**Hash Table**

**Introduction**

Hash Table is a data structure which organizes data using hash functions in order to support quick insertion and search.

There are two different kinds of hash tables: hash set and hash map.

* The hash set is one of the implementations of a set data structure to store no repeated values.
* The hash map is one of the implementations of a map data structure to store (key, value) pairs.

It is easy to use a hash table with the help of standard template libraries. Most common languages such as Java, C++ and Python support both hash set and hash map.

By choosing a proper hash function, the hash table can achieve wonderful performance in both insertion and search.

In this card, we will answer the following questions:

1. What is the principle of a hash table?
2. How to design a hash table?
3. How to use hash set to solve duplicates related problems?
4. How to use hash map to aggregate information by key?
5. How to design a proper key when using a hash table?

And we also provide exercises for you to be familiar with hash table.

Design a Hash Table

 The Principle of Hash Table

 Keys to Design a Hash Table

 Design HashSet

 Design HashMap

 Design Hash Table - Solution

 Complexity Analysis - Hash Table

Practical Application - Hash Set

 Hash Set - Usage

 Find Duplicates By Hash Set

 Contains Duplicate

 Single Number

 Intersection of Two Arrays

 Happy Number

Practical Application - Hash Map

 Hash Map - Usage

 Scenario I - Provide More Information

 Two Sum

 Isomorphic Strings

 Minimum Index Sum of Two Lists

 Scenario II - Aggregate by Key

 First Unique Character in a String

 Intersection of Two Arrays II

 Contains Duplicate II

 Logger Rate Limiter

Practical Application - Design the Key

 Design the Key

 Group Anagrams

 Group Shifted Strings

 Valid Sudoku

 Find Duplicate Subtrees

 Design the Key - Summary

Conclusion

 Jewels and Stones

 Longest Substring Without Repeating Characters

 Two Sum III - Data structure design

 4Sum II

 Top K Frequent Elements

 Unique Word Abbreviation

 Insert Delete GetRandom O(1)

## Design a Hash Table

In this chapter, we will discuss the underlying principle of the hash table.

After completing this chapter, you should be able to answer the following questions:

1. What is the principle of hash table?
2. Which factors will influence the choice of hash function and collision resolution strategy?
3. Understand the difference between a hash set and a hash map.
4. How to design a simple version of hash set and a hash map as in a typical standard template library.
5. What is the complexity of insertion and lookup operations?

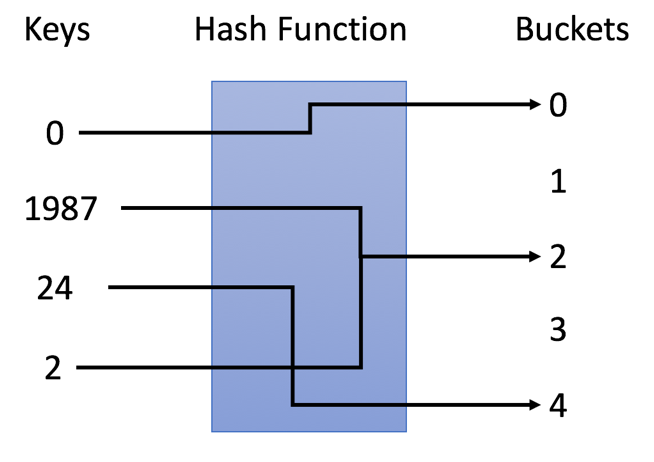
**The Principle of Hash Table**

As we mentioned in the introduction,  Hash Table is a data structure which organizes data using hash functions in order to support quick insertion and search. In this article, we will take a look at the principle of the hash table.

### ***The Principle of Hash Table***

The key idea of Hash Table is to use a hash function to map keys to buckets. To be more specific,

1. When we insert a new key, the hash function will decide which bucket the key should be assigned and the key will be stored in the corresponding bucket;
2. When we want to search for a key, the hash table will use the same hash function to find the corresponding bucket and search only in the specific bucket.



In the example, we use y = x % 5 as our hash function. Let's go through the insertion and search strategies using this example:

1. Insertion: we parse the keys through the hash function to map them into the corresponding bucket.
   * e.g. 1987 is assigned to bucket 2 while 24 is assigned to bucket 4.
2. Search: we parse the keys through the same hash function and search only in the specific bucket.
   * e.g. if we search for 1987, we will use the same hash function to map 1987 to 2. So we search in bucket 2 and we successfully find out 1987 in that bucket.
   * e.g. if we search for 23, will map 23 to 3 and search in bucket 3. And We find out that 23 is not in bucket 3 which means 23 is not in the hash table.

**Keys to Design a Hash Table**

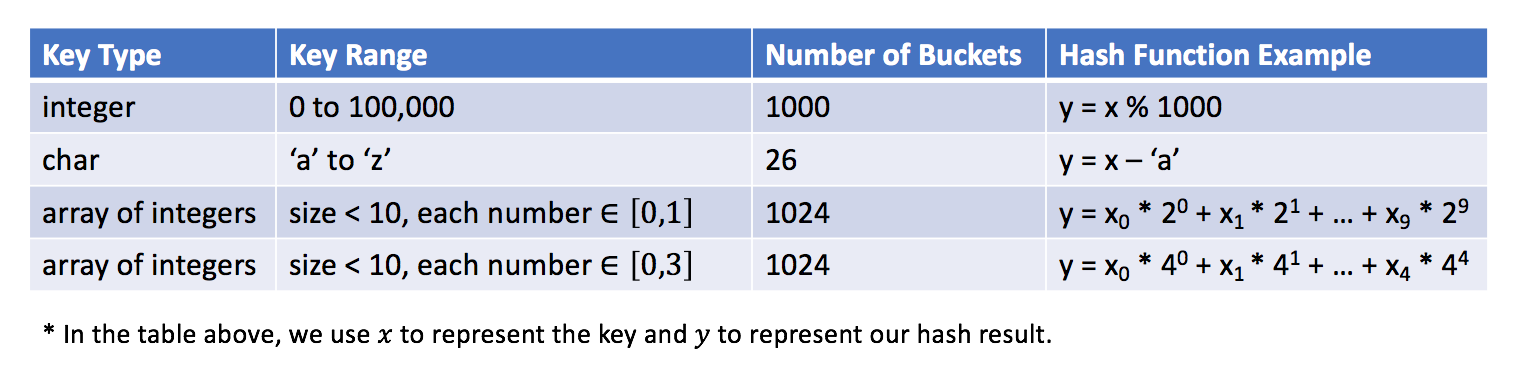
There are two essential factors that you should pay attention to when you are going to design a hash table.

### ***1. Hash Function***

The hash function is the most important component of a hash table which is used to map the key to a specific bucket. In the example in previous article, we use y = x % 5 as a hash function, where x is the key value and y is the index of the assigned bucket.

The hash function will depend on the range of key values and the number of buckets.

Here are some examples of hash functions:



It is an open problem to design a hash function. The idea is to try to assign the key to the bucket as uniform as you can. Ideally, a perfect hash function will be a one-one mapping between the key and the bucket. However, in most cases a hash function is not perfect and it is a tradeoff between the amount of buckets and the capacity of a bucket.

### ***2. Collision Resolution***

Ideally, if our hash function is a perfect one-one mapping, we will not need to handle collisions. Unfortunately, in most cases, collisions are almost inevitable. For instance, in our previous hash function (y = x % 5), both 1987 and 2 are assigned to bucket 2. That is a collision.

A collision resolution algorithm should solve the following questions:

1. How to organize the values in the same bucket?
2. What if too many values are assigned to the same bucket?
3. How to search a target value in a specific bucket?

These questions are related to the capacity of the bucket and the number of keys which might be mapped into the same bucket according to our hash function.

Let's assume that the bucket, which holds the maximum number of keys, has N keys.

Typically, if N is constant and small, we can simply use an array to store keys in the same bucket. If N is variable or large, we might need to use height-balanced binary search tree instead.

### ***Exercise***

By now, you should be able to implement a basic hash table. We provide the exercise for you to implement a hash set and a hash map. Read the requirement, determine your hash function and solve the collision if needed.

If you are not familiar with the concepts of hash set and hash map, you can go back to the introduction part to find out the answer.

Insertion and search are two basic operations in a hash table.

Besides, there are operations which are based on these two operations. For example, when we remove an element, we will first search the element and then remove the element from the corresponding position if the element exists.

**Design HashSet**

Design a HashSet without using any built-in hash table libraries.

To be specific, your design should include these functions:

* add(value): Insert a value into the HashSet.
* contains(value) : Return whether the value exists in the HashSet or not.
* remove(value): Remove a value in the HashSet. If the value does not exist in the HashSet, do nothing.

**Example:**

MyHashSet hashSet = new MyHashSet();

hashSet.add(1);

hashSet.add(2);

hashSet.contains(1);    // returns true

hashSet.contains(3);    // returns false (not found)

hashSet.add(2);

hashSet.contains(2);    // returns true

hashSet.remove(2);

hashSet.contains(2);    // returns false (already removed)

**Note:**

* All values will be in the range of [0, 1000000].
* The number of operations will be in the range of [1, 10000].
* Please do not use the built-in HashSet library.

## Solution

#### **Intuition**

This is a classical question from textbook, which is intended to test one's knowledge on data structure. Therefore, needless to say, it is not desirable to solve the problem with any build-in HashSet data structure.

There are two key questions that one should address, in order to implement the HashSet data structure, namely ***hash function*** and ***collision handling***.

* ***hash function***: the goal of the hash function is to assign an address to store a given value. Ideally, each unique value should have a unique hash value.
* ***collision handling***: since the nature of a hash function is to map a value from a space A into a corresponding value in a **smaller** space B, it could happen that multiple values from space A might be mapped to the same value in space B. This is what we call **collision**. Therefore, it is indispensable for us to have a strategy to handle the collision.

Overall, there are several strategies to resolve the collisions:

* [Separate Chaining](https://en.wikipedia.org/wiki/Hash_table#Separate_chaining): for values with the same hash key, we keep them in a bucket, and each bucket is independent of each other.
* [Open Addressing](https://en.wikipedia.org/wiki/Hash_table#Open_addressing): whenever there is a collision, we keep on probing on the main space with certain strategy until a free slot is found.
* [2-Choice Hashing](https://en.wikipedia.org/wiki/2-choice_hashing): we use two hash functions rather than one, and we pick the generated address with fewer collision.

In this article, we focus on the strategy of ***separate chaining***. Here is how it works overall.

* Essentially, the primary storage underneath a HashSet is a continuous memory as Array. Each element in this array corresponds to a bucket that stores the actual values.
* Given a value, first we generate a key for the value via the hash function. The generated key serves as the index to locate the bucket.
* Once the bucket is located, we then perform the desired operations on the bucket, such as add, remove and contains.

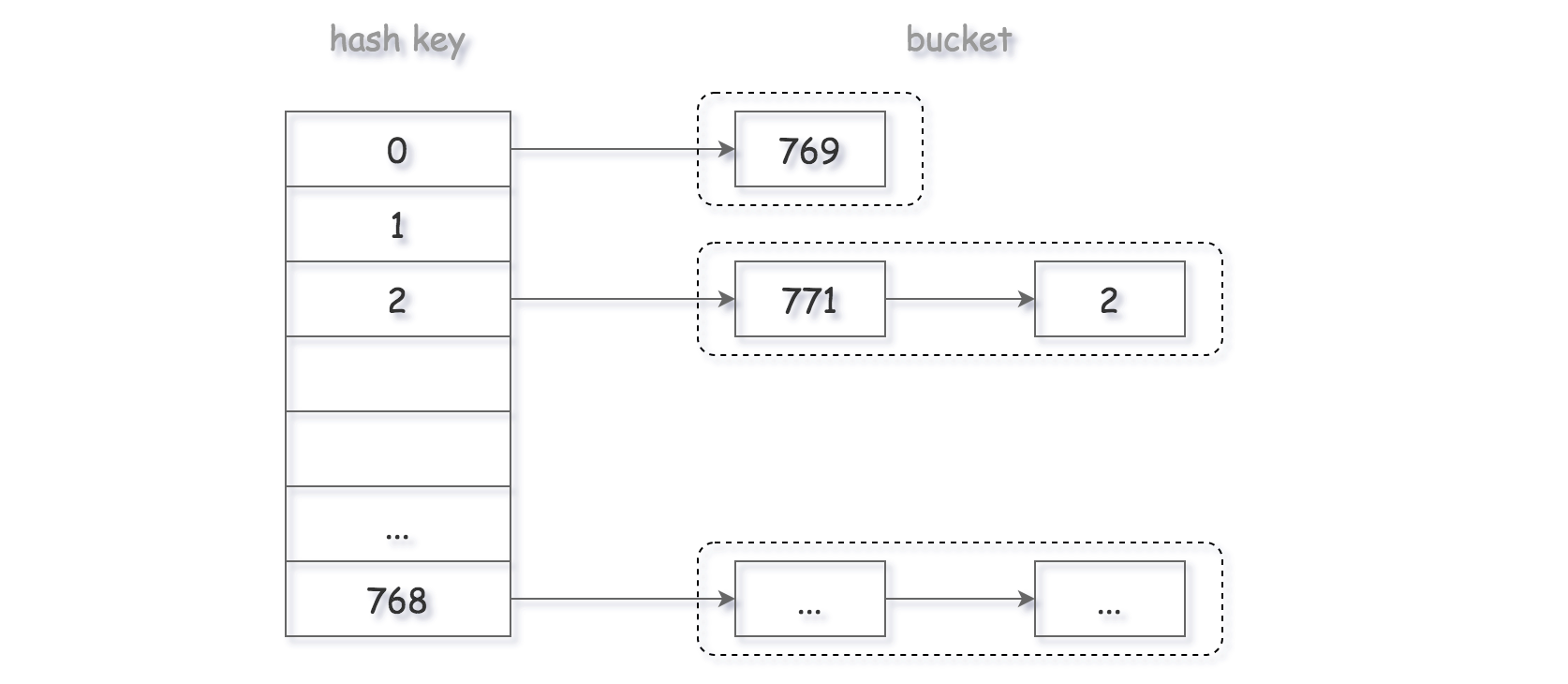
#### **Approach 1: LinkedList as Bucket**

**Intuition**

The common choice of hash function is the modulo operator, i.e. hash=value mod base. Here, the \text{base}base of modulo operation would determine the number of buckets that we would have at the end in the HashSet.

Theoretically, the more buckets we have (hence the larger the space would be), the less likely that we would have collisions. The choice of base is a tradeoff between the space and the collision.

In addition, it is generally advisable to use a prime number as the base of modulo, e.g. 769769, in order to reduce the potential collisions.



As to the design of bucket, again there are several options. One could simply use another Array as bucket to store all the values. However, one drawback with the Array data structure is that it would take O(*N*) time complexity to remove or insert an element, rather than the desired O(1).

Since for any update operation, we would need to scan the entire bucket first to avoid any duplicate, a better choice for the implementation of bucket would be the ***LinkedList***, which has a constant time complexity for the insertion as well as deletion, once we locate the position to update.

**Algorithm**

As we discussed in the above section, here we adopt the LinkedList to implement our bucket within the HashSet.

Essentially, we are implementing a LinkedList that does not contain any duplicate.

For each of the functions of add, remove and contains, we first generate the bucket index with the hash function. Then, we simply pass down the operation to the underlying bucket.

|  |
| --- |
| class MyHashSet {  private Bucket[] bucketArray;  private int keyRange;  /\*\* Initialize your data structure here. \*/  public MyHashSet() {  this.keyRange = 769;  this.bucketArray = new Bucket[this.keyRange];  for (int i = 0; i < this.keyRange; ++i)  this.bucketArray[i] = new Bucket();  }  protected int \_hash(int key) {  return (key % this.keyRange);  }  public void add(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].insert(key);  }  public void remove(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].delete(key);  }  /\*\* Returns true if this set contains the specified element \*/  public boolean contains(int key) {  int bucketIndex = this.\_hash(key);  return this.bucketArray[bucketIndex].exists(key);  }  }  class Bucket {  private LinkedList<Integer> container;  public Bucket() {  container = new LinkedList<Integer>();  }  public void insert(Integer key) {  int index = this.container.indexOf(key);  if (index == -1) {  this.container.addFirst(key);  }  }  public void delete(Integer key) {  this.container.remove(key);  }  public boolean exists(Integer key) {  int index = this.container.indexOf(key);  return (index != -1);  }  }  /\*\*  \* Your MyHashSet object will be instantiated and called as such:  \* MyHashSet obj = new MyHashSet();  \* obj.add(key);  \* obj.remove(key);  \* boolean param\_3 = obj.contains(key);  \*/ |

***Implementation Notes***

In the Python implementation, we employed a sort of **pseudo head** to keep a reference to the actual head of the LinkedList, which could simplify a bit the logic by reducing the number of branchings.

For a value that was never seen before, we insert it to the **head** of the bucket, though we could also append it to the tail. It is a choice that we made, which could **fit better** the scenario where redundant values are operated in nearby time windows, since it is more likely that we spot the value at the head of the bucket rather than walking through the entire bucket.

**Complexity Analysis**

* Time Complexity: O(*N/K*​) where *N* is the number of all possible values and *K* is the number of predefined buckets, which is 769.
  + Assuming that the values are evenly distributed, thus we could consider that the average size of bucket is N/K​.
  + Since for each operation, in the worst case, we would need to scan the entire bucket, hence the time complexity is O(N/K ​).
* Space Complexity: O(*K*+*M*) where *K* is the number of predefined buckets, and *M* is the number of unique values that have been inserted into the HashSet.

Approach 2: Binary Search Tree (BST) as Bucket

**Intuition**

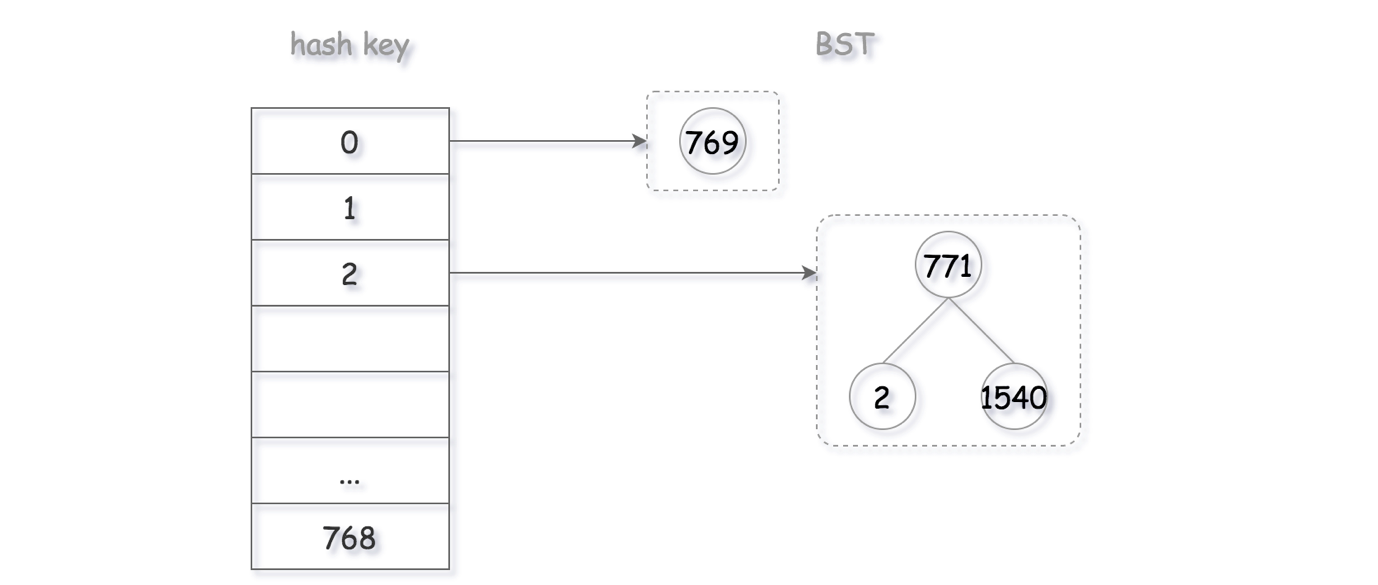
In the above approach, one of the drawbacks is that we have to scan the entire linkedlist in order to verify if a value already exists in the bucket (*i.e.* the lookup operation).

To optimize the above process, one of the strategies could be that we maintain a ***sorted list*** as the bucket. With the sorted list, we could obtain the O(log*N*) time complexity for the lookup operation, with the binary search algorithm, rather than a linear O(*N*) complexity as in the above approach.

On the other hand, if we implement the sorted list in a continuous space such as Array, it would incur a *linear* time complexity for the update operations (*e.g.* *insert* and *delete*), since we would need to shift the elements.

So the question is can we have a data structure that have O(log*N*) time complexity, for the operations of *search*, *insert* and *delete* ?

Well. The answer is yes, with ***Binary Search Tree*** (BST). Thanks to the properties of BST, we could optimize the time complexity of our first approach with LinkedList.



As a result, now the problem is boiled down to the implementation of a standard Binary Search Tree that serves as the *bucket* in the HashSet.

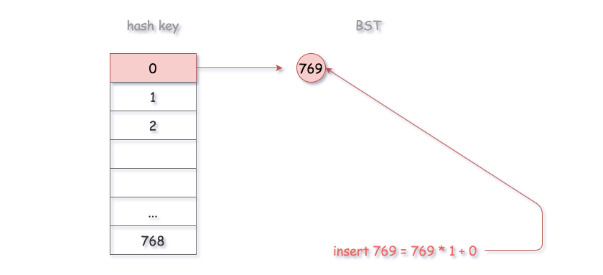
**Algorithm**

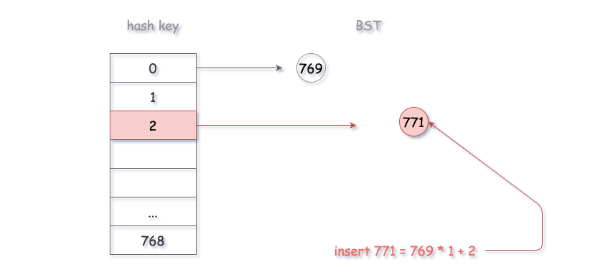
One could build upon the implementation of first approach for our second approach, by applying the [Façade design pattern](https://en.wikipedia.org/wiki/Facade_pattern).

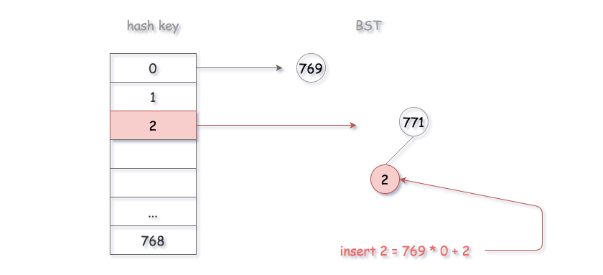
We have already defined a façade class (*i.e.* bucket) with three interfaces (exists, insert and delete), which hides all the underlying details from its users (*i.e.* HashSet).

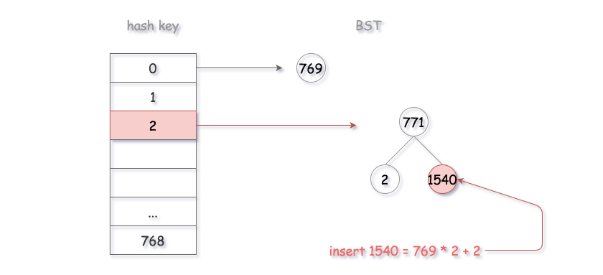
So we can keep the bulk of the code, and simply modify the implementation of bucket class with BST. For each of the interfaces in bucket, it corresponds exactly to an operation in BST.











Actually, we have each of the BST operations listed as an independent problem in LeetCode, as follows:

* [Article 700. Search in a BST](https://leetcode.com/articles/search-in-a-bst/)
* [Article 701. Insert in a BST](https://leetcode.com/articles/insert-into-a-bst/)
* [Article 450. Delete in a BST](https://leetcode.com/articles/delete-node-in-a-bst)

One could try these exercises first, and then combine them together to get a full implementation of BST.

|  |
| --- |
| class MyHashSet {  private Bucket[] bucketArray;  private int keyRange;  /\*\* Initialize your data structure here. \*/  public MyHashSet() {  this.keyRange = 769;  this.bucketArray = new Bucket[this.keyRange];  for (int i = 0; i < this.keyRange; ++i)  this.bucketArray[i] = new Bucket();  }  protected int \_hash(int key) {  return (key % this.keyRange);  }  public void add(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].insert(key);  }  public void remove(int key) {  int bucketIndex = this.\_hash(key);  this.bucketArray[bucketIndex].delete(key);  }  /\*\* Returns true if this set contains the specified element \*/  public boolean contains(int key) {  int bucketIndex = this.\_hash(key);  return this.bucketArray[bucketIndex].exists(key);  }  }  class Bucket {  private BSTree tree;  public Bucket() {  tree = new BSTree();  }  public void insert(Integer key) {  this.tree.root = this.tree.insertIntoBST(this.tree.root, key);  }  public void delete(Integer key) {  this.tree.root = this.tree.deleteNode(this.tree.root, key);  }  public boolean exists(Integer key) {  TreeNode node = this.tree.searchBST(this.tree.root, key);  return (node != null);  }  }  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) {  val = x;  }  }  class BSTree {  TreeNode root = null;  public TreeNode searchBST(TreeNode root, int val) {  if (root == null || val == root.val)  return root;  return val < root.val ? searchBST(root.left, val) : searchBST(root.right, val);  }  public TreeNode insertIntoBST(TreeNode root, int val) {  if (root == null)  return new TreeNode(val);  if (val > root.val)  // insert into the right subtree  root.right = insertIntoBST(root.right, val);  else if (val == root.val)  // skip the insertion  return root;  else  // insert into the left subtree  root.left = insertIntoBST(root.left, val);  return root;  }  /\*  \* One step right and then always left  \*/  public int successor(TreeNode root) {  root = root.right;  while (root.left != null)  root = root.left;  return root.val;  }  /\*  \* One step left and then always right  \*/  public int predecessor(TreeNode root) {  root = root.left;  while (root.right != null)  root = root.right;  return root.val;  }  public TreeNode deleteNode(TreeNode root, int key) {  if (root == null)  return null;  // delete from the right subtree  if (key > root.val)  root.right = deleteNode(root.right, key);  // delete from the left subtree  else if (key < root.val)  root.left = deleteNode(root.left, key);  // delete the current node  else {  // the node is a leaf  if (root.left == null && root.right == null)  root = null;  // the node is not a leaf and has a right child  else if (root.right != null) {  root.val = successor(root);  root.right = deleteNode(root.right, root.val);  }  // the node is not a leaf, has no right child, and has a left child  else {  root.val = predecessor(root);  root.left = deleteNode(root.left, root.val);  }  }  return root;  }  }  /\*\*  \* Your MyHashSet object will be instantiated and called as such:  \* MyHashSet obj = new MyHashSet();  \* obj.add(key);  \* obj.remove(key);  \* boolean param\_3 = obj.contains(key);  \*/ |

**Complexity Analysis**

* Time Complexity: O(log(*N/K)*​) where *N* is the number of all possible values and *K* is the number of predefined buckets, which is 769.
  + Assuming that the values are evenly distributed, we could consider that the average size of bucket is N/K​.
  + When we traverse the BST, we are conducting binary search, as a result, the final time complexity of each operation is O(log(*N/K)*​)
* Space Complexity: O(*K*+*M*) where *K* is the number of predefined buckets, and *M* is the number of unique values that have been inserted into the HashSet.

