling up infinitely.

Given an array points where points[i] = [xstart, xend], return the minimum number of arrows that must be shot to burst all balloons.

**Example 1:**

**Input:** points = [[10,16],[2,8],[1,6],[7,12]]

**Output:** 2

**Explanation:** One way is to shoot one arrow for example at x = 6 (bursting the balloons [2,8] and [1,6]) and another arrow at x = 11 (bursting the other two balloons).

**Example 2:**

**Input:** points = [[1,2],[3,4],[5,6],[7,8]]

**Output:** 4

**Example 3:**

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

**Output:** 2

**Example 4:**

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

**Output:** 1

**Example 5:**

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

**Output:** 1

**Constraints:**

* 0 <= points.length <= 104
* points[i].length == 2
* -231 <= xstart < xend <= 231 - 1

## Solution

#### Approach 1: Greedy

**Greedy algorithms**

Greedy problems usually look like "Find minimum number of something to do something" or "Find maximum number of something to fit in some conditions", and typically propose an unsorted input.

The idea of greedy algorithm is to pick the locally optimal move at each step, that will lead to the globally optimal solution.

The standard solution has \mathcal{O}(N \log N)O(*N*log*N*) time complexity and consists of two parts:

* Figure out how to sort the input data (\mathcal{O}(N \log N)O(*N*log*N*) time). That could be done directly by a sorting or indirectly by a heap usage. Typically sort is better than the heap usage because of gain in space.
* Parse the sorted input to have a solution (\mathcal{O}(N)O(*N*) time).

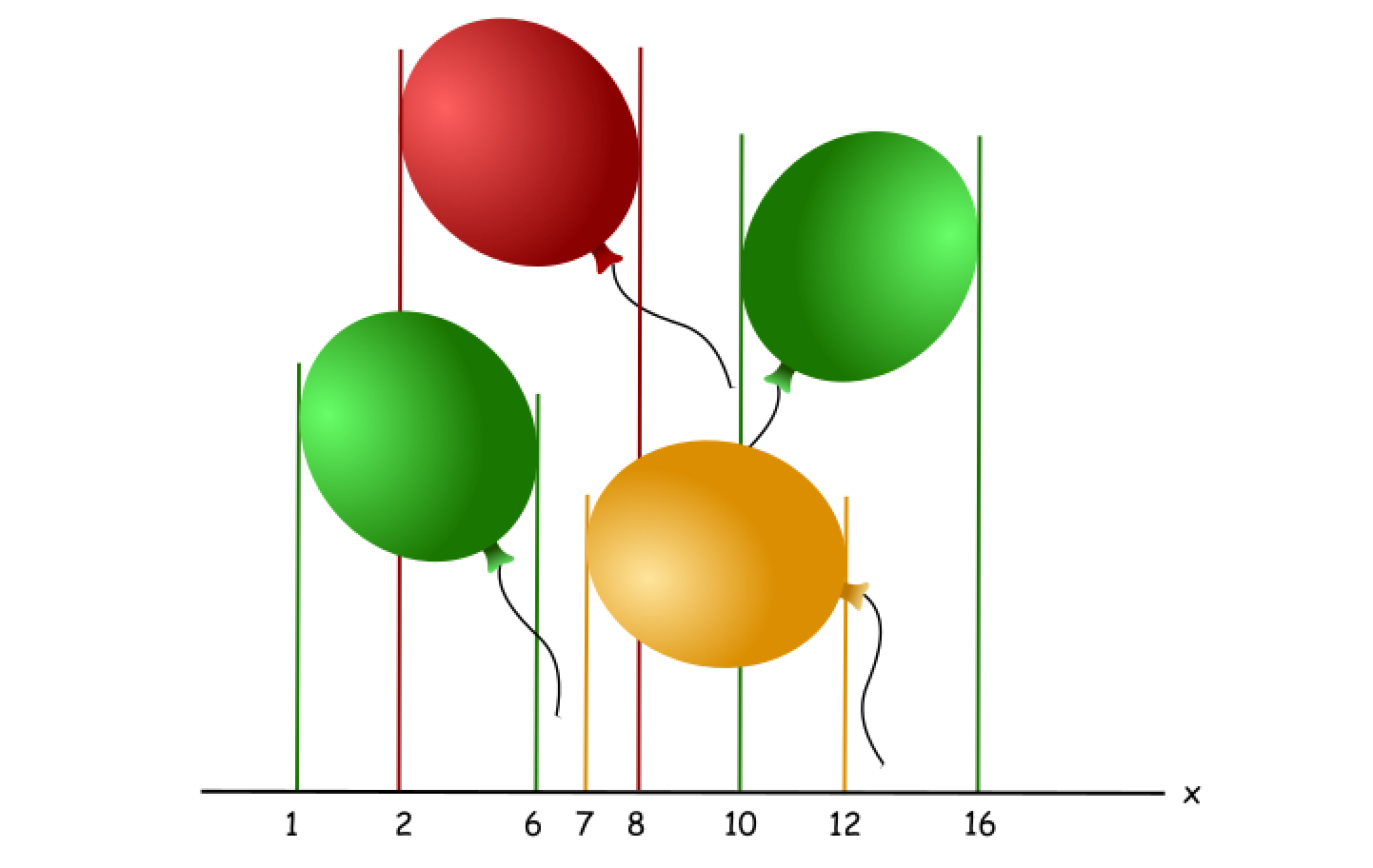
Please notice that in case of well-sorted input one doesn't need the first part and the greedy solution could have \mathcal{O}(N)O(*N*) time complexity, [here is an example](https://leetcode.com/articles/gas-station/).

How to prove that your greedy algorithm provides globally optimal solution?

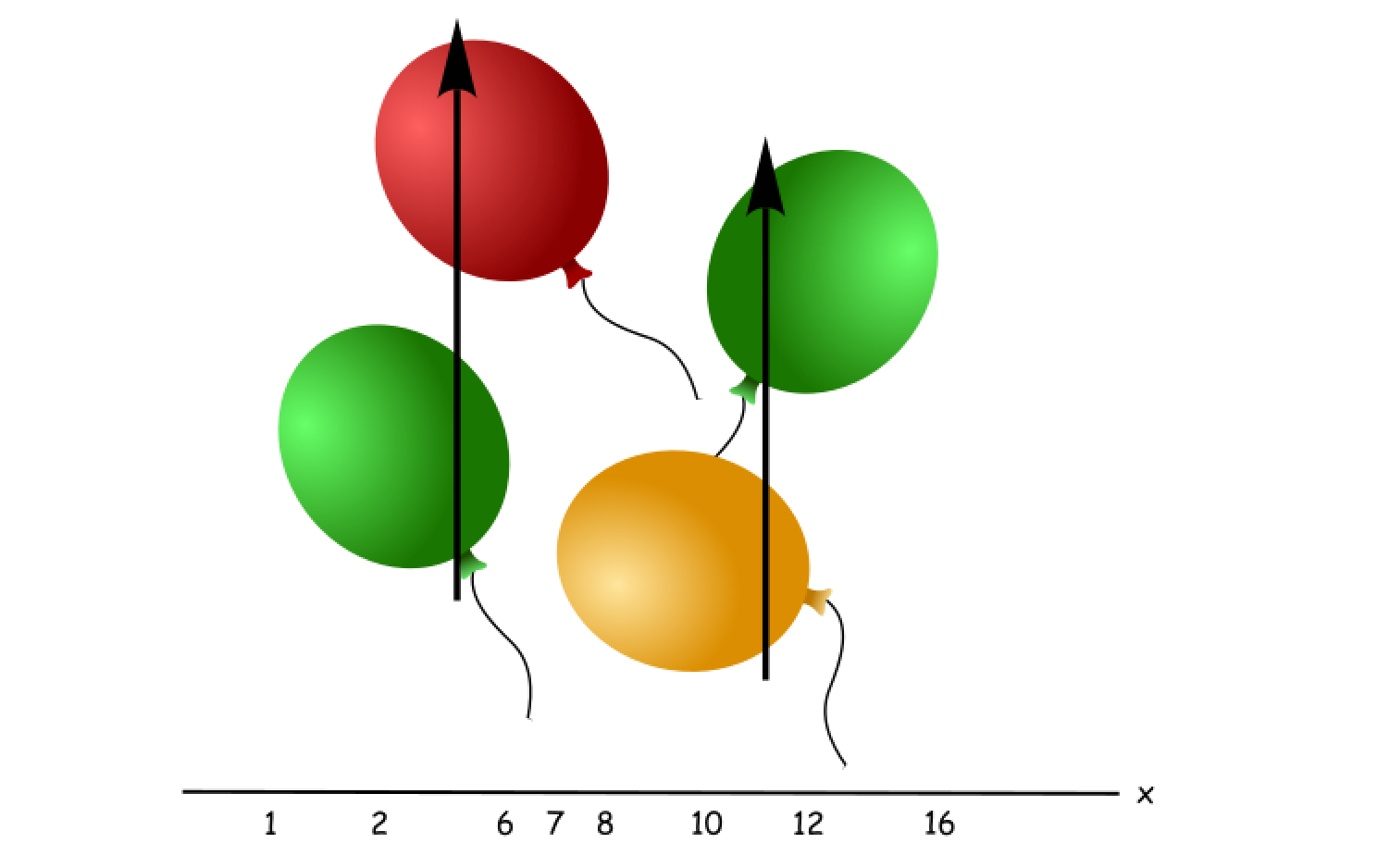
Usually you could use the [proof by contradiction](https://en.wikipedia.org/wiki/Proof_by_contradiction).

**Intuition**

Let's consider the following combinations of the balloons.



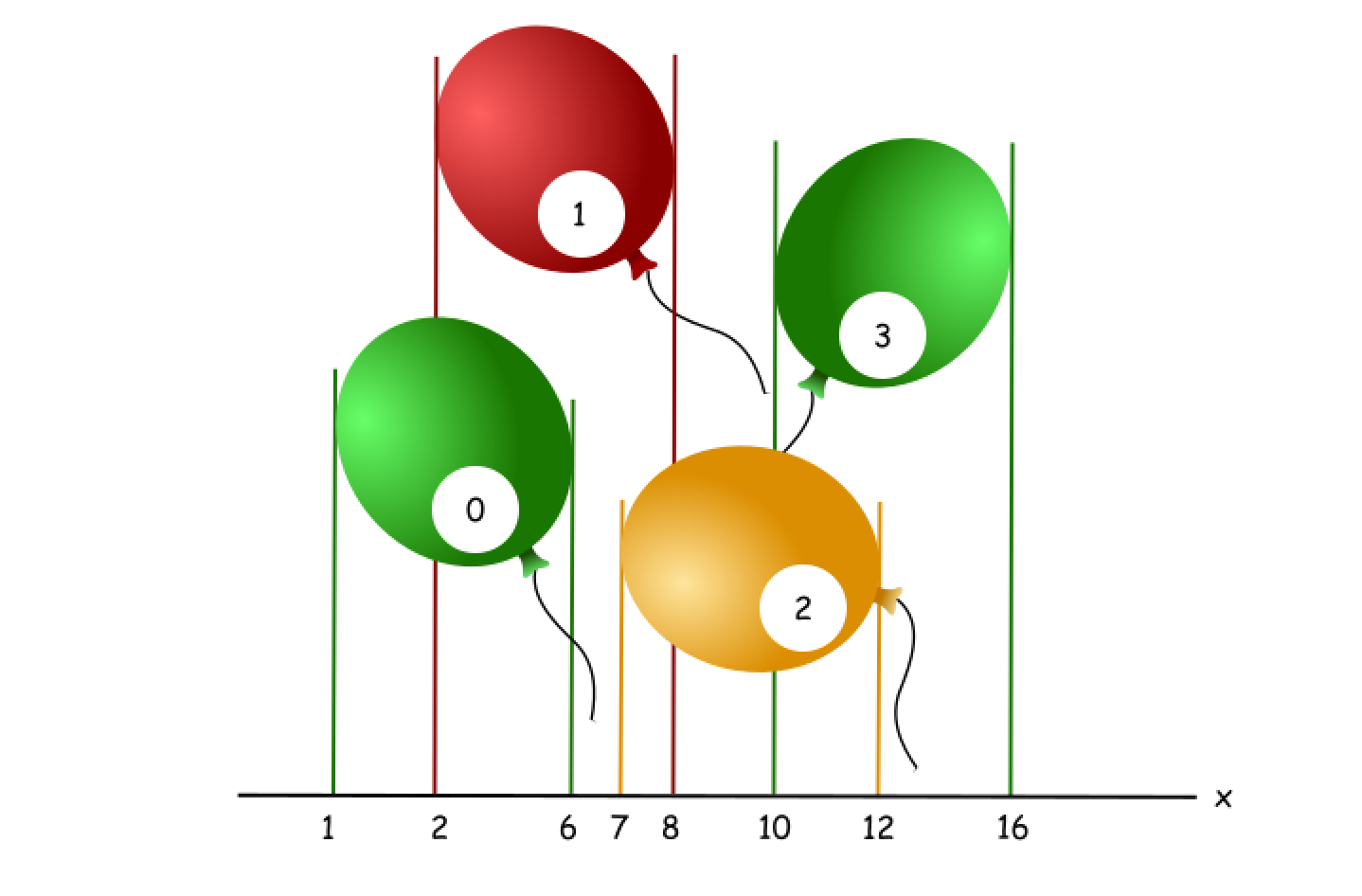
That's quite obvious that two arrows is enough to burst them all, let's figure out how to compute this result with the help of greedy algorithm.



Let's sort the balloons by the end coordinate, and then check them one by one. The first balloon is a green one number 0, it ends at coordinate 6, and there is no balloons ending before it because of sorting.

The other balloons have two possibilities :

* To have a start coordinate smaller than 6, like a red balloon. These ones could be burst together with the balloon 0 by one arrow.
* To have a start coordinate larger than 6, like a yellow balloon. These ones couldn't be burst together with the balloon 0 by one arrow, and hence one needs to increase the number of arrows here.



That means that one could always track the end of the current balloon, and ignore all the balloons which end before it. Once the current balloon is ended (= the next balloon starts after the current balloon), one has to increase the number of arrows by one and start to track the end of the next balloon.

**Algorithm**

Now the algorithm is straightforward :

* Sort the balloons by end coordinate x\_end.
* Initiate the end coordinate of a balloon which ends first : first\_end = points[0][1].
* Initiate number of arrows: arrows = 1.
* Iterate over all balloons:
  + If the balloon starts after first\_end:
    - Increase the number of arrows by one.
    - Set first\_end to be equal to the end of the current balloon.
* Return arrows.

**Implementation**

|  |
| --- |
| class Solution {  public int findMinArrowShots(int[][] points) {  if (points.length == 0) return 0;  // sort by x\_end  Arrays.sort(points, (o1, o2) -> {  // We can't simply use the o1[1] - o2[1] trick, as this will cause an  // integer overflow for very large or small values.  if (o1[1] == o2[1]) return 0;  if (o1[1] < o2[1]) return -1;  return 1;  });  int arrows = 1;  int xStart, xEnd, firstEnd = points[0][1];  for (int[] p: points) {  xStart = p[0];  xEnd = p[1];  // if the current balloon starts after the end of another one,  // one needs one more arrow  if (firstEnd < xStart) {  arrows++;  firstEnd = xEnd;  }  }  return arrows;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N \log N)O(*N*log*N*) because of sorting of input data.
* Space complexity : \mathcal{O}(N)O(*N*) or \mathcal{O}(\log{N})O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the list.sort() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is \mathcal{O}(N)O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is \mathcal{O}(\log{N})O(log*N*).

**Remove Duplicate Letters**

Given a string s, remove duplicate letters so that every letter appears once and only once. You must make sure your result is **the smallest in lexicographical order** among all possible results.

**Note:** This question is the same as 1081: <https://leetcode.com/problems/smallest-subsequence-of-distinct-characters/>

**Example 1:**

**Input:** s = "bcabc"

**Output:** "abc"

**Example 2:**

**Input:** s = "cbacdcbc"

**Output:** "acdb"

**Constraints:**

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

   Hide Hint #1

Greedily try to add one missing character. How to check if adding some character will not cause problems ? Use bit-masks to check whether you will be able to complete the sub-sequence if you add the character at some index i.

## Solution

#### Intuition

First we should make sure we understand what "lexicographical order" means. Comparing strings doesn't work the same way as comparing numbers. Strings are compared from the first character to the last one. Which string is greater depends on the comparison between the first unequal corresponding character in the two strings. As a result any string beginning with a will always be less than any string beginning with b, regardless of the ends of both strings.

Because of this, the optimal solution will have the smallest characters as early as possible. We draw two conclusions that provide different methods of solving this problem in O(N)*O*(*N*):

1. The leftmost letter in our solution will be the smallest letter such that the suffix from that letter contains every other. This is because we know that the solution must have one copy of every letter, and we know that the solution will have the lexicographically smallest leftmost character possible.

If there are multiple smallest letters, then we pick the leftmost one simply because it gives us more options. We can always eliminate more letters later on, so the optimal solution will always remain in our search space.

1. As we iterate over our string, if character i is greater than character i+1 and another occurrence of character i exists later in the string, deleting character i will **always** lead to the optimal solution. Characters that come later in the string i don't matter in this calculation because i is in a more significant spot. Even if character i+1 isn't the best yet, we can always replace it for a smaller character down the line if possible.

Since we try to remove characters as early as possible, and picking the best letter at each step leads to the best solution, "greedy" should be going off like an alarm.

#### Approach 1: Greedy - Solving Letter by Letter

**Algorithm**

We use idea number one from the intuition. In each iteration, we determine leftmost letter in our solution. This will be **the smallest character such that its suffix contains at least one copy of every character in the string**. We determine the rest our answer by recursively calling the function on the suffix we generate from the original string (leftmost letter is removed).

**Implementation**

|  |
| --- |
| public class Solution {  public String removeDuplicateLetters(String s) {  // find pos - the index of the leftmost letter in our solution  // we create a counter and end the iteration once the suffix doesn't have each unique character  // pos will be the index of the smallest character we encounter before the iteration ends  int[] cnt = new int[26];  int pos = 0;  for (int i = 0; i < s.length(); i++) cnt[s.charAt(i) - 'a']++;  for (int i = 0; i < s.length(); i++) {  if (s.charAt(i) < s.charAt(pos)) pos = i;  if (--cnt[s.charAt(i) - 'a'] == 0) break;  }  // our answer is the leftmost letter plus the recursive call on the remainder of the string  // note that we have to get rid of further occurrences of s[pos] to ensure that there are no duplicates  return s.length() == 0 ? "" : s.charAt(pos) + removeDuplicateLetters(s.substring(pos + 1).replaceAll("" + s.charAt(pos), ""));  }  } |

Note that the code in this section is a translated / commented version of the code [in this post](https://leetcode.com/problems/remove-duplicate-letters/discuss/76768/A-short-O(n)-recursive-greedy-solution) originally written by [lixx2100](https://leetcode.com/lixx2100/).

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*). Each recursive call will take O(N)*O*(*N*). The number of recursive calls is bounded by a constant (26 letters in the alphabet), so we have O(N) \* C = O(N)*O*(*N*)∗*C*=*O*(*N*).
* Space complexity : O(N)*O*(*N*). Each time we slice the string we're creating a new one (strings are immutable). The number of slices is bound by a constant, so we have O(N) \* C = O(N)*O*(*N*)∗*C*=*O*(*N*).

#### Approach 2: Greedy - Solving with Stack

**Algorithm**

We use idea number two from the intuition. We will keep a stack to store the solution we have built as we iterate over the string, and we will delete characters off the stack whenever it is possible and it makes our string smaller.

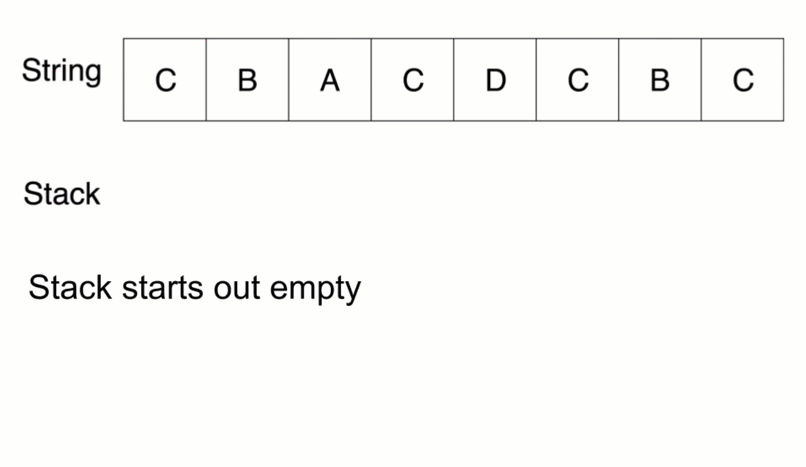
Each iteration we add the current character to the solution if it hasn't already been used. We try to remove as many characters as possible off the top of the stack, and then add the current character

The conditions for deletion are:

1. The character is greater than the current characters
2. The character can be removed because it occurs later on

At each stage in our iteration through the string, we greedily keep what's on the stack as small as possible.

The following animation makes this more clear:



|  |
| --- |
| class Solution {  public String removeDuplicateLetters(String s) {  Stack<Character> stack = new Stack<>();  // this lets us keep track of what's in our solution in O(1) time  HashSet<Character> seen = new HashSet<>();  // this will let us know if there are any more instances of s[i] left in s  HashMap<Character, Integer> last\_occurrence = new HashMap<>();  for(int i = 0; i < s.length(); i++) last\_occurrence.put(s.charAt(i), i);  for(int i = 0; i < s.length(); i++){  char c = s.charAt(i);  // we can only try to add c if it's not already in our solution  // this is to maintain only one of each character  if (!seen.contains(c)){  // if the last letter in our solution:  // 1. exists  // 2. is greater than c so removing it will make the string smaller  // 3. it's not the last occurrence  // we remove it from the solution to keep the solution optimal  while(!stack.isEmpty() && c < stack.peek() && last\_occurrence.get(stack.peek()) > i){  seen.remove(stack.pop());  }  seen.add(c);  stack.push(c);  }  }  StringBuilder sb = new StringBuilder(stack.size());  for (Character c : stack) sb.append(c.charValue());  return sb.toString();  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*). Although there is a loop inside a loop, the time complexity is still O(N)*O*(*N*). This is because the inner while loop is bounded by the total number of elements added to the stack (each time it fires an element goes). This means that the total amount of time spent in the inner loop is bounded by O(N)*O*(*N*), giving us a total time complexity of O(N)*O*(*N*)
* Space complexity : O(1)*O*(1). At first glance it looks like this is O(N)*O*(*N*), but that is not true! seen will only contain unique elements, so it's bounded by the number of characters in the alphabet (a constant). You can only add to stack if an element has not been seen, so stack also only consists of unique elements. This means that both stack and seen are bounded by constant, giving us O(1)*O*(1) space complexity.

**Buddy Strings**

Given two strings A and B of lowercase letters, return true if you can swap two letters in A so the result is equal to B, otherwise, return false.

Swapping letters is defined as taking two indices i and j (0-indexed) such that i != j and swapping the characters at A[i] and A[j]. For example, swapping at indices 0 and 2 in "abcd" results in "cbad".

**Example 1:**

**Input:** A = "ab", B = "ba"

**Output:** true

**Explanation:** You can swap A[0] = 'a' and A[1] = 'b' to get "ba", which is equal to B.

**Example 2:**

**Input:** A = "ab", B = "ab"

**Output:** false

**Explanation:** The only letters you can swap are A[0] = 'a' and A[1] = 'b', which results in "ba" != B.

**Example 3:**

**Input:** A = "aa", B = "aa"

**Output:** true

**Explanation:** You can swap A[0] = 'a' and A[1] = 'a' to get "aa", which is equal to B.

**Example 4:**

**Input:** A = "aaaaaaabc", B = "aaaaaaacb"

**Output:** true

**Example 5:**

**Input:** A = "", B = "aa"

**Output:** false

**Constraints:**

* 0 <= A.length <= 20000
* 0 <= B.length <= 20000
* A and B consist of lowercase letters.

## Solution

#### Approach 1: Enumerate Cases

**Intuition**

If the characters at the index of i in both strings are identical, i.e. A[i] == B[i], we call the characters at the index i as matched.

If swapping A[i] and A[j] would demonstrate that A and B are buddy strings, then A[i] == B[j] and A[j] == B[i]. That means among the four free variables A[i], A[j], B[i], B[j], there are only two cases: either A[i] == A[j] or not.

**Algorithm**

Let's work through the cases.

In the case A[i] == A[j] == B[i] == B[j], then the strings A and B are equal. So if A == B, we should check each index i for two matches with the same value.

In the case A[i] == B[j], A[j] == B[i], (A[i] != A[j]), the rest of the indices match. So if A and B have only two unmatched indices (say i and j), we should check that the equalities A[i] == B[j] and A[j] == B[i] hold.

|  |
| --- |
| class Solution {  public boolean buddyStrings(String A, String B) {  if (A.length() != B.length()) return false;  if (A.equals(B)) {  int[] count = new int[26];  for (int i = 0; i < A.length(); ++i)  count[A.charAt(i) - 'a']++;  for (int c: count)  if (c > 1) return true;  return false;  } else {  int first = -1, second = -1;  for (int i = 0; i < A.length(); ++i) {  if (A.charAt(i) != B.charAt(i)) {  if (first == -1)  first = i;  else if (second == -1)  second = i;  else  return false;  }  }  return (second != -1 && A.charAt(first) == B.charAt(second) &&  A.charAt(second) == B.charAt(first));  }  }  } |

**Complexity Analysis**

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

**House Robber II**

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers nums representing the amount of money of each house, return the maximum amount of money you can rob tonight ***without alerting the police***.

**Example 1:**

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

**Output:** 3

**Explanation:** You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

**Example 2:**

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

**Output:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

**Example 3:**

**Input:** nums = [0]

**Output:** 0

**Constraints:**

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

   Hide Hint #1

Since House[1] and House[n] are adjacent, they cannot be robbed together. Therefore, the problem becomes to rob either House[1]-House[n-1] or House[2]-House[n], depending on which choice offers more money. Now the problem has degenerated to the [House Robber](https://leetcode.com/problems/house-robber/description/), which is already been solved.

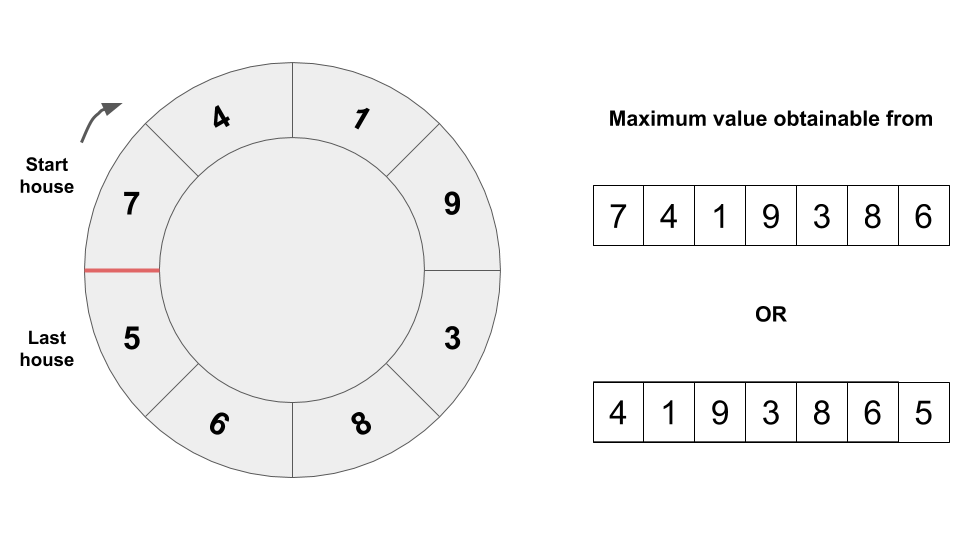
## Solution

This problem is a minor extension to the original [House Robber Problem](https://leetcode.com/problems/house-robber/). The only difference is that the first and the last houses are adjacent to each other and therefore, if the thief has robbed the first house, they cannot steal the last house and vice versa. As Hint 1 in the question suggests, "the problem becomes to rob either House[1]-House[n-1] or House[2]-House[n], depending on which choice offers more money. Now the problem has degenerated to the House Robber".

#### Approach 1: Dynamic Programming

**Intuition**

Assume we have nums of [7,4,1,9,3,8,6,5] as shown in the figure. Since the start house and last house are adjacent to each other, if the thief decides to rob the start house 7, they cannot rob the last house 5. Similarly, if they select last house 5, they have to start from a house with value 4. Therefore, the final solution that we are looking for is to take the maximum value the thief can rob between houses of list [7,4,1,9,3,8,6] and [4,1,9,3,8,6,5]. For each of the lists, all we need to do is to figure the maximum value the thief can get using the approach in the original [House Robber Problem](https://leetcode.com/problems/house-robber/).



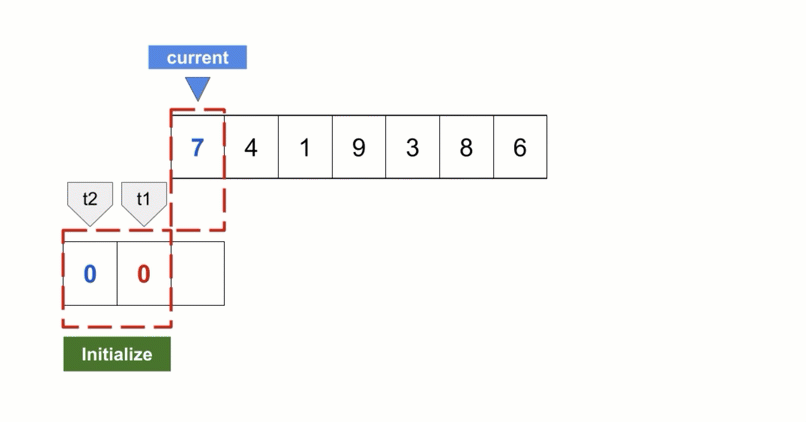
**Solving Original House Robber Problem with Dynamic Programming**

Trivial cases:

* If there is one house, the answer is the value of that house.
* If there are two houses, the answer is max(house1, house2).
* If there are three houses, you can either pick the middle house or the sum of the first and the last house. Therefore, it boils down to max(house3 + house1, house2).

To make the example more illustrative, imagine two thieves (t1 and t2) coordinating a grand robbery. They are equipped with walkie-talkies to communicate the values of houses to each other.

* Before entering any of the houses, both t1 and t2 have values of zero.
* t1, enters the first house and record the value of the house. If that is the only house to rob, they can rob this house and be done with it.
* If there is more than one house, t1 will leave a note of maximum value reaped until this point (which is just the value of the first house) and move to the next house while t2 moves into the house t1 was in. Now, t1 and t2 are going to communicate over the walkie-talkie to ask who has the most value. At this point, t2 will read the note left by t1 when the values are compared. If they have only two houses to rob, they would rob the house with the most value and be done with it.
* If there are three houses, t1 will leave a note of the maximum value reaped until this point and move to the next house. Then t1 will compare the value of the sum of the current house and the house which t2 is in with the value of the house t1 was in. The maximum value between those two will be chosen and t2 will move into the house next to it.
* If there are four houses, t1 will leave a note of the maximum value reaped until this point and move to the next house. Then t1 will compare the value of the sum of the current house and the house which t2 is in with the value of the house t1 was in. The maximum value between those two will be chosen and t2 will move into the house next to it.
* This procedure is done over and over again as long as there are houses in nums. If t1 has reached to the end of nums, t1 should have reaped the maximum amount obtainable from houses in nums.



|  |
| --- |
| class Solution {  public int rob(int[] nums) {  if (nums.length == 0)  return 0;  if (nums.length == 1)  return nums[0];  int max1 = rob\_simple(nums, 0, nums.length - 2);  int max2 = rob\_simple(nums, 1, nums.length - 1);  return Math.max(max1, max2);  }  public int rob\_simple(int[] nums, int start, int end) {  int t1 = 0;  int t2 = 0;  for (int i = start; i <= end; i++) {  int temp = t1;  int current = nums[i];  t1 = Math.max(current + t2, t1);  t2 = temp;  }  return t1;  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*) where N*N* is the size of nums. We are accumulating results as we are scanning nums.
* Space complexity : O(1)*O*(1) since we are not consuming additional space other than variables for two previous results and a temporary variable to hold one of the previous results.

**Repeated DNA Sequences**

All DNA is composed of a series of nucleotides abbreviated as 'A', 'C', 'G', and 'T', for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

**Example 1:**

**Input:** s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

**Output:** ["AAAAACCCCC","CCCCCAAAAA"]

**Example 2:**

**Input:** s = "AAAAAAAAAAAAA"

**Output:** ["AAAAAAAAAA"]

**Constraints:**

* 0 <= s.length <= 105
* s[i] is 'A', 'C', 'G', or 'T'.

## Solution

#### Overview

Follow-up here is to solve the same problem for arbitrary sequence length L*L*, and to check the situation when L*L* is quite large. Hence let's use L = 10*L*=10 notation everywhere to ease the problem generalisation.

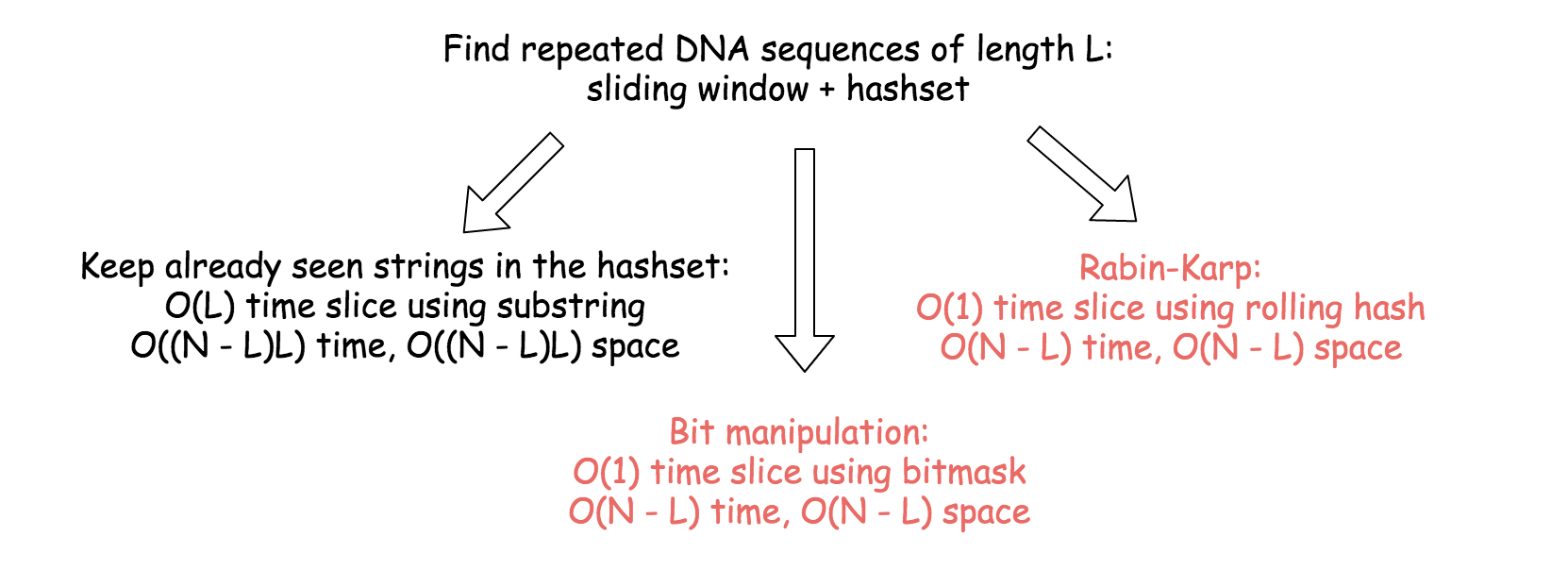
We will discuss three different ideas how to proceed. They are all based on sliding window + hashset. The key point is how to implement a window slice.

Linear-time window slice \mathcal{O}(L)O(*L*) is easy stupid, just take a substring. Overall that would result in \mathcal{O}((N - L) L)O((*N*−*L*)*L*) time complexity and huge space consumption in the case of large sequences.

Constant-time slice \mathcal{O}(1)O(1) is where the fun starts, because the way you choose will show your actual background. There are two ways to proceed:

* Rabin-Karp = constant-time slice using rolling hash algorithm.
* Bit manipulation = constant-time slice using bitmasks.

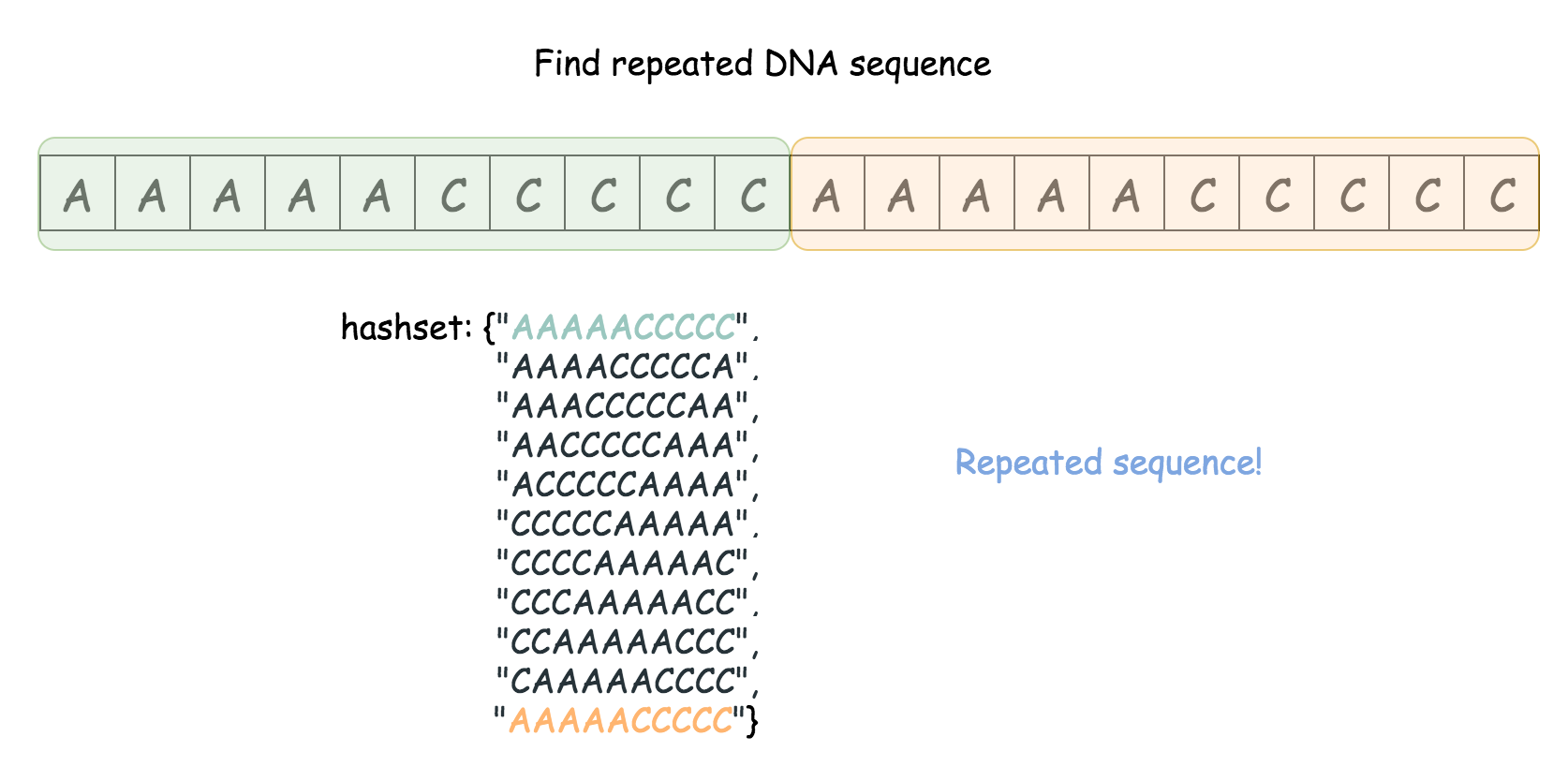
Last two approaches have \mathcal{O}(N - L)O(*N*−*L*) time complexity and moderate space consumption even in the case of large sequences.



#### Approach 1: Linear-time Slice Using Substring + HashSet

The idea is straightforward :

* Move a sliding window of length L along the string of length N.
* Check if the sequence in the sliding window is in the hashset of already seen sequences.
  + If yes, the repeated sequence is right here. Update the output.
  + If not, save the sequence in the sliding window in the hashset.



|  |
| --- |
| class Solution {  public List<String> findRepeatedDnaSequences(String s) {  int L = 10, n = s.length();  HashSet<String> seen = new HashSet(), output = new HashSet();  // iterate over all sequences of length L  for (int start = 0; start < n - L + 1; ++start) {  String tmp = s.substring(start, start + L);  if (seen.contains(tmp)) output.add(tmp);  seen.add(tmp);  }  return new ArrayList<String>(output);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}((N - L)L)O((*N*−*L*)*L*), that is \mathcal{O}(N)O(*N*) for the constant L = 10*L*=10. In the loop executed N - L + 1*N*−*L*+1 one builds a substring of length L*L*. Overall that results in \mathcal{O}((N - L)L)O((*N*−*L*)*L*) time complexity.
* Space complexity : \mathcal{O}((N - L)L)O((*N*−*L*)*L*) to keep the hashset, that results in \mathcal{O}(N)O(*N*) for the constant L = 10*L*=10.

#### Approach 2: Rabin-Karp : Constant-time Slice Using Rolling Hash

Rabin-Karp algorithm is used to perform a multiple pattern search. It's used for plagiarism detection and in bioinformatics to look for similarities in two or more proteins.

Detailed implementation of Rabin-Karp algorithm for quite a complex case you could find in the article [Longest Duplicate Substring](https://leetcode.com/articles/longest-duplicate-substring/), here we do a very basic implementation.

The idea is to slice over the string and to compute the hash of the sequence in the sliding window, both in a constant time.

Let's use string AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT as an example. First, convert string to integer array:

* 'A' -> 0, 'C' -> 1, 'G' -> 2, 'T' -> 3

AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT -> 00000111110000011111100000222333. Time to compute hash for the first sequence of length L: 0000011111. The sequence could be considered as a number in a [numeral system](https://en.wikipedia.org/wiki/Numeral_system) with the base 4 and hashed as

h\_0 = \sum\_{i = 0}^{L - 1}{c\_i 4^{L - 1 - i}}*h*0​=∑*i*=0*L*−1​*ci*​4*L*−1−*i*

Here c\_{0..4} = 0*c*0..4​=0 and c\_{5..9} = 1*c*5..9​=1 are digits of 0000011111.

Now let's consider the slice AAAAACCCCC -> AAAACCCCCA. For int arrays that means 0000011111 -> 0000111110, to remove leading 0 and to add trailing 0. One integer in, and one out, let's recompute the hash:

h\_1 = (h\_0 \times 4 - c\_0 4^L) + c\_{L + 1}*h*1​=(*h*0​×4−*c*0​4*L*)+*cL*+1​.

Voila, window slice and hash recomputation are both done in a constant time.

**Algorithm**

* Iterate over the start position of sequence : from 1 to N - L*N*−*L*.
  + If start == 0, compute the hash of the first sequence s[0: L].
  + Otherwise, compute rolling hash from the previous hash value.
  + If hash is in the hashset, one met a repeated sequence, time to update the output.
  + Otherwise, add hash in the hashset.
* Return output list.

**Implementation**

|  |
| --- |
| class Solution {  public List<String> findRepeatedDnaSequences(String s) {  int L = 10, n = s.length();  if (n <= L) return new ArrayList();  // rolling hash parameters: base a  int a = 4, aL = (int)Math.pow(a, L);  // convert string to array of integers  Map<Character, Integer> toInt = new  HashMap() {{put('A', 0); put('C', 1); put('G', 2); put('T', 3); }};  int[] nums = new int[n];  for(int i = 0; i < n; ++i) nums[i] = toInt.get(s.charAt(i));  int h = 0;  Set<Integer> seen = new HashSet();  Set<String> output = new HashSet();  // iterate over all sequences of length L  for (int start = 0; start < n - L + 1; ++start) {  // compute hash of the current sequence in O(1) time  if (start != 0)  h = h \* a - nums[start - 1] \* aL + nums[start + L - 1];  // compute hash of the first sequence in O(L) time  else  for(int i = 0; i < L; ++i) h = h \* a + nums[i];  // update output and hashset of seen sequences  if (seen.contains(h)) output.add(s.substring(start, start + L));  seen.add(h);  }  return new ArrayList<String>(output);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N - L)O(*N*−*L*), that is \mathcal{O}(N)O(*N*) for the constant L = 10*L*=10. In the loop executed N - L + 1*N*−*L*+1 one builds a hash in a constant time, that results in \mathcal{O}(N - L)O(*N*−*L*) time complexity.
* Space complexity : \mathcal{O}(N - L)O(*N*−*L*) to keep the hashset, that results in \mathcal{O}(N)O(*N*) for the constant L = 10*L*=10.

#### Approach 3: Bit Manipulation : Constant-time Slice Using Bitmask

The idea is to slice over the string and to compute the bitmask of the sequence in the sliding window, both in a constant time.