Notes on Hash Function

In all the above approaches, the range of address is fixed, since the base of modulo operator is fixed.

Sometimes, it might be more desirable to have a ***dynamic space*** that goes with the increase of elements in the HashSet. One could set up a threshold on the *load factor* (*i.e.* ratio between the number of elements and the size of space) of the HashSet, and double the range of address, once the load factor exceeds the threshold.

The increase of address space could potentially ***reduce*** the collisions, therefore improve the overall performance of HashSet. However, one should also take into account the cost of ***rehashing*** and redistributing the existing values.

In another scenario, one could adopt the ***2-choice hashing*** as we mentioned at the beginning, which could help the values to be more ***evenly*** distributed in the address space.

**Design HashMap**

Design a HashMap without using any built-in hash table libraries.

To be specific, your design should include these functions:

* put(key, value) : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.
* get(key): Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.
* remove(key) : Remove the mapping for the value key if this map contains the mapping for the key.

**Example:**

MyHashMap hashMap = new MyHashMap();

hashMap.put(1, 1);

hashMap.put(2, 2);

hashMap.get(1);            // returns 1

hashMap.get(3);            // returns -1 (not found)

hashMap.put(2, 1);          // update the existing value

hashMap.get(2);            // returns 1

hashMap.remove(2);          // remove the mapping for 2

hashMap.get(2);            // returns -1 (not found)

**Note:**

* All keys and values will be in the range of [0, 1000000].
* The number of operations will be in the range of [1, 10000].
* Please do not use the built-in HashMap library.

## Solution Intuition

Hashmap is a common data structure that is implemented in various forms in different programming languages, e.g. dict in Python and HashMap in Java. The most distinguish characteristic about hashmap is that it provides a fast access to a **value** that is associated with a given **key**.

There are two main issues that we should tackle, in order to design an efficient hashmap data structure: 1). hash function design and 2). collision handling.

* **1). hash function design**: the purpose of hash function is to map a key value to an address in the storage space, similarly to the system that we assign a postcode to each mail address. As one can image, for a good hash function, it should map different keys **evenly** across the storage space, so that we don't end up with the case that the majority of the keys are concentrated in a few spaces.
* **2). collision handling**: essentially the hash function reduces the vast key space into a limited address space. As a result, there could be the case where two different keys are mapped to the same address, which is what we call 'collision'. Since the collision is inevitable, it is important that we have a strategy to handle the collision.

Depending on how we deal with each of the above two issues, we could have various implementation of hashmap data structure.

#### **Approach 1: Modulo + Array**

**Intuition**

As one of the most intuitive implementations, we could adopt the modulo operator as the hash function, since the key value is of integer type. In addition, in order to minimize the potential collisions, it is advisable to use a prime number as the base of modulo, e.g. 2069.

We organize the storage space as an **array** where each element is indexed with the output value of the hash function.

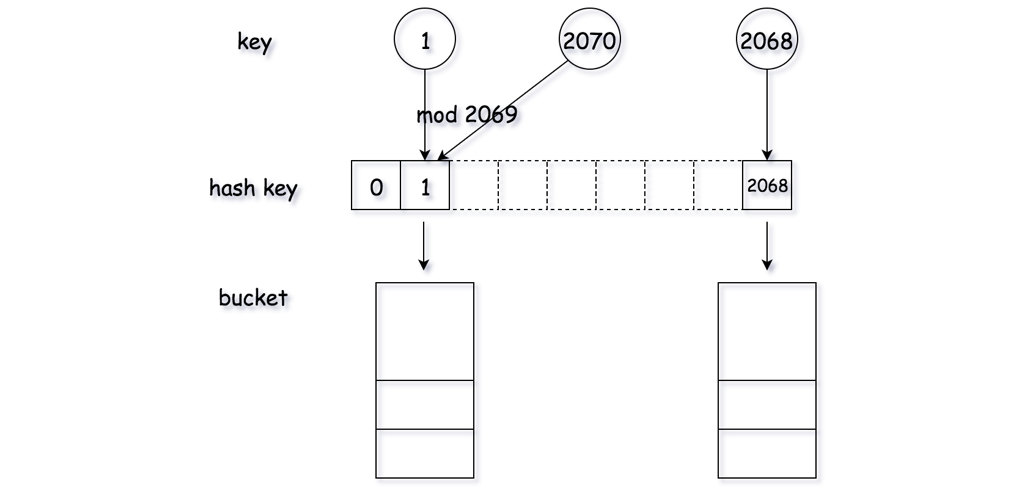
In case of collision, where two different keys are mapped to the same address, we use a **bucket** to hold all the values. The bucket is a container that hold all the values that are assigned by the hash function. We could use either a LinkedList or an Array to implement the bucket data structure.

**Algorithm**

For each of the methods in hashmap data structure, namely get(), put() and remove(), it all boils down to the method to locate the value that is stored in hashmap, given the key.

This localization process can be done in two steps:

* For a given key value, first we apply the hash function to generate a hash key, which corresponds to the address in our main storage. With this hash key, we would find the bucket where the value should be stored.
* Now that we found the bucket, we simply iterate through the bucket to check if the desired <key, value> pair does exist.



|  |
| --- |
| class Pair<U, V> {  public U first;  public V second;  public Pair(U first, V second) {  this.first = first;  this.second = second;  }  }  class Bucket {  private List<Pair<Integer, Integer>> bucket;  public Bucket() {  this.bucket = new LinkedList<Pair<Integer, Integer>>();  }  public Integer get(Integer key) {  for (Pair<Integer, Integer> pair : this.bucket) {  if (pair.first.equals(key))  return pair.second;  }  return -1;  }  public void update(Integer key, Integer value) {  boolean found = false;  for (Pair<Integer, Integer> pair : this.bucket) {  if (pair.first.equals(key)) {  pair.second = value;  found = true;  }  }  if (!found)  this.bucket.add(new Pair<Integer, Integer>(key, value));  }  public void remove(Integer key) {  for (Pair<Integer, Integer> pair : this.bucket) {  if (pair.first.equals(key)) {  this.bucket.remove(pair);  break;  }  }  }  }  class MyHashMap {  private int key\_space;  private List<Bucket> hash\_table;  /\*\* Initialize your data structure here. \*/  public MyHashMap() {  this.key\_space = 2069;  this.hash\_table = new ArrayList<Bucket>();  for (int i = 0; i < this.key\_space; ++i) {  this.hash\_table.add(new Bucket());  }  }  /\*\* value will always be non-negative. \*/  public void put(int key, int value) {  int hash\_key = key % this.key\_space;  this.hash\_table.get(hash\_key).update(key, value);  }  /\*\*  \* Returns the value to which the specified key is mapped, or -1 if this map contains no mapping  \* for the key  \*/  public int get(int key) {  int hash\_key = key % this.key\_space;  return this.hash\_table.get(hash\_key).get(key);  }  /\*\* Removes the mapping of the specified value key if this map contains a mapping for the key \*/  public void remove(int key) {  int hash\_key = key % this.key\_space;  this.hash\_table.get(hash\_key).remove(key);  }  }  /\*\*  \* Your MyHashMap object will be instantiated and called as such: MyHashMap obj = new MyHashMap();  \* obj.put(key,value); int param\_2 = obj.get(key); obj.remove(key);  \*/ |

Note that in the above implementations, we use Array to implement the bucket in Python, while we use LinkedList in Java.

**Complexity Analysis**

* Time Complexity: for each of the methods, the time complexity is O(*N/K*​) where *N* is the number of all possible keys and *K* is the number of predefined buckets in the hashmap, which is 2069 in our case.
  + In the ideal case, the keys are evenly distributed in all buckets. As a result, on average, we could consider the size of the bucket is *N/K*​​.
  + Since in the worst case we need to iterate through a bucket to find the desire value, the time complexity of each method is O(*N/K*​​).
* Space Complexity: O(*K*+*M*) where *K* is the number of predefined buckets in the hashmap and *M* is the number of unique keys that have been inserted into the hashmap.

**Design Hash Table - Solution**

Here are C++ and Java solutions for your reference. In our solution, we use an array to represent the hash set. Each element in the array is a bucket. And in each bucket, we use the array list (or vector in C++) to store all the values.

### ***Hash Set***

|  |
| --- |
| class MyHashSet {  private final int MAX\_LEN = 100000; // the amount of buckets  private List<Integer>[] set; // hash set implemented by array    /\*\* Returns the corresponding bucket index. \*/  private int getIndex(int key) {  return key % MAX\_LEN;  }    /\*\* Search the key in a specific bucket. Returns -1 if the key does not existed. \*/  private int getPos(int key, int index) {  // Each bucket contains a list.  List<Integer> temp = set[index];  if (temp == null) {  return -1;  }  // Iterate all the elements in the bucket to find the target key.  for (int i = 0; i < temp.size(); ++i) {  if (temp.get(i) == key) {  return i;  }  }  return -1;  }    /\*\* Initialize your data structure here. \*/  public MyHashSet() {  set = (List<Integer>[])new ArrayList[MAX\_LEN];  }    public void add(int key) {  int index = getIndex(key);  int pos = getPos(key, index);  if (pos < 0) {  // Add new key if key does not exist.  if (set[index] == null) {  set[index] = new ArrayList<Integer>();  }  set[index].add(key);  }  }    public void remove(int key) {  int index = getIndex(key);  int pos = getPos(key, index);  if (pos >= 0) {  // Remove the key if key exists.  set[index].remove(pos);  }  }    /\*\* Returns true if this set did not already contain the specified element \*/  public boolean contains(int key) {  int index = getIndex(key);  int pos = getPos(key, index);  return pos >= 0;  }  }  /\*\*  \* Your MyHashSet object will be instantiated and called as such:  \* MyHashSet obj = new MyHashSet();  \* obj.add(key);  \* obj.remove(key);  \* boolean param\_3 = obj.contains(key);  \*/ |

### ***Hash Map***

|  |
| --- |
| import javafx.util.Pair;  class MyHashMap {  private final int MAX\_LEN = 100000; // the amount of buckets  private List<Pair<Integer, Integer>>[] map; // hash map implemented by array    /\*\* Returns the corresponding bucket index. \*/  private int getIndex(int key) {  return key % MAX\_LEN;  }    /\*\* Search the key in a specific bucket. Returns -1 if the key does not existed. \*/  private int getPos(int key, int index) {  // Each bucket contains a list.  List<Pair<Integer, Integer>> temp = map[index];  if (temp == null) {  return -1;  }  // Iterate all the elements in the bucket to find the target key.  for (int i = 0; i < temp.size(); ++i) {  if (temp.get(i).getKey() == key) {  return i;  }  }  return -1;  }  /\*\* Initialize your data structure here. \*/  public MyHashMap() {  map = (List<Pair<Integer, Integer>>[])new ArrayList[MAX\_LEN];  }    /\*\* value will always be positive. \*/  public void put(int key, int value) {  int index = getIndex(key);  int pos = getPos(key, index);  if (pos < 0) {  // Add new (key, value) pair if key is not existed.  if (map[index] == null) {  map[index] = new ArrayList<Pair<Integer, Integer>>();  }  map[index].add(new Pair(key, value));  } else {  // Update the value if key is existed.  map[index].set(pos, new Pair(key, value));  }  }    /\*\* Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key \*/  public int get(int key) {  int index = getIndex(key);  int pos = getPos(key, index);  if (pos < 0) {  return -1;  } else {  return map[index].get(pos).getValue();  }  }    /\*\* Removes the mapping of the specified value key if this map contains a mapping for the key \*/  public void remove(int key) {  int index = getIndex(key);  int pos = getPos(key, index);  if (pos >= 0) {  map[index].remove(pos);  }  }  }  /\*\*  \* Your MyHashMap object will be instantiated and called as such:  \* MyHashMap obj = new MyHashMap();  \* obj.put(key,value);  \* int param\_2 = obj.get(key);  \* obj.remove(key);  \*/ |

### ***More***

Let's take a look at the operation "remove". After we find out the position of the element, we need to remove the element from the array list.

Let's assume that we are going to remove the ith element and the size of the array list is n.

The strategy used in the built-in function is to move all the elements after ith element one position forward. That is to say, you have to move n - i times. So the time complexity to remove an element from an array list will be O(n).

Consider different value of i. In average, we will move ((n - 1) + (n - 2) + ... + 1 + 0) / n = (n - 1) / 2 times.

Hopefully, there are two solutions to reduce the time complexity from O(n) to O(1).

**1. Swap**

There is a tricky strategy we can use. First, swap the element which we want to remove with the last element in the bucket. Then remove the last element. By this way, we successfully remove the element in O(1) time complexity.

**2. Linked List**

Another way to achieve this goal is to use a linked list instead of an array list. By this way, we can remove the element in O(1) time complexity without modifying the order in the list.

**Complexity Analysis - Hash Table**

In this article, we are going to discuss the performance of hash table.

### ***Complexity Analysis***

If there are M keys in total, we can achieve the space complexity of O(M) easily when using a hash table.

However, you might have noticed that the time complexity of hash table has a strong relationship with the design.

Most of us might have used an array in each bucket to store values in the same bucket. Ideally, the bucket size is small enough to be regarded as a constant. The time complexity of both insertion and search will be O(1).

But in the worst case, the maximum bucket size will be N. And the time complexity will be O(1) for insertion but O(N) for search.

### ***The Principle of Built-in Hash Table***

The typical design of built-in hash table is:

1. The key value can be any hashable type. And a value which belongs to a hashable type will have a hashcode. This code will be used in the mapping function to get the bucket index.
2. Each bucket contains an array to store all the values in the same bucket initially.
3. If there are too many values in the same bucket, these values will be maintained in a height-balanced binary search tree instead.

The average time complexity of both insertion and search is still O(1). And the time complexity in the worst case is O(logN) for both insertion and search by using height-balanced BST. It is a trade-off between insertion and search.

## Practical Application - Hash Set

In the previous chapter, we talked about how to design a hash table and the great performance of insertion and search in a hash table.

From this chapter on, we will focus on the practical applications.

In this chapter, we are going to talk about how to use the hash set with the help of standard template libraries and when we should use a hash set.

**Hash Set - Usage**

The hash set is one of the implementations of a set which is a data structure to store no repeated values.

We provide an example of using the hash set in Java, C++ and Python. If you are not familiar with the usage of the hash set, it will be helpful to go through the example.

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  // 1. initialize the hash set  Set<Integer> hashSet = new HashSet<>();  // 2. add a new key  hashSet.add(3);  hashSet.add(2);  hashSet.add(1);  // 3. remove the key  hashSet.remove(2);  // 4. check if the key is in the hash set  if (!hashSet.contains(2)) {  System.out.println("Key 2 is not in the hash set.");  }  // 5. get the size of the hash set  System.out.println("The size of has set is: " + hashSet.size());  // 6. iterate the hash set  for (Integer i : hashSet) {  System.out.print(i + " ");  }  System.out.println("are in the hash set.");  // 7. clear the hash set  hashSet.clear();  // 8. check if the hash set is empty  if (hashSet.isEmpty()) {  System.out.println("hash set is empty now!");  }  }  } |

**Find Duplicates By Hash Set**

As we know, it is easy and effective to insert a new value and check if a value is in a hash set or not.

Therefore, typically, a hash set is used to check if a value has ever appeared or not.

***An Example***

Let's look at an example:

Given an array of integers, find if the array contains any duplicates.

This is a typical problem which can be solved by a hash set.

You can simply iterate each value and insert the value into the set. If a value has already been in the hash set, there is a duplicate.

### ***Template***

Here we provide a template for you to solve this kind of problems:

|  |
| --- |
| /\*  \* Template for using hash set to find duplicates.  \*/  boolean findDuplicates(List<Type> keys) {  // Replace Type with actual type of your key  Set<Type> hashset = new HashSet<>();  for (Type key : keys) {  if (hashset.contains(key)) {  return true;  }  hashset.add(key);  }  return false;  } |

**Contains Duplicate**

Given an array of integers, find if the array contains any duplicates.

Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

**Example 1:**

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

**Output:** true

**Example 2:**

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

**Output:** false

**Example 3:**

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

**Output:** true

## Summary

This article is for beginners. It introduces the following ideas: Loop Invariant, Linear Search, Sorting and Hash Table.

## Solution

#### **Approach #1 (Naive Linear Search) [Time Limit Exceeded]**

**Intuition**

For an array of n*n* integers, there are  pairs of integers. Thus, we may check all  pairs and see if there is any pair with duplicates.

**Algorithm**

To apply this idea, we employ the linear search algorithm which is the simplest search algorithm. Linear search is a method of finding if a particular value is in a list by checking each of its elements, one at a time and in sequence until the desired one is found.

For our problem, we loop through all *n* integers. For the *i*th integer nums[i], we search in the previous i-1 integers for the duplicate of nums[i]. If we find one, we return true; if not, we continue. Return false at the end of the program.

To prove the correctness of the algorithm, we define the loop invariant. A loop invariant is a property that holds before (and after) each iteration. Knowing its invariant(s) is essential for understanding the effect of a loop. Here is the loop invariant:

Before the next search, there are no duplicate integers in the searched integers.

The loop invariant holds true before the loop because there is no searched integer. Each time through the loop we look for any any possible duplicate of the current element. If we found a duplicate, the function exits by returning true; If not, the invariant still holds true.

Therefore, if the loop finishes, the invariant tells us that there is no duplicate in all n*n* integers.

**Java**

public boolean containsDuplicate(int[] nums) {

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

for (int j = 0; j < i; ++j) {

if (nums[j] == nums[i]) return true;

}

}

return false;

}

// Time Limit Exceeded

**Complexity Analysis**

* Time complexity : O(n2). In the worst case, there are  pairs of integers to check. Therefore, the time complexity is O(n2).
* Space complexity : *O*(1). We only used constant extra space.

**Note**

This approach will get Time Limit Exceeded on LeetCode. Usually, if an algorithm is O(n2), it can handle *n* up to around 104. It gets Time Limit Exceeded when *n*≥105.

#### **Approach #2 (Sorting) [Accepted]**

**Intuition**

If there are any duplicate integers, they will be consecutive after sorting.

**Algorithm**

This approach employs sorting algorithm. Since comparison sorting algorithm like heapsort is known to provide *O*(*n*log*n*) worst-case performance, sorting is often a good preprocessing step. After sorting, we can sweep the sorted array to find if there are any two consecutive duplicate elements.

**Java**

public boolean containsDuplicate(int[] nums) {

Arrays.sort(nums);

for (int i = 0; i < nums.length - 1; ++i) {

if (nums[i] == nums[i + 1]) return true;

}

return false;

}

**Complexity Analysis**

* Time complexity : *O*(*n*log*n*). Sorting is *O*(*n*log*n*) and the sweeping is *O*(*n*). The entire algorithm is dominated by the sorting step, which is *O*(*n*log*n*).
* Space complexity : *O*(1). Space depends on the sorting implementation which, usually, costs *O*(1) auxiliary space if heapsort is used.

**Note**

The implementation here modifies the original array by sorting it. In general, it is not a good practice to modify the input unless it is clear to the caller that the input will be modified. One may make a copy of nums and operate on the copy instead.

#### **Approach #3 (Hash Table) [Accepted]**

**Intuition**

Utilize a dynamic data structure that supports fast search and insert operations.

**Algorithm**

From [Approach #1](https://leetcode.com/problems/contains-duplicate/solution/#approach-1-naive-linear-search-time-limit-exceeded) we know that search operations is *O*(*n*) in an unsorted array and we did so repeatedly. Utilizing a data structure with faster search time will speed up the entire algorithm.

There are many data structures commonly used as dynamic sets such as Binary Search Tree and Hash Table. The operations we need to support here are search() and insert(). For a self-balancing Binary Search Tree (TreeSet or TreeMap in Java), search() and insert() are both *O*(log*n*) time. For a Hash Table (HashSet or HashMap in Java), search() and insert() are both *O*(1) on average. Therefore, by using hash table, we can achieve linear time complexity for finding the duplicate in an unsorted array.

**Java**

public boolean containsDuplicate(int[] nums) {

Set<Integer> set = new HashSet<>(nums.length);

for (int x: nums) {

if (set.contains(x)) return true;

set.add(x);

}

return false;

}

**Complexity Analysis**

* Time complexity : *O*(*n*). We do search() and insert() for *n* times and each operation takes constant time.
* Space complexity : *O*(*n*). The space used by a hash table is linear with the number of elements in it.

**Note**

For certain test cases with not very large *n*, the runtime of this method can be slower than [Approach #2](https://leetcode.com/problems/contains-duplicate/solution/#approach-2-sorting-accepted). The reason is hash table has some overhead in maintaining its property. One should keep in mind that real world performance can be different from what the Big-O notation says. The Big-O notation only tells us that for sufficiently large input, one will be faster than the other. Therefore, when *n* is not sufficiently large, an *O*(*n*) algorithm can be slower than an *O*(*n*log*n*) algorithm.

## See Also

* [Problem 219 Contains Duplicate II](https://leetcode.com/articles/contains-duplicate-ii/)
* [Problem 220 Contains Duplicate III](https://leetcode.com/articles/contains-duplicate-iii/)

**Single Number**

Given a **non-empty** array of integers nums, every element appears twice except for one. Find that single one.

**Follow up:** Could you implement a solution with a linear runtime complexity and without using extra memory?

**Example 1:**

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

**Output:** 1

**Example 2:**

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

**Output:** 4

**Example 3:**

**Input:** nums = [1]

**Output:** 1

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -3 \* 104 <= nums[i] <= 3 \* 104
* Each element in the array appears twice except for one element which appears only once.

## Solution

#### **Approach 1: List operation**

**Algorithm**

1. Iterate over all the elements in nums
2. If some number in  nums is new to array, append it
3. If some number is already in the array, remove it

|  |
| --- |
| class Solution {  public int singleNumber(int[] nums) {  List<Integer> no\_duplicate\_list = new ArrayList<>();  for (int i : nums) {  if (!no\_duplicate\_list.contains(i)) {  no\_duplicate\_list.add(i);  } else {  no\_duplicate\_list.remove(new Integer(i));  }  }  return no\_duplicate\_list.get(0);  }  } |

**Complexity Analysis**

* Time complexity : O(n2). We iterate through nums, taking *O*(*n*) time. We search the whole list to find whether there is duplicate number, taking *O*(*n*) time. Because search is in the for loop, so we have to multiply both time complexities which is O(n2).
* Space complexity : *O*(*n*). We need a list of size *n* to contain elements in nums.

#### **Approach 2: Hash Table**

**Algorithm**

We use hash table to avoid the *O*(*n*) time required for searching the elements.

1. Iterate through all elements in nums and set up key/value pair.
2. Return the element which appeared only once.

|  |
| --- |
| **class Solution {**  **public int singleNumber(int[] nums) {**  **HashMap<Integer, Integer> hash\_table = new HashMap<>();**  **for (int i : nums) {**  **hash\_table.put(i, hash\_table.getOrDefault(i, 0) + 1);**  **}**  **for (int i : nums) {**  **if (hash\_table.get(i) == 1) {**  **return i;**  **}**  **}**  **return 0;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : *O*(*n*⋅1)=*O*(*n*). Time complexity of for loop is *O*(*n*). Time complexity of hash table(dictionary in python) operation pop is *O*(1).
* Space complexity : *O*(*n*). The space required by *hash*\_*table* is equal to the number of elements in nums.

#### **Approach 3: Math**

**Concept**

2∗(*a*+*b*+*c*)−(*a*+*a*+*b*+*b*+*c*)=*c*

|  |
| --- |
| **class Solution {**  **public int singleNumber(int[] nums) {**  **int sumOfSet = 0, sumOfNums = 0;**  **Set<Integer> set = new HashSet();**  **for (int num : nums) {**  **if (!set.contains(num)) {**  **set.add(num);**  **sumOfSet += num;**  **}**  **sumOfNums += num;**  **}**  **return 2 \* sumOfSet - sumOfNums;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : *O*(*n*+*n*)=*O*(*n*). sum will call next to iterate through nums. We can see it as sum(list(i, for i in nums)) which means the time complexity is *O*(*n*) because of the number of elements(*n*) in nums.
* Space complexity : *O*(*n*+*n*)=*O*(*n*). set needs space for the elements in nums

#### **Approach 4: Bit Manipulation**

**Concept**

* If we take XOR of zero and some bit, it will return that bit
  + *a*⊕0=*a*
* If we take XOR of two same bits, it will return 0
  + *a*⊕*a*=0
* *a*⊕*b*⊕*a* = (*a*⊕*a*)⊕*b* = 0 ⊕*b* = *b*

So we can XOR all bits together to find the unique number.

|  |
| --- |
| **class Solution {**  **public int singleNumber(int[] nums) {**  **int a = 0;**  **for (int i : nums) {**  **a ^= i;**  **}**  **return a;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : O(n) We only iterate through nums, so the time complexity is the number of elements in nums.
* Space complexity : *O*(1).

**Intersection of Two Arrays**

Given two arrays, write a function to compute their intersection.

**Example 1:**

**Input:** nums1 = [1,2,2,1], nums2 = [2,2]

**Output:** [2]

**Example 2:**

**Input:** nums1 = [4,9,5], nums2 = [9,4,9,8,4]

**Output:** [9,4]

**Note:**

* Each element in the result must be unique.
* The result can be in any order.

## Solution

#### **Approach 1: Two Sets**

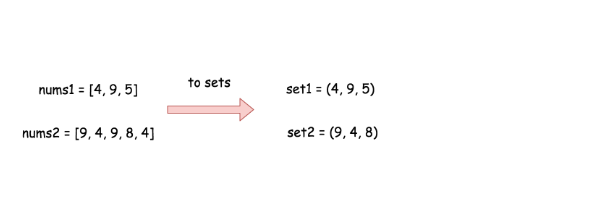
**Intuition**

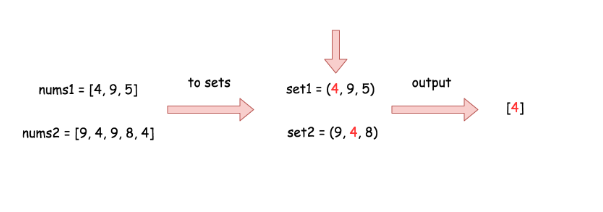
The naive approach would be to iterate along the first array nums1 and to check for each value if this value in nums2 or not. If yes - add the value to output. Such an approach would result in a pretty bad  O(*n*×*m*) time complexity, where n and m are arrays' lengths.

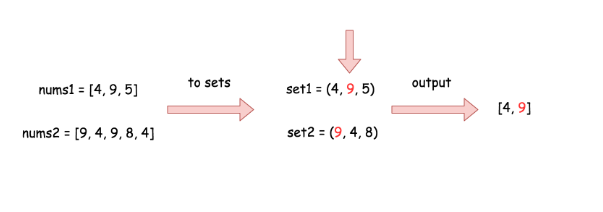
To solve the problem in linear time, let's use the structure set, which provides in/contains operation in  O(1) time in average case.

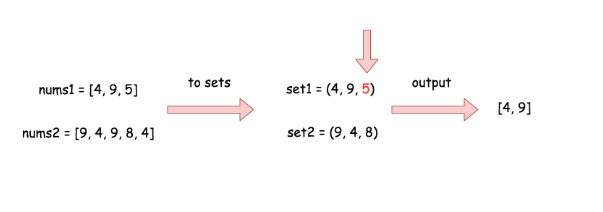
The idea is to convert both arrays into sets, and then iterate over the smallest set checking the presence of each element in the larger set. Time complexity of this approach is  O(*n*+*m*) in the average case.

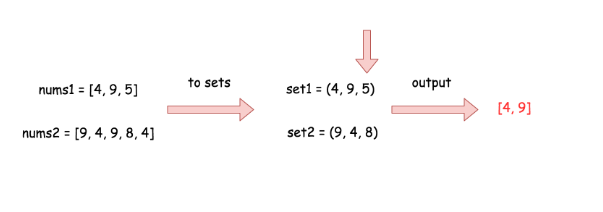












|  |
| --- |
| class Solution {  public int[] set\_intersection(HashSet<Integer> set1, HashSet<Integer> set2) {  int [] output = new int[set1.size()];  int idx = 0;  for (Integer s : set1)  if (set2.contains(s)) output[idx++] = s;  return Arrays.copyOf(output, idx);  }  public int[] intersection(int[] nums1, int[] nums2) {  HashSet<Integer> set1 = new HashSet<Integer>();  for (Integer n : nums1) set1.add(n);  HashSet<Integer> set2 = new HashSet<Integer>();  for (Integer n : nums2) set2.add(n);  if (set1.size() < set2.size()) return set\_intersection(set1, set2);  else return set\_intersection(set2, set1);  }  } |

**Complexity Analysis**

* Time complexity : O(*n*+*m*), where n and m are arrays' lengths. O(*n*) time is used to convert nums1 into set, O(*m*) time is used to convert nums2, and contains/in operations are O(1) in the average case.
* Space complexity : O(*m*+*n*) in the worst case when all elements in the arrays are different.

#### **Approach 2: Built-in Set Intersection**

**Intuition**

There are built-in intersection facilities, which provide  O(*n*+*m*) time complexity in the average case and  O(*n*×*m*) time complexity in the worst case.

In Python it's [intersection operator](https://wiki.python.org/moin/TimeComplexity#set), in Java - [retainAll() function](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractCollection.html#retainAll-java.util.Collection-).

**Implementation**

|  |
| --- |
| class Solution {  public int[] intersection(int[] nums1, int[] nums2) {  HashSet<Integer> set1 = new HashSet<Integer>();  for (Integer n : nums1) set1.add(n);  HashSet<Integer> set2 = new HashSet<Integer>();  for (Integer n : nums2) set2.add(n);  set1.retainAll(set2);  int [] output = new int[set1.size()];  int idx = 0;  for (int s : set1) output[idx++] = s;  return output;  }  } |

**Complexity Analysis**

* Time complexity :  O(*n*+*m*) in the average case and O(*n*×*m*) [in the worst case when load factor is high enough](https://wiki.python.org/moin/TimeComplexity#set).
* Space complexity : O(*n*+*m*) in the worst case when all elements in the arrays are different.

**Happy Number**

Write an algorithm to determine if a number n is happy.

A **happy number** is a number defined by the following process:

* Starting with any positive integer, replace the number by the sum of the squares of its digits.
* Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
* Those numbers for which this process **ends in 1** are happy.

Return true if n is a happy number, and false if not.

**Example 1:**

**Input:** n = 19

**Output:** true

**Explanation:**

12 + 92 = 82

82 + 22 = 68

62 + 82 = 100

12 + 02 + 02 = 1

**Example 2:**

**Input:** n = 2

**Output:** false

**Constraints:**

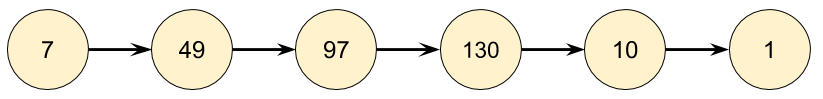
* 1 <= n <= 231 - 1

## Solution

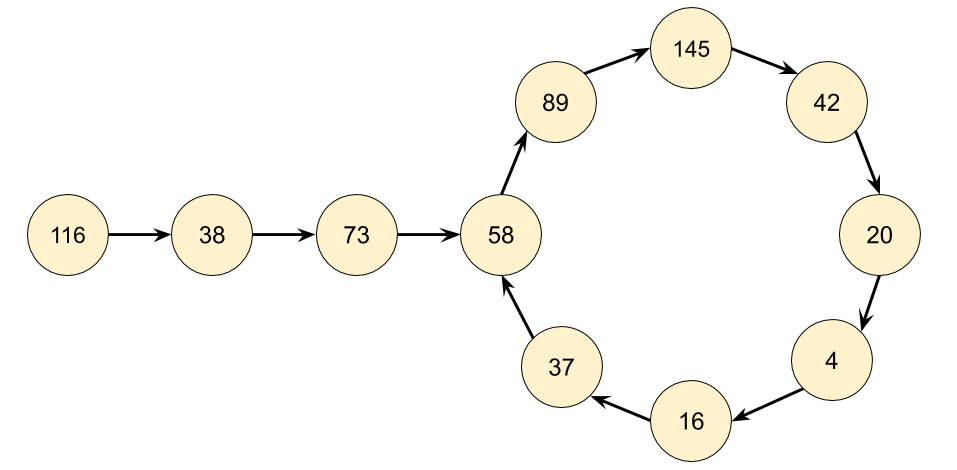
#### **Approach 1: Detect Cycles with a HashSet**

**Intuition**

A good way to get started with a question like this is to make a couple of examples. Let's start with the number 7. The next number will be 49 (as 7^2 = 49), and then the next after that will be 97 (as 4^2 + 9^2 = 97). We can continually repeat the process of squaring and then adding the digits until we get to 1. Because we got to 1, we know that 7 is a happy number, and the function should return true.



As another example, let's start with 116. By repeatedly applying the squaring and adding process, we eventually get to 58, and then a bit after that, we get back to 58. Because we are back at a number we've already seen, we know there is a cycle, and therefore it is impossible to ever reach 1. So for 116, the function should return false.



Based on our exploration so far, we'd expect continually following links to end in one of three ways.

1. It eventually gets to 1.
2. It eventually gets stuck in a cycle.
3. It keeps going higher and higher, up towards infinity.

That 3rd option sounds really annoying to detect and handle. How would we even know that it is going to continue going up, rather than eventually going back down, possibly to 1? Luckily, it turns out we don't need to worry about it. Think carefully about what the largest next number we could get for each number of digits is.

| **Digits** | **Largest** | **Next** |
| --- | --- | --- |
| 1 | 9 | 81 |
| 2 | 99 | 162 |
| 3 | 999 | 243 |
| 4 | 9999 | 324 |
| 13 | 9999999999999 | 1053 |

For a number with 3 digits, it's impossible for it to ever go larger than 243. This means it will have to either get stuck in a cycle below 243 or go down to 1. Numbers with 4 or more digits will always lose a digit at each step until they are down to 3 digits. So we know that at worst, the algorithm might cycle around all the numbers under 243 and then go back to one it's already been to (a cycle) or go to 1. But it won't go on indefinitely, allowing us to rule out the 3rd option.

Even though you don't need to handle the 3rd case in the code, you still need to understand why it can never happen, so that you can justify why you didn't handle it.

**Algorithm**

There are 2 parts to the algorithm we'll need to design and code.

1. Given a number *n*, what is its next number?
2. Follow a chain of numbers and detect if we've entered a cycle.

Part 1 can be done by using the division and modulus operators to repeatedly take digits off the number until none remain, and then squaring each removed digit and adding them together. Have a careful look at the code for this, "picking digits off one-by-one" is a useful technique you'll use for solving a lot of different problems.

Part 2 can be done using a **HashSet**. Each time we generate the next number in the chain, we check if it's already in our HashSet.

* If it is not in the HashSet, we should add it.
* If it is in the HashSet, that means we're in a cycle and so should return false.

The reason we use a **HashSet** and not a Vector, List, or Array is because we're repeatedly checking whether or not numbers are in it. Checking if a number is in a HashSet takes *O*(1) time, whereas for the other data structures it takes *O*(*n*) time. Choosing the correct data structures is an essential part of solving these problems.

|  |
| --- |
| class Solution {  private int getNext(int n) {  int totalSum = 0;  while (n > 0) {  int d = n % 10;  n = n / 10;  totalSum += d \* d;  }  return totalSum;  }  public boolean isHappy(int n) {  Set<Integer> seen = new HashSet<>();  while (n != 1 && !seen.contains(n)) {  seen.add(n);  n = getNext(n);  }  return n == 1;  }  } |

**Complexity Analysis**

Determining the time complexity for this problem is challenging for an "easy" level question. If you're new to these problems, have a go at calculating the time complexity for just the getNext(n) function (don't worry about how many numbers will be in the chain).

* Time complexity :  *O*(243⋅3+log*n*+loglog*n*+logloglog*n*)... = *O*(log*n*).

Finding the **next** value for a given number has a cost of *O*(log*n*) because we are processing each digit in the number, and the number of digits in a number is given by log*n*.

To work out the total time complexity, we'll need to think carefully about how many numbers are in the chain, and how big they are.

We determined above that once a number is below 243, it is impossible for it to go back up above 243. Therefore, based on our very shallow analysis we know for sure that once a number is below 243, it is impossible for it to take more than another 243 steps to terminate. Each of these numbers has at most 3 digits. With a little more analysis, we could replace the 243 with the length of the longest number chain below 243, however because the constant doesn't matter anyway, we won't worry about it.

For an *n* above 243, we need to consider the cost of each number in the chain that is above 243. With a little math, we can show that in the worst case, these costs will be  *O*(log*n*)+*O*(loglog*n*)+*O*(logloglog*n*).... Luckily for us, the *O*(log*n*) is the dominating part, and the others are all tiny in comparison (collectively, they add up to less than log*n*), so we can ignore them.

* Space complexity : *O*(log*n*). Closely related to the time complexity, and is a measure of what numbers we're putting in the HashSet, and how big they are. For a large enough *n*, the most space will be taken by *n* itself.

We can optimize to  *O*(243⋅3)=*O*(1) easily by only saving numbers in the set that are less than 243, as we have already shown that for numbers that are higher, it's impossible to get back to them anyway.

It might seem worrying that we're simply dropping such "large" constants. But this is what we do in Big O notation, which is a measure of how long the function will take, as the size of the input increases.

Think about what would happen if you had a number with 1 million digits in it. The first step of the algorithm would process those million digits, and then the next value would be, at most (pretend all the digits are 9), be 81∗1,000,000=81,000,000. In just one step, we've gone from a million digits, down to just 8. The largest possible 8 digit number we could get is 99,9999,999, which then goes down to 81∗8=648. And then from here, the cost will be the same as if we'd started with a 3 digit number. Starting with 2 million digits (a **massively** larger number than one with a 1 million digits) would only take roughly twice as long, as again, the dominant part is summing the squares of the 2 million digits, and the rest is tiny in comparison.

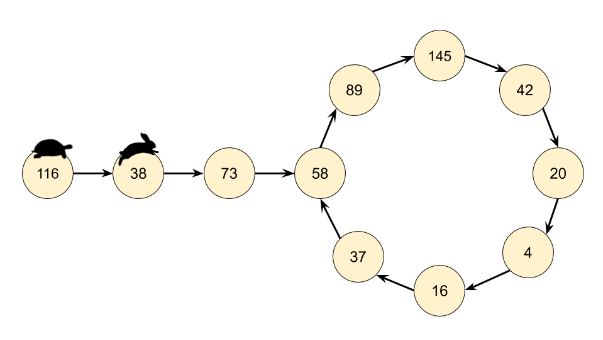
#### **Approach 2: Floyd's Cycle-Finding Algorithm**

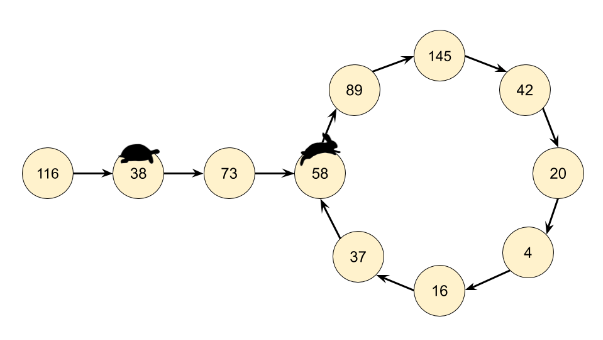
**Intuition**

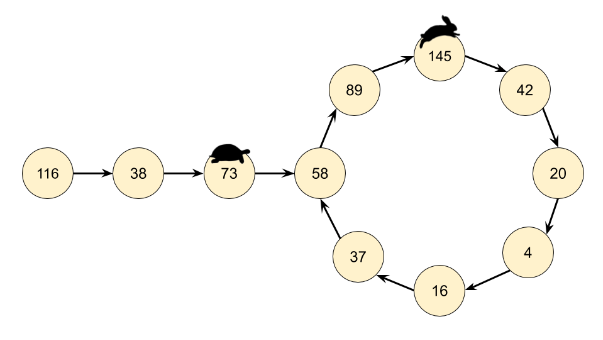
The chain we get by repeatedly calling getNext(n) is an implicit **LinkedList**. Implicit means we don't have actual LinkedNode's and pointers, but the data does still form a LinkedList structure. The starting number is the head "node" of the list, and all the other numbers in the chain are nodes. The next pointer is obtained with our getNext(n) function above.

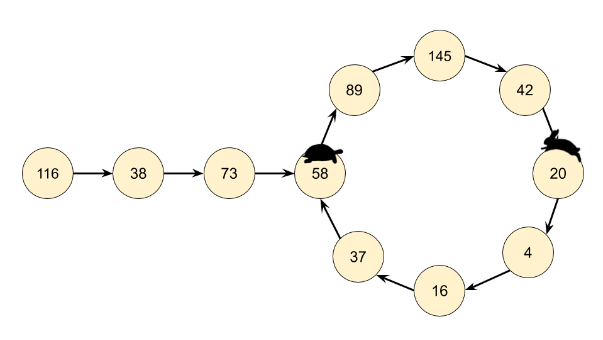
Recognizing that we actually have a LinkedList, it turns out that this question is almost the same as another Leetcode problem, [detecting if a linked list has a cycle](https://leetcode.com/problems/linked-list-cycle). As @Freezen [has pointed out](https://leetcode.com/problems/happy-number/discuss/56917/My-solution-in-C(-O(1)-space-and-no-magic-math-property-involved-)), we can therefore use Floyd's Cycle-Finding Algorithm here. This algorithm is based on 2 runners running around a circular race track, a fast runner and a slow runner. In reference to a famous fable, many people call the slow runner the "tortoise" and the fast runner the "hare".

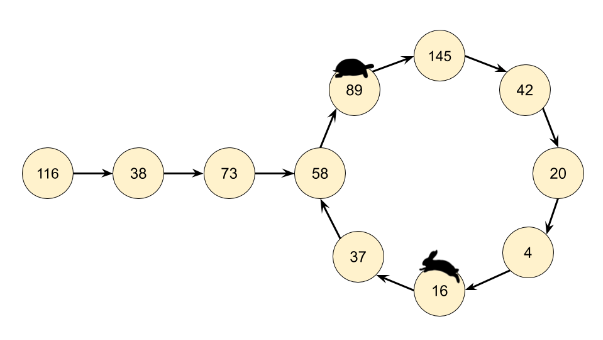
Regardless of where the tortoise and hare start in the cycle, they are guaranteed to eventually meet. This is because the hare moves one node closer to the tortoise (in their direction of movement) each step.

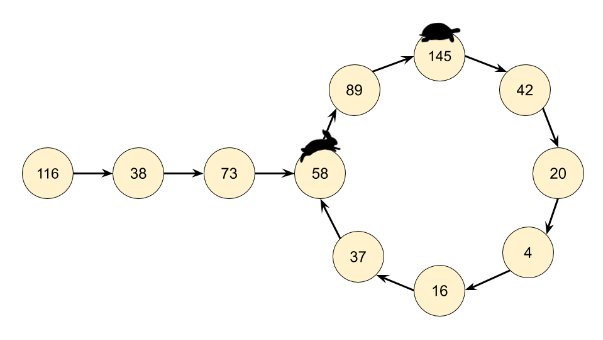


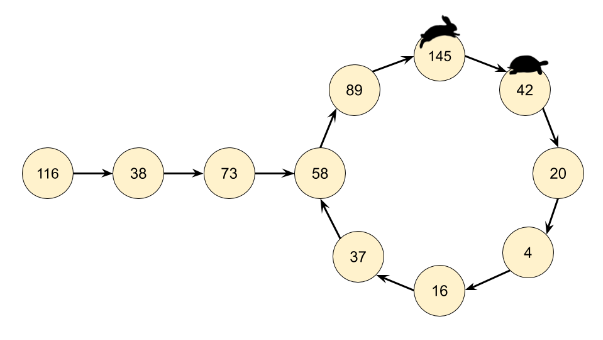


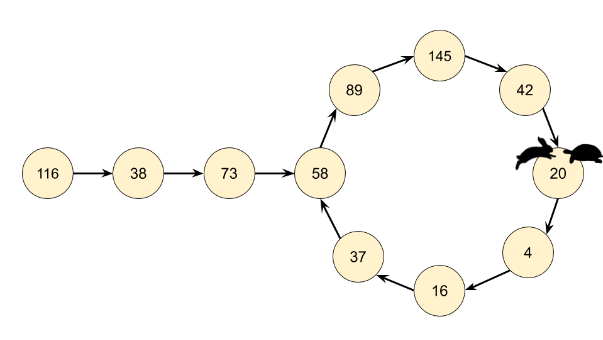












**Algorithm**

Instead of keeping track of just one value in the chain, we keep track of 2, called the slow runner and the fast runner. At each step of the algorithm, the slow runner goes forward by 1 number in the chain, and the fast runner goes forward by 2 numbers (nested calls to the getNext(n) function).

If n *is* a happy number, i.e. there is no cycle, then the fast runner will eventually get to 1 before the slow runner.

If n *is not* a happy number, then eventually the fast runner and the slow runner will be on the same number.

|  |
| --- |
| class Solution {  public int getNext(int n) {  int totalSum = 0;  while (n > 0) {  int d = n % 10;  n = n / 10;  totalSum += d \* d;  }  return totalSum;  }  public boolean isHappy(int n) {  int slowRunner = n;  int fastRunner = getNext(n);  while (fastRunner != 1 && slowRunner != fastRunner) {  slowRunner = getNext(slowRunner);  fastRunner = getNext(getNext(fastRunner));  }  return fastRunner == 1;  }  } |

**Complexity Analysis**

* Time complexity : *O*(log*n*). Builds on the analysis for the previous approach, except this time we need to analyse how much extra work is done by keeping track of two places instead of one, and how many times they'll need to go around the cycle before meeting.

If there is no cycle, then the fast runner will get to 1, and the slow runner will get halfway to 1. Because there were 2 runners instead of 1, we know that at worst, the cost was (2⋅log*n*)=*O*(log*n*).

Like above, we're treating the length of the chain to the cycle as insignificant compared to the cost of calculating the next value for the first n. Therefore, the only thing we need to do is show that the number of times the runners go back over previously seen numbers in the chain is constant.

Once both pointers are in the cycle (which will take constant time to happen) the fast runner will get one step closer to the slow runner at each cycle. Once the fast runner is one step behind the slow runner, they'll meet on the next step. Imagine there are *k* numbers in the cycle. If they started at *k*−1 places apart (which is the furthest apart they can start), then it will take *k*−1 steps for the fast runner to reach the slow runner, which again is constant for our purposes. Therefore, the dominating operation is still calculating the next value for the starting n, which is *O*(log*n*).

* Space complexity : *O*(1). For this approach, we don't need a HashSet to detect the cycles. The pointers require constant extra space.

#### **Approach 3: Hardcoding the Only Cycle (Advanced)**

**Intuition**

The previous two approaches are the ones you'd be expected to come up with in an interview. This third approach is ***not something you'd write in an interview***, but is aimed at the mathematically curious among you as it's quite interesting.

What's the biggest number that could have a next value bigger than itself? Well we know it has to be less than 243, from the analysis we did previously. Therefore, we know that any cycles must contain numbers *smaller* than 243, as anything bigger could not be cycled back to. With such small numbers, it's not difficult to write a brute force program that finds all the cycles.

If you do this, you'll find there's only *one* cycle: 4 →16→37→58→89→145→42→20→4. All other numbers are on chains that lead into this cycle, or on chains that lead into 1.

Therefore, we can just hardcode a HashSet containing these numbers, and if we ever reach one of them, then we know we're in the cycle. There's no need to keep track of where we've been previously.

**Algorithm**

|  |
| --- |
| class Solution {  private static Set<Integer> cycleMembers =  new HashSet<>(Arrays.asList(4, 16, 37, 58, 89, 145, 42, 20));  public int getNext(int n) {  int totalSum = 0;  while (n > 0) {  int d = n % 10;  n = n / 10;  totalSum += d \* d;  }  return totalSum;  }  public boolean isHappy(int n) {  while (n != 1 && !cycleMembers.contains(n)) {  n = getNext(n);  }  return n == 1;  }  } |

**Complexity Analysis**

Time complexity : *O*(log*n*). Same as above.

Space complexity : *O*(1). We are not maintaining any history of numbers we've seen. The hardcoded HashSet is of a constant size.

**An Alternative Implementation**

Thanks [@Manky](https://leetcode.com/manky/) for sharing this alternative with us!

This approach was based on the idea that all numbers either end at 1 or enter the cycle {4, 16, 37, 58, 89, 145, 42, 20}, wrapping around it infinitely.

An alternative approach would be to recognise that all numbers will either end at 1, or go past 4 (a member of the cycle) at some point. Therefore, instead of hardcoding the entire cycle, we can just hardcode the 4.

|  |
| --- |
| class Solution {    public int getNext(int n) {  int totalSum = 0;  while (n > 0) {  int d = n % 10;  n = n / 10;  totalSum += d \* d;  }  return totalSum;  }    public boolean isHappy(int n) {  while (n != 1 && n != 4) {  n = getNext(n);  }  return n == 1;  }  } |

This alternative has the same time and space complexity as approach 3, from a big-oh point of view. The time taken in practice for this alternative will be slower by a *constant* amount though. If the cycle was entered at 16, then the algorithm will traverse the entire cycle before getting back to 4. The space complexity will be *less* by a *constant* amount, because we're now only hardcoding 4 and not the other 7 numbers in the cycle.

**Hash Map - Usage**

The hash map is one of the implementations of a map which is used to store (key, value) pairs.

We provide an example of using the hash map in Java, C++ and Python. If you are not familiar with the usage of the hash map, it will be helpful to go through the example.

|  |
| --- |
| // "static void main" must be defined in a public class.  public class Main {  public static void main(String[] args) {  // 1. initialize a hash map  Map<Integer, Integer> hashmap = new HashMap<>();  // 2. insert a new (key, value) pair  hashmap.putIfAbsent(0, 0);  hashmap.putIfAbsent(2, 3);  // 3. insert a new (key, value) pair or update the value of existed key  hashmap.put(1, 1);  hashmap.put(1, 2);  // 4. get the value of specific key  System.out.println("The value of key 1 is: " + hashmap.get(1));  // 5. delete a key  hashmap.remove(2);  // 6. check if a key is in the hash map  if (!hashmap.containsKey(2)) {  System.out.println("Key 2 is not in the hash map.");  }  // 7. get the size of the hash map  System.out.println("The size of hash map is: " + hashmap.size());  // 8. iterate the hash map  for (Map.Entry<Integer, Integer> entry : hashmap.entrySet()) {  System.out.print("(" + entry.getKey() + "," + entry.getValue() + ") ");  }  System.out.println("are in the hash map.");  // 9. clear the hash map  hashmap.clear();  // 10. check if the hash map is empty  if (hashmap.isEmpty()) {  System.out.println("hash map is empty now!");  }  }  } |

**Scenario I - Provide More Information**

The first scenario to use a hash map is that we need more information rather than only the key. Then we can build a mapping relationship between key and information by hash map.

### ***An Example***

Let's look at an example:

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

In this example, if we only want to return true if there is a solution, we can use a hash set to store all the values when we iterate the array and check if target - current\_value is in the hash set or not.

However, we are asked to return more information which means we not only care about the value but also care about the index. We need to store not only the number as the key but also the index as the value. Therefore, we should use a hash map rather than a hash set.

### ***What's More***

In some cases, we need more information not just to return more information but also to help us with our decisions.

In the previous examples, when we meet a duplicated key, we will return the corresponding information immediately. But sometimes, we might want to check if the value of the key is acceptable first.

### ***Template***

Here we provide a template for you to solve this kind of problems:

|  |
| --- |
| /\*  \* Template for using hash map to find duplicates.  \* Replace ReturnType with the actual type of your return value.  \*/  ReturnType aggregateByKey\_hashmap(List<Type>& keys) {  // Replace Type and InfoType with actual type of your key and value  Map<Type, InfoType> hashmap = new HashMap<>();  for (Type key : keys) {  if (hashmap.containsKey(key)) {  if (hashmap.get(key) satisfies the requirement) {  return needed\_information;  }  }  // Value can be any information you needed (e.g. index)  hashmap.put(key, value);  }  return needed\_information;  } |

**Two Sum**

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to *target*.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Output:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

**Constraints:**

* 2 <= nums.length <= 103
* -109 <= nums[i] <= 109
* -109 <= target <= 109
* **Only one valid answer exists.**

Hint #1

A really brute force way would be to search for all possible pairs of numbers but that would be too slow. Again, it's best to try out brute force solutions for just for completeness. It is from these brute force solutions that you can come up with optimizations.