As for Rabin-Karp, let's start from conversion of string to 2-bits integer array:

A -> 0 = 00\_2, \quad C -> 1 = 01\_2, \quad G -> 2 = 10\_2, \quad T -> 3 = 11\_2*A*−>0=002​,*C*−>1=012​,*G*−>2=102​,*T*−>3=112​

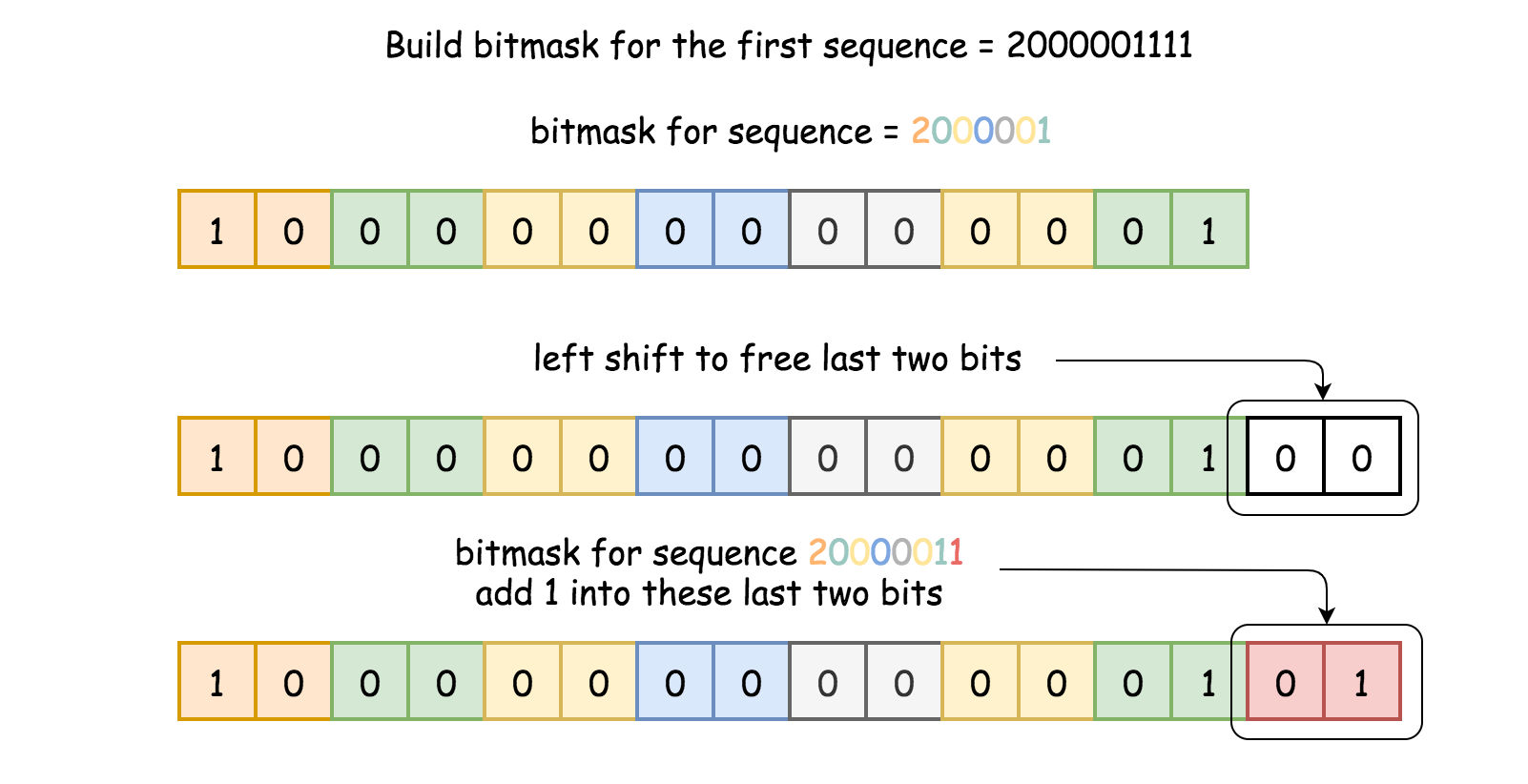
GAAAAACCCCCAAAAACCCCCCAAAAAGGGTTT -> 200000111110000011111100000222333.

Time to compute bitmask for the first sequence of length L: 2000001111. Each digit in the sequence (0, 1, 2 or 3) takes not more than 2 bits:

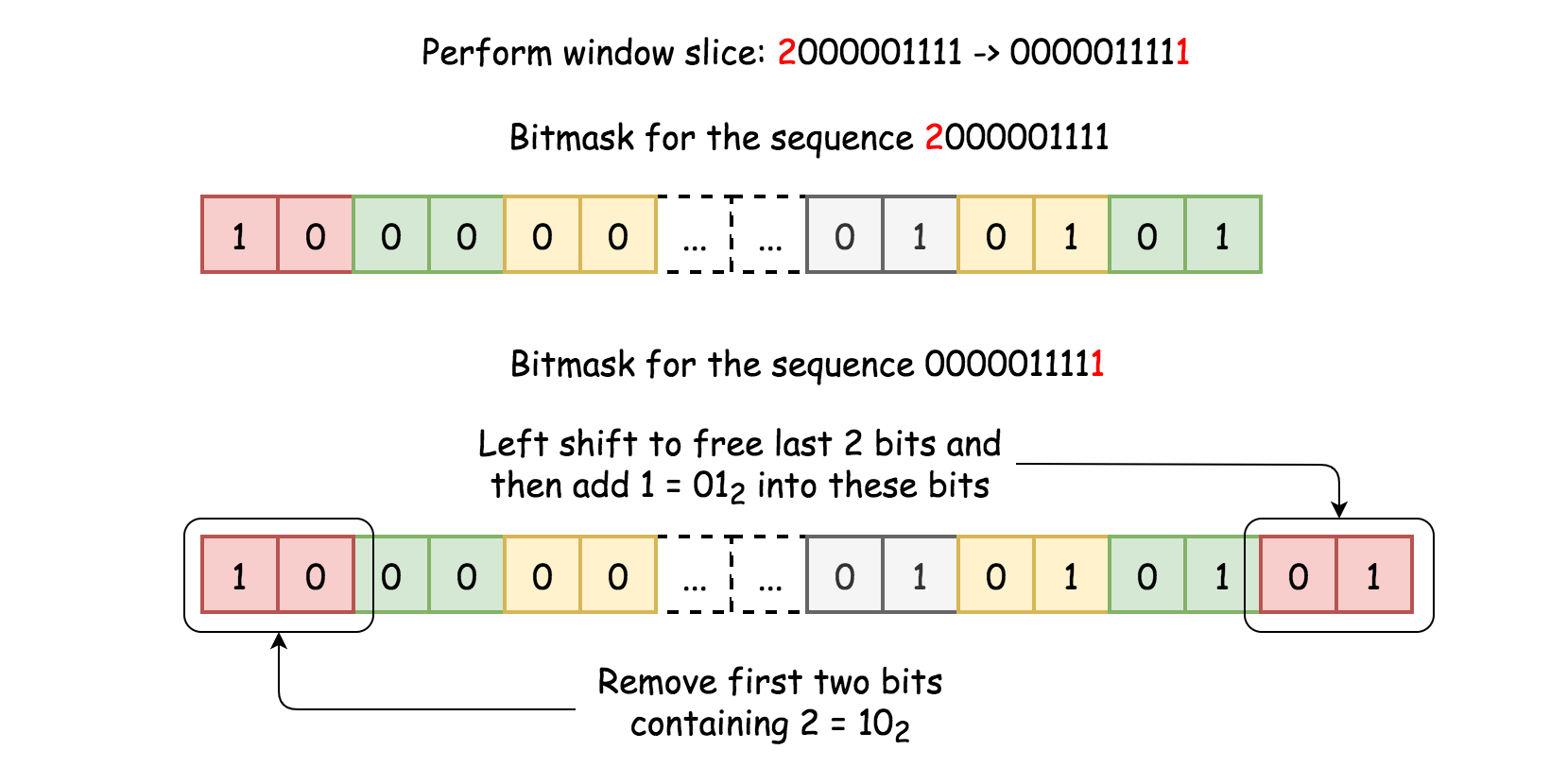
0 = 00\_2, \quad 1 = 01\_2, \quad 2 = 10\_2, \quad 3 = 11\_20=002​,1=012​,2=102​,3=112​

Hence the bitmask could be computed in the loop:

* Do left shift to free the last two bits: bitmask <<= 2
* Save current digit from 2000001111 in these last two bits: bitmask |= nums[i]



Now let's consider the slice GAAAAACCCCC -> AAAAACCCCC. For int arrays that means 20000011111 -> 0000011111, to remove leading 2 and to add trailing 1.



To add trailing 1 is simple, the same idea as just above:

* Do left shift to free the last two bits: bitmask <<= 2
* Save 1 into these last two bits: bitmask |= 1

Now the problem is to remove two leading bits, which contain 2. In other words, the problem is to set 2L-bit and (2L + 1)-bit to zero.

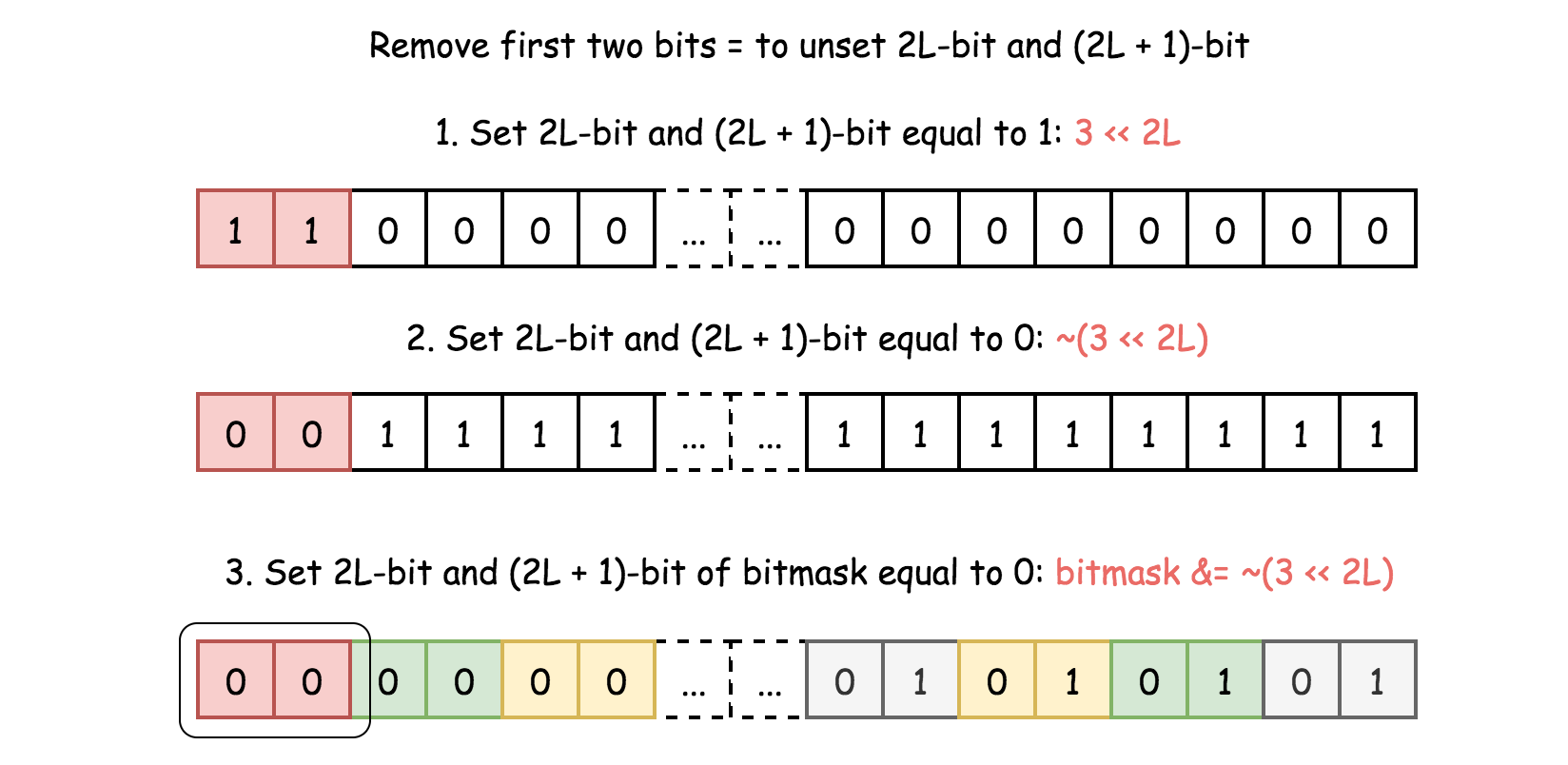
Let's use bitwise trick to unset n-th bit: bitmask &= ~(1 << n).

This trick is very simple:

* 1 << n is to set n-th bit equal to 1.
* ~(1 << n) is to set n-th bit equal to 0, and all lower bits to 1.
* bitmask &= ~(1 << n) is to set n-th bit of bitmask equal to 0.

Straightforward trick usage is to unset first 2L-bit and then (2L + 1)-bit: bitmask &= ~(1 << 2 \* L) & ~(1 << (2 \* L + 1). That could be simplified as bitmask &= ~(3 << 2 \* L):

* 3 = (11)\_23=(11)2​, and hence 3 << 2 \* L would set 2L-bit and (2L + 1)-bit equal to 1.
* ~(3 << 2 \* L) would set 2L-bit and (2L + 1)-bit equal to 0, and all lower bits to 1.
* bitmask &= ~(3 << 2 \* L) would set 2L-bit and (2L + 1)-bit of bitmask equal to 0.



Voila, window slice and bitmask recomputation are both done in a constant time.

**Algorithm**

* Iterate over the start position of sequence : from 1 to N - L*N*−*L*.
  + If start == 0, compute the bitmask of the first sequence s[0: L].
  + Otherwise, compute bitmask from the previous bitmask.
  + If bitmask is in the hashset, one met a repeated sequence, time to update the output.
  + Otherwise, add bitmask in the hashset.
* Return output list.

**Implementation**

|  |
| --- |
| class Solution {  public List<String> findRepeatedDnaSequences(String s) {  int L = 10, n = s.length();  if (n <= L) return new ArrayList();  // rolling hash parameters: base a  int a = 4, aL = (int)Math.pow(a, L);  // convert string to array of integers  Map<Character, Integer> toInt = new  HashMap() {{put('A', 0); put('C', 1); put('G', 2); put('T', 3); }};  int[] nums = new int[n];  for(int i = 0; i < n; ++i) nums[i] = toInt.get(s.charAt(i));  int bitmask = 0;  Set<Integer> seen = new HashSet();  Set<String> output = new HashSet();  // iterate over all sequences of length L  for (int start = 0; start < n - L + 1; ++start) {  // compute bitmask of the current sequence in O(1) time  if (start != 0) {  // left shift to free the last 2 bit  bitmask <<= 2;  // add a new 2-bits number in the last two bits  bitmask |= nums[start + L - 1];  // unset first two bits: 2L-bit and (2L + 1)-bit  bitmask &= ~(3 << 2 \* L);  }  // compute hash of the first sequence in O(L) time  else {  for(int i = 0; i < L; ++i) {  bitmask <<= 2;  bitmask |= nums[i];  }  }  // update output and hashset of seen sequences  if (seen.contains(bitmask)) output.add(s.substring(start, start + L));  seen.add(bitmask);  }  return new ArrayList<String>(output);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N - L)O(*N*−*L*), that is \mathcal{O}(N)O(*N*) for the constant L = 10*L*=10. In the loop executed N - L + 1*N*−*L*+1 one builds a bitmask in a constant time, that results in \mathcal{O}(N - L)O(*N*−*L*) time complexity.
* Space complexity : \mathcal{O}(N - L)O(*N*−*L*) to keep the hashset, that results in \mathcal{O}(N)O(*N*) for the constant L = 10*L*=10.

**Best Time to Buy and Sell Stock IV**

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

**Notice** that you may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** k = 2, prices = [2,4,1]

**Output:** 2

**Explanation:** Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

**Example 2:**

**Input:** k = 2, prices = [3,2,6,5,0,3]

**Output:** 7

**Explanation:** Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

**Constraints:**

* 0 <= k <= 100
* 0 <= prices.length <= 1000
* 0 <= prices[i] <= 1000

## Solution

### **Overview**

You probably can guess from the problem title, this is the fourth problem in the series of [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/) problem. It's strongly recommended that you should finish the previous problems before starting this one. Nevertheless, it's not necessary to finish the previous problems to understand this solution, and you can even use the methods we provide to help you solve the other problems.

Here, two approaches are introduced: Dynamic Programming approach, and Merging approach. Both are awesome, but the first method is more universal to other problems.

#### Approach 1: Dynamic Programming

**Intuition**

[Dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming) (dp) is a popular method among hard-level problems. Its basic idea is to store the previous result to reduce redundant calculations. However, it is hard for beginners to think of the dp method. Below, a step-by-step tutorial of how to think of dp is introduced. If you are already familiar with dp, you can jump to the algorithm part to check out the actual implementation.

Generally, there are two ways to come up with a dp solution. One way is to start with a brute force approach and reduce unnecessary calculations. Another way is to treat the stored results as "states", and try to jump from the starting state to the ending state.

For beginners, it is recommended to start with the brute force approach. So, how to brute force to solve this problem?

Back to (part of) the question:

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Cool, looks like we need to arrange at most k transactions. A natural idea is to iterate all the possible combinations of k transactions, and then find the best combination. As for those with less than k transactions, they are similar and can be considered later. A transaction consists of two parts: buying and selling. Therefore, we need to find 2k points in the stock line, k points for buying, and k points for selling.

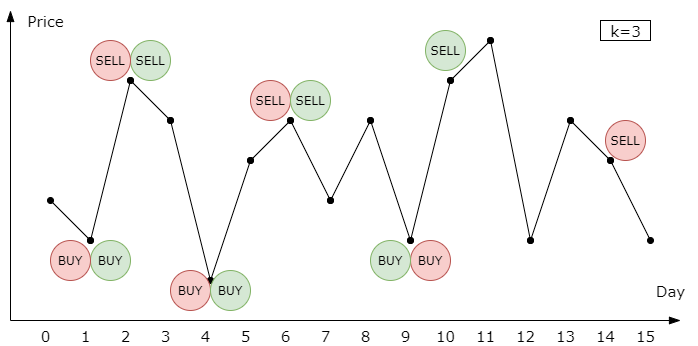
Now, we can roughly estimate the time complexity. Suppose there are n days in total, and we need to pick 2k days. The number of possible situations is about C^{2k}\_{n} = \frac{n!}{(2k)!(n-2k)!}*Cn*2*k*​=(2*k*)!(*n*−2*k*)!*n*!​. It's not a good result because it involves factorial, which is likely to cause Time Limit Exceeded (TLE). Usually what we need is a polynomial one. However, it includes some invalid situations so the actual number is smaller.

Another problem is that, what if 2k is larger than n? In this case, we are not able to pick 2k points from n points, which means we will not reach the limit no matter how we try. Therefore, all we need to do is to iterate each day, and if the price of day i arise, buy the stock in i-1th day and sell it at ith day.

2k > n is a special case and can be addressed easily.

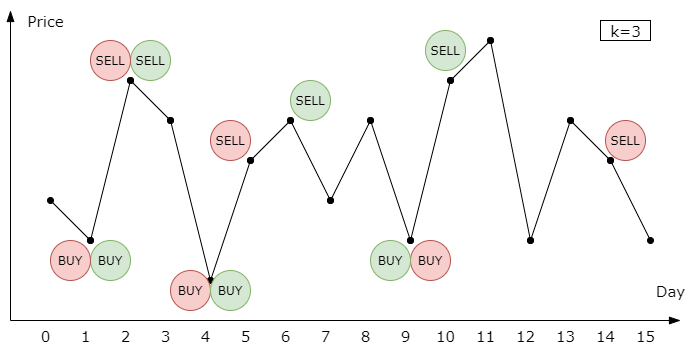
Back to our factorial number. The next step is to review our brute force approach and find out the possible redundant calculations. In our brute force approach, we need to iterate all the possible combinations and calculate the profit of each one to find the best. Can you find out where repeated calculations are?

Consider the following case, where the red color represents a possible combination, and the green represents another one:



The two combinations are the same before day 10. If we calculate the profits separately, we need to calculate the profit before day 10 twice. Here is where dp comes! We can store the current balance on day 9, and reuse it later. Therefore, we can store the result in a dict, where the key is the day number and the transactions we made before, and the value is the balance. Wait a minute, can we do better?

Consider another case:



The only difference is that the red sells stock at a lower price during the second transaction. Therefore, the red has a lower profit on day 10 than the green has. In this case, we need not calculate the rest profit of the red, since it can not beat the green in the future.

Therefore, we can compare those reds, and continue the next day with the one with the highest profit. However, we need to ensure that the best one will not be beaten by the "losers" in the future, so they should have the same "resources" at the time we store and compare the balances.

Hence, we can use three characteristics to store the profit: the day number, the transaction number used, and the stock holding status. You can use other representations of resources, such as using "the day remained" instead of "the day number". Feel free to try. Now, let's go to the algorithm part.

**Algorithm**

In the previous part, we introduced an intuitive idea from brute force to dp method, and here we need to decide the details of the algorithm.

We can either store the dp results in a dict or an array. Array costs less time for accessing and updating than dict, so we always prefer an array when possible. Because of three needed characteristics (day number, transaction number used, stock holding status), a three-dimensional array is our choice. We can use dp[day\_number][used\_transaction\_number][stock\_holding\_status] to represent our states, where stock\_holding\_status is a 0/1 number representing whether you hold the stock or not.

The value of dp[i][j][l] represents the best profit we can have at the end of the i-th day, with j remaining transactions to make and l stocks.

The next step is finding out the so-called "transition equation", which is a method that tells you how to jump from one state to another.

We start with dp[0][0][0] = 0 and dp[0][0][1]=-prices[0], and our final aim is max of dp[n-1][j][0] from j=0 to j=k. Now, we need to fill out the entire array to find out the result. Assume we have gotten the results before day i, and we need to calculate the profit of day i. There are only four possible actions we can do on day i: 1. keep holding the stock, 2. keep not holding the stock, 3. buy the stock, or 4. sell the stock. The profit is easy to calculate.

1. Keep holding the stock:

dp[i][j][1] = dp[i-1][j][1]*dp*[*i*][*j*][1]=*dp*[*i*−1][*j*][1]

1. Keep not holding the stock:

dp[i][j][0] = dp[i-1][j][0]*dp*[*i*][*j*][0]=*dp*[*i*−1][*j*][0]

1. Buying, when j>0:

dp[i][j][1] = dp[i-1][j-1][0]-prices[i]*dp*[*i*][*j*][1]=*dp*[*i*−1][*j*−1][0]−*prices*[*i*]

1. Selling:

dp[i][j][0] = dp[i-1][j][1]+prices[i]*dp*[*i*][*j*][0]=*dp*[*i*−1][*j*][1]+*prices*[*i*]

We can combine they together to find the maximum profit:

dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0]-prices[i])*dp*[*i*][*j*][1]=*max*(*dp*[*i*−1][*j*][1],*dp*[*i*−1][*j*−1][0]−*prices*[*i*])

dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1]+prices[i])*dp*[*i*][*j*][0]=*max*(*dp*[*i*−1][*j*][0],*dp*[*i*−1][*j*][1]+*prices*[*i*])

Awesome! Now we can use for-loop to calculate the whole dp array and achieve our final result. Remember to solve the special cases when 2k > n.

|  |
| --- |
| public class Solution {  public int maxProfit(int k, int[] prices) {  int n = prices.length;  // solve special cases  if (n <= 0 || k <= 0) {  return 0;  }  if (2 \* k > n) {  int res = 0;  for (int i = 1; i < n; i++) {  res += Math.max(0, prices[i] - prices[i - 1]);  }  return res;  }  // dp[i][used\_k][ishold] = balance  // ishold: 0 nothold, 1 hold  int[][][] dp = new int[n][k + 1][2];  // initialize the array with -inf  // we use -1e9 here to prevent overflow  for (int i = 0; i < n; i++) {  for (int j = 0; j <= k; j++) {  dp[i][j][0] = -1000000000;  dp[i][j][1] = -1000000000;  }  }  // set starting value  dp[0][0][0] = 0;  dp[0][1][1] = -prices[0];  // fill the array  for (int i = 1; i < n; i++) {  for (int j = 0; j <= k; j++) {  // transition equation  dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j][1] + prices[i]);  // you can't hold stock without any transaction  if (j > 0) {  dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]);  }  }  }  int res = 0;  for (int j = 0; j <= k; j++) {  res = Math.max(res, dp[n - 1][j][0]);  }  return res;  }  } |

There a few points you should notice from the code above:

1. Take care of the initial values in dp array. Generally, it's ok to initialize them to zero. However, in this case, we need to make them -inf to mark impossible situations, such as dp[0][0][1].
2. You can reverse the order of filling the dp array, with some modifications in the transition equation. For example, decreasing j instead of increasing it.
3. Some state-compressed method can be applied if you want. For example, we only need dp[i-1], when calculating dp[i], therefore we can delete other useless dp to save memory. Just using two arrays to storing dp[i-1] and dp[i] and refreshing them every iteration will do.
4. The code above is not the fastest because we prioritize the readability. It would be faster if you put the larger dimension in the inner array since it uses CPU cache more efficiently.

**Complexity**

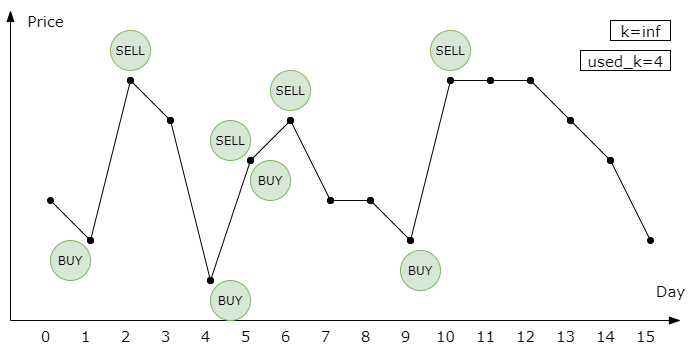
* Time Complexity: \mathcal{O}(nk)O(*nk*) if 2k \le n2*k*≤*n* , \mathcal{O}(n)O(*n*) if 2k > n2*k*>*n*, where n*n* is the length of the prices sequence, since we have two for-loop.
* Space Complexity: \mathcal{O}(nk)O(*nk*) without state-compressed, and \mathcal{O}(k)O(*k*) with state-compressed, where n*n* is the length of the prices sequence.

#### Approach 2: Merging

**Intuition**

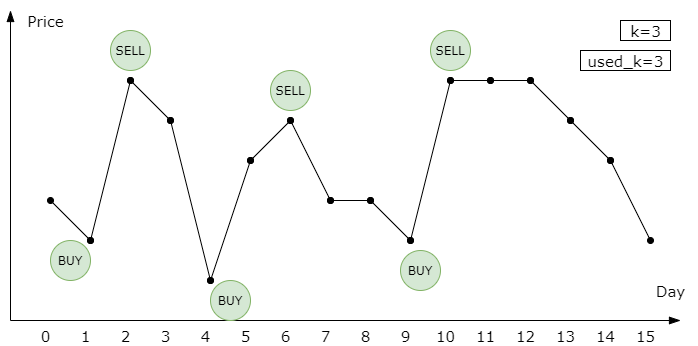
This approach starts from a simple situation with k=infinity, and drecrease k one by one.

Consider a weakened problem when k=infinity. Since we already know the prices of tomorrow, our solution is to trade whenever prices[i-1] < prices[i]. Below is an example.



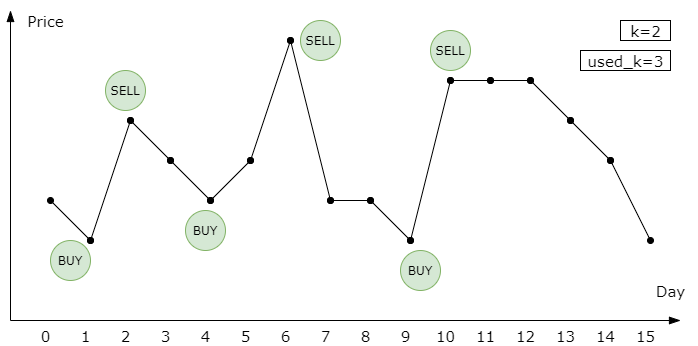
We only used 4 transactions! However, what we need to solve is the case with an actual k. Let's decrease k from inf and see what happens. Our solution can handle all the k >=4, since we only used 4 transactions. But what if k=3?

Notice that at day 5, we buy and sell the stock at the same time. We can cancel the redundant transaction without impact the final profit!



We can conclude that for the consecutively increasing subsequence, we only need to buy once at the start and sell once at the end.

How about k=2? Maybe we need to delete one transaction. We can iterate all the transactions and delete the one with least revenue. However, deleting can not always achieve our best solution. Consider the following example:



When k=2, the best solution is to buy at day 1 and day 9, and to sell on day 6 and day 10. Deleting any transactions cannot reach this solution. However, we can merge the previous two transactions to get to this. A naive approach is iterating all the near transactions and find out the pair with the lowest impact on the revenue. Since we decrease k one by one, reducing one transaction is enough. Ok, let's go to the algorithm part to check the detail.

**Algorithm**

The general idea is to store all consecutively increasing subsequence as the initial solution. Then delete or merge transactions until the number of transactions less than or equal to k.

|  |
| --- |
| public class Solution {  public int maxProfit(int k, int[] prices) {  int n = prices.length;  // solve special cases  if (n <= 0 || k <= 0) {  return 0;  }  // find all consecutively increasing subsequence  ArrayList<int[]> transactions = new ArrayList<>();  int start = 0;  int end = 0;  for (int i = 1; i < n; i++) {  if (prices[i] >= prices[i - 1]) {  end = i;  } else {  if (end > start) {  int[] t = { start, end };  transactions.add(t);  }  start = i;  }  }  if (end > start) {  int[] t = { start, end };  transactions.add(t);  }  while (transactions.size() > k) {  // check delete loss  int delete\_index = 0;  int min\_delete\_loss = Integer.MAX\_VALUE;  for (int i = 0; i < transactions.size(); i++) {  int[] t = transactions.get(i);  int profit\_loss = prices[t[1]] - prices[t[0]];  if (profit\_loss < min\_delete\_loss) {  min\_delete\_loss = profit\_loss;  delete\_index = i;  }  }  // check merge loss  int merge\_index = 0;  int min\_merge\_loss = Integer.MAX\_VALUE;  for (int i = 1; i < transactions.size(); i++) {  int[] t1 = transactions.get(i - 1);  int[] t2 = transactions.get(i);  int profit\_loss = prices[t1[1]] - prices[t2[0]];  if (profit\_loss < min\_merge\_loss) {  min\_merge\_loss = profit\_loss;  merge\_index = i;  }  }  // delete or merge  if (min\_delete\_loss <= min\_merge\_loss) {  transactions.remove(delete\_index);  } else {  int[] t1 = transactions.get(merge\_index - 1);  int[] t2 = transactions.get(merge\_index);  t1[1] = t2[1];  transactions.remove(merge\_index);  }  }  int res = 0;  for (int[] t : transactions) {  res += prices[t[1]] - prices[t[0]];  }  return res;  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(n(n-k))O(*n*(*n*−*k*)) if 2k \le n2*k*≤*n* , \mathcal{O}(n)O(*n*) if 2k > n2*k*>*n*, where n*n* is the length of the price sequence. The maximum size of transactions is \mathcal{O}(n)O(*n*), and we need \mathcal{O}(n-k)O(*n*−*k*) iterations.
* Space Complexity: \mathcal{O}(n)O(*n*), since we need a list to store transactions.

**Minimum Domino Rotations For Equal Row**

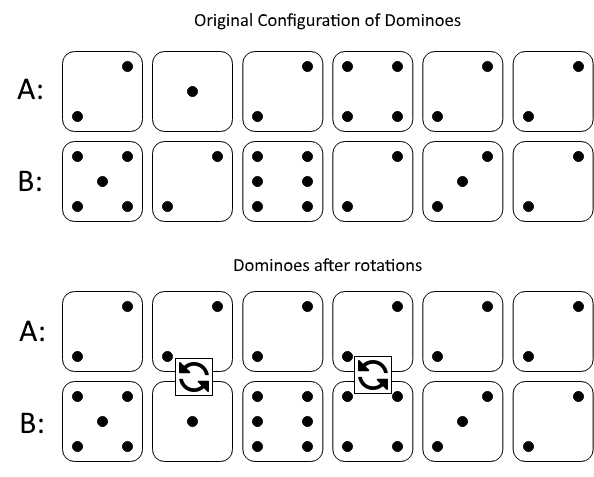
In a row of dominoes, A[i] and B[i] represent the top and bottom halves of the ith domino.  (A domino is a tile with two numbers from 1 to 6 - one on each half of the tile.)

We may rotate the ith domino, so that A[i] and B[i] swap values.

Return the minimum number of rotations so that all the values in A are the same, or all the values in B are the same.

If it cannot be done, return -1.

**Example 1:**



**Input:** A = [2,1,2,4,2,2], B = [5,2,6,2,3,2]

**Output:** 2

**Explanation:**

The first figure represents the dominoes as given by A and B: before we do any rotations.

If we rotate the second and fourth dominoes, we can make every value in the top row equal to 2, as indicated by the second figure.

**Example 2:**

**Input:** A = [3,5,1,2,3], B = [3,6,3,3,4]

**Output:** -1

**Explanation:**

In this case, it is not possible to rotate the dominoes to make one row of values equal.

**Constraints:**

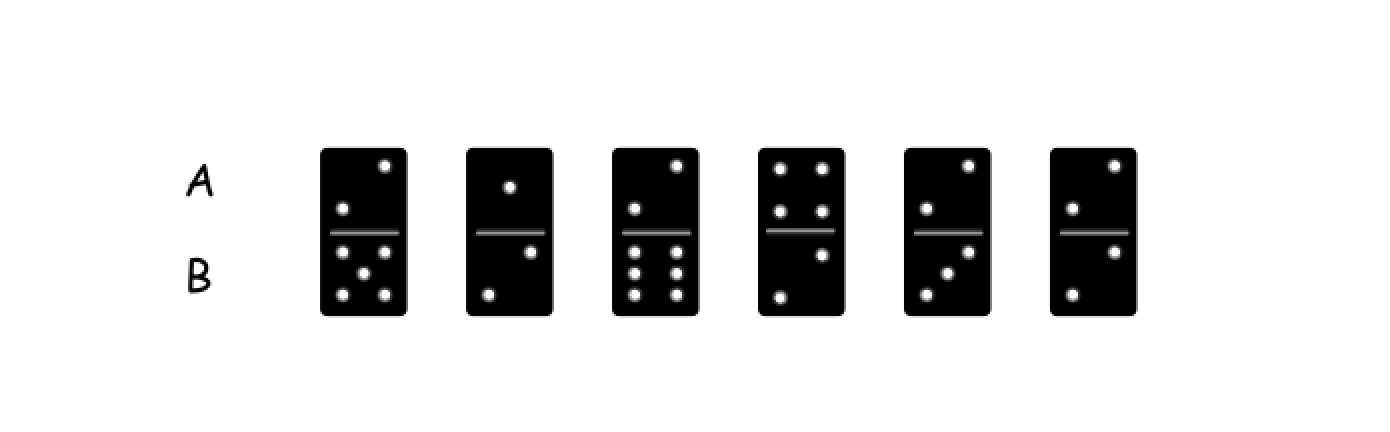
* 2 <= A.length == B.length <= 2 \* 104
* 1 <= A[i], B[i] <= 6

## Solution

#### Approach 1: Greedy.

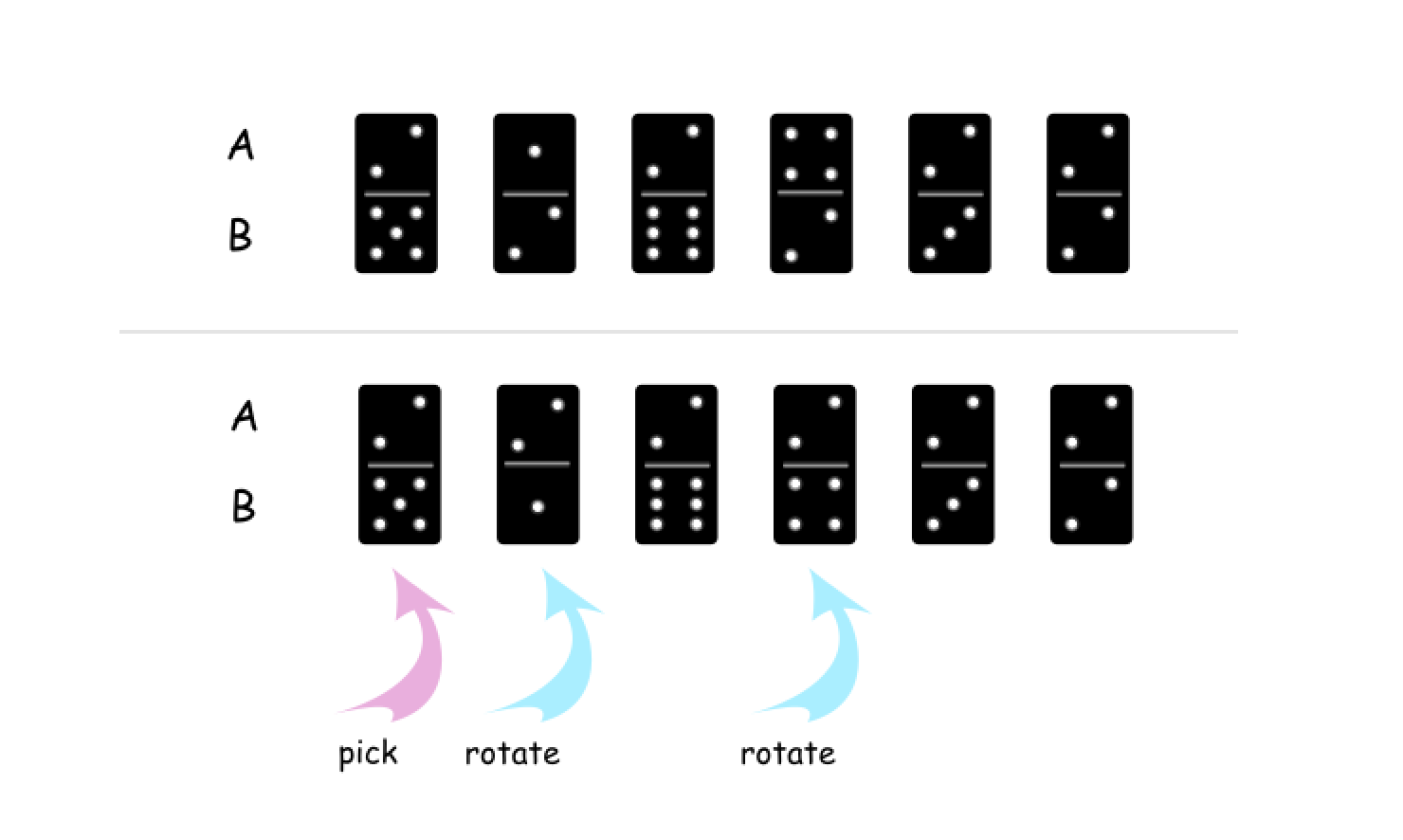
**Intuition**

Let's pick up an arbitrary ith domino element in the configuration. The element has two sides, A[i] is an upper side and B[i] is a lower side.

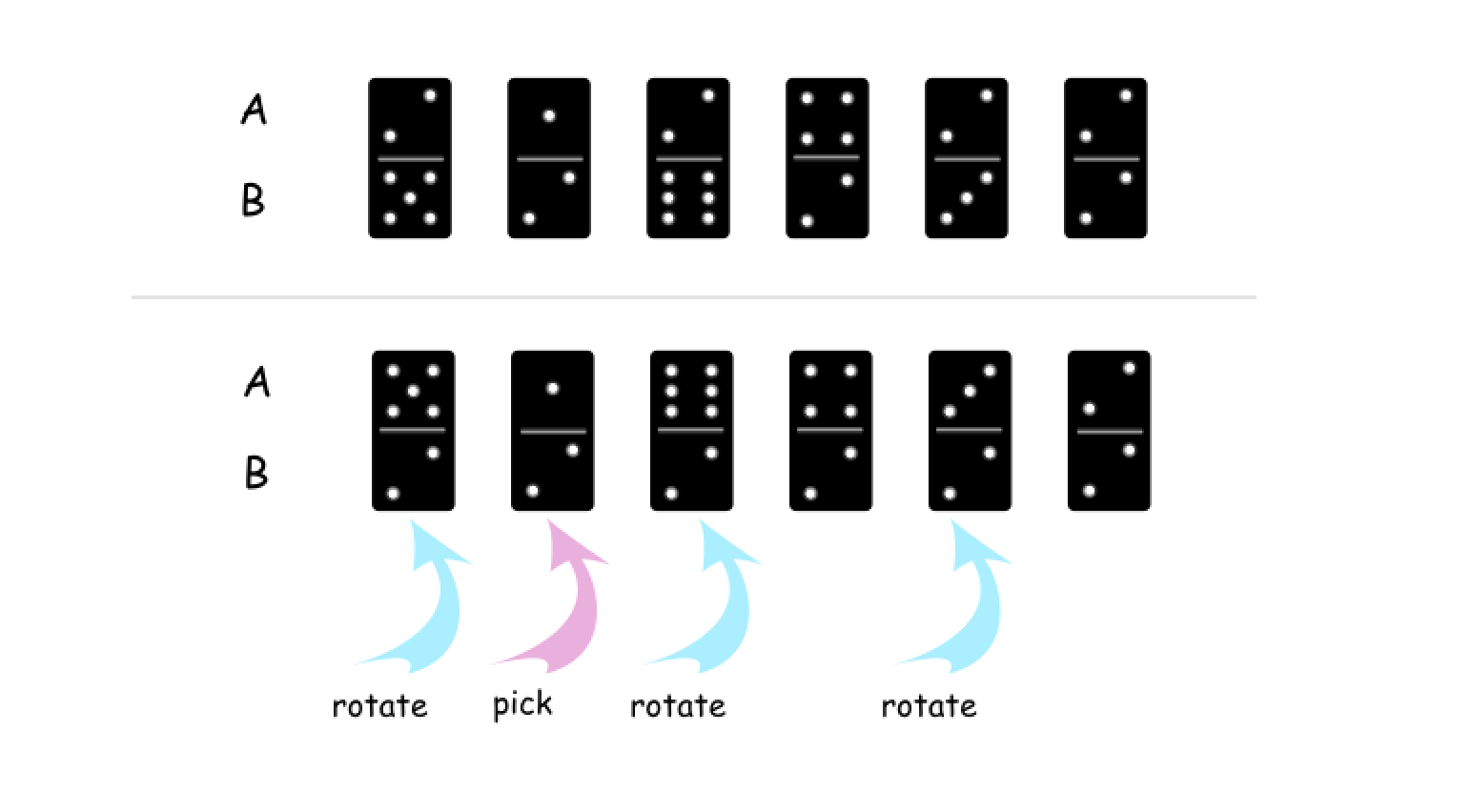


There could be three possible situations here

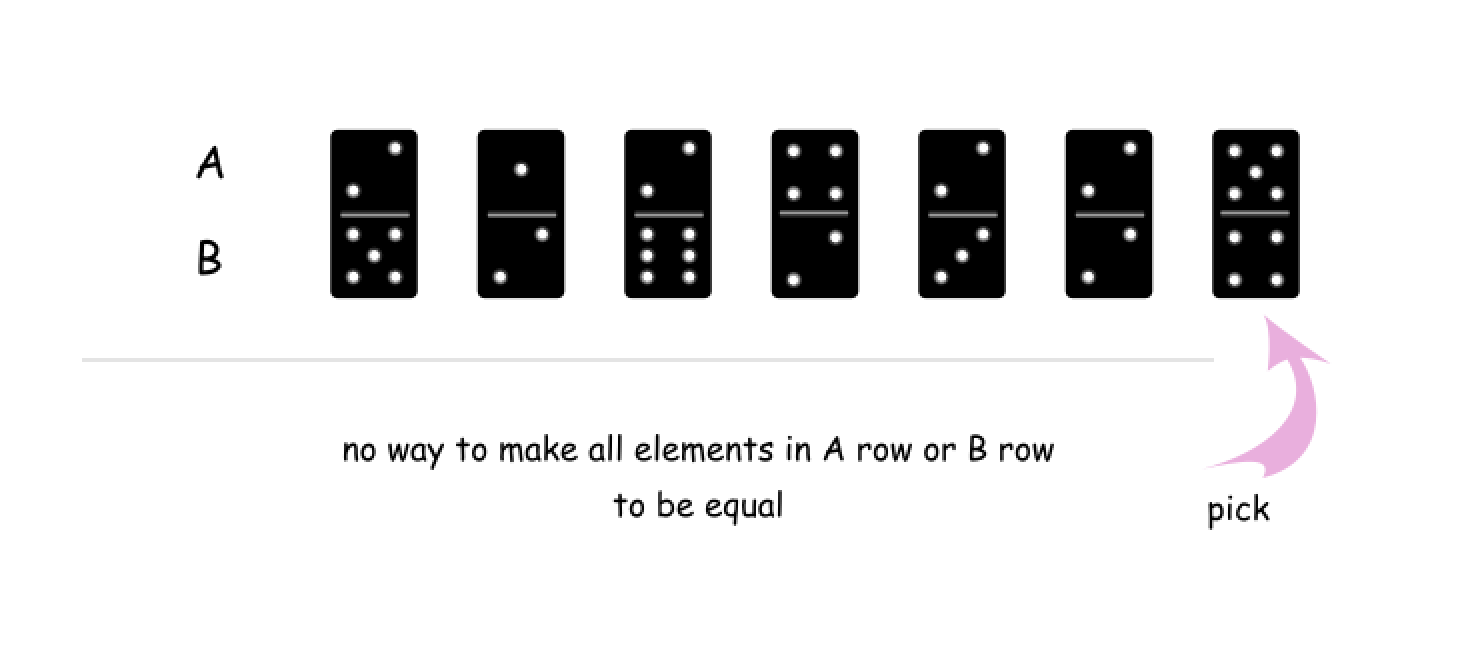
1 . One could make all elements of A row or B row to be the same and equal to A[i] value. For example, if one picks up the 0th element, it's possible to make all elements of A row to be equal to 2.