Hint #2

So, if we fix one of the numbers, say

x

, we have to scan the entire array to find the next number

y

which is

value - x

where value is the input parameter. Can we change our array somehow so that this search becomes faster?

Hint #3

The second train of thought is, without changing the array, can we use additional space somehow? Like maybe a hash map to speed up the search?

## Solution

#### **Approach 1: Brute Force**

The brute force approach is simple. Loop through each element x*x* and find if there is another value that equals to *target*−*x*.

|  |
| --- |
| public int[] twoSum(int[] nums, int target) {  for (int i = 0; i < nums.length; i++) {  for (int j = i + 1; j < nums.length; j++) {  if (nums[j] == target - nums[i]) {  return new int[] { i, j };  }  }  }  throw new IllegalArgumentException("No two sum solution");  } |

**Complexity Analysis**

* Time complexity : O(n^2). For each element, we try to find its complement by looping through the rest of array which takes *O*(*n*) time. Therefore, the time complexity is O(n^2).
* Space complexity : *O*(1).

#### **Approach 2: Two-pass Hash Table**

To improve our run time complexity, we need a more efficient way to check if the complement exists in the array. If the complement exists, we need to look up its index. What is the best way to maintain a mapping of each element in the array to its index? A hash table.

We reduce the look up time from *O*(*n*) to *O*(1) by trading space for speed. A hash table is built exactly for this purpose, it supports fast look up in near constant time. I say "near" because if a collision occurred, a look up could degenerate to *O*(*n*) time. But look up in hash table should be amortized *O*(1) time as long as the hash function was chosen carefully.

A simple implementation uses two iterations. In the first iteration, we add each element's value and its index to the table. Then, in the second iteration we check if each element's complement (*target*−*nums*[*i*]) exists in the table. Beware that the complement must not be *nums*[*i*] itself!

|  |
| --- |
| public int[] twoSum(int[] nums, int target) {  Map<Integer, Integer> map = new HashMap<>();  for (int i = 0; i < nums.length; i++) {  map.put(nums[i], i);  }  for (int i = 0; i < nums.length; i++) {  int complement = target - nums[i];  if (map.containsKey(complement) && map.get(complement) != i) {  return new int[] { i, map.get(complement) };  }  }  throw new IllegalArgumentException("No two sum solution");  } |

**Complexity Analysis:**

* Time complexity : *O*(*n*). We traverse the list containing *n* elements exactly twice. Since the hash table reduces the look up time to *O*(1), the time complexity is *O*(*n*).
* Space complexity : *O*(*n*). The extra space required depends on the number of items stored in the hash table, which stores exactly *n* elements.

#### **Approach 3: One-pass Hash Table**

It turns out we can do it in one-pass. While we iterate and inserting elements into the table, we also look back to check if current element's complement already exists in the table. If it exists, we have found a solution and return immediately.

|  |
| --- |
| public int[] twoSum(int[] nums, int target) {  Map<Integer, Integer> map = new HashMap<>();  for (int i = 0; i < nums.length; i++) {  int complement = target - nums[i];  if (map.containsKey(complement)) {  return new int[] { map.get(complement), i };  }  map.put(nums[i], i);  }  throw new IllegalArgumentException("No two sum solution");  } |

**Complexity Analysis:**

* Time complexity : *O*(*n*). We traverse the list containing *n* elements only once. Each look up in the table costs only *O*(1) time.
* Space complexity : *O*(*n*). The extra space required depends on the number of items stored in the hash table, which stores at most *n* elements.

**Isomorphic Strings**

Given two strings s and t, determine if they are isomorphic.

Two strings s and t are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

**Example 1:**

**Input:** s = "egg", t = "add"

**Output:** true

**Example 2:**

**Input:** s = "foo", t = "bar"

**Output:** false

**Example 3:**

**Input:** s = "paper", t = "title"

**Output:** true

**Constraints:**

* 1 <= s.length <= 5 \* 104
* t.length == s.length
* s and t consist of any valid ascii character.

**Minimum Index Sum of Two Lists**

Suppose Andy and Doris want to choose a restaurant for dinner, and they both have a list of favorite restaurants represented by strings.

You need to help them find out their **common interest** with the **least list index sum**. If there is a choice tie between answers, output all of them with no order requirement. You could assume there always exists an answer.

**Example 1:**

**Input:** list1 = ["Shogun","Tapioca Express","Burger King","KFC"], list2 = ["Piatti","The Grill at Torrey Pines","Hungry Hunter Steakhouse","Shogun"]

**Output:** ["Shogun"]

**Explanation:** The only restaurant they both like is "Shogun".

**Example 2:**

**Input:** list1 = ["Shogun","Tapioca Express","Burger King","KFC"], list2 = ["KFC","Shogun","Burger King"]

**Output:** ["Shogun"]

**Explanation:** The restaurant they both like and have the least index sum is "Shogun" with index sum 1 (0+1).

**Example 3:**

**Input:** list1 = ["Shogun","Tapioca Express","Burger King","KFC"], list2 = ["KFC","Burger King","Tapioca Express","Shogun"]

**Output:** ["KFC","Burger King","Tapioca Express","Shogun"]

**Example 4:**

**Input:** list1 = ["Shogun","Tapioca Express","Burger King","KFC"], list2 = ["KNN","KFC","Burger King","Tapioca Express","Shogun"]

**Output:** ["KFC","Burger King","Tapioca Express","Shogun"]

**Example 5:**

**Input:** list1 = ["KFC"], list2 = ["KFC"]

**Output:** ["KFC"]

**Constraints:**

* 1 <= list1.length, list2.length <= 1000
* 1 <= list1[i].length, list2[i].length <= 30
* list1[i] and list2[i] consist of spaces ' ' and English letters.
* All the stings of list1 are **unique**.
* All the stings of list2 are **unique**.

## Solution

#### **Approach #1 Using HashMap [Accepted]**

In this approach, we compare every string in *list*1 and *list*2 by traversing over the whole list *list*2 for every string chosen from *list*1. We make use of a hashmap *map*, which contains elements of the form (*sum*:*listsum*​). Here, *sum* refers to the sum of indices of matching elements and *listsum*​ refers to the list of matching strings whose indices' sum equals *sum*.

Thus, while doing the comparisons, whenever a match between a string at *ith* index of list1*list*1 and *jth* index of *list*2 is found, we make an entry in the *map* corresponding to the sum *i*+*j*, if this entry isn't already present. If an entry with this sum already exists, we need to keep a track of all the strings which lead to the same index sum. Thus, we append the current string to the list of strings corresponding to sum *i*+*j*.

At the end, we traverse over the keys of the *map* and find out the list of strings corresponding to the key representing the minimum sum.

|  |
| --- |
| public class Solution {  public String[] findRestaurant(String[] list1, String[] list2) {  HashMap < Integer, List < String >> map = new HashMap < > ();  for (int i = 0; i < list1.length; i++) {  for (int j = 0; j < list2.length; j++) {  if (list1[i].equals(list2[j])) {  if (!map.containsKey(i + j))  map.put(i + j, new ArrayList < String > ());  map.get(i + j).add(list1[i]);  }  }  }  int min\_index\_sum = Integer.MAX\_VALUE;  for (int key: map.keySet())  min\_index\_sum = Math.min(min\_index\_sum, key);  String[] res = new String[map.get(min\_index\_sum).size()];  return map.get(min\_index\_sum).toArray(res);  }  } |

**Complexity Analysis**

* Time complexity : *O*(*l*1​∗*l*2​∗*x*). Every item of *list*1 is compared with all the items of *list*2. l\_1*l*1​ and *l*2​ are the lengths of *list*1 and *list*2 respectively. And *x* refers to average string length.
* Space complexity : *O*(*l*1​∗*l*2​∗*x*). In worst case all items of *list*1 and *list*2 are same. In that case, hashmap size grows upto *l*1​∗*l*2​∗*x*, where x*x* refers to average string length.

#### **Approach #2 Without Using HashMap [Accepted]**

**Algorithm**

Another method could be to traverse over the various *sum*(index sum) values and determine if any such string exists in *list*1 and *list*2 such that the sum of its indices in the two lists equals *sum*.

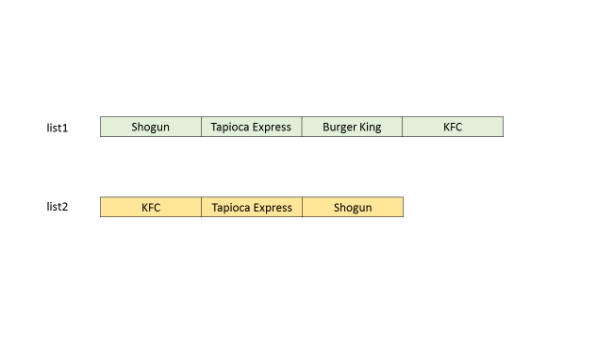
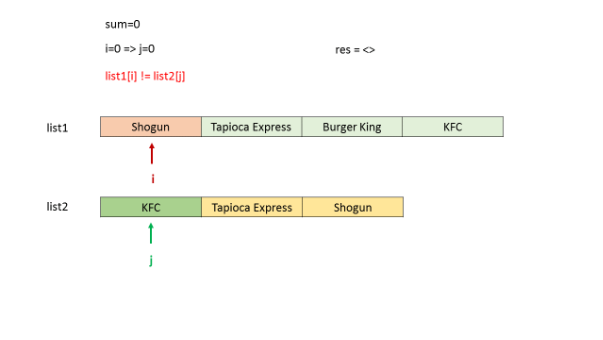
Now, we know that the value of index sum, *sum* could range from 0 to *m*+*n*−1. Here, *m* and *n* refer to the length of lists *list*1 and *list*2 respectively. Thus, we choose every value of *sum* in ascending order. For every *sum* chosen, we iterate over *list*1. Suppose, currently the string at *ith* index in *list*1 is being considered. Now, in order for the index sum *sum* to be the one corresponding to matching strings in list1*list*1 and *list*2, the string at index *j* in *list*2 should match the string at index *i* in *list*1, such that *sum*=*i*+*j*.

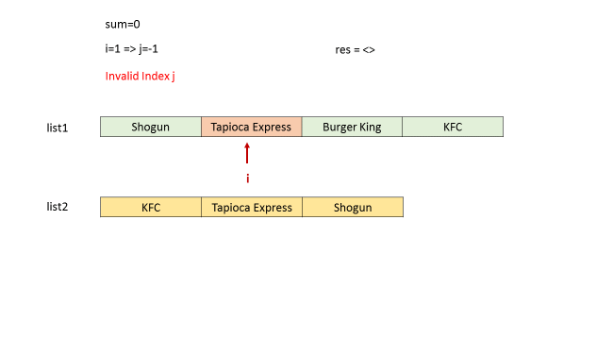
Or, stating in other terms, the string at index *j* in *list*2 should be equal to the string at index *i* in *list*1, such that *j*=*sum*−*i*. Thus, for a particular sum*sum* and *i*(from *list*1), we can directly determine that we need to check the element at index *j*=*sum*−*i* in *list*2, instead of traversing over the whole *list*2.

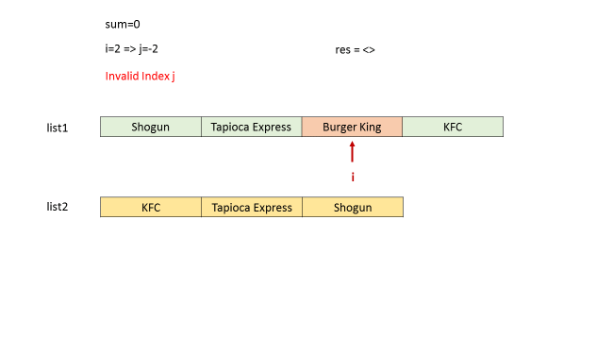
Doing such checks/comparisons, iterate over all the indices of *list*1 for every *sum* value chosen. Whenver a match occurs between *list*1 and *list*2, we put the matching string in a list *res*.

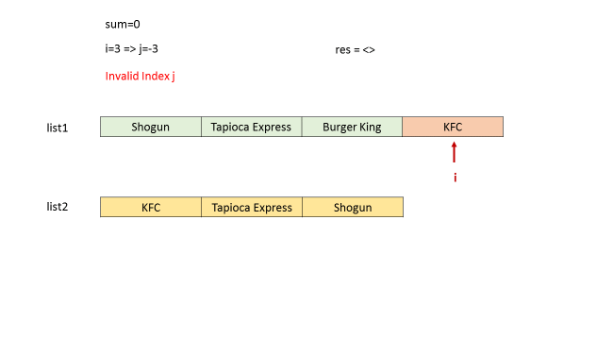
We do the same process of checking the strings for all the values of *sum* in ascending order. After completing every iteration over *list*1 for a particular *sum*, we check if the *res* list is empty or not. If it is empty, we need to continue the process with the next *sum* value considered. If not, the current *res* gives the required list with minimum index sum. This is because we are already considering the index sum values in ascending order. So, the first list to be found is the required resultant list.

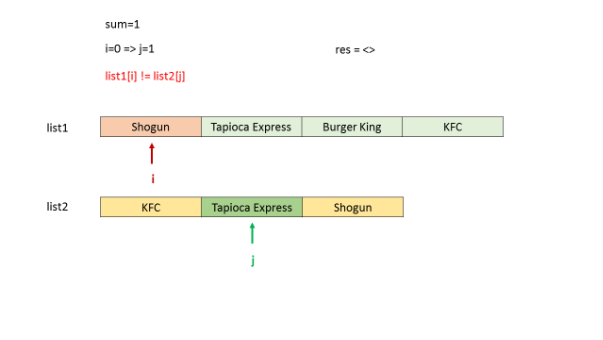
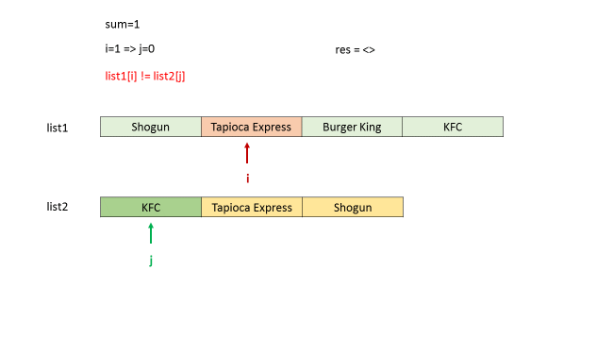
The following example depicts the process:

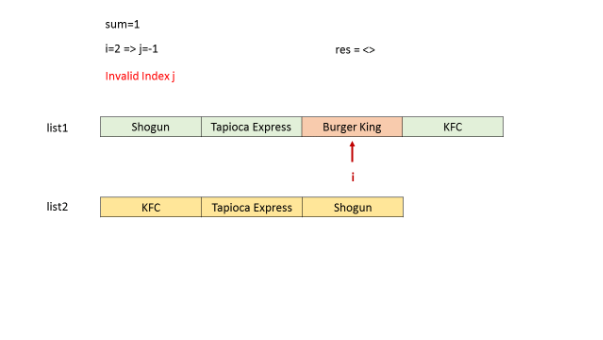
 

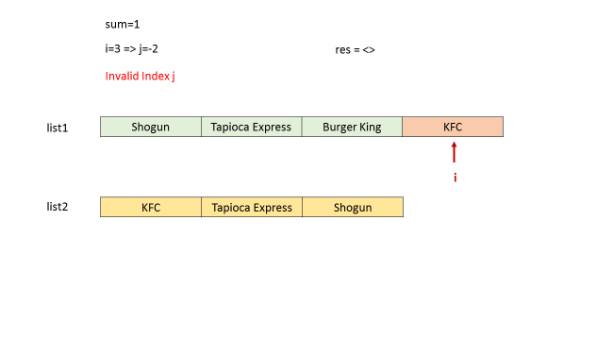


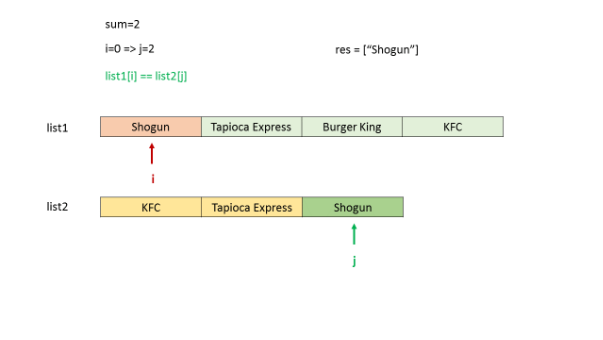


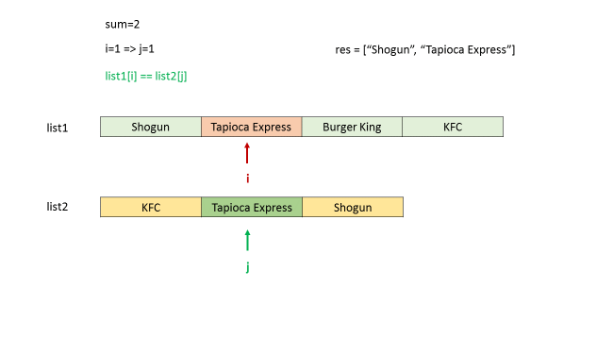


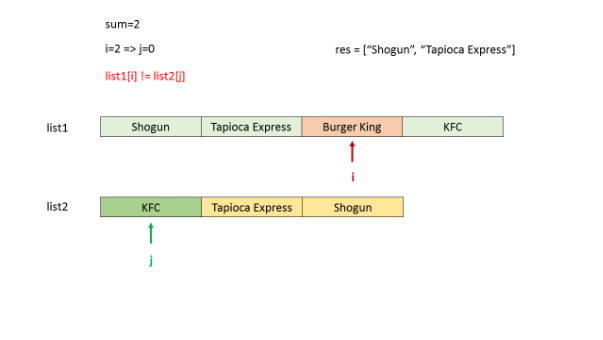
 

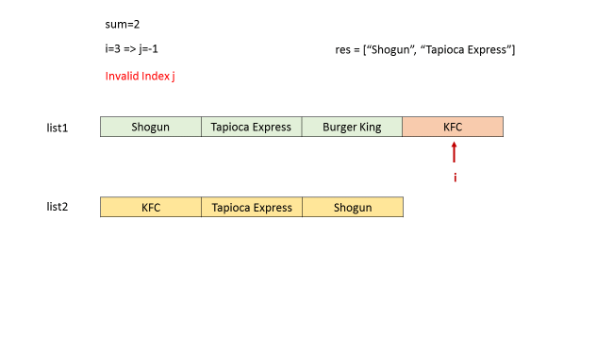














|  |
| --- |
| public class Solution {  public String[] findRestaurant(String[] list1, String[] list2) {  List < String > res = new ArrayList < > ();  for (int sum = 0; sum < list1.length + list2.length - 1; sum++) {  for (int i = 0; i <= sum; i++) {  if (i < list1.length && sum - i < list2.length && list1[i].equals(list2[sum - i]))  res.add(list1[i]);  }  if (res.size() > 0)  break;  }  return res.toArray(new String[res.size()]);  }  } |

**Complexity Analysis**

* Time complexity : *O*((*l*1​+*l*2​)2∗*x*). There are two nested loops upto *l*1​+*l*2​ and string comparison takes *x* time. Here, *x* refers to the average string length.
* Space complexity : *O*(*r*∗*x*). *res* list is used to store the result. Assuming *r* is the length of *res*.

#### **Approach #3 Using HashMap (linear) [Accepted]**

We make use of a HashMap to solve the given problem in a different way in this approach. Firstly, we traverse over the whole *list*1 and create an entry for each element of *list*1 in a HashMap *map*, of the form (*list*[*i*],*i*). Here, i*i* refers to the index of the *ith* element, and *list*[*i*] is the *ith* element itself. Thus, we create a mapping from the elements of *list*1 to their indices.

Now, we traverse over *list*2. For every element ,*list*2[*j*], of *list*2 encountered, we check if the same element already exists as a key in the *map*. If so, it means that the element exists in both *list*1 and *list*2. Thus, we find out the sum of indices corresponding to this element in the two lists, given by *sum*=*map*.*get*(*list*[*j*])+*j*. If this *sum* is lesser than the minimum sum obtained till now, we update the resultant list to be returned, *res*, with the element *list*2[*j*] as the only entry in it.

If the *sum* is equal to the minimum sum obtained till now, we put an extra entry corresponding to the element *list*2[*j*] in the *res* list.

Below code is inspired by [@cloud.runner](http://leetcode.com/cloud.runner)

|  |
| --- |
| public class Solution {  public String[] findRestaurant(String[] list1, String[] list2) {  HashMap < String, Integer > map = new HashMap < String, Integer > ();  for (int i = 0; i < list1.length; i++)  map.put(list1[i], i);  List < String > res = new ArrayList < > ();  int min\_sum = Integer.MAX\_VALUE, sum;  for (int j = 0; j < list2.length && j <= min\_sum; j++) {  if (map.containsKey(list2[j])) {  sum = j + map.get(list2[j]);  if (sum < min\_sum) {  res.clear();  res.add(list2[j]);  min\_sum = sum;  } else if (sum == min\_sum)  res.add(list2[j]);  }  }  return res.toArray(new String[res.size()]);  }  } |

**Complexity Analysis**

* Time complexity : *O*(*l*1​+*l*2​). Every item of *list*2 is checked in a map of *list*1. *l*1​ and *l*2​ are the lengths of *list*1 and *list*2 respectively.
* Space complexity : *O*(*l*1​∗*x*). hashmap size grows upto *l*1​∗*x*, where x*x* refers to average string length.

**Scenario II - Aggregate by Key**

Another frequent scenario is to aggregate all the information by key. We can also use a hash map to achieve this goal.

### ***An Example***

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

A simple way to solve this problem is to count the occurrence of each character first. And then go through the results to find out the first unique character.

Therefore, we can maintain a hashmap whose key is the character while the value is a counter for the corresponding character. Each time when we iterate a character, we just add the corresponding value by 1.

***What's more***

The key to solving this kind of problem is to decide your strategy when you encounter an existing key.

In the example above, our strategy is to count the occurrence. Sometimes, we might sum all the values up. And sometimes, we might replace the original value with the newest one. The strategy depends on the problem and practice will help you make a right decision.

***Template***

Here we provide a template for you to solve this kind of problems:

|  |
| --- |
| /\*  \* Template for using hash map to find duplicates.  \* Replace ReturnType with the actual type of your return value.  \*/  ReturnType aggregateByKey\_hashmap(List<Type>& keys) {  // Replace Type and InfoType with actual type of your key and value  Map<Type, InfoType> hashmap = new HashMap<>();  for (Type key : keys) {  if (hashmap.containsKey(key)) {  hashmap.put(key, updated\_information);  }  // Value can be any information you needed (e.g. index)  hashmap.put(key, value);  }  return needed\_information;  } |

**First Unique Character in a String**

Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1.

**Examples:**

s = "leetcode"

return 0.

s = "loveleetcode"

return 2.