2 . One could make all elements of A row or B row to be the same and equal to B[i] value. For example, if one picks up the 1th element, it's possible to make all elements of B row to be equal to 2.



3 . It's impossible to make all elements of A row or B row to have the same A[i] or B[i] value.



The third situation means that it's impossible to make all elements in A row or B row to be equal.

Yes, the only one domino element was checked here, and still it's enough because the rotation is the only allowed operation here.

**Algorithm**

* Pick up the first element. It has two sides: A[0] and B[0].
* Check if one could make all elements in A row or B row to be equal to A[0]. If yes, return the minimum number of rotations needed.
* Check if one could make all elements in A row or B row to be equal to B[0]. If yes, return the minimum number of rotations needed.
* Otherwise return -1.

**Implementation**

|  |
| --- |
| class Solution {  /\*  Return min number of rotations  if one could make all elements in A or B equal to x.  Else return -1.  \*/  public int check(int x, int[] A, int[] B, int n) {  // how many rotations should be done  // to have all elements in A equal to x  // and to have all elements in B equal to x  int rotations\_a = 0, rotations\_b = 0;  for (int i = 0; i < n; i++) {  // rotations coudn't be done  if (A[i] != x && B[i] != x) return -1;  // A[i] != x and B[i] == x  else if (A[i] != x) rotations\_a++;  // A[i] == x and B[i] != x  else if (B[i] != x) rotations\_b++;  }  // min number of rotations to have all  // elements equal to x in A or B  return Math.min(rotations\_a, rotations\_b);  }  public int minDominoRotations(int[] A, int[] B) {  int n = A.length;  int rotations = check(A[0], B, A, n);  // If one could make all elements in A or B equal to A[0]  if (rotations != -1 || A[0] == B[0]) return rotations;  // If one could make all elements in A or B equal to B[0]  else return check(B[0], B, A, n);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since here one iterates over the arrays not more than two times.
* Space complexity : \mathcal{O}(1)O(1) since it's a constant space solution.

**Asteroid Collision**

We are given an array asteroids of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

**Example 1:**

**Input:** asteroids = [5,10,-5]

**Output:** [5,10]

**Explanation:** The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

**Example 2:**

**Input:** asteroids = [8,-8]

**Output:** []

**Explanation:** The 8 and -8 collide exploding each other.

**Example 3:**

**Input:** asteroids = [10,2,-5]

**Output:** [10]

**Explanation:** The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

**Example 4:**

**Input:** asteroids = [-2,-1,1,2]

**Output:** [-2,-1,1,2]

**Explanation:** The -2 and -1 are moving left, while the 1 and 2 are moving right. Asteroids moving the same direction never meet, so no asteroids will meet each other.

**Constraints:**

* 2 <= asteroids.length <= 104
* -1000 <= asteroids[i] <= 1000
* asteroids[i] != 0

   Hide Hint #1

Say a row of asteroids is stable. What happens when a new asteroid is added on the right?

#### Approach #1: Stack [Accepted]

**Intuition**

A row of asteroids is stable if no further collisions will occur. After adding a new asteroid to the right, some more collisions may happen before it becomes stable again, and all of those collisions (if they happen) must occur right to left. This is the perfect situation for using a stack.

**Algorithm**

Say we have our answer as a stack with rightmost asteroid top, and a new asteroid comes in. If new is moving right (new > 0), or if top is moving left (top < 0), no collision occurs.

Otherwise, if abs(new) < abs(top), then the new asteroid will blow up; if abs(new) == abs(top) then both asteroids will blow up; and if abs(new) > abs(top), then the top asteroid will blow up (and possibly more asteroids will, so we should continue checking.)

|  |
| --- |
| class Solution {  public int[] asteroidCollision(int[] asteroids) {  Stack<Integer> stack = new Stack();  for (int ast: asteroids) {  collision: {  while (!stack.isEmpty() && ast < 0 && 0 < stack.peek()) {  if (stack.peek() < -ast) {  stack.pop();  continue;  } else if (stack.peek() == -ast) {  stack.pop();  }  break collision;  }  stack.push(ast);  }  }  int[] ans = new int[stack.size()];  for (int t = ans.length - 1; t >= 0; --t) {  ans[t] = stack.pop();  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of asteroids. Our stack pushes and pops each asteroid at most once.
* Space Complexity: O(N)*O*(*N*). We use a stack to keep track of the intermediate results. In the worst case, the states do not evolve at the end, i.e. we need O(N)*O*(*N*) space where N*N* is the number of input asteroids.

**Search in a Sorted Array of Unknown Size**

Given an integer array sorted in ascending order, write a function to search target in nums.  If target exists, then return its index, otherwise return -1. **However, the array size is unknown to you**. You may only access the array using an ArrayReader interface, where ArrayReader.get(k) returns the element of the array at index k (0-indexed).

You may assume all integers in the array are less than 10000, and if you access the array out of bounds, ArrayReader.get will return 2147483647.

**Example 1:**

**Input:** array = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** array = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**Constraints:**

* You may assume that all elements in the array are unique.
* The value of each element in the array will be in the range [-9999, 9999].
* The length of the array will be in the range [1, 10^4].

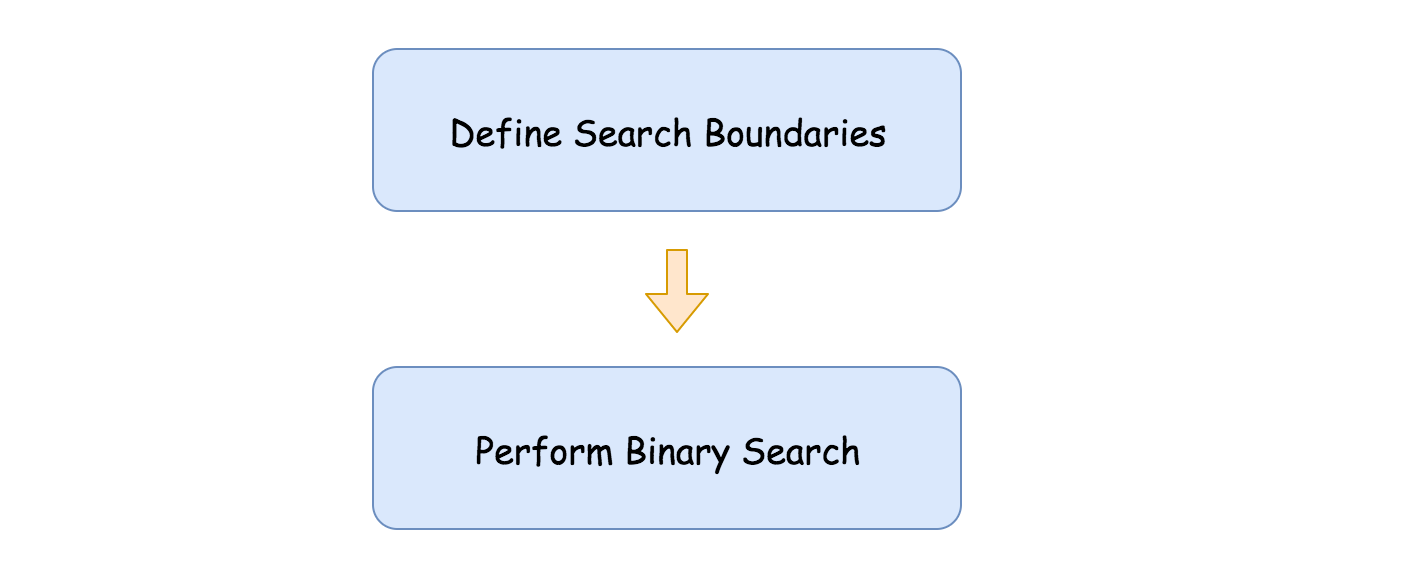
## Solution

#### Approach 1: Binary Search

**Split into Two Subproblems**

The array is sorted, i.e. one could try to fit into a logarithmic time complexity. That means two subproblems, and both should be done in a logarithmic time:

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

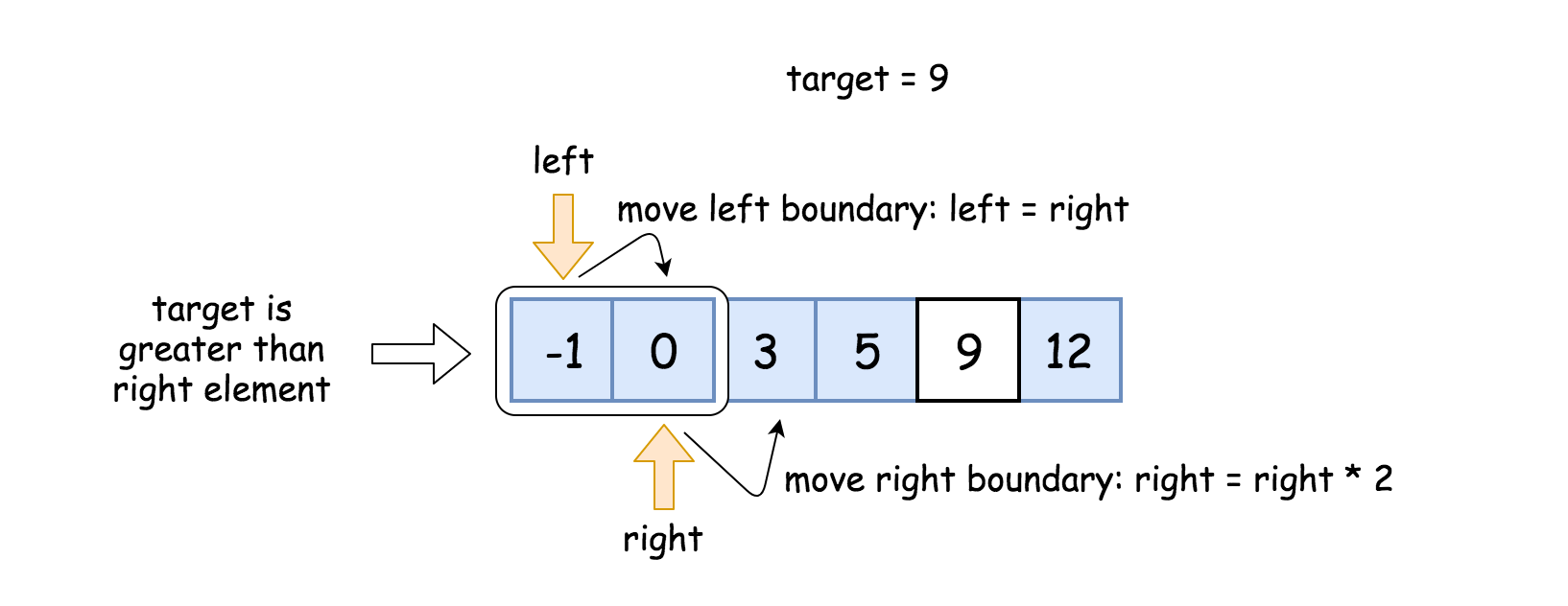


**Define Search Boundaries**

This is a key subproblem here.

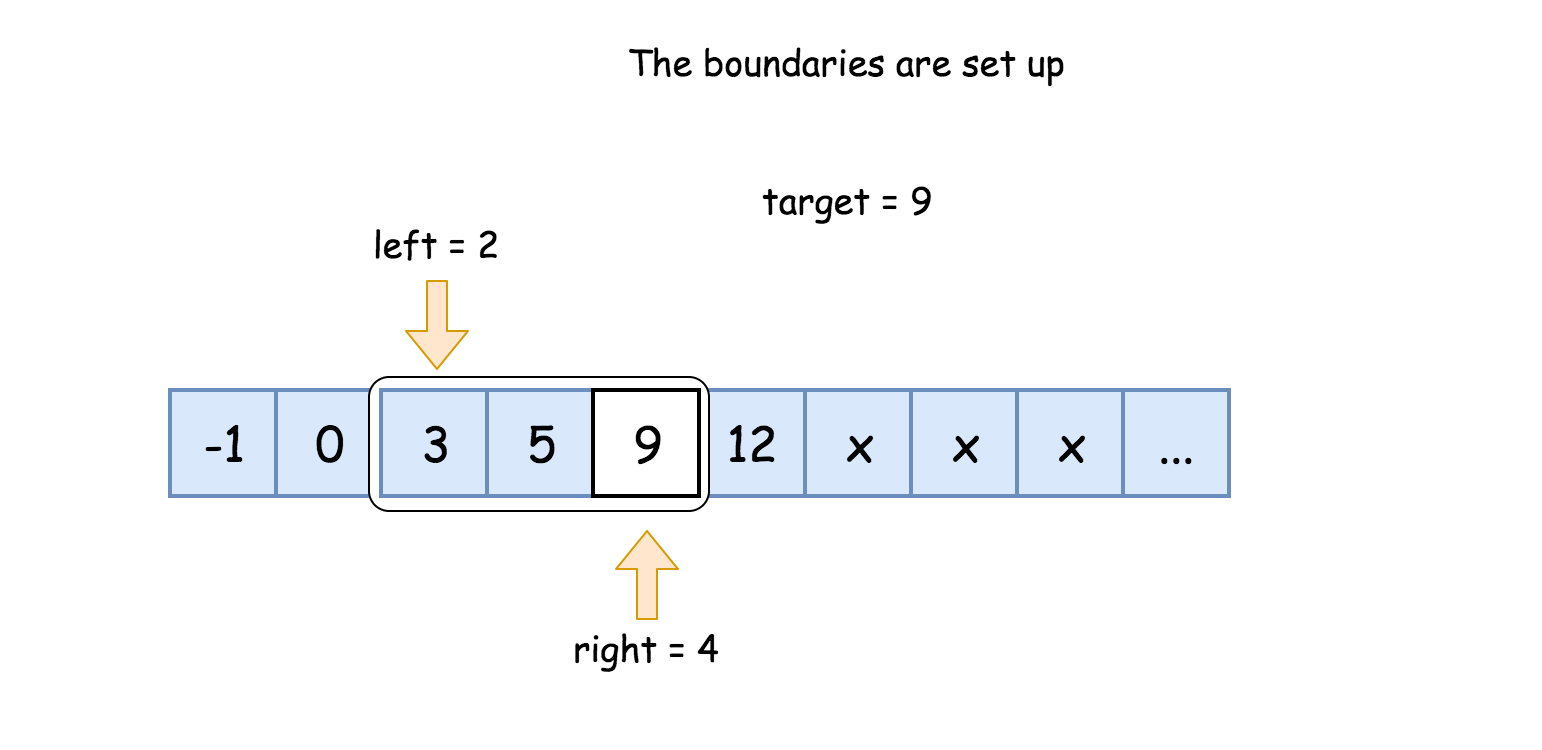
The idea is quite simple. Let's take two first indexes, 0 and 1, as left and right boundaries. If the target value is not among these zeroth and the first element, then it's outside the boundaries, on the right.

That means that the left boundary could 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.



If the target now is less than the right element, we're done, the boundaries are set. If not, repeat these two steps till the boundaries are established:

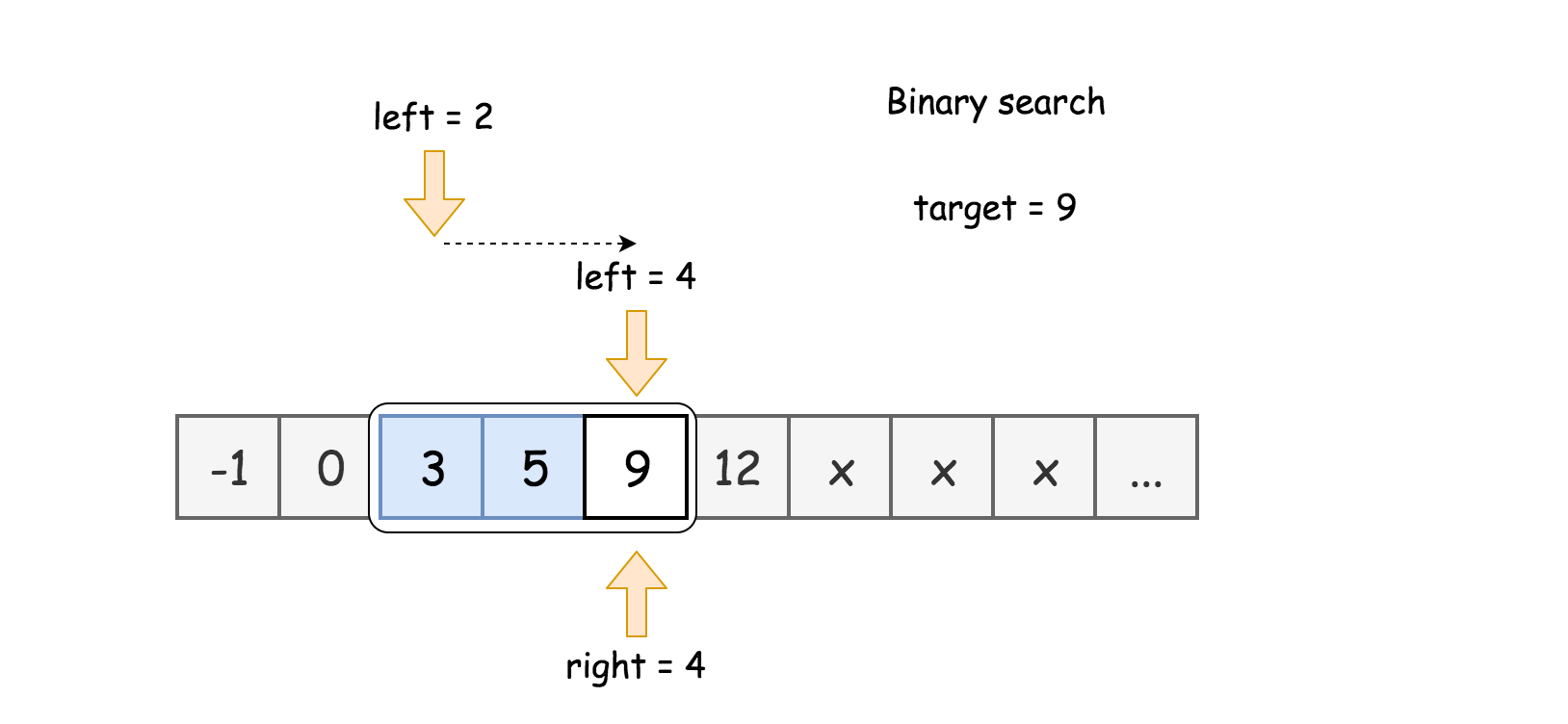
* Move the left boundary to the right: left = right.
* Extend the right boundary: right = right \* 2.



**Binary Search**

[Binary search](https://leetcode.com/explore/learn/card/binary-search/) is a textbook algorithm with a logarithmic time complexity. It's based on the idea to compare the target value to the middle element of the array.

* If the target value is equal to the middle element - we're done.
* If the target value is smaller - continue to search on the left.
* If the target value is larger - continue to search on the right.



**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**

Define boundaries:

* Initiate left = 0 and right = 1.
* While target is on the right to the right boundary: reader.get(right) < target:
  + 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 index in the middle: pivot = (left + right) / 2. To avoid overflow, use the form pivot = left + ((right - left) >> 1) instead of straightforward expression above.
  + Retrieve the element at this index: num = reader.get(pivot).
  + Compare middle element num to the target value.
    - If the middle element is the target num == target: return pivot.
    - If the target is not yet found:
      * If num > target, continue to search on the left right = pivot - 1.
      * Else continue to search on the right left = pivot + 1.
* We're here because target is not found. Return -1.

**Implementation**

|  |
| --- |
| class Solution {  public int search(ArrayReader reader, int target) {  if (reader.get(0) == target) return 0;  // search boundaries  int left = 0, right = 1;  while (reader.get(right) < target) {  left = right;  right <<= 1;  }  // binary search  int pivot, num;  while (left <= right) {  pivot = left + ((right - left) >> 1);  num = reader.get(pivot);  if (num == target) return pivot;  if (num > target) right = pivot - 1;  else left = pivot + 1;  }  // there is no target element  return -1;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(\log T)O(log*T*), where T*T* is an index of target value.

There are two operations here: to define search boundaries and to perform binary search.

Let's first find the number of steps k to setup the boundaries. On the first step, the boundaries are 2^0 .. 2^{0 + 1}20..20+1, on the second step 2^1 .. 2^{1 + 1}21..21+1, etc. When everything is done, the boundaries are 2^k .. 2^{k + 1}2*k*..2*k*+1 and 2^k < T \le 2^{k + 1}2*k*<*T*≤2*k*+1. That means one needs k = \log T*k*=log*T* steps to setup the boundaries, that means \mathcal{O}(\log T)O(log*T*) time complexity.

Now let's discuss the complexity of the binary search. There are 2^{k + 1} - 2^k = 2^k2*k*+1−2*k*=2*k* elements in the boundaries, i.e. 2^{\log T} = T2log*T*=*T* elements. [As discussed](https://leetcode.com/articles/binary-search/), binary search has logarithmic complexity, that results in \mathcal{O}(\log T)O(log*T*) time complexity.

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

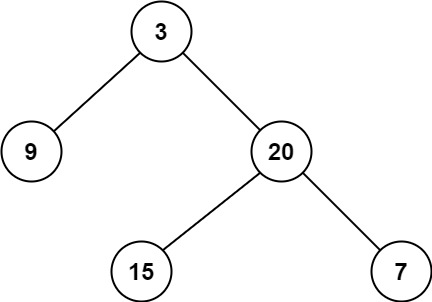
**Minimum Depth of Binary Tree**

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

**Note:** A leaf is a node with no children.

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 2

**Example 2:**

**Input:** root = [2,null,3,null,4,null,5,null,6]

**Output:** 5

**Constraints:**

* The number of nodes in the tree is in the range [0, 105].
* -1000 <= Node.val <= 1000

## Solution

**Tree definition**

First of all, here is the definition of the TreeNode which we would use.

|  |
| --- |
| // Definition for a binary tree node.  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) {  val = x;  }  } |

#### Approach 1: Recursion

**Algorithm**

The intuitive approach is to solve the problem by recursion. Here we demonstrate an example with the DFS (Depth First Search) strategy.

|  |
| --- |
| class Solution {  public int minDepth(TreeNode root) {  if (root == null) {  return 0;  }  if ((root.left == null) && (root.right == null)) {  return 1;  }  int min\_depth = Integer.MAX\_VALUE;  if (root.left != null) {  min\_depth = Math.min(minDepth(root.left), min\_depth);  }  if (root.right != null) {  min\_depth = Math.min(minDepth(root.right), min\_depth);  }  return min\_depth + 1;  }  } |

**Complexity analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*), where N*N* is the number of nodes.
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only one child node, the recursion call would occur N*N* times (the height of the tree), therefore the storage to keep the call stack would be \mathcal{O}(N)O(*N*). But in the best case (the tree is completely balanced), the height of the tree would be \log(N)log(*N*). Therefore, the space complexity in this case would be \mathcal{O}(\log(N))O(log(*N*)).

#### Approach 2: DFS Iteration

We could also convert the above recursion into iteration, with the help of stack.

The idea is to visit each leaf with the DFS strategy, while updating the minimum depth when we reach the leaf node.

So we start from a stack which contains the root node and the corresponding depth which is 1. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The minimum depth is updated at each leaf node.

|  |
| --- |
| class Solution {  public int minDepth(TreeNode root) {  LinkedList<Pair<TreeNode, Integer>> stack = new LinkedList<>();  if (root == null) {  return 0;  }  else {  stack.add(new Pair(root, 1));  }  int min\_depth = Integer.MAX\_VALUE;  while (!stack.isEmpty()) {  Pair<TreeNode, Integer> current = stack.pollLast();  root = current.getKey();  int current\_depth = current.getValue();  if ((root.left == null) && (root.right == null)) {  min\_depth = Math.min(min\_depth, current\_depth);  }  if (root.left != null) {  stack.add(new Pair(root.left, current\_depth + 1));  }  if (root.right != null) {  stack.add(new Pair(root.right, current\_depth + 1));  }  }  return min\_depth;  }  } |

**Complexity analysis**

* Time complexity : each node is visited exactly once and time complexity is \mathcal{O}(N)O(*N*).
* Space complexity : in the worst case we could keep up to the entire tree, that results in \mathcal{O}(N)O(*N*) space complexity.

#### Approach 3: BFS Iteration

The drawback of the DFS approach in this case is that all nodes should be visited to ensure that the minimum depth would be found. Therefore, this results in a \mathcal{O}(N)O(*N*) complexity. One way to optimize the complexity is to use the BFS strategy. We iterate the tree level by level, and the first leaf we reach corresponds to the minimum depth. As a result, we do not need to iterate all nodes.

|  |
| --- |
| class Solution {  public int minDepth(TreeNode root) {  LinkedList<Pair<TreeNode, Integer>> queue = new LinkedList<>();  if (root == null) {  return 0;  }  else {  queue.add(new Pair(root, 1));  }  int current\_depth = 0;  while (!queue.isEmpty()) {  Pair<TreeNode, Integer> current = queue.poll();  root = current.getKey();  current\_depth = current.getValue();  if ((root.left == null) && (root.right == null)) {  break;  }  if (root.left != null) {  queue.add(new Pair(root.left, current\_depth + 1));  }  if (root.right != null) {  queue.add(new Pair(root.right, current\_depth + 1));  }  }  return current\_depth;  }  } |

**Complexity analysis**

* Time complexity : in the worst case for a balanced tree we need to visit all nodes level by level up to the tree height, that excludes the bottom level only. This way we visit N/2*N*/2 nodes, and thus the time complexity is \mathcal{O}(N)O(*N*).
* Space complexity : is the same as time complexity here \mathcal{O}(N)O(*N*).

**132 Pattern**

Given an array of n integers nums, a **132 pattern** is a subsequence of three integers nums[i], nums[j] and nums[k] such that i < j < k and nums[i] < nums[k] < nums[j].

Return *true* if there is a ***132 pattern*** in *nums*, otherwise, return *false*.

**Follow up:**The O(n^2) is trivial, could you come up with the O(n logn) or the O(n) solution?

**Example 1:**

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

**Output:** false

**Explanation:** There is no 132 pattern in the sequence.

**Example 2:**

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

**Output:** true

**Explanation:** There is a 132 pattern in the sequence: [1, 4, 2].

**Example 3:**

**Input:** nums = [-1,3,2,0]

**Output:** true

**Explanation:** There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] and [-1, 2, 0].

**Constraints:**

* n == nums.length
* 1 <= n <= 104
* -109 <= nums[i] <= 109

## Solution Article

#### Approach 1: Brute Force

The simplest solution is to consider every triplet (i, j, k)(*i*,*j*,*k*) and check if the corresponding numbers satisfy the 132 criteria. If any such triplet is found, we can return a True value. If no such triplet is found, we need to return a False value.

|  |
| --- |
| public class Solution {  public boolean find132pattern(int[] nums) {  for (int i = 0; i < nums.length - 2; i++) {  for (int j = i + 1; j < nums.length - 1; j++) {  for (int k = j + 1; k < nums.length; k++) {  if (nums[k] > nums[i] && nums[j] > nums[k])  return true;  }  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : O(n^3)*O*(*n*3). Three loops are used to consider every possible triplet. Here, n*n* refers to the size of nums*nums* array.
* Space complexity : O(1)*O*(1). Constant extra space is used.

#### Approach 2: Better Brute Force

**Algorithm**

We can improve the last approach to some extent, if we make use of some observations. We can note that for a particular number nums[j]*nums*[*j*] chosen as 2nd element in the 132 pattern, if we don't consider nums[k]*nums*[*k*](the 3rd element) for the time being, our job is to find out the first element, nums[i]*nums*[*i*](i<j*i*<*j*) which is lesser than nums[j]*nums*[*j*].

Now, assume that we have somehow found a nums[i],nums[j]*nums*[*i*],*nums*[*j*] pair. Our task now reduces to finding out a nums[k]*nums*[*k*](Kk>j>i)*Kk*>*j*>*i*), which falls in the range (nums[i], nums[j])(*nums*[*i*],*nums*[*j*]). Now, to maximize the likelihood of a nums[k]*nums*[*k*] falling in this range, we need to increase this range as much as possible.

Since, we started off by fixing a nums[j]*nums*[*j*], the only option in our hand is to choose a minimum value of nums[i]*nums*[*i*] given a particular nums[j]*nums*[*j*]. Once, this pair nums[i],nums[j]*nums*[*i*],*nums*[*j*], has been found out, we simply need to traverse beyond the index j*j* to find if a nums[k]*nums*[*k*] exists for this pair satisfying the 132 criteria.

Based on the above observations, while traversing over the nums*nums* array choosing various values of nums[j]*nums*[*j*], we simultaneously keep a track of the minimum element found so far(excluding nums[j]*nums*[*j*]). This minimum element always serves as the nums[i]*nums*[*i*] for the current nums[j]*nums*[*j*]. Thus, we only need to traverse beyond the j^{th}*jth* index to check the nums[k]*nums*[*k*]'s to determine if any of them satisfies the 132 criteria.

|  |
| --- |
| public class Solution {  public boolean find132pattern(int[] nums) {  int min\_i = Integer.MAX\_VALUE;  for (int j = 0; j < nums.length - 1; j++) {  min\_i = Math.min(min\_i, nums[j]);  for (int k = j + 1; k < nums.length; k++) {  if (nums[k] < nums[j] && min\_i < nums[k])  return true;  }  }  return false;  }  } |

**Complexity Analysis**

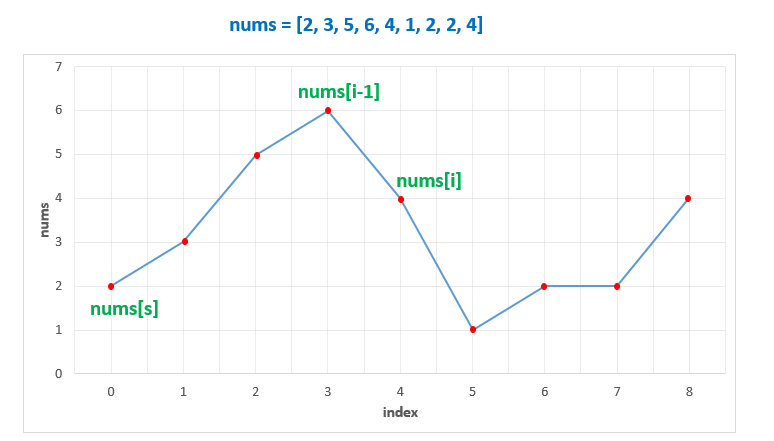
* Time complexity : O(n^2)*O*(*n*2). Two loops are used to find the nums[j],nums[k]*nums*[*j*],*nums*[*k*] pairs. Here, n*n* refers to the size of nums*nums* array.
* Space complexity : O(1)*O*(1). Constant extra space is used.

#### Approach 3: Searching Intervals

**Algorithm**

As discussed in the last approach, once we've fixed a nums[i],nums[j]*nums*[*i*],*nums*[*j*] pair, we just need to determine a nums[k]*nums*[*k*] which falls in the range (nums[i],nums[j])(*nums*[*i*],*nums*[*j*]). Further, to maximize the likelihood of any arbitrary nums[k]*nums*[*k*] falling in this range, we need to try to keep this range as much as possible. But, in the last approach, we tried to work only on nums[i]*nums*[*i*]. But, it'll be a better choice, if we can somehow work out on nums[j]*nums*[*j*] as well.

To do so, we can look at the given nums*nums* array in the form of a graph, as shown below:



From the above graph, which consists of rising and falling slopes, we know, the best qualifiers to act as the nums[i],nums[j]*nums*[*i*],*nums*[*j*] pair, as discussed above, to maximize the range nums[i], nums[j]*nums*[*i*],*nums*[*j*], at any instant, while traversing the nums*nums* array, will be the points at the endpoints of a local rising slope. Thus, once we've found such points, we can traverse over the nums*nums* array to find a nums[k]*nums*[*k*] satisfying the given 132 criteria.

To find these points at the ends of a local rising slope, we can traverse over the given nums*nums* array. While traversing, we can keep a track of the minimum point found after the last peak(nums[s]*nums*[*s*]).

Now, whenever we encounter a falling slope, say, at index i*i*, we know, that nums[i-1]*nums*[*i*−1] was the endpoint of the last rising slope found. Thus, we can scan over the k*k* indices(k>i), to find a 132 pattern.

But, instead of traversing over nums*nums* to find a k*k* satisfying the 132 pattern for every such rising slope, we can store this range (nums[s], nums[i-1])(*nums*[*s*],*nums*[*i*−1])(acting as (nums[i], nums[j])(*nums*[*i*],*nums*[*j*])) in, say an intervals*intervals* array.

While traversing over the nums*nums* array to check the rising/falling slopes, whenever we find any rising slope, we can keep adding the endpoint pairs to this intervals*intervals* array. At the same time, we can also check if the current element falls in any of the ranges found so far. If so, this element satisfies the 132 criteria for that range.

If no such element is found till the end, we need to return a False value.

|  |
| --- |
| public class Solution {  public boolean find132pattern(int[] nums) {  List < int[] > intervals = new ArrayList < > ();  int i = 1, s = 0;  while (i < nums.length) {  if (nums[i] < nums[i - 1]) {  if (s < i - 1)  intervals.add(new int[] {nums[s], nums[i - 1]});  s = i;  }  for (int[] a: intervals)  if (nums[i] > a[0] && nums[i] < a[1])  return true;  i++;  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). We traverse over the nums*nums* array of size n*n* once to find the slopes. But for every element, we also need to traverse over the intervals*intervals* to check if any element falls in any range found so far. This array can contain atmost (n/2)(*n*/2) pairs, in the case of an alternate increasing-decreasing sequence(worst case e.g.[5 6 4 7 3 8 2 9]).
* Space complexity : O(n)*O*(*n*). intervals*intervals* array can contain atmost n/2*n*/2 pairs, in the worst case(alternate increasing-decreasing sequence).

#### Approach 4: Stack

**Algorithm**

In Approach 2, we found out nums[i]*nums*[*i*] corresponding to a particular nums[j]*nums*[*j*] directly without having to consider every pair possible in nums*nums* to find this nums[i],nums[j]*nums*[*i*],*nums*[*j*] pair. If we do some preprocessing, we can make the process of finding a nums[k]*nums*[*k*] corresponding to this nums[i],nums[j]*nums*[*i*],*nums*[*j*] pair also easy.

The preprocessing required is to just find the best nums[i]*nums*[*i*] value corresponding to every nums[j]*nums*[*j*] value. This is done in the same manner as in the second approach i.e. we find the minimum element found till the j^{th}*jth* element which acts as the nums[i]*nums*[*i*] for the current nums[j]*nums*[*j*]. We maintain thes values in a min*min* array. Thus, min[j]*min*[*j*] now refers to the best nums[i]*nums*[*i*] value for a particular nums[j]*nums*[*j*].

Now, we traverse back from the end of the nums*nums* array to find the nums[k]*nums*[*k*]'s. Suppose, we keep a track of the nums[k]*nums*[*k*] values which can potentially satisfy the 132 criteria for the current nums[j]*nums*[*j*]. We know, one of the conditions to be satisfied by such a nums[k]*nums*[*k*] is that it must be greater than nums[i]*nums*[*i*]. Or in other words, we can also say that it must be greater than min[j]*min*[*j*] for a particular nums[j]*nums*[*j*] chosen.

Once it is ensured that the elements left for competing for the nums[k]*nums*[*k*] are all greater than min[j]*min*[*j*](or nums[i]*nums*[*i*]), our only task is to ensure that it should be lesser than nums[j]*nums*[*j*]. Now, the best element from among the competitors, for satisfying this condition will be the minimum one from out of these elements.

If this element, nums[k]*nums*[*k*] satisfies nums[k] < nums[j]*nums*[*k*]<*nums*[*j*], we've found a 132 pattern. If not, no other element will satisfy this criteria, since they are all greater than or equal to nums[min]*nums*[*min*] and thus greater than or equal to nums[j]*nums*[*j*] as well.

To keep a track of these potential nums[k]*nums*[*k*] values for a particular nums[i],nums[j]*nums*[*i*],*nums*[*j*] considered currently, we maintain a stack*stack* on which these potential nums[k]*nums*[*k*]'s satisfying the 132 criteria lie in a descending order(minimum element on the top). We need not sort these elements on the stack*stack*, but they'll be sorted automatically as we'll discuss along with the process.

After creating a min*min* array, we start traversing the nums[j]*nums*[*j*] array in a backward manner. Let's say, we are currently at the j^{th}*jth* element and let's also assume that the stack*stack* is sorted right now. Now, firstly, we check if nums[j] > min[j]*nums*[*j*]>*min*[*j*]. If not, we continue with the (j-1)^{th}(*j*−1)*th* element and the stack*stack* remains sorted. If not, we keep on popping the elements from the top of the stack*stack* till we find an element, stack[top]*stack*[*top*] such that, stack[top] > min[j]*stack*[*top*]>*min*[*j*](or stack[top] > nums[i]*stack*[*top*]>*nums*[*i*]).

Once the popping is done, we're sure that all the elements pending on the stack*stack* are greater than nums[i]*nums*[*i*] and are thus, the potential candidates for nums[k]*nums*[*k*] satisfying the 132 criteria. We can also note that the elements which have been popped from the stack*stack*, all satisfy stack[top] ≤ min[j].

Since, in the min*min* array, min[p] ≤ min[q], for every p > q*p*>*q*, these popped elements also satisfy stack[top] ≤ min[k], for all 0 ≤ k < j. Thus, they are not the potential nums[k]*nums*[*k*] candidates for even the preceding elements. Even after doing the popping, the stack*stack* remains sorted.

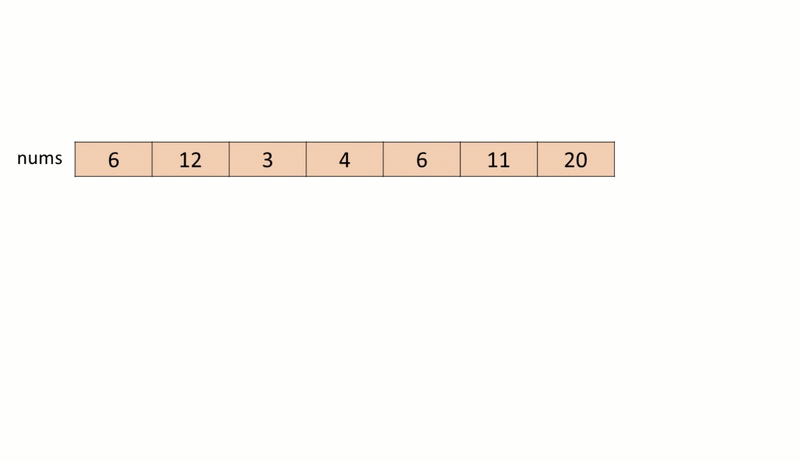
After the popping is done, we've got the minimum element from amongst all the potential nums[k]*nums*[*k*]'s on the top of the stack*stack*(as per the assumption). We can check if it is less than or equal to nums[j]*nums*[*j*] to satisfy the 132 criteria(we've already checked stack[top] > nums[i]*stack*[*top*]>*nums*[*i*]). If this element satisfies the 132 criteria, we can return a True value. If not, we know that for the current j*j*, nums[j] > min[j]*nums*[*j*]>*min*[*j*]. Thus, the element nums[j]*nums*[*j*] could be a potential nums[k]*nums*[*k*] value, for the preceding nums[i]'s*nums*[*i*]′*s*.

Thus, we push it over the stack*stack*. We can note that, we need to push this element nums[j]*nums*[*j*] on the stack*stack* only when it didn't satisfy stack[top]<nums[j]*stack*[*top*]<*nums*[*j*]. Thus, nums[j] ≤ stack[top]. Thus, even after pushing this element on the stack*stack*, the stack*stack* remains sorted. Thus, we've seen by induction, that the stack*stack* always remains sorted.

Also, note that in case nums[j] ≤ min[j], we don't push nums[j]*nums*[*j*] onto the stack*stack*. This is because this nums[j]*nums*[*j*] isn't greater than even the minimum element lying towards its left and thus can't act as nums[k]*nums*[*k*] in the future.

If no element is found satisfying the 132 criteria till reaching the first element, we return a False value.

The following animation better illustrates the process.



|  |
| --- |
| public class Solution {  public boolean find132pattern(int[] nums) {  if (nums.length < 3)  return false;  Stack < Integer > stack = new Stack < > ();  int[] min = new int[nums.length];  min[0] = nums[0];  for (int i = 1; i < nums.length; i++)  min[i] = Math.min(min[i - 1], nums[i]);  for (int j = nums.length - 1; j >= 0; j--) {  if (nums[j] > min[j]) {  while (!stack.isEmpty() && stack.peek() <= min[j])  stack.pop();  if (!stack.isEmpty() && stack.peek() < nums[j])  return true;  stack.push(nums[j]);  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). We travesre over the nums*nums* array of size n*n* once to fill the min*min* array. After this, we traverse over nums*nums* to find the nums[k]*nums*[*k*]. During this process, we also push and pop the elements on the stack*stack*. But, we can note that atmost n*n* elements can be pushed and popped off the stack*stack* in total. Thus, the second traversal requires only O(n)*O*(*n*) time.
* Space complexity : O(n)*O*(*n*). The stack*stack* can grow upto a maximum depth of n*n*. Furhter, min*min* array of size n*n* is used.

#### Approach 5: Binary Search

**Algorithm**

In the last approach, we've made use of a separate stack*stack* to push and pop the nums[k]*nums*[*k*]'s. But, we can also note that when we reach the index j*j* while scanning backwards for finding nums[k]*nums*[*k*], the stack*stack* can contain atmost n-j-1*n*−*j*−1 elements. Here, n*n* refers to the number of elements in nums*nums* array.

We can also note that this is the same number of elements which lie beyond the j^{th}*jth* index in nums*nums* array. We also know that these elements lying beyond the j^{th}*jth* index won't be needed in the future ever again. Thus, we can make use of this space in nums*nums* array instead of using a separate stack*stack*. The rest of the process can be carried on in the same manner as discussed in the last approach.

We can try to go for another optimization here. Since, we've got an array for storing the potential nums[k]*nums*[*k*] values now, we need not do the popping process for a min[j]*min*[*j*] to find an element just larger than min[j]*min*[*j*] from amongst these potential values.

Instead, we can make use of Binary Search to directly find an element, which is just larger than min[j]*min*[*j*] in the required interval, if it exists. If such an element is found, we can compare it with nums[j]*nums*[*j*] to check the 132 criteria. Otherwise, we continue the process as in the last approach.

|  |
| --- |
| public class Solution {  public boolean find132pattern(int[] nums) {  if (nums.length < 3)  return false;  int[] min = new int[nums.length];  min[0] = nums[0];  for (int i = 1; i < nums.length; i++)  min[i] = Math.min(min[i - 1], nums[i]);  for (int j = nums.length - 1, k = nums.length; j >= 0; j--) {  if (nums[j] > min[j]) {  k = Arrays.binarySearch(nums, k, nums.length, min[j] + 1);  if (k < 0)  k = -1 - k;  if (k < nums.length && nums[k] < nums[j])  return true;  nums[--k] = nums[j];  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : O\big(n \log n\big)*O*(*n*log*n*). Filling min*min* array requires O(n)*O*(*n*) time. The second traversal is done over the whole nums*nums* array of length n*n*. For every current nums[j]*nums*[*j*] we need to do the Binary Search, which requires O\big(\log n\big)*O*(log*n*). In the worst case, this Binary Search will be done for all the n*n* elements, and the required element won't be found in any case, leading to a complexity of O\big(n \log n\big)*O*(*n*log*n*).
* Space complexity : O(n)*O*(*n*). min*min* array of size n*n* is used.

#### Approach 6: Using Array as a Stack

**Algorithm**

In the last approach, we've seen that in the worst case, the required element won't be found for all the n*n* elements and thus Binary Search is done at every step increasing the time complexity.

To remove this problem, we can follow the same steps as in Approach 4 i.e. We can remove those elements(update the index k*k*) which aren't greater than nums[i]*nums*[*i*](min[j]*min*[*j*]). Thus, in case no element is larger than min[j]*min*[*j*] the index k*k* reaches the last element.

Now, at every step, only nums[j]*nums*[*j*] will be added and removed from consideration in the next step, improving the time complexity in the worst case. The rest of the method remains the same as in Approach 4.

This approach is inspired by [@fun4leetcode](https://leetcode.com/fun4leetcode/)

|  |
| --- |
| public class Solution {  public boolean find132pattern(int[] nums) {  if (nums.length < 3)  return false;  int[] min = new int[nums.length];  min[0] = nums[0];  for (int i = 1; i < nums.length; i++)  min[i] = Math.min(min[i - 1], nums[i]);  for (int j = nums.length - 1, k = nums.length; j >= 0; j--) {  if (nums[j] > min[j]) {  while (k < nums.length && nums[k] <= min[j])  k++;  if (k < nums.length && nums[k] < nums[j])  return true;  nums[--k] = nums[j];  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). We travesre over the nums*nums* array of size n*n* once to fill the min*min* array. After this, we traverse over nums*nums* to find the nums[k]*nums*[*k*]. Atmost n*n* elements can be put in and out of the nums*nums* array in total. Thus, the second traversal requires only O(n)*O*(*n*) time.
* Space complexity : O(n)*O*(*n*). min*min* array of size n*n* is used.