**Note:** You may assume the string contains only lowercase English letters.

## Solution

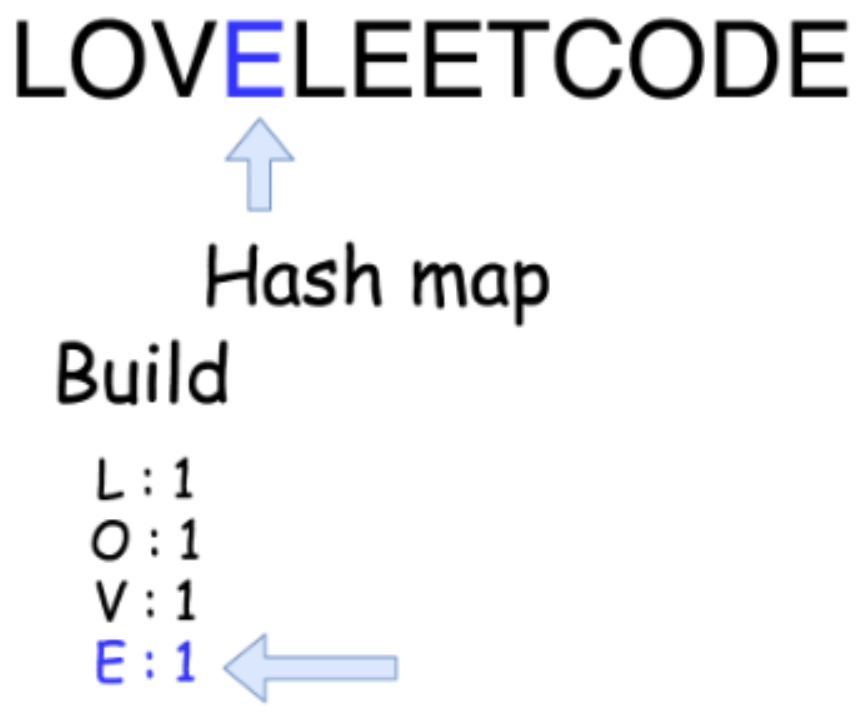
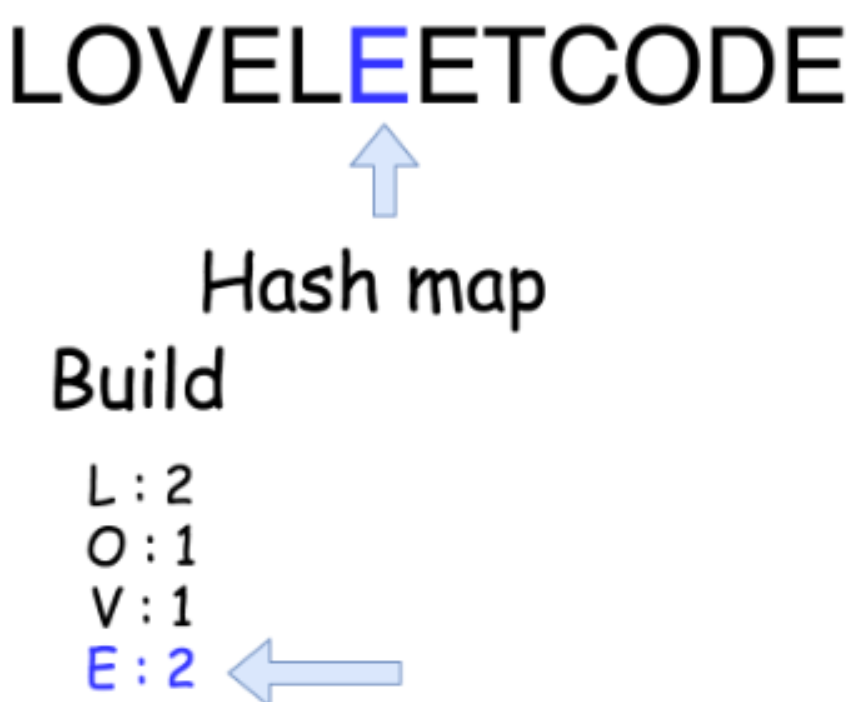
#### **Approach 1: Linear time solution**

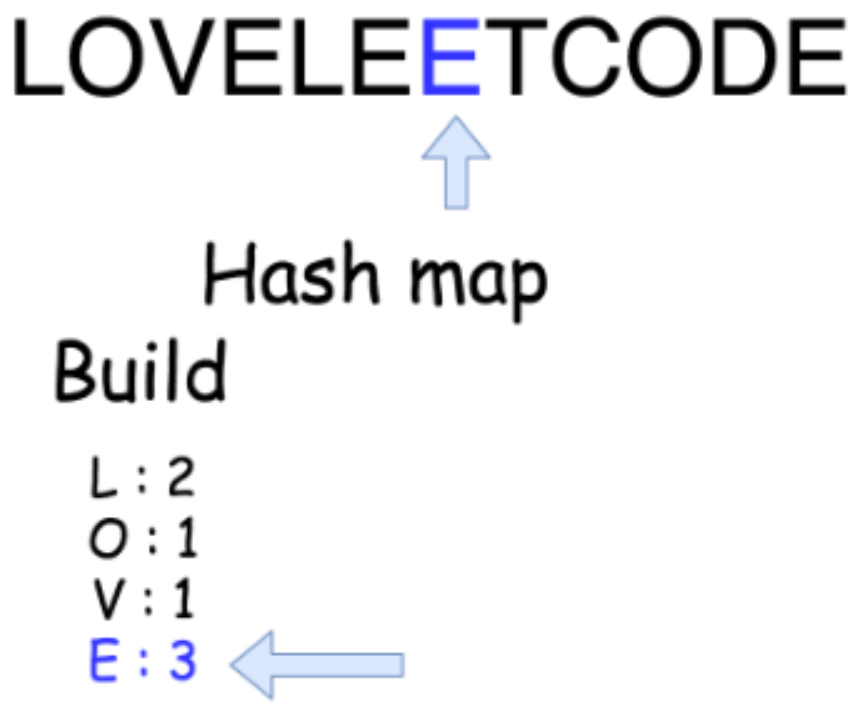
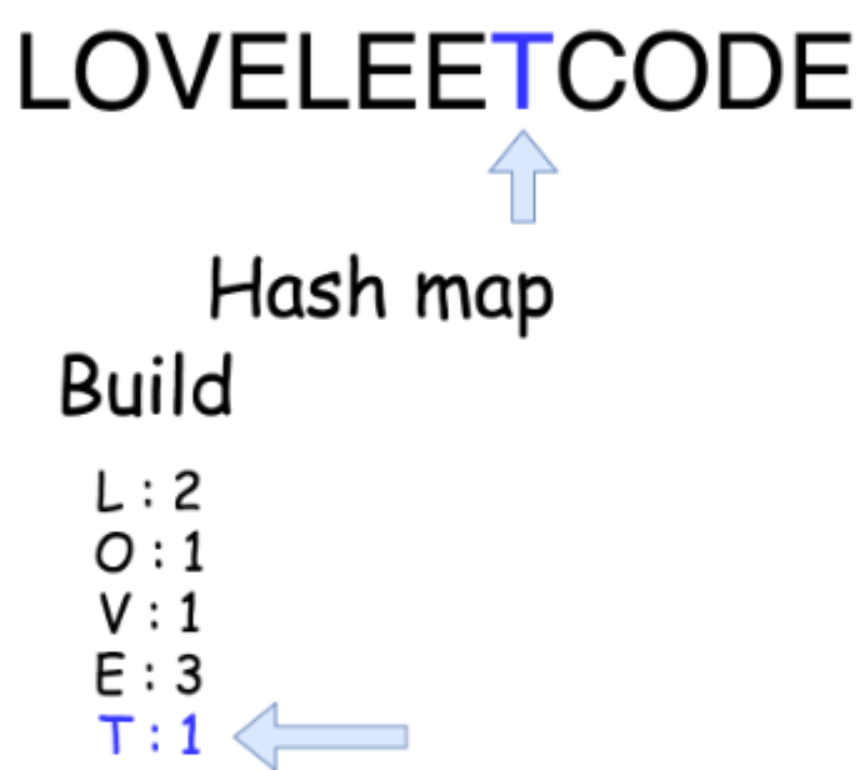
The best possible solution here could be of a linear time because to ensure that the character is unique you have to check the whole string anyway.

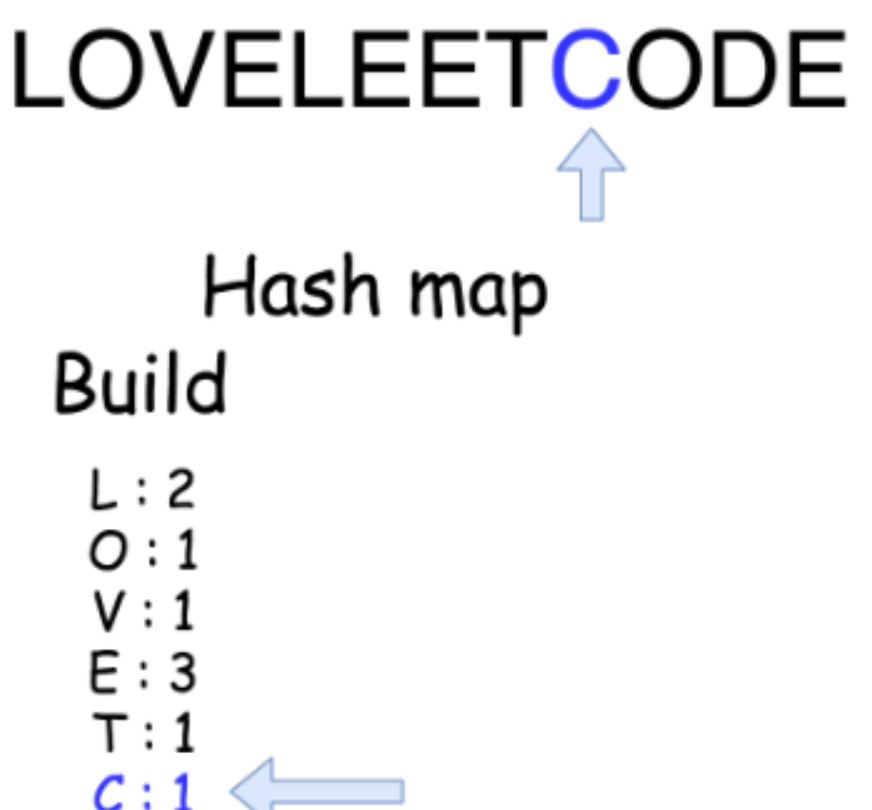
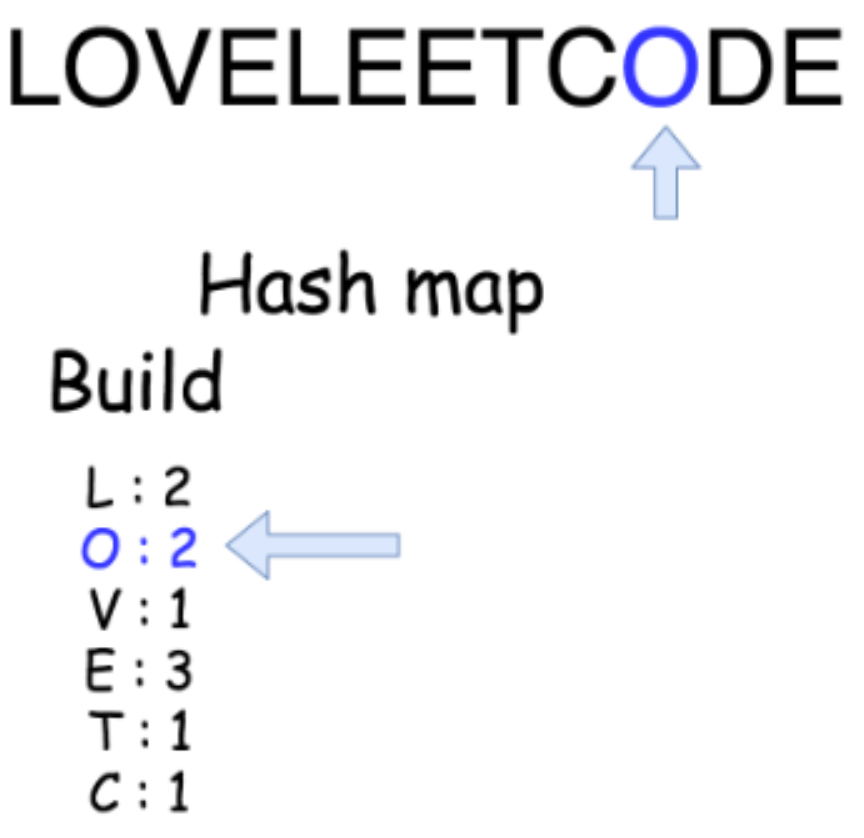
The idea is to go through the string and save in a hash map the number of times each character appears in the string. That would take O(*N*) time, where N is a number of characters in the string.

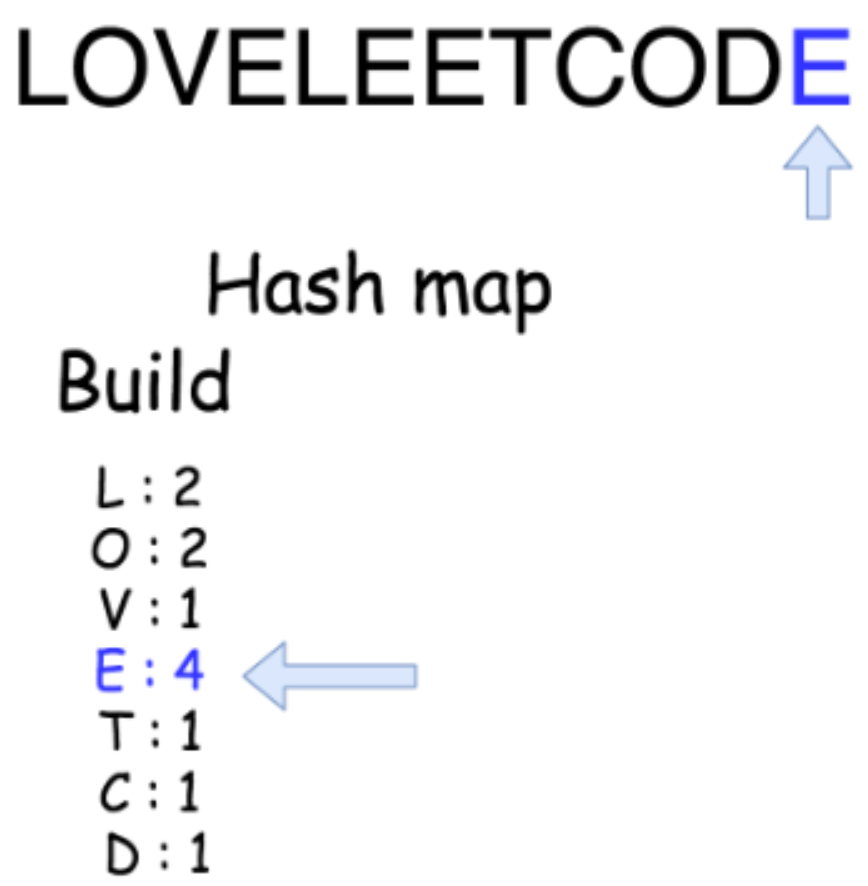
And then we go through the string the second time, this time we use the hash map as a reference to check if a character is unique or not.  
If the character is unique, one could just return its index. The complexity of the second iteration is O(*N*) as well.

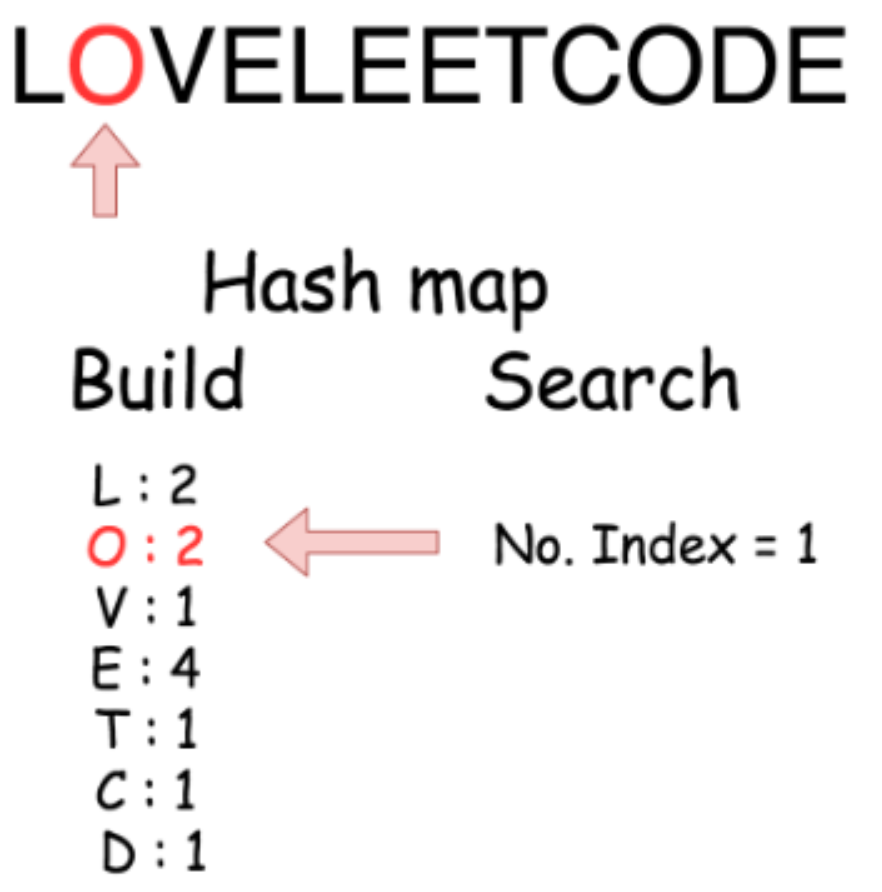


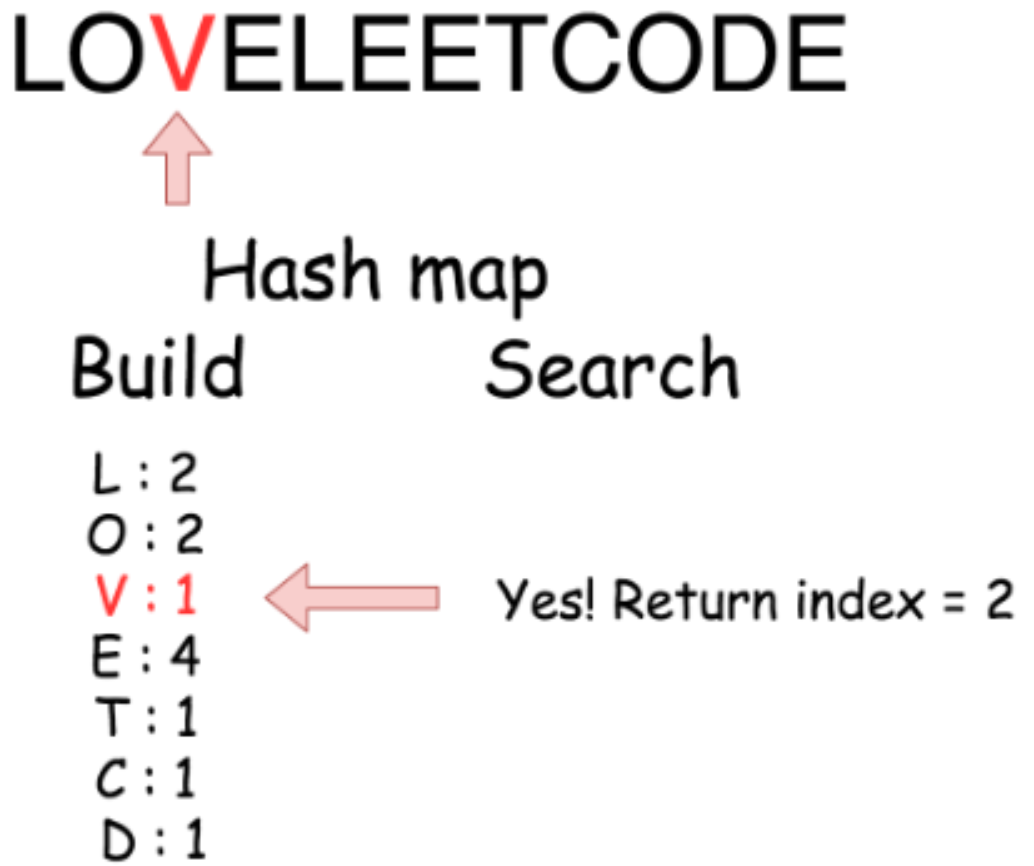
  
 



|  |
| --- |
| class Solution {  public int firstUniqChar(String s) {  HashMap<Character, Integer> count = new HashMap<Character, Integer>();  int n = s.length();  // build hash map : character and how often it appears  for (int i = 0; i < n; i++) {  char c = s.charAt(i);  count.put(c, count.getOrDefault(c, 0) + 1);  }    // find the index  for (int i = 0; i < n; i++) {  if (count.get(s.charAt(i)) == 1)  return i;  }  return -1;  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) since we go through the string of length N two times.
* Space complexity : O(1) because English alphabet contains 26 letters.

**Intersection of Two Arrays II**

Given two arrays, write a function to compute their intersection.

**Example 1:**

**Input:** nums1 = [1,2,2,1], nums2 = [2,2]

**Output:** [2,2]

**Example 2:**

**Input:** nums1 = [4,9,5], nums2 = [9,4,9,8,4]

**Output:** [4,9]

**Note:**

* Each element in the result should appear as many times as it shows in both arrays.
* The result can be in any order.

**Follow up:**

* What if the given array is already sorted? How would you optimize your algorithm?
* What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better?
* What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

## Solution

If an interviewer gives you this problem, your first question should be - how should I handle duplicates? Your second question, perhaps, can be about the order of inputs and outputs. Such questions manifest your problem-solving skills, and help you steer to the right solution.

The [solution](https://leetcode.com/problems/intersection-of-two-arrays/solution/) for the previous problem, [349. Intersection of Two Arrays](https://leetcode.com/problems/intersection-of-two-arrays/), talks about approaches when each number in the output must be unique. For this problem, we need to adapt those approaches so that numbers in the result appear as many times as they do in both arrays.

#### **Approach 1: Hash Map**

For the previous problem, we used a hash set to achieve a linear time complexity. Here, we need to use a hash map to track the count for each number.

We collect numbers and their counts from one of the arrays into a hash map. Then, we iterate along the second array, and check if the number exists in the hash map and its count is positive. If so - add the number to the result and decrease its count in the hash map.



It's a good idea to check array sizes and use a hash map for the smaller array. It will reduce memory usage when one of the arrays is very large.

**Algorithm**

1. If nums1 is larger than nums2, swap the arrays.
2. For each element in nums1:
   * Add it to the hash map m.
     + Increment the count if the element is already there.
3. Initialize the insertion pointer (k) with zero.
4. Iterate along nums2:
   * If the current number is in the hash map and count is positive:
     + Copy the number into nums1[k], and increment k.
     + Decrement the count in the hash map.
5. Return first k elements of nums1.

For our solutions here, we use one of the arrays to store the result. As we find common numbers, we copy them to the first array starting from the beginning. This idea is from [this solution](https://leetcode.com/problems/intersection-of-two-arrays-ii/discuss/82405/Simple-Java-Solution) by [sankitgupta](https://leetcode.com/sankitgupta/).

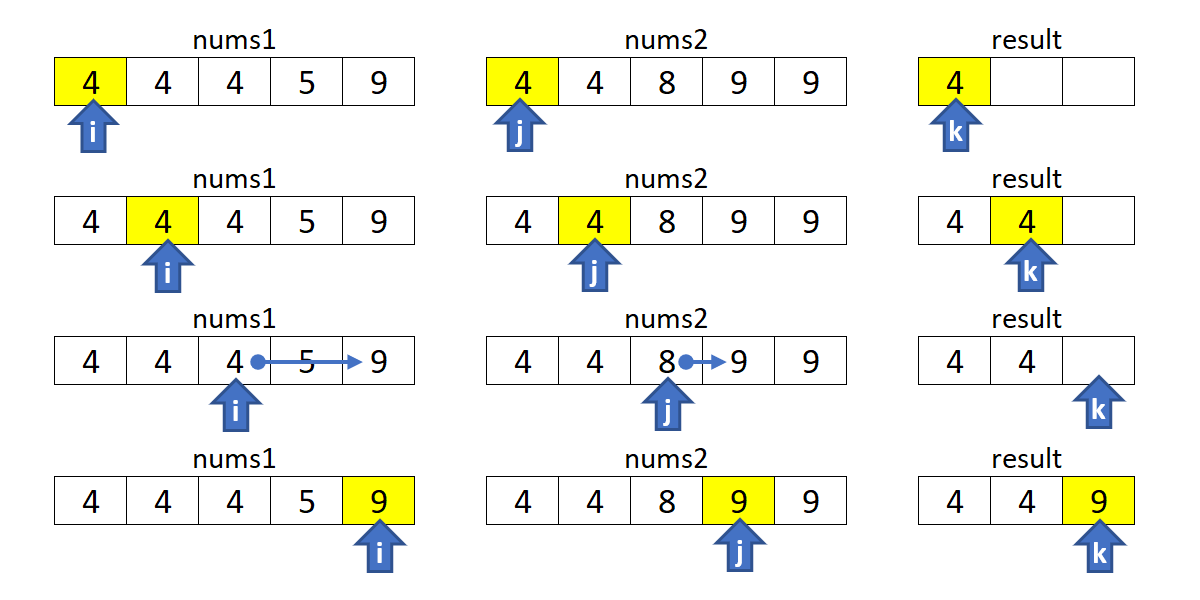
|  |
| --- |
| public int[] intersect(int[] nums1, int[] nums2) {  if (nums1.length > nums2.length) {  return intersect(nums2, nums1);  }  HashMap<Integer, Integer> m = new HashMap<>();  for (int n : nums1) {  m.put(n, m.getOrDefault(n, 0) + 1);  }  int k = 0;  for (int n : nums2) {  int cnt = m.getOrDefault(n, 0);  if (cnt > 0) {  nums1[k++] = n;  m.put(n, cnt - 1);  }  }  return Arrays.copyOfRange(nums1, 0, k);  } |

**Complexity Analysis**

* Time Complexity: O(*n*+*m*), where *n* and *m* are the lengths of the arrays. We iterate through the first, and then through the second array; insert and lookup operations in the hash map take a constant time.
* Space Complexity:  O(min(*n*,*m*)). We use hash map to store numbers (and their counts) from the smaller array.

#### **Approach 2: Sort**

You can recommend this method when the input is sorted, or when the output needs to be sorted. Here, we sort both arrays (assuming they are not sorted) and use two pointers to find common numbers in a single scan.



**Algorithm**

1. Sort nums1 and nums2.
2. Initialize i, j and k with zero.
3. Move indices i along nums1, and j through nums2:
   * Increment i if nums1[i] is smaller.
   * Increment j if nums2[j] is smaller.
   * If numbers are the same, copy the number into nums1[k], and increment i, j and k.
4. Return first k elements of nums1.

|  |
| --- |
| 1. public int[] intersect(int[] nums1, int[] nums2) { 2. Arrays.sort(nums1); 3. Arrays.sort(nums2); 4. int i = 0, j = 0, k = 0; 5. while (i < nums1.length && j < nums2.length) { 6. if (nums1[i] < nums2[j]) { 7. ++i; 8. } else if (nums1[i] > nums2[j]) { 9. ++j; 10. } else { 11. nums1[k++] = nums1[i++]; 12. ++j; 13. } 14. } 15. return Arrays.copyOfRange(nums1, 0, k); 16. } |

**Complexity Analysis**

* Time Complexity:  O(*n*log*n*+*m*log*m*), where *n* and *m* are the lengths of the arrays. We sort two arrays independently, and then do a linear scan.
* Space Complexity: from O(log*n*+log*m*) to O(*n*+*m*), depending on the implementation of the sorting algorithm. For the complexity analysis purposes, we ignore the memory required by inputs and outputs.

#### **Approach 3: Built-in Intersection**

This is similar to [Approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-2-sort). Instead of iterating with two pointers, we use a built-in function to find common elements. In C++, we can use set\_intersection for sorted arrays (or multisets).

The [retainAll](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractCollection.html#retainAll-java.util.Collection-) method in Java, unfortunately, does not care how many times an element occurs in the other collection. You can use the [retainOccurrences](https://guava.dev/releases/23.0/api/docs/com/google/common/collect/Multisets.html#retainOccurrences-com.google.common.collect.Multiset-com.google.common.collect.Multiset-) method of the multiset implementation in [Guava](https://guava.dev/releases/16.0/api/docs/com/google/common/collect/Multiset.html).

**Algorithm**

Note that set\_intersection returns the position past the end of the produced range, so it can be used as an input for the erase function. The idea is from [this solution](https://leetcode.com/problems/intersection-of-two-arrays-ii/discuss/82269/Short-Python-C%2B%2B) by [StefanPochmann](https://leetcode.com/stefanpochmann/).

|  |
| --- |
| vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {  sort(begin(nums1), end(nums1));  sort(begin(nums2), end(nums2));  nums1.erase(set\_intersection(begin(nums1), end(nums1),  begin(nums2), end(nums2), begin(nums1)), end(nums1));  return nums1;  } |

**Complexity Analysis**

* Same as for [approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach2complexity) above.

#### **Follow-up Questions**

1. What if the given array is already sorted? How would you optimize your algorithm?
   * We can use either [Approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-2-sort) or [Approach 3](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-3-built-in-intersection), dropping the sort of course. It will give us linear time and constant memory complexity.
2. What if nums1's size is small compared to nums2's size? Which algorithm is better?
   * [Approach 1](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-1-hash-map) is a good choice here as we use a hash map for the smaller array.
3. What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?
   * If nums1 fits into the memory, we can use [Approach 1](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-1-hash-map) to collect counts for nums1 into a hash map. Then, we can sequentially load and process nums2.
   * If neither of the arrays fit into the memory, we can apply some partial processing strategies:
     + Split the numeric range into subranges that fits into the memory. Modify [Approach 1](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-1-hash-map) to collect counts only within a given subrange, and call the method multiple times (for each subrange).
     + Use an external sort for both arrays. Modify [Approach 2](https://leetcode.com/problems/intersection-of-two-arrays-ii/solution/#approach-2-sort) to load and process arrays sequentially.

**Contains Duplicate II**

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the **absolute** difference between *i* and *j* is at most *k*.

**Example 1:**

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

**Output:** true

**Example 2:**

**Input:** nums = [1,0,1,1], k = 1

**Output:** true

**Example 3:**

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

**Output:** false

## Summary

This article is for beginners. It introduces the following ideas: Linear Search, Binary Search Tree and Hash Table.

## Solution

#### **Approach #1: Naive Linear Search**

**Intuition**

Look for duplicate element in the previous *k* elements.

**Algorithm**

This algorithm is the same as [Approach #1 in Contains Duplicate solution](https://leetcode.com/articles/contains-duplicate/#approach-1-naive-linear-search-time-limit-exceeded), except that it looks at previous *k* elements instead of all its previous elements.

Another perspective of this algorithm is to keep a virtual sliding window of the previous *k* elements. We scan for the duplicate in this window.

|  |
| --- |
| public boolean containsNearbyDuplicate(int[] nums, int k) {  for (int i = 0; i < nums.length; ++i) {  for (int j = Math.max(i - k, 0); j < i; ++j) {  if (nums[i] == nums[j]) return true;  }  }  return false;  }  // Time Limit Exceeded. |

**Complexity Analysis**

* Time complexity : *O*(*n*min(*k*,*n*)). It costs *O*(min(*k*,*n*)) time for each linear search. Apparently we do at most *n* comparisons in one search even if *k* can be larger than *n*.
* Space complexity : *O*(1).

#### **Approach #2: Binary Search Tree**

**Intuition**

Keep a sliding window of *k* elements using self-balancing Binary Search Tree (BST).

**Algorithm**

The key to improve upon [Approach #1](https://leetcode.com/problems/contains-duplicate-ii/solution/#approach-1-naive-linear-search-time-limit-exceeded) above is to reduce the search time of the previous *k* elements. Can we use an auxiliary data structure to maintain a sliding window of *k* elements with more efficient search, delete, and insert operations? Since elements in the sliding window are strictly First-In-First-Out (FIFO), queue is a natural data structure. A queue using a linked list implementation supports constant time delete and insert operations, however the search costs linear time, which is no better than [Approach #1](https://leetcode.com/problems/contains-duplicate-ii/solution/#approach-1-naive-linear-search-time-limit-exceeded).

A better option is to use a self-balancing BST. A BST supports search, delete and insert operations all in *O*(log*k*) time, where k*k* is the number of elements in the BST. In most interviews you are not required to implement a self-balancing BST, so you may think of it as a black box. Most programming languages provide implementations of this useful data structure in its standard library. In Java, you may use a TreeSet or a TreeMap. In C++ STL, you may use a std::set or a std::map.

If you already have such a data structure available, the pseudocode is:

* Loop through the array, for each element do
  + Search current element in the BST, return true if found
  + Put current element in the BST
  + If the size of the BST is larger than *k*, remove the oldest item.
* Return false

|  |
| --- |
| public boolean containsNearbyDuplicate(int[] nums, int k) {  Set<Integer> set = new TreeSet<>();  for (int i = 0; i < nums.length; ++i) {  if (set.contains(nums[i])) return true;  set.add(nums[i]);  if (set.size() > k) {  set.remove(nums[i - k]);  }  }  return false;  }  // Time Limit Exceeded. |

**Complexity Analysis**

* Time complexity :  *O*(*n*log(min(*k*,*n*))). We do *n* operations of search, delete and insert. Each operation costs logarithmic time complexity in the sliding window which size is min(*k*,*n*). Note that even if *k* can be greater than *n*, the window size can never exceed *n*.
* Space complexity : *O*(min(*n*,*k*)). Space is the size of the sliding window which should not exceed *n* or *k*.

**Note**

The algorithm still gets Time Limit Exceeded for large *n* and *k*.

#### **Approach #3: Hash Table**

**Intuition**

Keep a sliding window of *k* elements using Hash Table.

**Algorithm**

From the previous approaches, we know that even logarithmic performance in search is not enough. In this case, we need a data structure supporting constant time search, delete and insert operations. Hash Table is the answer. The algorithm and implementation are almost identical to [Approach #2](https://leetcode.com/problems/contains-duplicate-ii/solution/#approach-2-binary-search-tree-time-limit-exceeded).

* Loop through the array, for each element do
  + Search current element in the HashTable, return true if found
  + Put current element in the HashTable
  + If the size of the HashTable is larger than *k*, remove the oldest item.
* Return false

|  |
| --- |
| public boolean containsNearbyDuplicate(int[] nums, int k) {  Set<Integer> set = new HashSet<>();  for (int i = 0; i < nums.length; ++i) {  if (set.contains(nums[i])) return true;  set.add(nums[i]);  if (set.size() > k) {  set.remove(nums[i - k]);  }  }  return false;  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). We do *n* operations of search, delete and insert, each with constant time complexity.
* Space complexity : *O*(min(*n*,*k*)). The extra space required depends on the number of items stored in the hash table, which is the size of the sliding window, min(*n*,*k*).

## See Also

* [Problem 217 Contains Duplicate](https://leetcode.com/articles/contains-duplicate/)
* [Problem 220 Contains Duplicate III](https://leetcode.com/articles/contains-duplicate-iii/)

**Logger Rate Limiter**

Design a logger system that receives a stream of messages along with their timestamps. Each **unique** message should only be printed **at most every 10 seconds** (i.e. a message printed at timestamp t will prevent other identical messages from being printed until timestamp t + 10).

All messages will come in chronological order. Several messages may arrive at the same timestamp.

Implement the Logger class:

* Logger() Initializes the logger object.
* bool shouldPrintMessage(int timestamp, string message) Returns true if the message should be printed in the given timestamp, otherwise returns false.

**Example 1:**

**Input**

["Logger", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage"]

[[], [1, "foo"], [2, "bar"], [3, "foo"], [8, "bar"], [10, "foo"], [11, "foo"]]

**Output**

[null, true, true, false, false, false, true]

**Explanation**

Logger logger = new Logger();

logger.shouldPrintMessage(1, "foo"); // return true, next allowed timestamp for "foo" is 1 + 10 = 11

logger.shouldPrintMessage(2, "bar"); // return true, next allowed timestamp for "bar" is 2 + 10 = 12

logger.shouldPrintMessage(3, "foo"); // 3 < 11, return false

logger.shouldPrintMessage(8, "bar"); // 8 < 12, return false

logger.shouldPrintMessage(10, "foo"); // 10 < 11, return false

logger.shouldPrintMessage(11, "foo"); // 11 >= 11, return true, next allowed timestamp for "foo" is

// 11 + 10 = 21

**Constraints:**

* 0 <= timestamp <= 109
* Every timestamp will be passed in non-decreasing order (chronological order).
* 1 <= message.length <= 30
* At most 104 calls will be made to shouldPrintMessage.

## Solution

#### **Approach 1: Queue + Set**

**Intuition**

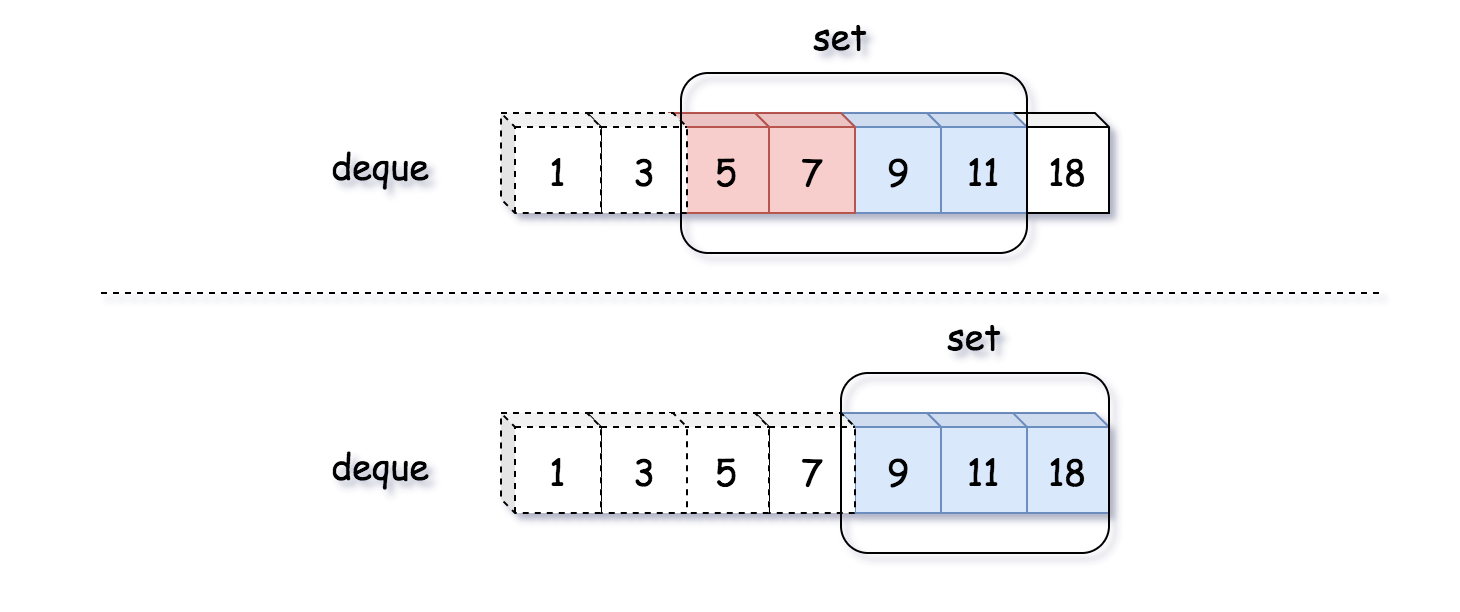
Before we tackle the problem, it is imperative to clarify the conditions of the problem, since it was not explicit in the problem description. Here is one important note:

It is possible that several messages arrive roughly at the same time.

We could interpret that the input messages are in chronological order, i.e. the timestamps of the messages are monotonically increasing, though not strictly. This constraint is critical, since it would simplify the task, as one will see in the following solutions.

As a first solution, let us build a solution intuitively following the tasks described in the problem.

We keep the incoming messages in a **queue**. In addition, to accelerate the check of duplicates, we use a **set** data structure to index the messages.



As one see can from the above example where the number indicates the timestamp of each message, the arrival of the message with the timestamp 18 would invalidate both the messages with the timestamp of 5 and 7 which go beyond the time window of 10 seconds.

**Algorithm**

* First of all, we use a queue as a sort of sliding window to keep all the printable messages in certain time frame (10 seconds).
* At the arrival of each incoming message, it comes with a timestamp. This timestamp implies the evolution of the sliding windows. Therefore, we should first invalidate those expired messages in our queue.
* Since the queue and set data structures should be in sync with each other, we would also remove those expired messages from our message set.
* After the updates of our message queue and set, we then simply check if there is any duplicate for the new incoming message. If not, we add the message to the queue as well as the set.

|  |
| --- |
| class Pair<U, V> {  public U first;  public V second;  public Pair(U first, V second) {  this.first = first;  this.second = second;  }  }  class Logger {  private LinkedList<Pair<String, Integer>> msgQueue;  private HashSet<String> msgSet;  /\*\* Initialize your data structure here. \*/  public Logger() {  msgQueue = new LinkedList<Pair<String, Integer>>();  msgSet = new HashSet<String>();  }  /\*\*  \* Returns true if the message should be printed in the given timestamp, otherwise returns false.  \*/  public boolean shouldPrintMessage(int timestamp, String message) {  // clean up.  while (msgQueue.size() > 0) {  Pair<String, Integer> head = msgQueue.getFirst();  if (timestamp - head.second >= 10) {  msgQueue.removeFirst();  msgSet.remove(head.first);  } else  break;  }  if (!msgSet.contains(message)) {  Pair<String, Integer> newEntry = new Pair<String, Integer>(message, timestamp);  msgQueue.addLast(newEntry);  msgSet.add(message);  return true;  } else  return false;  }  } |

As one can see, the usage of set data structure is not absolutely necessary. One could simply iterate the message queue to check if there is any duplicate.

Another important note is that if the messages are not chronologically ordered then we would have to iterate through the entire queue to remove the expired messages, rather than having early stopping. Or one could use some sorted queue such as **Priority Queue** to keep the messages.

**Complexity Analysis**

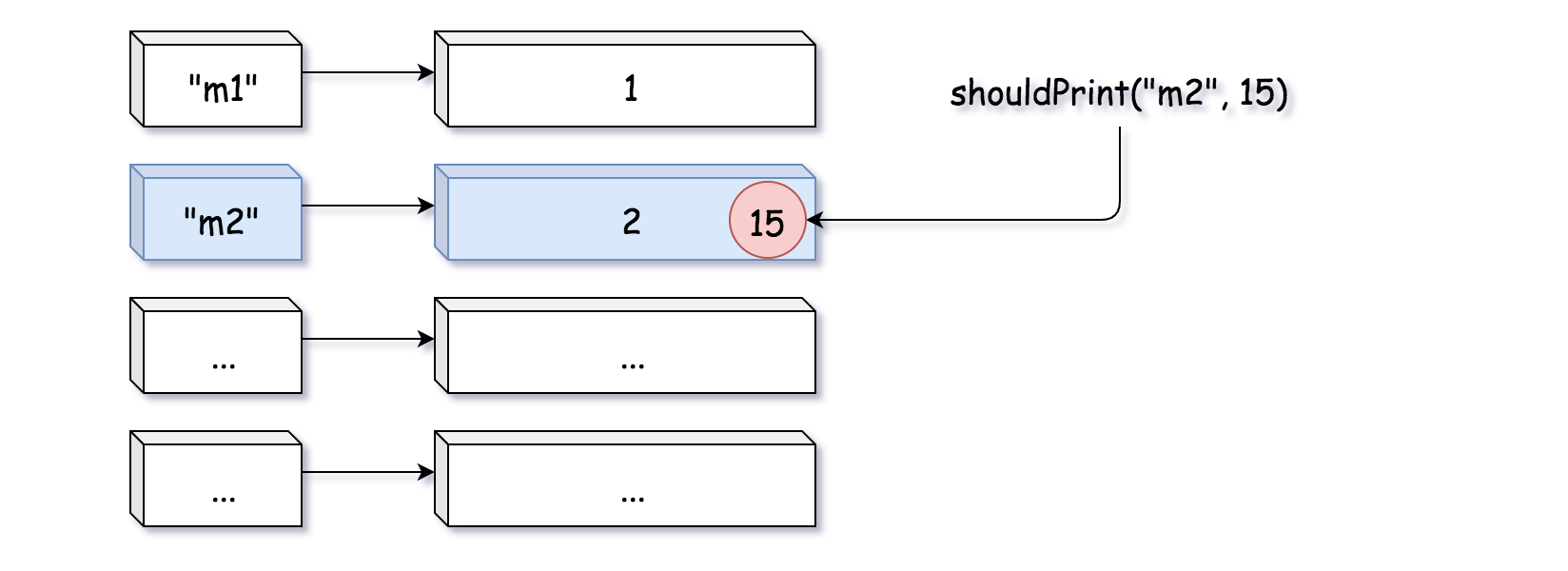
* Time Complexity: O(*N*) where *N* is the size of the queue. In the worst case, all the messages in the queue become obsolete. As a result, we need clean them up.
* Space Complexity: O(*N*) where *N* is the size of the queue. We keep the incoming messages in both the queue and set. The upper bound of the required space would be 2*N*, if we have no duplicate at all.

#### **Approach 2: Hashtable / Dictionary**

**Intuition**

One could combine the queue and set data structure into a **hashtable** or **dictionary**, which gives us the capacity of keeping all unique messages as of queue as well as the capacity to quickly evaluate the duplication of messages as of set.

The idea is that we keep a hashtable/dictionary with the message as key, and its timestamp as the value. The hashtable keeps all the unique messages along with the latest timestamp that the message was printed.



As one can see from the above example, there is an entry in the hashtable with the message m2 and the timestamp 2. Then there comes another message m2 with the timestamp 15. Since the message was printed 13 seconds before (i.e. beyond the buffer window), it is therefore eligible to print again the message. As a result, the timestamp of the message m2 would be updated to 15.

**Algorithm**

* We initialize a hashtable/dictionary to keep the messages along with the timestamp.
* At the arrival of a new message, the message is eligible to be printed with either of the two conditions as follows:
  + case 1). we have never seen the message before.
  + case 2). we have seen the message before, and it was printed more than 10 seconds ago.
* In both of the above cases, we would then update the entry that is associated with the message in the hashtable, with the latest timestamp.

|  |
| --- |
| class Logger {  private HashMap<String, Integer> msgDict;  /\*\* Initialize your data structure here. \*/  public Logger() {  msgDict = new HashMap<String, Integer>();  }  /\*\*  \* Returns true if the message should be printed in the given timestamp, otherwise returns false.  \*/  public boolean shouldPrintMessage(int timestamp, String message) {  if (!this.msgDict.containsKey(message)) {  this.msgDict.put(message, timestamp);  return true;  }  Integer oldTimestamp = this.msgDict.get(message);  if (timestamp - oldTimestamp >= 10) {  this.msgDict.put(message, timestamp);  return true;  } else  return false;  }  } |

Note: for clarity, we separate the two cases into two blocks. One could combine the two blocks together to have a more concise solution.

The main difference between this approach with hashtable and the previous approach with queue is that in previous approach we do proactive cleaning, i.e. at each invocation of function, we first remove those expired messages.

While in this approach, we keep all the messages even when they are expired. This characteristics might become problematic, since the usage of memory would keep on growing over the time. Sometimes it might be more desirable to have the garbage collection property of the previous approach.

**Complexity Analysis**

* Time Complexity: O(1). The lookup and update of the hashtable takes a constant time.
* Space Complexity: O(*M*) where *M* is the size of all incoming messages. Over the time, the hashtable would have an entry for each unique message that has appeared.

## Practical Application - Design the Key

Another problem you might encounter is that when you meet some problems which you thought can be solved by a hash table, you are not able to figure out a proper key.

In this chapter, we are going to solve this problem. We will discuss with some examples, provide some exercise and finally, come to a conclusion to give you some tips to solve some classical problems.

**Design the Key**

In the previous problems, the choice of key is comparatively straightforward. Unfortunately, sometimes you have to think it over to design a suitable key when using a hash table.

 Let's look at an example:

Given an array of strings, group anagrams together.

As we know, a hash map can perform really well in grouping information by key. But we cannot use the original string as key directly. We have to design a proper key to present the type of anagrams. For instance, there are two strings "eat" and "ate" which should be in the same group. While "eat" and "act" should not be grouped together.

***Solution***

Actually, designing a key is to build a mapping relationship by yourself between the original information and the actual key used by hash map. When you design a key, you need to guarantee that:

1. All values belong to the same group will be mapped in the same group.

2. Values which needed to be separated into different groups will not be mapped into the same group.

This process is similar to design a hash function, but here is an essential difference. A hash function satisfies the first rule but might not satisfy the second one. But your mapping function should satisfy both of them.

In the example above, our mapping strategy can be: sort the string and use the sorted string as the key. That is to say, both "eat" and "ate" will be mapped to "aet".

The mapping strategy can be really tricky sometimes. We provide some exercise for you in this chapter and will give a summary after that.

**Group Anagrams**

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lower-case English letters.

#### **Approach 1: Categorize by Sorted String**

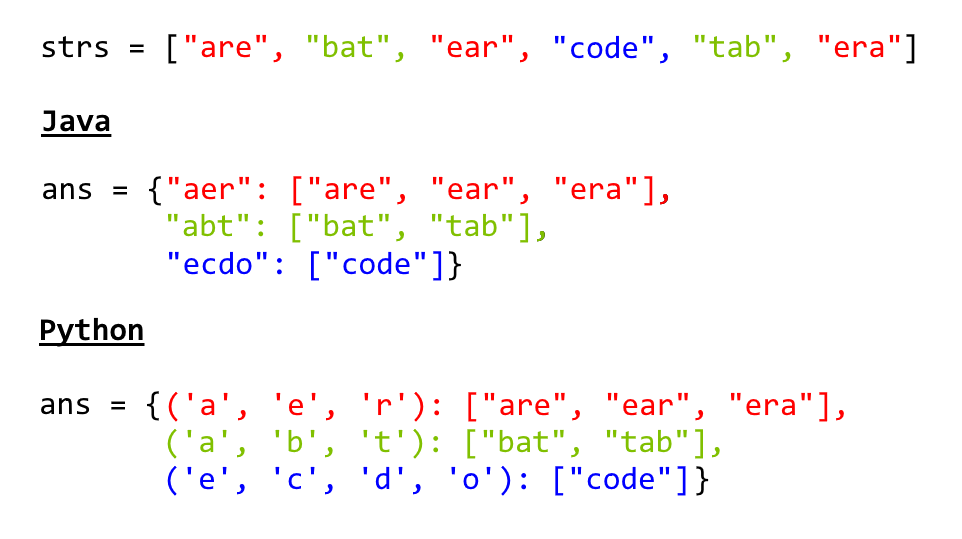
**Intuition**

Two strings are anagrams if and only if their sorted strings are equal.

**Algorithm**

Maintain a map ans : {String -> List} where each key K is a sorted string, and each value is the list of strings from the initial input that when sorted, are equal to K.

In Java, we will store the key as a string, eg. code. In Python, we will store the key as a hashable tuple, eg. ('c', 'o', 'd', 'e').



|  |
| --- |
| class Solution {  public List<List<String>> groupAnagrams(String[] strs) {  if (strs.length == 0) return new ArrayList();  Map<String, List> ans = new HashMap<String, List>();  for (String s : strs) {  char[] ca = s.toCharArray();  Arrays.sort(ca);  String key = String.valueOf(ca);  if (!ans.containsKey(key)) ans.put(key, new ArrayList());  ans.get(key).add(s);  }  return new ArrayList(ans.values());  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*NK*log*K*), where *N* is the length of strs, and *K* is the maximum length of a string in strs. The outer loop has complexity *O*(*N*) as we iterate through each string. Then, we sort each string in *O*(*K*log*K*) time.
* Space Complexity: *O*(*NK*), the total information content stored in ans.

#### **Approach 1: Categorize by Sorted String**

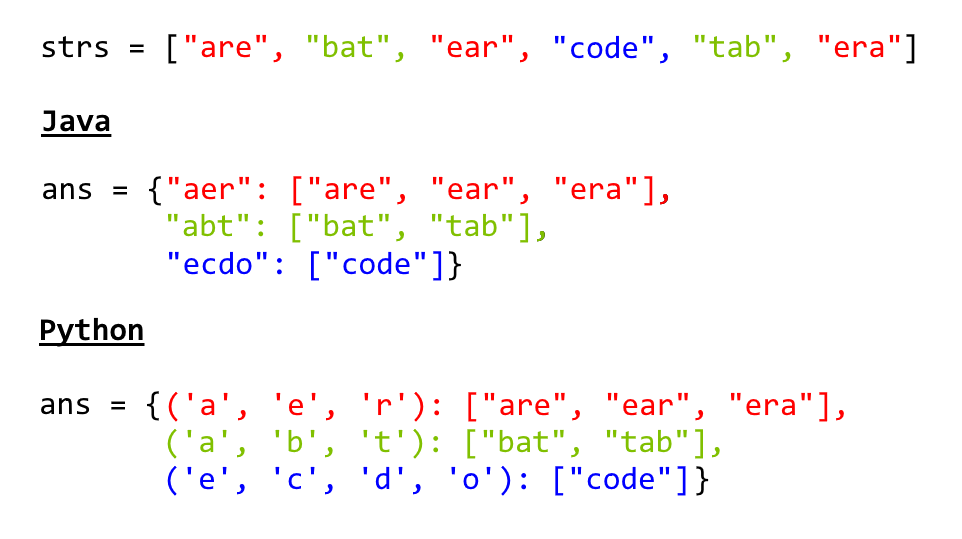
**Intuition**

Two strings are anagrams if and only if their sorted strings are equal.

**Algorithm**

Maintain a map ans : {String -> List} where each key \text{K}K is a sorted string, and each value is the list of strings from the initial input that when sorted, are equal to \text{K}K.

In Java, we will store the key as a string, eg. code. In Python, we will store the key as a hashable tuple, eg. ('c', 'o', 'd', 'e').



**Complexity Analysis**

* Time Complexity: O(NK \log K)*O*(*NK*log*K*), where N*N* is the length of strs, and K*K* is the maximum length of a string in strs. The outer loop has complexity O(N)*O*(*N*) as we iterate through each string. Then, we sort each string in O(K \log K)*O*(*K*log*K*) time.
* Space Complexity: O(NK)*O*(*NK*), the total information content stored in ans.

#### **Approach 2: Categorize by Count**

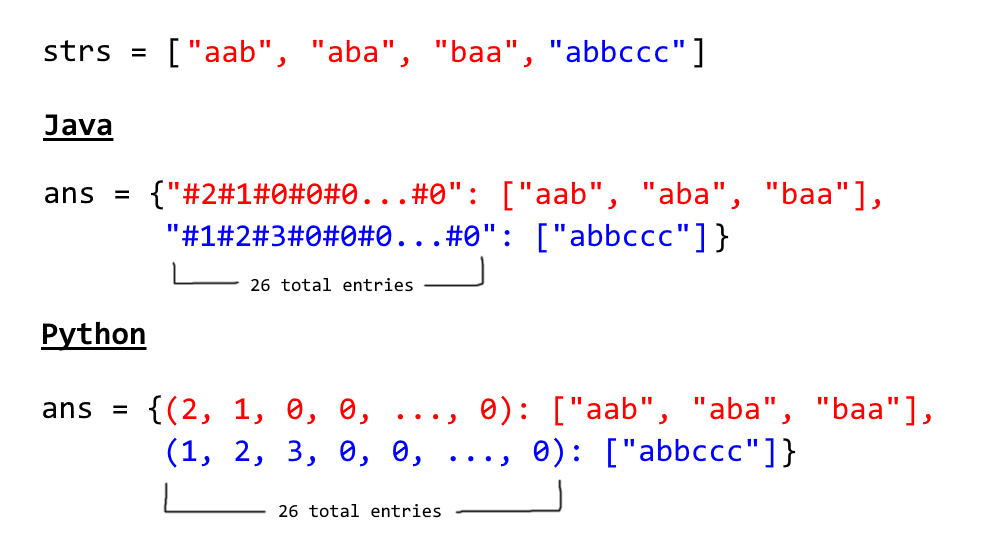
**Intuition**

Two strings are anagrams if and only if their character counts (respective number of occurrences of each character) are the same.