**Bag of Tokens**

You have an initial **power** of P, an initial **score** of 0, and a bag of tokens where tokens[i] is the value of the ith token (0-indexed).

Your goal is to maximize your total **score** by potentially playing each token in one of two ways:

* If your current **power** is at least tokens[i], you may play the ith token face up, losing tokens[i] **power** and gaining 1 **score**.
* If your current **score** is at least 1, you may play the ith token face down, gaining tokens[i] **power** and losing 1 **score**.

Each token may be played **at most** once and **in any order**. You do **not** have to play all the tokens.

Return the largest possible ***score*** you can achieve after playing any number of tokens.

**Example 1:**

**Input:** tokens = [100], P = 50

**Output:** 0

**Explanation:** Playing the only token in the bag is impossible because you either have too little power or too little score.

**Example 2:**

**Input:** tokens = [100,200], P = 150

**Output:** 1

**Explanation:** Play the 0th token (100) face up, your power becomes 50 and score becomes 1.

There is no need to play the 1st token since you cannot play it face up to add to your score.

**Example 3:**

**Input:** tokens = [100,200,300,400], P = 200

**Output:** 2

**Explanation:** Play the tokens in this order to get a score of 2:

1. Play the 0th token (100) face up, your power becomes 100 and score becomes 1.

2. Play the 3rd token (400) face down, your power becomes 500 and score becomes 0.

3. Play the 1st token (200) face up, your power becomes 300 and score becomes 1.

4. Play the 2nd token (300) face up, your power becomes 0 and score becomes 2.

**Constraints:**

* 0 <= tokens.length <= 1000
* 0 <= tokens[i], P < 104

## Solution

#### Approach 1: Greedy

**Intuition**

If we play a token face up, we might as well play the one with the smallest value. Similarly, if we play a token face down, we might as well play the one with the largest value.

**Algorithm**

We don't need to play anything until absolutely necessary. Let's play tokens face up until we can't, then play a token face down and continue.

We must be careful, as it is easy for our implementation to be incorrect if we do not handle corner cases correctly. We should always play tokens face up until exhaustion, then play one token face down and continue.

Our loop must be constructed with the right termination condition: we can either play a token face up or face down.

Our final answer could be any of the intermediate answers we got after playing tokens face up (but before playing them face down.)

|  |
| --- |
| class Solution {  public int bagOfTokensScore(int[] tokens, int P) {  Arrays.sort(tokens);  int lo = 0, hi = tokens.length - 1;  int points = 0, ans = 0;  while (lo <= hi && (P >= tokens[lo] || points > 0)) {  while (lo <= hi && P >= tokens[lo]) {  P -= tokens[lo++];  points++;  }  ans = Math.max(ans, points);  if (lo <= hi && points > 0) {  P += tokens[hi--];  points--;  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of tokens.
* Space complexity : \mathcal{O}(N)O(*N*) or \mathcal{O}(\log{N})O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the sorted() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is \mathcal{O}(N)O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is \mathcal{O}(\log{N})O(log*N*).

**Stone Game IV**

Alice and Bob take turns playing a game, with Alice starting first.

Initially, there are n stones in a pile.  On each player's turn, that player makes a move consisting of removing **any** non-zero **square number** of stones in the pile.

Also, if a player cannot make a move, he/she loses the game.

Given a positive integer n. Return True if and only if Alice wins the game otherwise return False, assuming both players play optimally.

**Example 1:**

**Input:** n = 1

**Output:** true

**Explanation:** Alice can remove 1 stone winning the game because Bob doesn't have any moves.

**Example 2:**

**Input:** n = 2

**Output:** false

**Explanation:** Alice can only remove 1 stone, after that Bob removes the last one winning the game (2 -> 1 -> 0).

**Example 3:**

**Input:** n = 4

**Output:** true

**Explanation:** n is already a perfect square, Alice can win with one move, removing 4 stones (4 -> 0).

**Example 4:**

**Input:** n = 7

**Output:** false

**Explanation:** Alice can't win the game if Bob plays optimally.

If Alice starts removing 4 stones, Bob will remove 1 stone then Alice should remove only 1 stone and finally Bob removes the last one (7 -> 3 -> 2 -> 1 -> 0).

If Alice starts removing 1 stone, Bob will remove 4 stones then Alice only can remove 1 stone and finally Bob removes the last one (7 -> 6 -> 2 -> 1 -> 0).

**Example 5:**

**Input:** n = 17

**Output:** false

**Explanation:** Alice can't win the game if Bob plays optimally.

**Constraints:**

* 1 <= n <= 10^5

   Hide Hint #1

Use dynamic programming to keep track of winning and losing states. Given some number of stones, Alice can win if she can force Bob onto a losing state.

## Solution

### **Overview**

You probably can guess from the problem title, this is the fourth problem in the series of [Stone Game](https://leetcode.com/problems/stone-game/) problems. Those problems are similar, but have considerable differences, making their solutions quite different. It's highly recommended to finish them all.

Here, two approaches are introduced: DFS with memorization and DP approach.

#### Approach 1: DFS with memorization

**Intuition**

First, let's analyze the problem.

According to [Zermelo's\_theorem](https://en.wikipedia.org/wiki/Zermelo%27s_theorem_(game_theory)), given n stones, either Alice has a must-win strategy, or Bob has one. Therefore, for Alice, the current state is either "must-win" or "must-lose". But how to determine which one it is?

For convenience, in the following context, "the current player" refers to the player now removing the stones, and "state i" refers to when there is i stones remaining.

Now the problem asks whether the current player will win in state n.

If we can go to a known state where Bob must lose, then we know Alice must win in the current state. All Alice has to do is to move the corresponding number of stones to go to that state. Therefore we need to find out which state Bob must lose.

Note that after going to the next state, Bob becomes the player removing the stones, which is the position of Alice in the current state. Therefore, to find out whether Bob will lose in the next state, we just need to check whether our function gives False for remaining stones.

**Algorithm**

Let function dfs(remain) represents whether the current player must win with remain stones remaining.

To find out the result of dfs(n), we need to iterate k from 0 to check whether there exits dfs(remain - k\*k)==False. To prevent redundant calculate, use a map to store the result of dfs function.

Don't forget the base case dfs(0)==False and dfs(1)==True.

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

|  |
| --- |
| class Solution {  public boolean winnerSquareGame(int n) {  HashMap<Integer, Boolean> cache = new HashMap<>();  cache.put(0, false);  return dfs(cache, n);  }  public static boolean dfs(HashMap<Integer, Boolean> cache, int remain) {  if (cache.containsKey(remain)) {  return cache.get(remain);  }  int sqrt\_root = (int) Math.sqrt(remain);  for (int i = 1; i <= sqrt\_root; i++) {  // if there is any chance to make the opponent lose the game in the next round,  // then the current player will win.  if (!dfs(cache, remain - i \* i)) {  cache.put(remain, true);  return true;  }  }  cache.put(remain, false);  return false;  }  } |

There some tricks that we used in the code above.

In the Python code, we use @lru\_cache instead of a map to store the result of dfs. It's a useful grammar sugar in Python.

In the Java code, we don't have things like @lru\_cache in Python, so here we use a simple HashMap. However, we can still use some tricks, if you want -- using an array instead of a map: we can use 0 to mark the unvisited nodes, use 1 to mark the true results, and use 2 to mark the false results. Also, we can just use an array of bytes, which uses less memory than an array of ints.

Note that the speed would be a little faster if you iterate i from sqrt\_root to 0 due to the data characteristics.

**Complexity Analysis**

Assume N*N* is the length of arr.

* Time complexity: \mathcal{O}(N\sqrt{N})O(*NN*​) since we spend \mathcal{O}(\sqrt{N})O(*N*​) at most for each dfs call, and there are \mathcal{O}({N})O(*N*) dfs calls in total.
* Space complexity: \mathcal{O}(N)O(*N*) since we need spaces of \mathcal{O}(N)O(*N*) to store the result of dfs.

#### Approach 2: DP

**Intuition**

DFS with memorization is very similar to dp. We can also use dp to solve this problem.

We can just use a single dp[i] array to store whether the player now removing stones wins with i stones remaining.

**Algorithm**

Let dp[i] represents the result of the game with i stones. dp[i]==True means the current player must win, and dp[i]==False means the current player must lose, when both players play optimally.

The next step is to find out how to calculate dp[i].

We can iterate all possible movements, and check if we can move to a False state. If we can, then we found a must-win strategy, otherwise, we must lose since the opponent has a must-win strategy in this case.

More clearly, we can iterate k from 0 and check if there exists dp[i - k\*k]==False. Of course, i - k\*k >= 0.

Finally, we only need to return dp[n].

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

|  |
| --- |
| class Solution {  public boolean winnerSquareGame(int n) {  boolean[] dp = new boolean[n + 1];  for (int i = 0; i < n + 1; i++) {  for (int k = 1; k \* k <= i; k++) {  if (dp[i - k \* k] == false) {  dp[i] = true;  break;  }  }  }  return dp[n];  }  } |

Also, we can employ DP in a slightly different way.

**Intuition**

Let's think in the backtrack way. If we have a state i that we know the current player must lose, what can we infer?

-- Any other states that can go to i must be True.

Let's say in another state j the current player in j can go to i by removing stones. In this case, the state j is True since the current player must win.

How to find all the state j? Well, we can iterate over the square numbers and add them to i.

**Algorithm**

Still, let dp[i] represent the result of the game of i stones. dp[i]==True means the first player (Alice) must win, and dp[i]==False means the second player (Bob) must win when both players play optimally.

When we get to a False state, we mark all accessible states as True. When we get to a True state, just continue (Why? Well... because it's useless).

Finally, we only need to return dp[n].

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

|  |
| --- |
| class Solution {  public boolean winnerSquareGame(int n) {  boolean[] dp = new boolean[n + 1];  for (int i = 0; i <= n; i++) {  if (dp[i]) {  continue;  }  for (int k = 1; i + k \* k <= n; k++) {  dp[i + k \* k] = true;  }  }  return dp[n];  }  } |

**Complexity Analysis**

Assume N*N* is the length of arr.

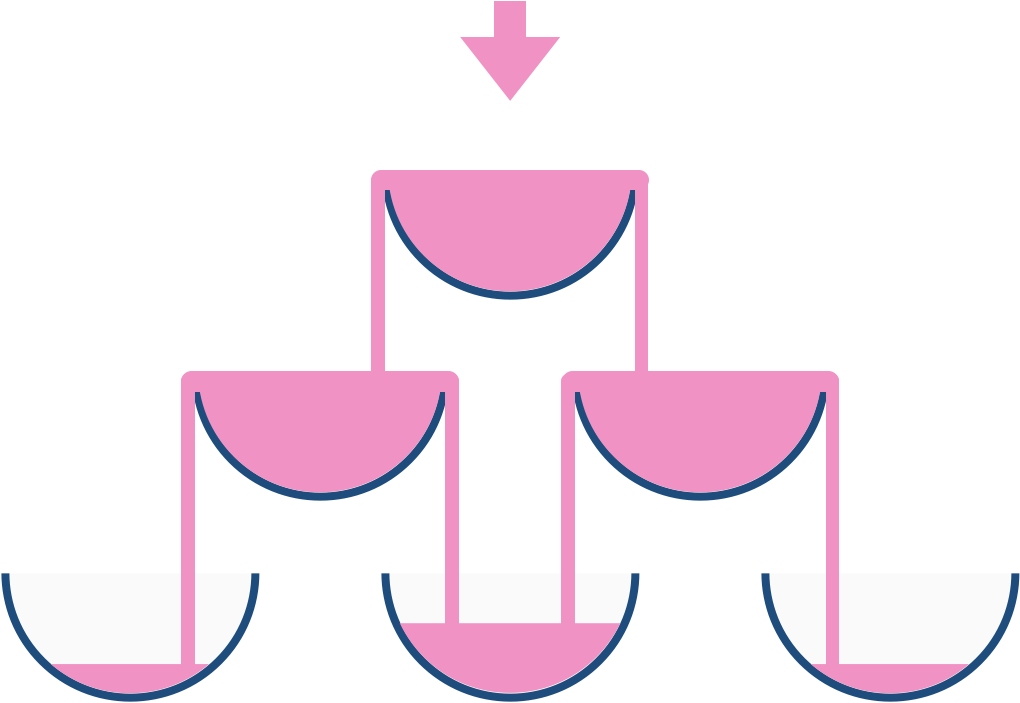
* Time complexity: \mathcal{O}(N\sqrt{N})O(*NN*​) since we iterate over the dp array and spend \mathcal{O}(\sqrt{N})O(*N*​) at most on each element.
* Space complexity: \mathcal{O}(N)O(*N*) since we need a dp array.

**Champagne Tower**

We stack glasses in a pyramid, where the **first** row has 1 glass, the **second** row has 2 glasses, and so on until the 100th row.  Each glass holds one cup of champagne.

Then, some champagne is poured into the first glass at the top.  When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it.  When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on.  (A glass at the bottom row has its excess champagne fall on the floor.)

For example, after one cup of champagne is poured, the top most glass is full.  After two cups of champagne are poured, the two glasses on the second row are half full.  After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now.  After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.



Now after pouring some non-negative integer cups of champagne, return how full the jth glass in the ith row is (both i and j are 0-indexed.)

**Example 1:**

**Input:** poured = 1, query\_row = 1, query\_glass = 1

**Output:** 0.00000

**Explanation:** We poured 1 cup of champange to the top glass of the tower (which is indexed as (0, 0)). There will be no excess liquid so all the glasses under the top glass will remain empty.

**Example 2:**

**Input:** poured = 2, query\_row = 1, query\_glass = 1

**Output:** 0.50000

**Explanation:** We poured 2 cups of champange to the top glass of the tower (which is indexed as (0, 0)). There is one cup of excess liquid. The glass indexed as (1, 0) and the glass indexed as (1, 1) will share the excess liquid equally, and each will get half cup of champange.

**Example 3:**

**Input:** poured = 100000009, query\_row = 33, query\_glass = 17

**Output:** 1.00000

**Constraints:**

* 0 <= poured <= 109
* 0 <= query\_glass <= query\_row < 100

#### Approach #1: Simulation [Accepted]

**Intuition**

Instead of keeping track of how much champagne should end up in a glass, keep track of the total amount of champagne that flows through a glass. For example, if poured = 10 cups are poured at the top, then the total flow-through of the top glass is 10; the total flow-through of each glass in the second row is 4.5, and so on.

**Algorithm**

In general, if a glass has flow-through X, then Q = (X - 1.0) / 2.0 quantity of champagne will equally flow left and right. We can simulate the entire pour for 100 rows of glasses. A glass at (r, c) will have excess champagne flow towards (r+1, c) and (r+1, c+1).

|  |
| --- |
| class Solution {  public double champagneTower(int poured, int query\_row, int query\_glass) {  double[][] A = new double[102][102];  A[0][0] = (double) poured;  for (int r = 0; r <= query\_row; ++r) {  for (int c = 0; c <= r; ++c) {  double q = (A[r][c] - 1.0) / 2.0;  if (q > 0) {  A[r+1][c] += q;  A[r+1][c+1] += q;  }  }  }  return Math.min(1, A[query\_row][query\_glass]);  }  } |

**Complexity Analysis**

* Time Complexity: O(R^2)*O*(*R*2), where R*R* is the number of rows. As this is fixed, we can consider this complexity to be O(1)*O*(1).
* Space Complexity: O(R^2)*O*(*R*2), or O(1)*O*(1) by the reasoning above.

**Summary Ranges**

You are given a **sorted unique** integer array nums.

Return the ***smallest sorted*** list of ranges that ***cover all the numbers in the array exactly***. That is, each element of nums is covered by exactly one of the ranges, and there is no integer x such that x is in one of the ranges but not in nums.

Each range [a,b] in the list should be output as:

* "a->b" if a != b
* "a" if a == b

**Example 1:**

**Input:** nums = [0,1,2,4,5,7]

**Output:** ["0->2","4->5","7"]

**Explanation:** The ranges are:

[0,2] --> "0->2"

[4,5] --> "4->5"

[7,7] --> "7"

**Example 2:**

**Input:** nums = [0,2,3,4,6,8,9]

**Output:** ["0","2->4","6","8->9"]

**Explanation:** The ranges are:

[0,0] --> "0"

[2,4] --> "2->4"

[6,6] --> "6"

[8,9] --> "8->9"

**Example 3:**

**Input:** nums = []

**Output:** []

**Example 4:**

**Input:** nums = [-1]

**Output:** ["-1"]

**Example 5:**

**Input:** nums = [0]

**Output:** ["0"]

**Constraints:**

* 0 <= nums.length <= 20
* -231 <= nums[i] <= 231 - 1
* All the values of nums are **unique**.
* nums is sorted in ascending order.

## Solution

**Intuition**

A range covers consecutive elements. If two adjacent elements have difference larger than 11, the two elements does not belong to the same range.

**Algorithm**

To summarize the ranges, we need to know how to separate them. The array is sorted and without duplicates. In such array, two adjacent elements have difference either 1 or larger than 1. If the difference is 1, they should be put in the same range; otherwise, separate ranges.

We also need to know the start index of a range so that we can put it in the result list. Thus, we keep two indices, representing the two boundaries of current range. For each new element, we check if it extends the current range. If not, we put the current range into the list.

Don't forget to put the last range into the list. One can do this by either a special condition in the loop or putting the last range in to the list after the loop.

**Java**

public class Solution {

public List<String> summaryRanges(int[] nums) {

List<String> summary = new ArrayList<>();

for (int i = 0, j = 0; j < nums.length; ++j) {

// check if j + 1 extends the range [nums[i], nums[j]]

if (j + 1 < nums.length && nums[j + 1] == nums[j] + 1)

continue;

// put the range [nums[i], nums[j]] into the list

if (i == j)

summary.add(nums[i] + "");

else

summary.add(nums[i] + "->" + nums[j]);

i = j + 1;

}

return summary;

}

}

**Java (Alternative)**

public class Solution {

public List<String> summaryRanges(int[] nums) {

List<String> summary = new ArrayList<>();

for (int i, j = 0; j < nums.length; ++j){

i = j;

// try to extend the range [nums[i], nums[j]]

while (j + 1 < nums.length && nums[j + 1] == nums[j] + 1)

++j;

// put the range into the list

if (i == j)

summary.add(nums[i] + "");

else

summary.add(nums[i] + "->" + nums[j]);

}

return summary;

}

}

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Each element is visited constant times: either in comparison with neighbor or put in the result list.
* Space complexity : O(1)*O*(1). All the auxiliary space we need is the two indices, if we don't count the return list.

**Number of Longest Increasing Subsequence**

Given an integer array nums, return the number of longest increasing subsequences.

**Notice** that the sequence has to be **strictly** increasing.

**Example 1:**

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

**Output:** 2

**Explanation:** The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].

**Example 2:**

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

**Output:** 5

**Explanation:** The length of longest continuous increasing subsequence is 1, and there are 5 subsequences' length is 1, so output 5.

**Constraints:**

* 1 <= nums.length <= 2000
* -106 <= nums[i] <= 106

#### Approach 1: Dynamic Programming

**Intuition and Algorithm**

Suppose for sequences ending at nums[i], we knew the length length[i] of the longest sequence, and the number count[i] of such sequences with that length.

For every i < j with A[i] < A[j], we might append A[j] to a longest subsequence ending at A[i]. It means that we have demonstrated count[i] subsequences of length length[i] + 1.

Now, if those sequences are longer than length[j], then we know we have count[i] sequences of this length. If these sequences are equal in length to length[j], then we know that there are now count[i] additional sequences to be counted of that length (ie. count[j] += count[i]).

|  |
| --- |
| class Solution {  public int findNumberOfLIS(int[] nums) {  int N = nums.length;  if (N <= 1) return N;  int[] lengths = new int[N]; //lengths[i] = length of longest ending in nums[i]  int[] counts = new int[N]; //count[i] = number of longest ending in nums[i]  Arrays.fill(counts, 1);  for (int j = 0; j < N; ++j) {  for (int i = 0; i < j; ++i) if (nums[i] < nums[j]) {  if (lengths[i] >= lengths[j]) {  lengths[j] = lengths[i] + 1;  counts[j] = counts[i];  } else if (lengths[i] + 1 == lengths[j]) {  counts[j] += counts[i];  }  }  }  int longest = 0, ans = 0;  for (int length: lengths) {  longest = Math.max(longest, length);  }  for (int i = 0; i < N; ++i) {  if (lengths[i] == longest) {  ans += counts[i];  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2) where N*N* is the length of nums. There are two for-loops and the work inside is O(1)*O*(1).
* Space Complexity: O(N)*O*(*N*), the space used by lengths and counts.

#### Approach 2: Segment Tree

**Intuition**

Suppose we knew for each length L, the number of sequences with length L ending in x. Then when considering the next element of nums, updating our knowledge hinges on knowing the number of sequences with length L-1 ending in y < x. This type of query over an interval is a natural fit for using some sort of tree.

We could try using Fenwick trees, but we would have to store N*N* of them, which naively might be O(N^2)*O*(*N*2) space. Here, we focus on an implementation of a Segment Tree.

**Algorithm**

Implementing Segment Trees is discussed in more detail [here](https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/). In this approach, we will attempt a variant of segment tree that doesn't use lazy propagation.

First, let us call the "value" of an interval, the longest length of an increasing subsequence, and the number of such subsequences in that interval.

Each node knows about the interval of nums values it is considering [node.range\_left, node.range\_right], and it knows about node.val, which contains information on the value of interval. Nodes also have node.left and node.right children that represents the left and right half of the interval node considers. These child nodes are created on demand as appropriate.

Now, query(node, key) will tell us the value of the interval specified by node, except we'll exclude anything above key. When key is outside the interval specified by node, we return the answer. Otherwise, we'll divide the interval into two and query both intervals, then merge the result.

What does merge(v1, v2) do? Suppose two nodes specify adjacent intervals, and have corresponding values v1 = node1.val, v2 = node2.val. What should the aggregate value, v = merge(v1, v2) be? If there are longer subsequences in one node, then v will just be that. If both nodes have longest subsequences of equal length, then we should count subsequences in both nodes. Note that we did not have to consider cases where larger subsequences were made, since these were handled by insert.

What does insert(node, key, val) do? We repeatedly insert the key into the correct half of the interval that node specifies (possibly a point), and after such insertion this node's value could change, so we merge the values together again.