**Algorithm**

We can transform each string s into a character count, count, consisting of 26 non-negative integers representing the number of a's, b's, c's, etc. We use these counts as the basis for our hash map.

In Java, the hashable representation of our count will be a string delimited with '**#**' characters. For example, abbccc will be #1#2#3#0#0#0...#0 where there are 26 entries total. In python, the representation will be a tuple of the counts. For example, abbccc will be (1, 2, 3, 0, 0, ..., 0), where again there are 26 entries total.



|  |
| --- |
| class Solution {  public List<List<String>> groupAnagrams(String[] strs) {  if (strs.length == 0) return new ArrayList();  Map<String, List> ans = new HashMap<String, List>();  int[] count = new int[26];  for (String s : strs) {  Arrays.fill(count, 0);  for (char c : s.toCharArray()) count[c - 'a']++;  StringBuilder sb = new StringBuilder("");  for (int i = 0; i < 26; i++) {  sb.append('#');  sb.append(count[i]);  }  String key = sb.toString();  if (!ans.containsKey(key)) ans.put(key, new ArrayList());  ans.get(key).add(s);  }  return new ArrayList(ans.values());  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*NK*), where *N* is the length of strs, and *K* is the maximum length of a string in strs. Counting each string is linear in the size of the string, and we count every string.
* Space Complexity: *O*(*NK*), the total information content stored in ans.

**Group Shifted Strings**

Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which forms the sequence:

"abc" -> "bcd" -> ... -> "xyz"

Given a list of **non-empty** strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

**Example:**

**Input:** ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],

**Output:**

[

["abc","bcd","xyz"],

["az","ba"],

["acef"],

["a","z"]

]

**Valid Sudoku**

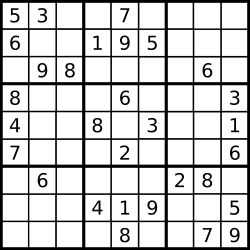
Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

**Note:**

* A Sudoku board (partially filled) could be valid but is not necessarily solvable.
* Only the filled cells need to be validated according to the mentioned rules.

**Example 1:**



**Input:** board =

[["5","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

**Output:** true

**Example 2:**

**Input:** board =

[["8","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

**Output:** false

**Explanation:** Same as Example 1, except with the **5** in the top left corner being modified to **8**. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

**Constraints:**

* board.length == 9
* board[i].length == 9
* board[i][j] is a digit or '.'.

## Solution Intuition

The naive solution would be to iterate three times over the board to ensure that :

* There is no rows with duplicates.
* There is no columns with duplicates.
* There is no sub-boxes with duplicates.

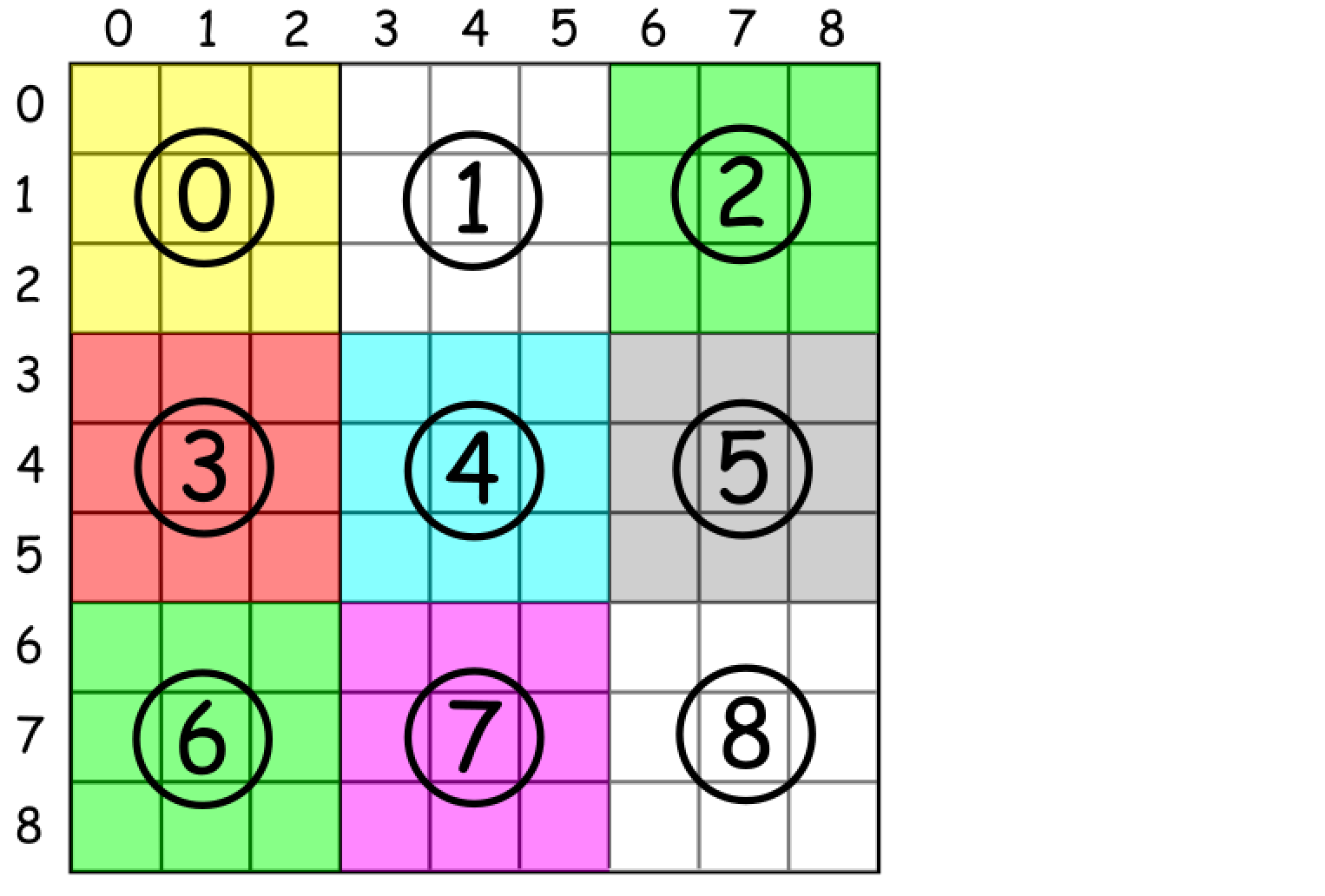
Actually, all this could be done in just one iteration.

#### **Approach 1: One iteration**

Let's first discuss two questions.

* How to enumerate sub-boxes?

One could use box\_index = (row / 3) \* 3 + col / 3 where / is an integer division, row is a row number, and col is a column number.

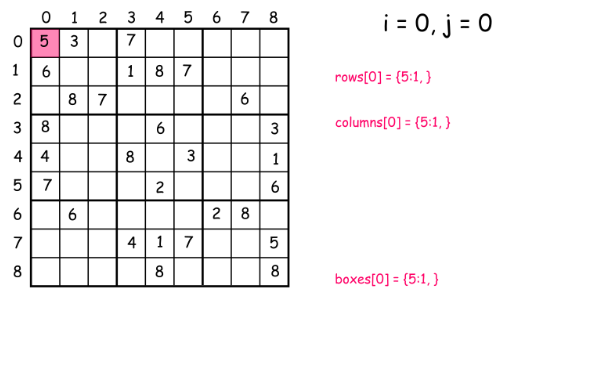


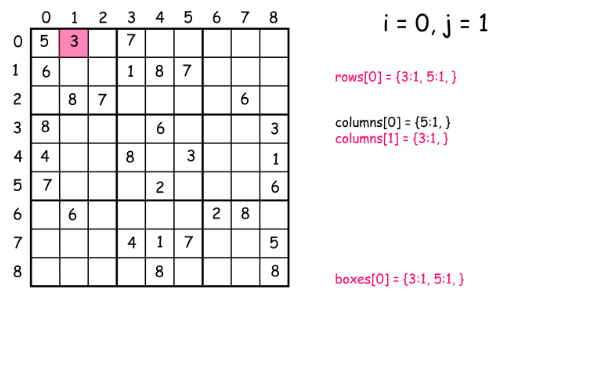
* How to ensure that there is no duplicates in a row / column / box?

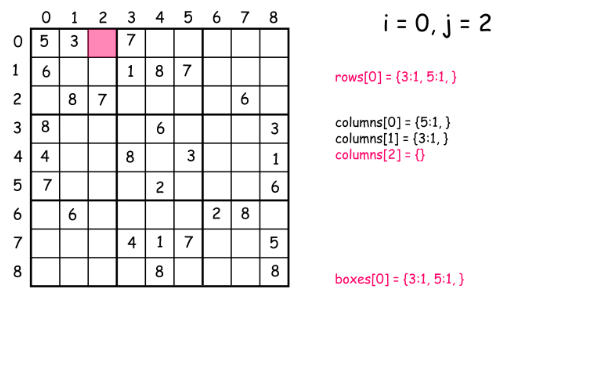
One could just track all values which were already encountered in a hash map value -> count.

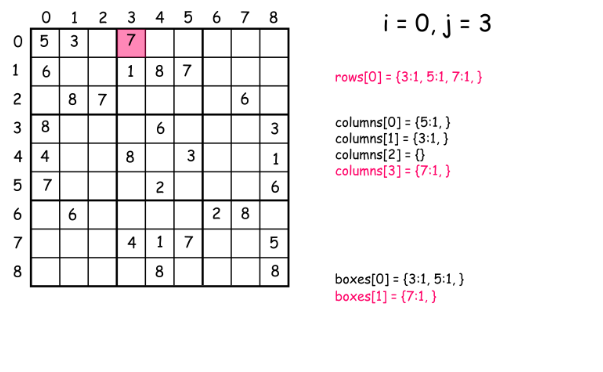
Now everything is ready for the overall algorithm :

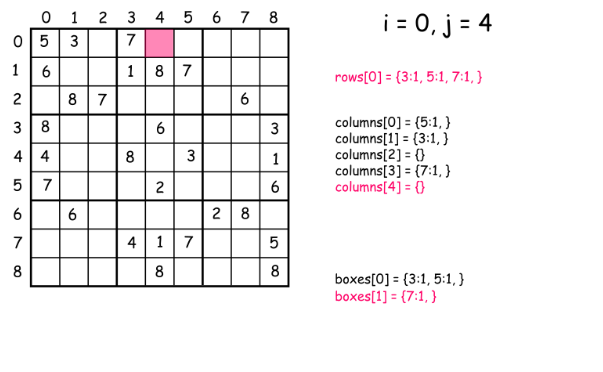
* Move along the board.
  + Check for each cell value if it was seen already in the current row / column / box :
    - Return false if yes.
    - Keep this value for a further tracking if no.
* Return true.

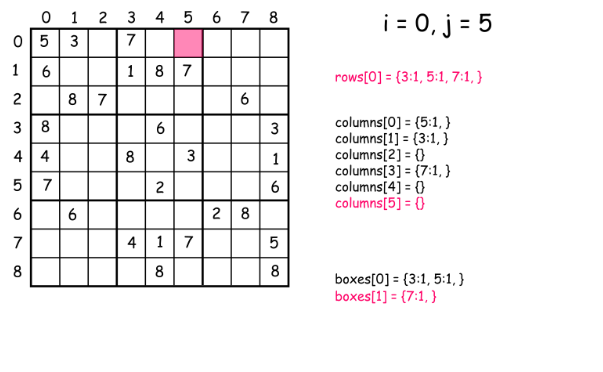


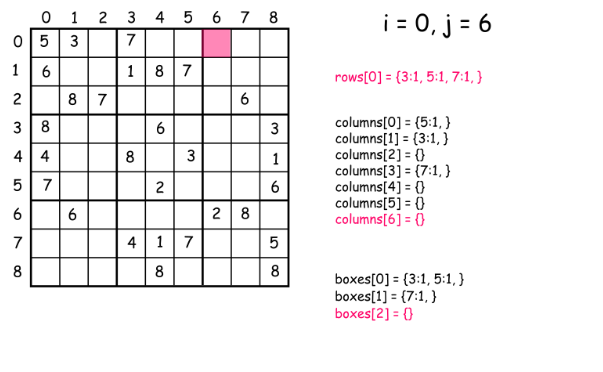


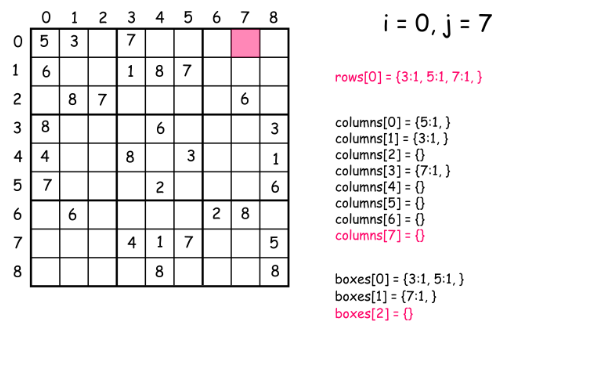


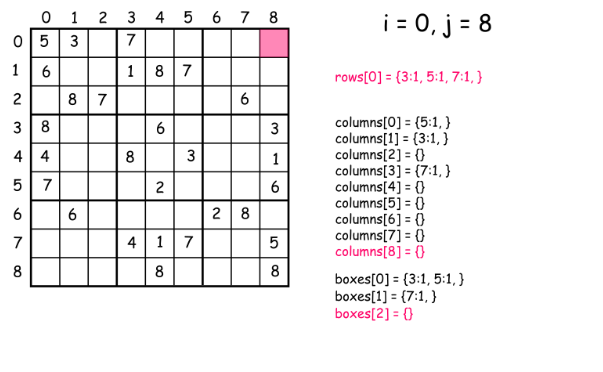


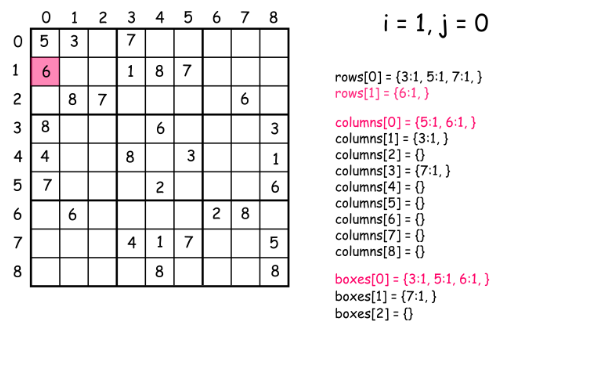


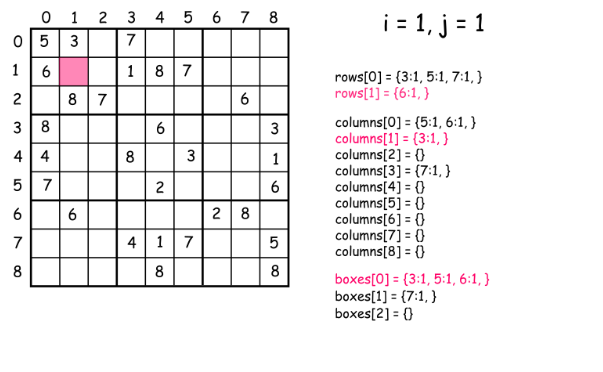




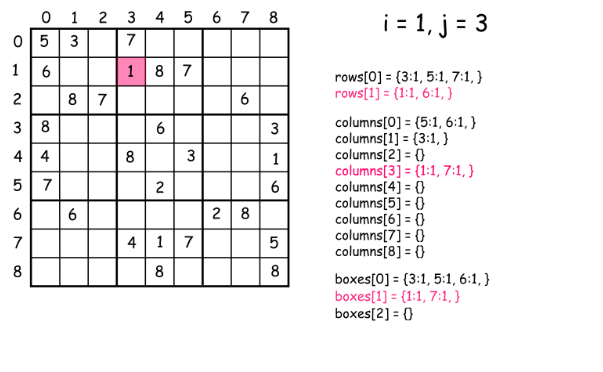


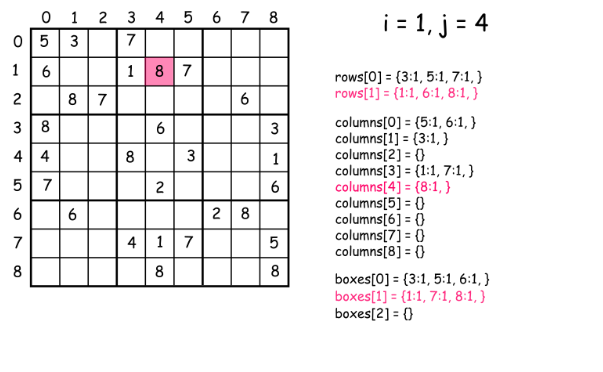


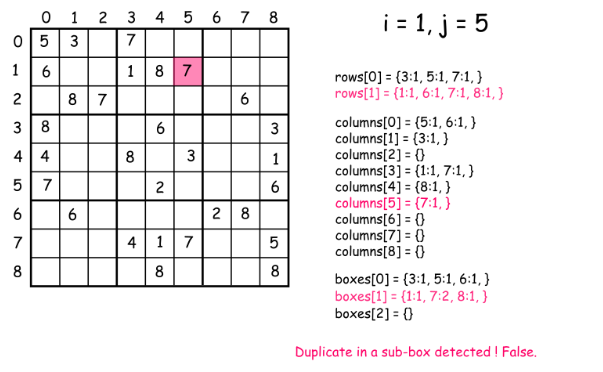












|  |
| --- |
| class Solution {  public boolean isValidSudoku(char[][] board) {  // init data  HashMap<Integer, Integer> [] rows = new HashMap[9];  HashMap<Integer, Integer> [] columns = new HashMap[9];  HashMap<Integer, Integer> [] boxes = new HashMap[9];  for (int i = 0; i < 9; i++) {  rows[i] = new HashMap<Integer, Integer>();  columns[i] = new HashMap<Integer, Integer>();  boxes[i] = new HashMap<Integer, Integer>();  }  // validate a board  for (int i = 0; i < 9; i++) {  for (int j = 0; j < 9; j++) {  char num = board[i][j];  if (num != '.') {  int n = (int)num;  int box\_index = (i / 3 ) \* 3 + j / 3;  // keep the current cell value  rows[i].put(n, rows[i].getOrDefault(n, 0) + 1);  columns[j].put(n, columns[j].getOrDefault(n, 0) + 1);  boxes[box\_index].put(n, boxes[box\_index].getOrDefault(n, 0) + 1);  // check if this value has been already seen before  if (rows[i].get(n) > 1 || columns[j].get(n) > 1 || boxes[box\_index].get(n) > 1)  return false;  }  }  }  return true;  }  } |

**Complexity Analysis**

* Time complexity : O(1) since all we do here is just one iteration over the board with 81 cells.
* Space complexity : O(1).

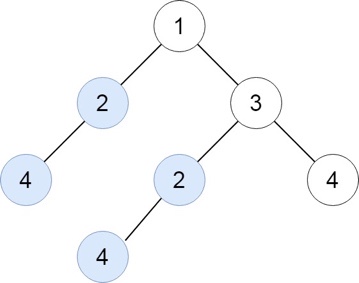
**Find Duplicate Subtrees**

Given the root of a binary tree, return all **duplicate subtrees**.

For each kind of duplicate subtrees, you only need to return the root node of any **one** of them.

Two trees are **duplicate** if they have the **same structure** with the **same node values**.

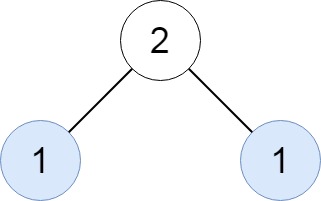
**Example 1:**



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

**Output:** [[2,4],[4]]

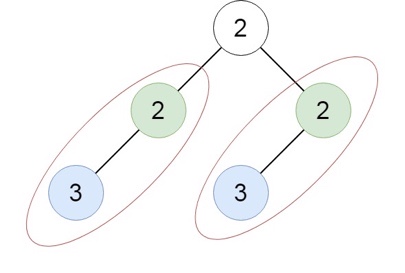
**Example 2:**



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

**Output:** [[1]]

**Example 3:**



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

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

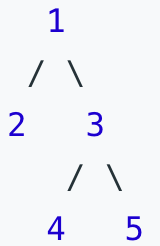
**Constraints:**

* The number of the nodes in the tree will be in the range [1, 10^4]
* -200 <= Node.val <= 200

#### **Approach #1: Depth-First Search [Accepted]**

**Intuition**

We can serialize each subtree. For example, the tree



can be represented as the serialization 1,2,#,#,3,4,#,#,5,#,#, which is a unique representation of the tree.

**Algorithm**

Perform a depth-first search, where the recursive function returns the serialization of the tree. At each node, record the result in a map, and analyze the map after to determine duplicate subtrees.

|  |
| --- |
| class Solution {  Map<String, Integer> count;  List<TreeNode> ans;  public List<TreeNode> findDuplicateSubtrees(TreeNode root) {  count = new HashMap();  ans = new ArrayList();  collect(root);  return ans;  }  public String collect(TreeNode node) {  if (node == null) return "#";  String serial = node.val + "," + collect(node.left) + "," + collect(node.right);  count.put(serial, count.getOrDefault(serial, 0) + 1);  if (count.get(serial) == 2)  ans.add(node);  return serial;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2), where *N* is the number of nodes in the tree. We visit each node once, but each creation of serial may take *O*(*N*) work.
* Space Complexity: O(N^2), the size of count.

#### **Approach #2: Unique Identifier [Accepted]**

**Intuition**

Suppose we have a unique identifier for subtrees: two subtrees are the same if and only if they have the same id.

Then, for a node with left child id of x and right child id of y, (node.val, x, y) uniquely determines the tree.

**Algorithm**

If we have seen the triple (node.val, x, y) before, we can use the identifier we've remembered. Otherwise, we'll create a new one.

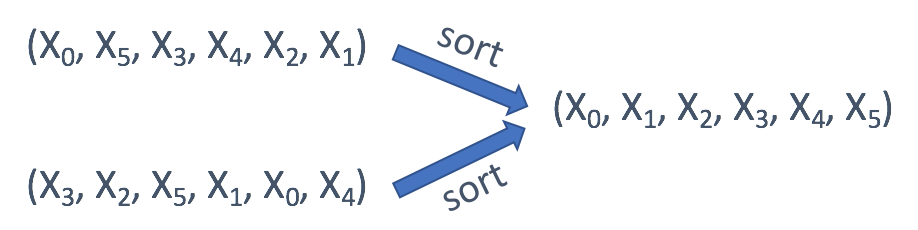
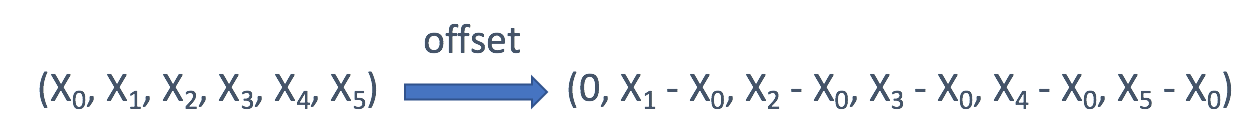
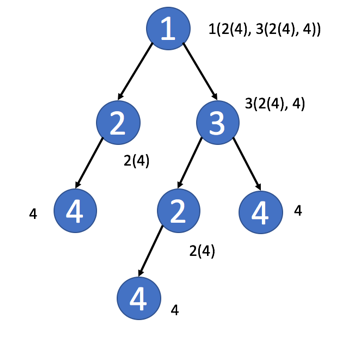
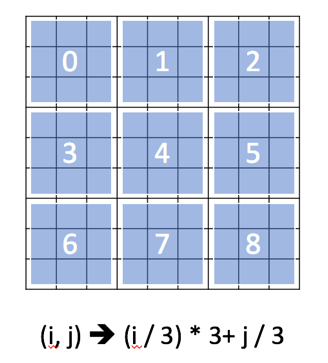
|  |
| --- |
| class Solution {  int t;  Map<String, Integer> trees;  Map<Integer, Integer> count;  List<TreeNode> ans;  public List<TreeNode> findDuplicateSubtrees(TreeNode root) {  t = 1;  trees = new HashMap();  count = new HashMap();  ans = new ArrayList();  lookup(root);  return ans;  }  public int lookup(TreeNode node) {  if (node == null) return 0;  String serial = node.val + "," + lookup(node.left) + "," + lookup(node.right);  int uid = trees.computeIfAbsent(serial, x-> t++);  count.put(uid, count.getOrDefault(uid, 0) + 1);  if (count.get(uid) == 2)  ans.add(node);  return uid;  }  } |

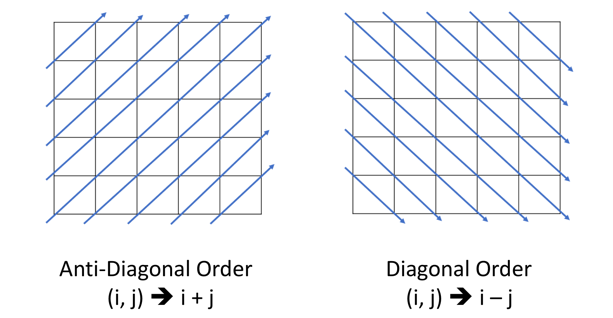
**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the number of nodes in the tree. We visit each node once.
* Space Complexity: *O*(*N*). Every structure we use is using *O*(1) storage per node.

**Design the Key - Summary**

Here are some takeaways about how to design the key for you.