Finally, in our main algorithm, for each num in nums we query for the value length, count of the interval below num, and we know it will lead to count additional sequences of length length + 1. We then update our tree with that knowledge.

|  |
| --- |
| class Solution {  public Value merge(Value v1, Value v2) {  if (v1.length == v2.length) {  if (v1.length == 0) return new Value(0, 1);  return new Value(v1.length, v1.count + v2.count);  }  return v1.length > v2.length ? v1 : v2;  }  public void insert(Node node, int key, Value val) {  if (node.range\_left == node.range\_right) {  node.val = merge(val, node.val);  return;  } else if (key <= node.getRangeMid()) {  insert(node.getLeft(), key, val);  } else {  insert(node.getRight(), key, val);  }  node.val = merge(node.getLeft().val, node.getRight().val);  }  public Value query(Node node, int key) {  if (node.range\_right <= key) return node.val;  else if (node.range\_left > key) return new Value(0, 1);  else return merge(query(node.getLeft(), key), query(node.getRight(), key));  }  public int findNumberOfLIS(int[] nums) {  if (nums.length == 0) return 0;  int min = nums[0], max = nums[0];  for (int num: nums) {  min = Math.min(min, num);  max = Math.max(max, num);  }  Node root = new Node(min, max);  for (int num: nums) {  Value v = query(root, num-1);  insert(root, num, new Value(v.length + 1, v.count));  }  return root.val.count;  }  }  class Node {  int range\_left, range\_right;  Node left, right;  Value val;  public Node(int start, int end) {  range\_left = start;  range\_right = end;  left = null;  right = null;  val = new Value(0, 1);  }  public int getRangeMid() {  return range\_left + (range\_right - range\_left) / 2;  }  public Node getLeft() {  if (left == null) left = new Node(range\_left, getRangeMid());  return left;  }  public Node getRight() {  if (right == null) right = new Node(getRangeMid() + 1, range\_right);  return right;  }  }  class Value {  int length;  int count;  public Value(int len, int ct) {  length = len;  count = ct;  }  } |

**Complexity Analysis**

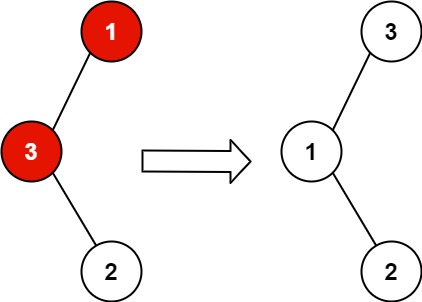
* Time Complexity: O(N\log {N})*O*(*N*log*N*) where N*N* is the length of nums. In our main for loop, we do O(\log{N})*O*(log*N*) work to query and insert.
* Space Complexity: O(N)*O*(*N*), the space used by the segment tree.

**Recover Binary Search Tree**

You are given the root of a binary search tree (BST), where exactly two nodes of the tree were swapped by mistake. Recover the tree without changing its structure.

**Follow up:** A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

**Example 1:**

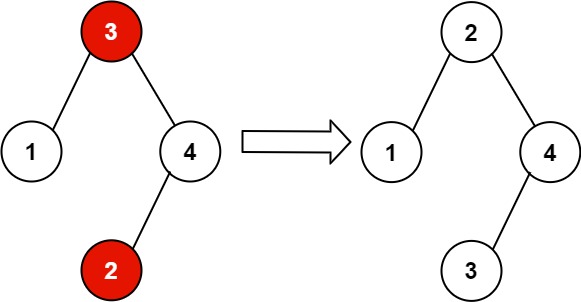


**Input:** root = [1,3,null,null,2]

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

**Explanation:** 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BST valid.

**Example 2:**



**Input:** root = [3,1,4,null,null,2]

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

**Explanation:** 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 makes the BST valid.

**Constraints:**

* The number of nodes in the tree is in the range [2, 1000].
* -231 <= Node.val <= 231 - 1

## Solution

#### Approach 1: Sort an Almost Sorted Array Where Two Elements Are Swapped

**Intuition**

Let's start from straightforward but not optimal solution with a linear time and space complexity. This solution serves to identify and discuss all subproblems.

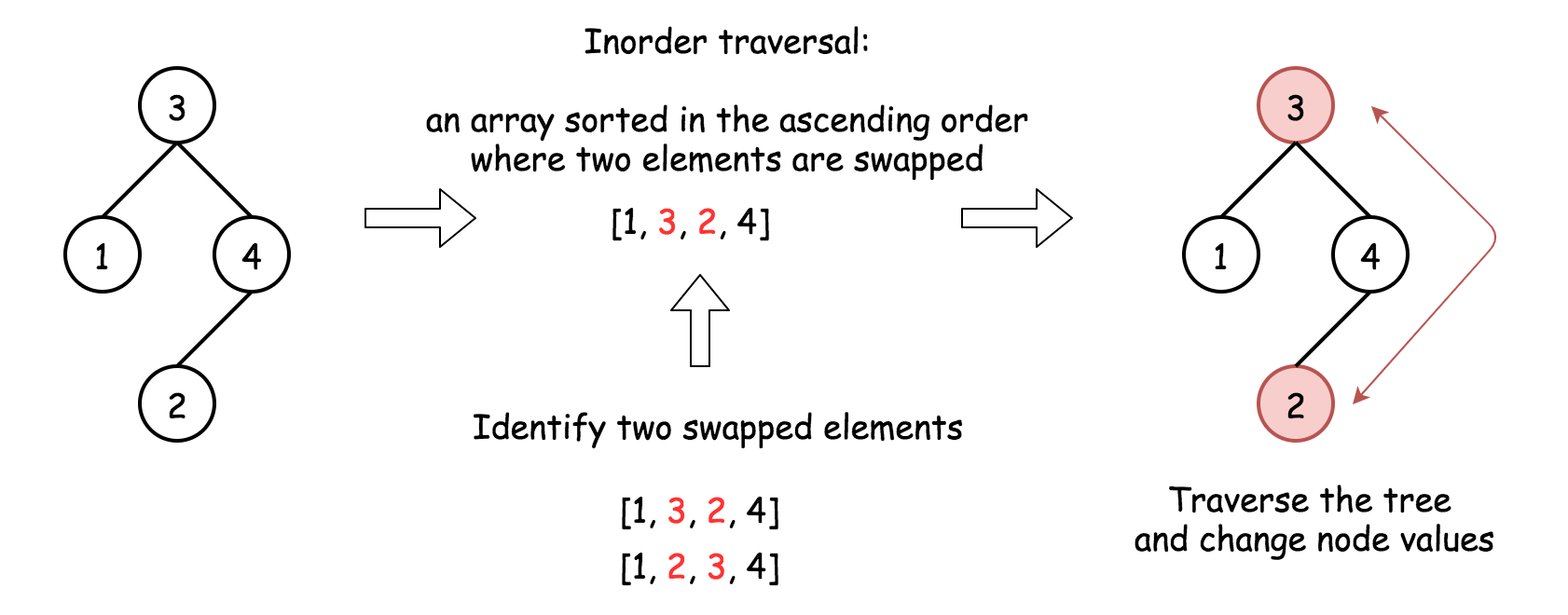
It's known that [inorder traversal of BST is an array sorted in the ascending order](https://leetcode.com/articles/delete-node-in-a-bst/). Here is how one could compute an inorder traversal

|  |
| --- |
| public void inorder(TreeNode root, List<Integer> nums) {  if (root == null) return;  inorder(root.left, nums);  nums.add(root.val);  inorder(root.right, nums);  } |

Here two nodes are swapped, and hence inorder traversal is an almost sorted array where only two elements are swapped. To identify two swapped elements in a sorted array is a classical problem that could be solved in linear time. Here is a solution code

|  |
| --- |
| public int[] findTwoSwapped(List<Integer> nums) {  int n = nums.size();  int x = -1, y = -1;  for(int i = 0; i < n - 1; ++i) {  if (nums.get(i + 1) < nums.get(i)) {  y = nums.get(i + 1);  // first swap occurence  if (x == -1) x = nums.get(i);  // second swap occurence  else break;  }  }  return new int[]{x, y};  } |

When swapped nodes are known, one could traverse the tree again and swap their values.



**Algorithm**

Here is the algorithm:

1. Construct inorder traversal of the tree. It should be an almost sorted list where only two elements are swapped.
2. Identify two swapped elements x and y in an almost sorted array in linear time.
3. Traverse the tree again. Change value x to y and value y to x.

**Implementation**

|  |
| --- |
| class Solution {  public void inorder(TreeNode root, List<Integer> nums) {  if (root == null) return;  inorder(root.left, nums);  nums.add(root.val);  inorder(root.right, nums);  }  public int[] findTwoSwapped(List<Integer> nums) {  int n = nums.size();  int x = -1, y = -1;  for(int i = 0; i < n - 1; ++i) {  if (nums.get(i + 1) < nums.get(i)) {  y = nums.get(i + 1);  // first swap occurence  if (x == -1) x = nums.get(i);  // second swap occurence  else break;  }  }  return new int[]{x, y};  }  public void recover(TreeNode r, int count, int x, int y) {  if (r != null) {  if (r.val == x || r.val == y) {  r.val = r.val == x ? y : x;  if (--count == 0) return;  }  recover(r.left, count, x, y);  recover(r.right, count, x, y);  }  }  public void recoverTree(TreeNode root) {  List<Integer> nums = new ArrayList();  inorder(root, nums);  int[] swapped = findTwoSwapped(nums);  recover(root, 2, swapped[0], swapped[1]);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*). To compute inorder traversal takes \mathcal{O}(N)O(*N*) time, to identify and to swap back swapped nodes \mathcal{O}(N)O(*N*) in the worst case.
* Space complexity : \mathcal{O}(N)O(*N*) since we keep inorder traversal nums with N elements.

#### What Is Coming Next

In approach 1 we discussed three easy subproblems of this hard problem:

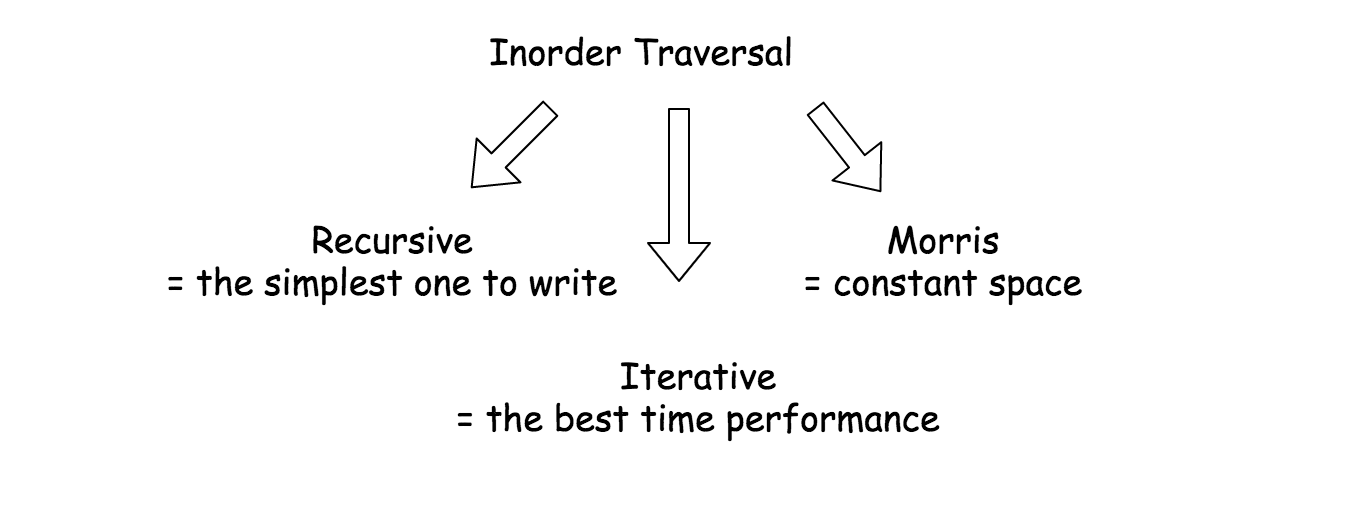
1. Construct inorder traversal.
2. Find swapped elements in an almost sorted array where only two elements are swapped.
3. Swap values of two nodes.

Now we will discuss three more approaches, and basically they are all the same :

* Merge steps 1 and 2, i.e. identify swapped nodes during the inorder traversal.
* Swap node values.

The difference in-between the following approaches is in a chosen method to implement inorder traversal :

* Approach 2 : Iterative.
* Approach 3 : Recursive.
* Approach 4 : Morris.



Iterative and recursive approaches here do less than one pass, and they both need up to \mathcal{O}(H)O(*H*) space to keep stack, where H is a tree height.

Morris approach is two pass approach, but it's a constant-space one.

#### Approach 2: Iterative Inorder Traversal

**Intuition**

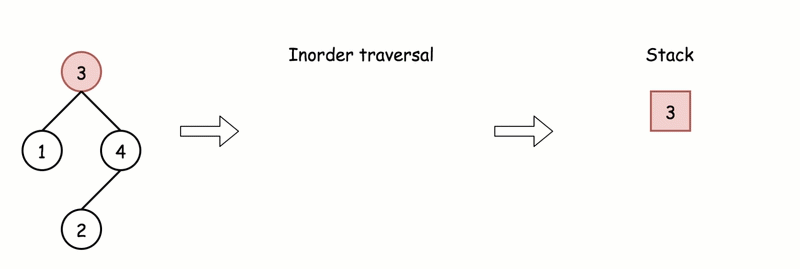
Here we construct inorder traversal by iterations and identify swapped nodes at the same time, in one pass.

Iterative inorder traversal is simple: go left as far as you can, then one step right. Repeat till the end of nodes in the tree.

To identify swapped nodes, track the last node pred in the inorder traversal (i.e. the predecessor of the current node) and compare it with current node value. If the current node value is smaller than its predecessor pred value, the swapped node is here.

There are only two swapped nodes here, and hence one could break after having the second node identified.

Doing so, one could get directly nodes (and not only their values), and hence swap node values in \mathcal{O}(1)O(1) time, drastically reducing the time needed for step 3.



**Implementation**

[Don't use Stack in Java, use ArrayDeque instead](https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html).

|  |
| --- |
| class Solution {  public void swap(TreeNode a, TreeNode b) {  int tmp = a.val;  a.val = b.val;  b.val = tmp;  }  public void recoverTree(TreeNode root) {  Deque<TreeNode> stack = new ArrayDeque();  TreeNode x = null, y = null, pred = null;  while (!stack.isEmpty() || root != null) {  while (root != null) {  stack.add(root);  root = root.left;  }  root = stack.removeLast();  if (pred != null && root.val < pred.val) {  y = root;  if (x == null) x = pred;  else break;  }  pred = root;  root = root.right;  }  swap(x, y);  }  } |

**Complexity Analysis**

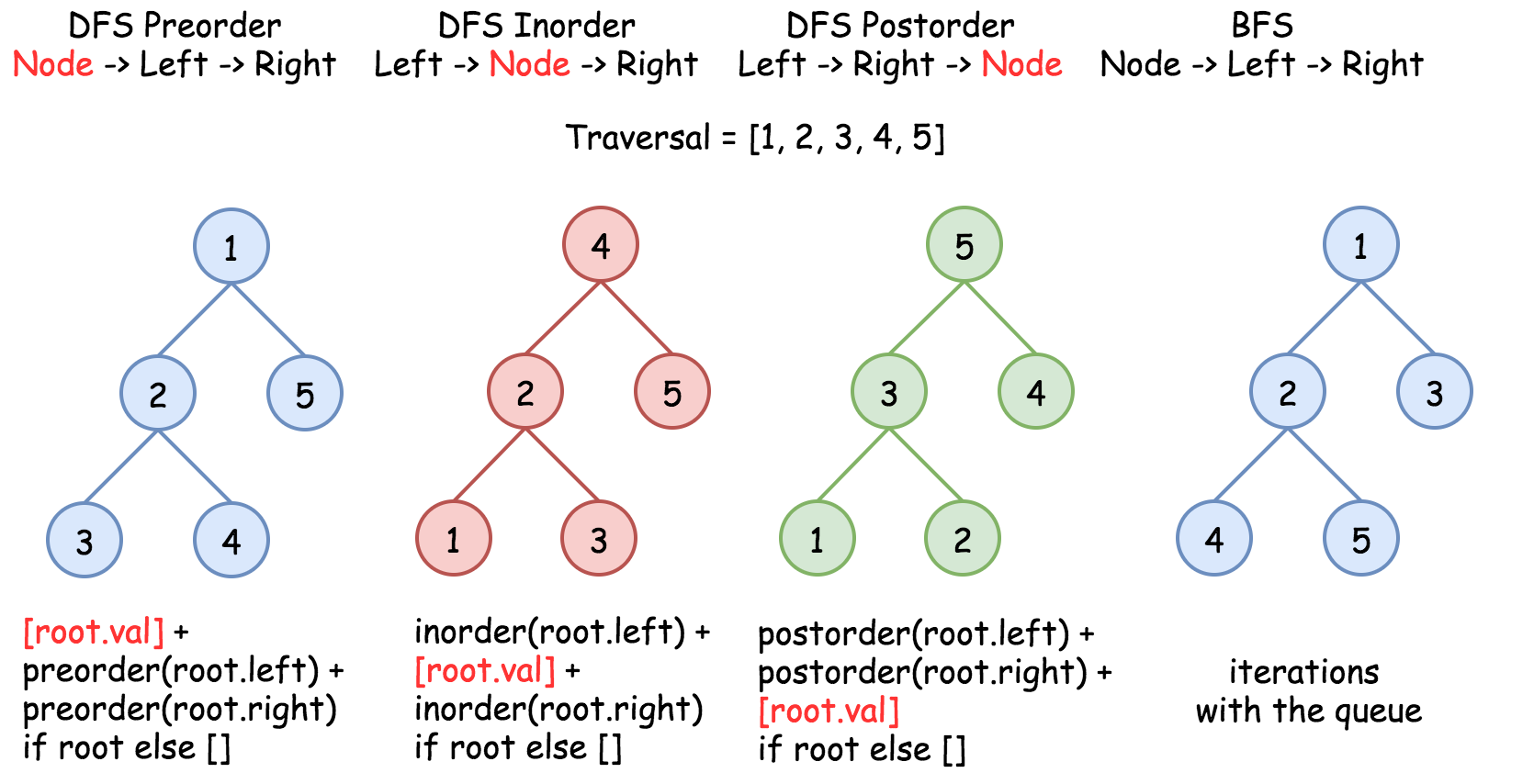
* Time complexity: \mathcal{O}(N)O(*N*) in the worst case when one of the swapped nodes is a rightmost leaf.
* Space complexity : up to \mathcal{O}(N)O(*N*) to keep the stack in the worst case when the tree is completely lean.

#### Approach 3: Recursive Inorder Traversal

Iterative approach 2 could be converted into recursive one.

Recursive inorder traversal is extremely simple: follow Left->Node->Right direction, i.e. do the recursive call for the left child, then do all the business with the node (= if the node is the swapped one or not), and then do the recursive call for the right child.

On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different DFS strategies.



**Implementation**

|  |
| --- |
| class Solution {  TreeNode x = null, y = null, pred = null;  public void swap(TreeNode a, TreeNode b) {  int tmp = a.val;  a.val = b.val;  b.val = tmp;  }  public void findTwoSwapped(TreeNode root) {  if (root == null) return;  findTwoSwapped(root.left);  if (pred != null && root.val < pred.val) {  y = root;  if (x == null) x = pred;  else return;  }  pred = root;  findTwoSwapped(root.right);  }  public void recoverTree(TreeNode root) {  findTwoSwapped(root);  swap(x, y);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*) in the worst case when one of the swapped nodes is a rightmost leaf.
* Space complexity : up to \mathcal{O}(N)O(*N*) to keep the stack in the worst case when the tree is completely lean.

#### Approach 4: Morris Inorder Traversal

We discussed already iterative and recursive inorder traversals, which both have great time complexity though use up to \mathcal{O}(N)O(*N*) to keep stack. We could trade in performance to save space.

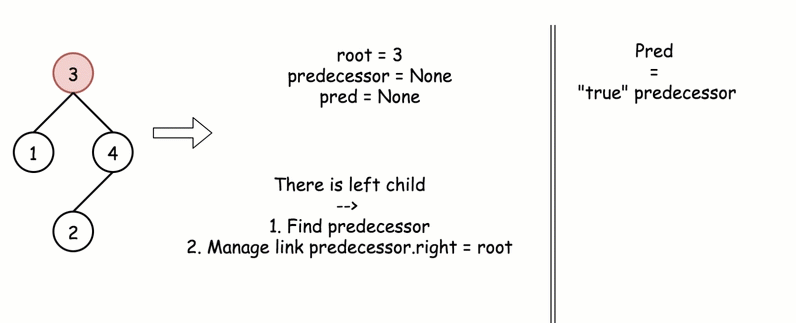
The idea of Morris inorder traversal is simple: to use no space but to traverse the tree.

How that could be even possible? At each node one has to decide where to go: left or right, traverse left subtree or traverse right subtree. How one could know that the left subtree is already done if no additional memory is allowed?

The idea of [Morris](https://www.sciencedirect.com/science/article/pii/0020019079900681) algorithm is to set the temporary link between the node and its [predecessor](https://leetcode.com/articles/delete-node-in-a-bst/): predecessor.right = root. So one starts from the node, computes its predecessor and verifies if the link is present.

* There is no link? Set it and go to the left subtree.
* There is a link? Break it and go to the right subtree.

There is one small issue to deal with : what if there is no left child, i.e. there is no left subtree? Then go straightforward to the right subtree.



|  |
| --- |
| class Solution {  public void swap(TreeNode a, TreeNode b) {  int tmp = a.val;  a.val = b.val;  b.val = tmp;  }  public void recoverTree(TreeNode root) {  // predecessor is a Morris predecessor.  // In the 'loop' cases it could be equal to the node itself predecessor == root.  // pred is a 'true' predecessor,  // the previous node in the inorder traversal.  TreeNode x = null, y = null, pred = null, predecessor = null;  while (root != null) {  // If there is a left child  // then compute the predecessor.  // If there is no link predecessor.right = root --> set it.  // If there is a link predecessor.right = root --> break it.  if (root.left != null) {  // Predecessor node is one step left  // and then right till you can.  predecessor = root.left;  while (predecessor.right != null && predecessor.right != root)  predecessor = predecessor.right;  // set link predecessor.right = root  // and go to explore left subtree  if (predecessor.right == null) {  predecessor.right = root;  root = root.left;  }  // break link predecessor.right = root  // link is broken : time to change subtree and go right  else {  // check for the swapped nodes  if (pred != null && root.val < pred.val) {  y = root;  if (x == null) x = pred;  }  pred = root;  predecessor.right = null;  root = root.right;  }  }  // If there is no left child  // then just go right.  else {  // check for the swapped nodes  if (pred != null && root.val < pred.val) {  y = root;  if (x == null) x = pred;  }  pred = root;  root = root.right;  }  }  swap(x, y);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since we visit each node up to two times.
* Space complexity : \mathcal{O}(1)O(1).