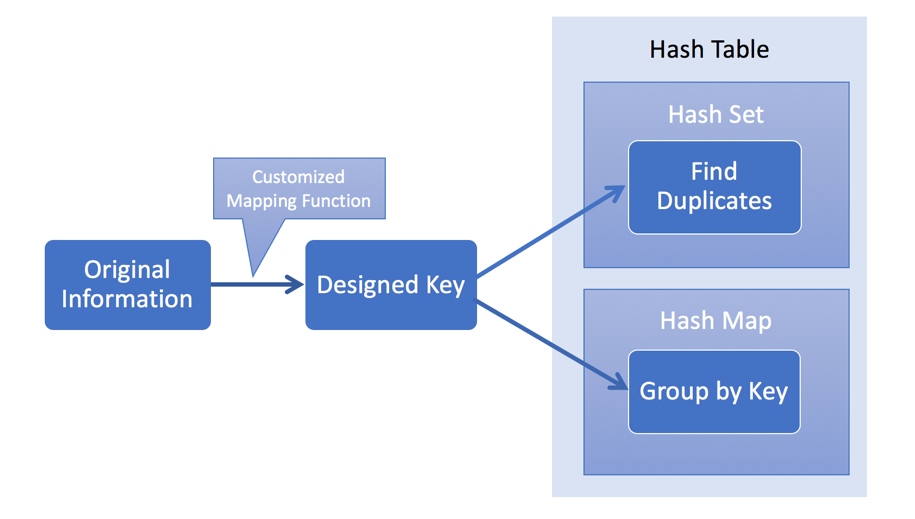
1. When the order of each element in the string/array doesn't matter, you can use the sorted string/array as the key.   
   
2. If you only care about the offset of each value, usually the offset from the first value, you can use the offset as the key.  
   
3. In a tree, you might want to directly use the TreeNode as key sometimes. But in most cases, the serialization of the subtree might be a better idea.  
   
4. In a matrix, you might want to use the row index or the column index as key.
5. In a Sudoku, you can combine the row index and the column index to identify which block this element belongs to.  
   
6. Sometimes, in a matrix, you might want to aggregate the values in the same diagonal line.



## Conclusion

We are now more familiar with the principle and usage of the hash table.

We have also talked about how to apply hash table from three respects in previous chapters. Here we combine them together and come up with a typical thinking process to solve problems by hash table flexibly.



What's more, we will meet more complicated problems sometimes. We might need to:

* use several hash tables together
* combine the hash table with other data structure
* combine the hash table with other algorithms
* ...

We provide some exercise in this chapter. After finishing this chapter, you will be more confident with hash table related problems.

**Jewels and Stones**

You're given strings jewels representing the types of stones that are jewels, and stones representing the stones you have. Each character in stones is a type of stone you have. You want to know how many of the stones you have are also jewels.

Letters are case sensitive, so "a" is considered a different type of stone from "A".

**Example 1:**

**Input:** jewels = "aA", stones = "aAAbbbb"

**Output:** 3

**Example 2:**

**Input:** jewels = "z", stones = "ZZ"

**Output:** 0

**Constraints:**

* 1 <= jewels.length, stones.length <= 50
* jewels and stones consist of only English letters.
* All the characters of jewels are **unique**.

   Hide Hint #1

For each stone, check if it is a jewel.

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

**Intuition and Algorithm**

For each stone, check whether it matches any of the jewels. We can check with a linear scan.

|  |
| --- |
| class Solution {  public int numJewelsInStones(String J, String S) {  int ans = 0;  for (char s: S.toCharArray()) // For each stone...  for (char j: J.toCharArray()) // For each jewel...  if (j == s) { // If the stone is a jewel...  ans++;  break; // Stop searching whether this stone 's' is a jewel  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*J*.length∗*S*.length)).
* Space Complexity: *O*(1) additional space complexity in Python. In Java, this can be *O*(*J*.length∗*S*.length)) because of the creation of new arrays.

#### **Approach #2: Hash Set [Accepted]**

**Intuition and Algorithm**

For each stone, check whether it matches any of the jewels. We can check efficiently with a Hash Set.

|  |
| --- |
| class Solution {  public int numJewelsInStones(String J, String S) {  Set<Character> Jset = new HashSet();  for (char j: J.toCharArray())  Jset.add(j);  int ans = 0;  for (char s: S.toCharArray())  if (Jset.contains(s))  ans++;  return ans;  }  } |

**Complexity Analysis**

* Time Complexity:  *O*(*J*.length+*S*.length). The *O*(*J*.length) part comes from creating J. The  *O*(*S*.length) part comes from searching S.
* Space Complexity:  *O*(*J*.length).

**Longest Substring Without Repeating Characters**

Given a string s, find the length of the **longest substring** without repeating characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Example 4:**

**Input:** s = ""

**Output:** 0

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of English letters, digits, symbols and spaces.

## Solution Article

#### **Approach 1: Brute Force**

**Intuition**

Check all the substring one by one to see if it has no duplicate character.

**Algorithm**

Suppose we have a function boolean allUnique(String substring) which will return true if the characters in the substring are all unique, otherwise false. We can iterate through all the possible substrings of the given string s and call the function allUnique. If it turns out to be true, then we update our answer of the maximum length of substring without duplicate characters.

Now let's fill the missing parts:

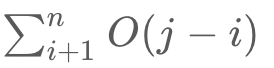
1. To enumerate all substrings of a given string, we enumerate the start and end indices of them. Suppose the start and end indices are *i* and *j*, respectively. Then we have 0≤*i*<*j*≤*n* (here end index *j* is exclusive by convention). Thus, using two nested loops with i*i* from 0 to *n*−1 and *j* from *i*+1 to *n*, we can enumerate all the substrings of s.
2. To check if one string has duplicate characters, we can use a set. We iterate through all the characters in the string and put them into the set one by one. Before putting one character, we check if the set already contains it. If so, we return false. After the loop, we return true.

|  |
| --- |
| public class Solution {  public int lengthOfLongestSubstring(String s) {  int n = s.length();  int ans = 0;  for (int i = 0; i < n; i++)  for (int j = i + 1; j <= n; j++)  if (allUnique(s, i, j)) ans = Math.max(ans, j - i);  return ans;  }  public boolean allUnique(String s, int start, int end) {  Set<Character> set = new HashSet<>();  for (int i = start; i < end; i++) {  Character ch = s.charAt(i);  if (set.contains(ch)) return false;  set.add(ch);  }  return true;  }  } |

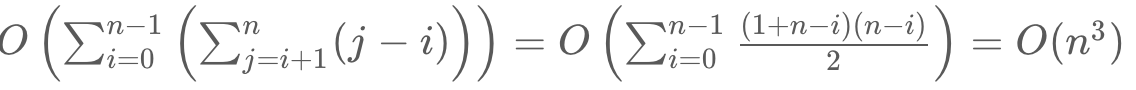
**Complexity Analysis**

* Time complexity : O(n^3).

To verify if characters within index range [*i*,*j*) are all unique, we need to scan all of them. Thus, it costs *O*(*j*−*i*) time.

For a given i, the sum of time costed by each *j*∈[*i*+1,*n*] is  


Thus, the sum of all the time consumption is:



* Space complexity : *O*(*min*(*n*,*m*)). We need *O*(*k*) space for checking a substring has no duplicate characters, where *k* is the size of the Set. The size of the Set is upper bounded by the size of the string *n* and the size of the charset/alphabet *m*.

#### **Approach 2: Sliding Window**

**Algorithm**

The naive approach is very straightforward. But it is too slow. So how can we optimize it?

In the naive approaches, we repeatedly check a substring to see if it has duplicate character. But it is unnecessary. If a substring *sij*​ from index *i* to *j*−1 is already checked to have no duplicate characters. We only need to check if *s*[*j*] is already in the substring *sij*​.

To check if a character is already in the substring, we can scan the substring, which leads to an O(n^2) algorithm. But we can do better.

By using HashSet as a sliding window, checking if a character in the current can be done in *O*(1).

A sliding window is an abstract concept commonly used in array/string problems. A window is a range of elements in the array/string which usually defined by the start and end indices, i.e. [*i*,*j*) (left-closed, right-open). A sliding window is a window "slides" its two boundaries to the certain direction. For example, if we slide [*i*,*j*) to the right by 1 element, then it becomes [*i*+1,*j*+1) (left-closed, right-open).

Back to our problem. We use HashSet to store the characters in current window [*i*,*j*) (*j*=*i* initially). Then we slide the index *j* to the right. If it is not in the HashSet, we slide *j* further. Doing so until s[j] is already in the HashSet. At this point, we found the maximum size of substrings without duplicate characters start with index *i*. If we do this for all *i*, we get our answer.

|  |
| --- |
| public class Solution {  public int lengthOfLongestSubstring(String s) {  int n = s.length();  Set<Character> set = new HashSet<>();  int ans = 0, i = 0, j = 0;  while (i < n && j < n) {  // try to extend the range [i, j]  if (!set.contains(s.charAt(j))){  set.add(s.charAt(j++));  ans = Math.max(ans, j - i);  }  else {  set.remove(s.charAt(i++));  }  }  return ans;  }  } |

**Complexity Analysis**

* Time complexity : *O*(2*n*)=*O*(*n*). In the worst case each character will be visited twice by *i* and *j*.
* Space complexity : *O*(*min*(*m*,*n*)). Same as the previous approach. We need *O*(*k*) space for the sliding window, where *k* is the size of the Set. The size of the Set is upper bounded by the size of the string *n* and the size of the charset/alphabet *m*.

#### **Approach 3: Sliding Window Optimized**

The above solution requires at most 2n steps. In fact, it could be optimized to require only n steps. Instead of using a set to tell if a character exists or not, we could define a mapping of the characters to its index. Then we can skip the characters immediately when we found a repeated character.

The reason is that if *s*[*j*] have a duplicate in the range [*i*,*j*) with index *j*′, we don't need to increase *i* little by little. We can skip all the elements in the range [*i*,*j*′] and let i*i* to be *j*′+1 directly.

**Java (Using HashMap)**

|  |
| --- |
| public class Solution {  public int lengthOfLongestSubstring(String s) {  int n = s.length(), ans = 0;  Map<Character, Integer> map = new HashMap<>(); // current index of character  // try to extend the range [i, j]  for (int j = 0, i = 0; j < n; j++) {  if (map.containsKey(s.charAt(j))) {  i = Math.max(map.get(s.charAt(j)), i);  }  ans = Math.max(ans, j - i + 1);  map.put(s.charAt(j), j + 1);  }  return ans;  }  } |

Here is a visualization of the above code.

**Java (Assuming ASCII 128)**

The previous implements all have no assumption on the charset of the string s.

If we know that the charset is rather small, we can replace the Map with an integer array as direct access table.

Commonly used tables are:

* int[26] for Letters 'a' - 'z' or 'A' - 'Z'
* int[128] for ASCII
* int[256] for Extended ASCII

|  |
| --- |
| public class Solution {  public int lengthOfLongestSubstring(String s) {  int n = s.length(), ans = 0;  int[] index = new int[128]; // current index of character  // try to extend the range [i, j]  for (int j = 0, i = 0; j < n; j++) {  i = Math.max(index[s.charAt(j)], i);  ans = Math.max(ans, j - i + 1);  index[s.charAt(j)] = j + 1;  }  return ans;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Index *j* will iterate *n* times.
* Space complexity (HashMap) : *O*(*min*(*m*,*n*)). Same as the previous approach.
* Space complexity (Table): *O*(*m*). *m* is the size of the charset.

**Two Sum III - Data structure design**

Design a data structure that accepts a stream of integers and checks if it has a pair of integers that sum up to a particular value.

Implement the TwoSum class:

* TwoSum() Initializes the TwoSum object, with an empty array initially.
* void add(int number) Adds number to the data structure.
* boolean find(int value) Returns true if there exists any pair of numbers whose sum is equal to value, otherwise, it returns false.

**Example 1:**

**Input**

["TwoSum", "add", "add", "add", "find", "find"]

[[], [1], [3], [5], [4], [7]]

**Output**

[null, null, null, null, true, false]

**Explanation**

TwoSum twoSum = new TwoSum();

twoSum.add(1); // [] --> [1]

twoSum.add(3); // [1] --> [1,3]

twoSum.add(5); // [1,3] --> [1,3,5]

twoSum.find(4); // 1 + 3 = 4, return true

twoSum.find(7); // No two integers sum up to 7, return false

**Constraints:**

* -105 <= number <= 105
* -231 <= value <= 231 - 1
* At most 5 \* 104 calls will be made to add and find.

## Solution

#### **Approach 1: Sorted List**

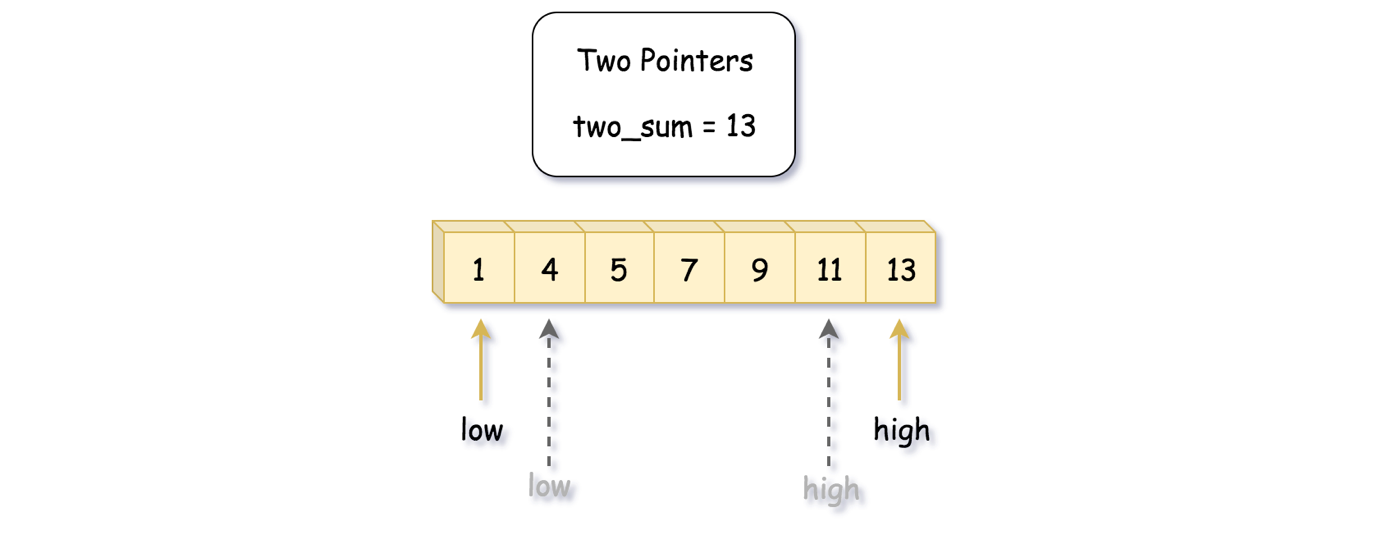
**Intuition**

First of all, the problem description is not terribly clear on the requirements of time and space complexity. But let us consider this as part of the challenge or a freedom of design. We could figure out the desired complexity for each function, by trial and error.

This is one of the followup problems to the first programming problem on LeetCode called [Two Sum](https://leetcode.com/problems/two-sum/), where one is asked to return the indice of two numbers from a **list** that could sum up to a given value.

Let us take the inspiration from the origin problem, by keeping all the incoming numbers in a list.

Given a list, one of the solutions to the Two Sum problem is called **Two-Pointers Iteration** where we iterate through the list from two directions with two pointers approaching each other.



However, one of the preconditions for the Two-Pointers Iteration solution is that the input list should be ***sorted***.

So now, here are the questions:

* Should we keep the list in order while inserting new numbers in the function add(number) ?
* Or should we do the sorting on demand, i.e. at the invocation of find(value) ?

We will address the above two questions later in the Algorithm section.

**Algorithm**

Let us first give the algorithm of Two-Pointers Iteration to find the two-sum solution from a sorted list:

* We initialize **two pointers** low and high which point to the head and the tail elements of the list respectively.
* With the two pointers, we start a **loop** to iterate the list. The loop would terminate either we find the two-sum solution or the two pointers meet each other.
* Within the loop, at each step, we would move either of the pointers, according to different conditions:
  + If the sum of the elements pointed by the current pointers is ***less than*** the desired value, then we should try to increase the sum to meet the desired value, i.e. we should move the low pointer forwards to have a larger value.
  + Similarly if the sum of the elements pointed by the current pointers is ***greater than*** the desired value, we then should try to reduce the sum by moving the high pointer towards the low pointer.
  + If the sum happen to the desired value, then we could simply do an **early return** of the function.
* If the loop is terminated at the case where the two pointers meet each other, then we can be sure that there is no solution to the desired value.

|  |
| --- |
| import java.util.Collections;  class TwoSum {  private ArrayList<Integer> nums;  private boolean is\_sorted;  /\*\* Initialize your data structure here. \*/  public TwoSum() {  this.nums = new ArrayList<Integer>();  this.is\_sorted = false;  }  /\*\* Add the number to an internal data structure.. \*/  public void add(int number) {  this.nums.add(number);  this.is\_sorted = false;  }  /\*\* Find if there exists any pair of numbers which sum is equal to the value. \*/  public boolean find(int value) {  if (!this.is\_sorted) {  Collections.sort(this.nums);  this.is\_sorted = true;  }  int low = 0, high = this.nums.size() - 1;  while (low < high) {  int twosum = this.nums.get(low) + this.nums.get(high);  if (twosum < value)  low += 1;  else if (twosum > value)  high -= 1;  else  return true;  }  return false;  }  } |

The usage pattern of the desired data structure in the online judge, as we would discover, is that the add(number) function would be called **frequently** which might be followed a less frequent call of find(value) function.

The usage pattern implies that we should try to minimize the cost of add(number) function. As a result, we sort the list within the find(value) function instead of the add(number) function.

So to the above questions about where to place the sort operation, actually both options are valid and correct. Due to the usage pattern of the two functions though, it is **less optimal** to sort the list at each add operation.

On the other hand, we do not do sorting at each occasion of find(value) neither. But rather, we sort on demand, i.e. only when the list is updated. As a result, we **amortize** the cost of the sorting over the time. And this is the optimization trick for the solution to pass the online judge.

**Complexity Analysis**

* Time Complexity:
  + For the add(number) function: O(1), since we simply append the element into the list.
  + For the find(value) function: O(*N*⋅log(*N*)). In the worst case, we would need to sort the list first, which is of  O(*N*⋅log(*N*)) time complexity normally. And later, again in the worst case we need to iterate through the entire list, which is of  O(*N*) time complexity. As a result, the overall time complexity of the function lies on O(*N*⋅log(*N*)) of the sorting operation, which dominates over the later iteration part.
* Space Complexity: the overall space complexity of the data structure is  O(*N*) where *N* is the total number of numbers that have been added.

#### **Approach 2: HashTable**

**Intuition**

As an alternative solution to the original [Two Sum](https://leetcode.com/problems/two-sum/) problem, one could employ the HashTable to index each number.

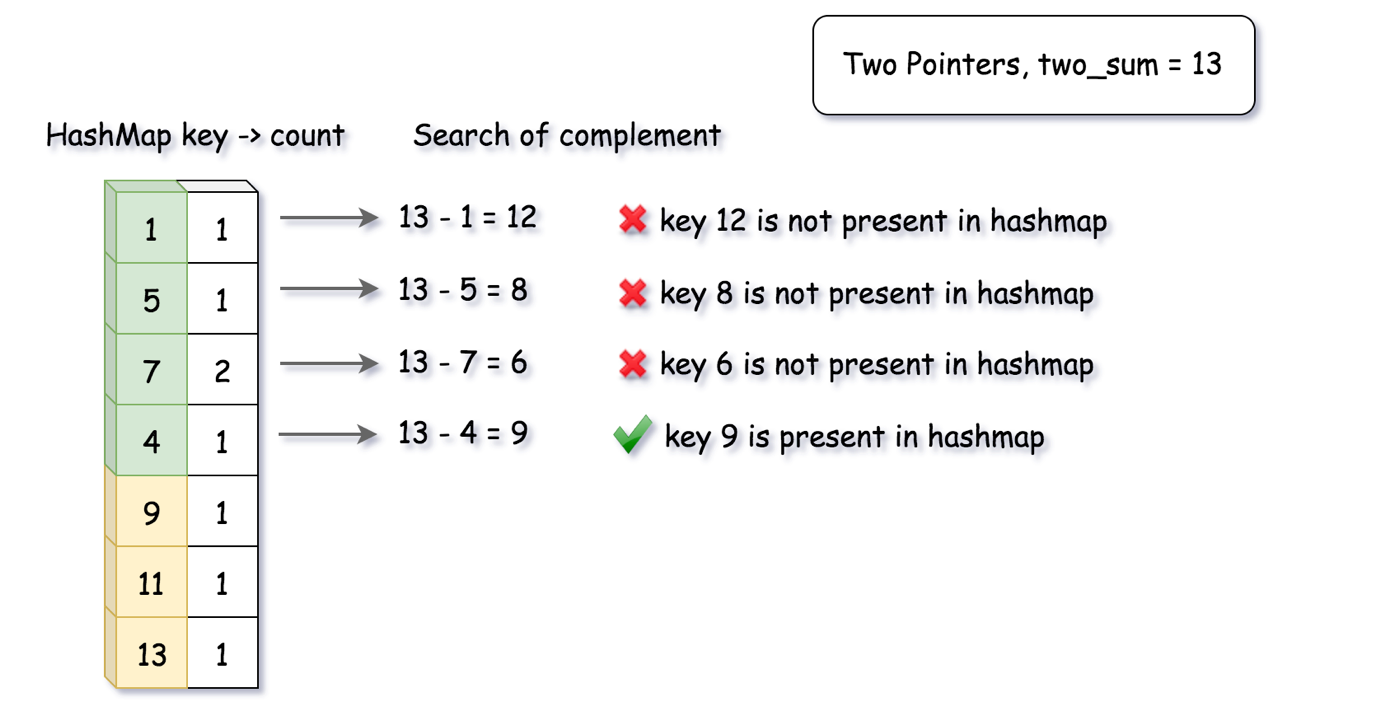
Given a desired sum value S, for each number a, we just need to verify if there exists a complement number (S-a) in the table.

As we know, the data structure of hashtable could offer us a quick lookup as well as insertion operations, which fits well with the above requirements.

**Algorithm**

* First, we initialize a hashtable container in our data structure.
* For the add(number) function, we build a frequency hashtable with the number as key and the frequency of the number as the value in the table.
* For the find(value) function, we then iterate through the hashtable over the keys. For each key (number), we check if there exists a complement (value - number) in the table. If so, we could terminate the loop and return the result.
* In a particular case, where the number and its complement are equal, we then need to check if there exists at least **two copies** of the number in the table.

We illustrate the algorithm in the following figure:



|  |
| --- |
| import java.util.HashMap;  class TwoSum {  private HashMap<Integer, Integer> num\_counts;  /\*\* Initialize your data structure here. \*/  public TwoSum() {  this.num\_counts = new HashMap<Integer, Integer>();  }  /\*\* Add the number to an internal data structure.. \*/  public void add(int number) {  if (this.num\_counts.containsKey(number))  this.num\_counts.replace(number, this.num\_counts.get(number) + 1);  else  this.num\_counts.put(number, 1);  }  /\*\* Find if there exists any pair of numbers which sum is equal to the value. \*/  public boolean find(int value) {  for (Map.Entry<Integer, Integer> entry : this.num\_counts.entrySet()) {  int complement = value - entry.getKey();  if (complement != entry.getKey()) {  if (this.num\_counts.containsKey(complement))  return true;  } else {  if (entry.getValue() > 1)  return true;  }  }  return false;  }  } |

**Complexity Analysis**

* Time Complexity:
  + For the add(number) function: O(1), since it takes a constant time to update an entry in hashtable.
  + For the find(value) function: O(*N*), where *N* is the total number of **unique** numbers. In the worst case, we would iterate through the entire table.
* Space Complexity: O(*N*), where *N* is the total number of **unique** numbers that we will see during the usage of the data structure.

**4Sum II**

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

To make problem a bit easier, all A, B, C, D have same length of N where 0 ≤ N ≤ 500. All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 1.

**Example:**

**Input:**

A = [ 1, 2]

B = [-2,-1]

C = [-1, 2]

D = [ 0, 2]

**Output:**

2

**Explanation:**

The two tuples are:

1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0

2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0

## Solution

This problem is a variation of [4Sum](https://leetcode.com/articles/4sum/), and we recommend checking that problem first. The main difference is that here we pick each element from a different array, while in 4Sum all elements come from the same array. For that reason, we cannot use the [Two Pointers](https://leetcode.com/articles/4sum/#approach-1-two-pointers) approach, where elements must be in the same sorted array.

On the bright side, we do not need to worry about using the same element twice - we pick one element at a time from each array. As you will see later, this help reduce the time complexity.

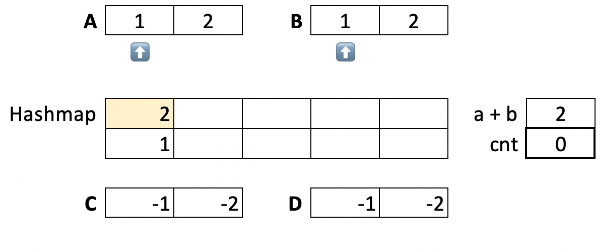
Finally, we do not need to return actual values and ensure they are unique; we just count each combination of four elements that sums to zero.

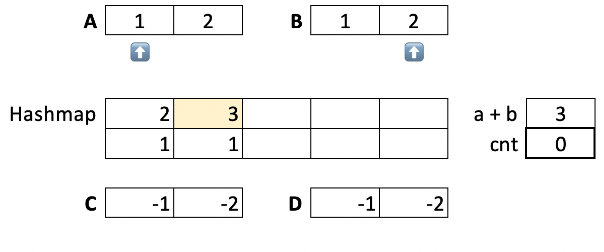
#### **Approach 1: Hashmap**

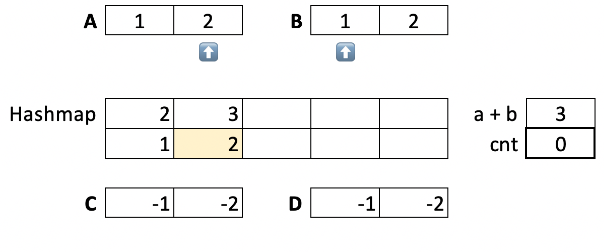
A brute force solution will be to enumerate all combinations of elements using four nested loops, which results in (n^4) time complexity. A faster approach is to use three nested loops, and, for each sum a + b + c, search for a complementary value d == -(a + b + c) in the fourth array. We can do the search in O(1) if we populate the fourth array into a hashmap.

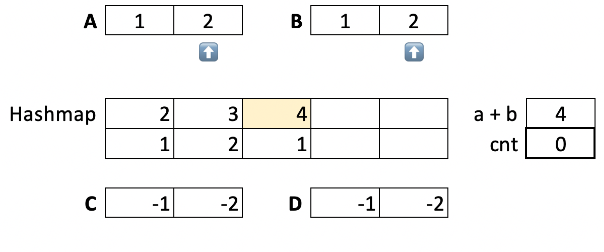
Note that we need to track the frequency of each element in the fourth array. If an element is repeated multiple times, it will form multiple quadruples. Therefore, we will use hashmap values to store counts.

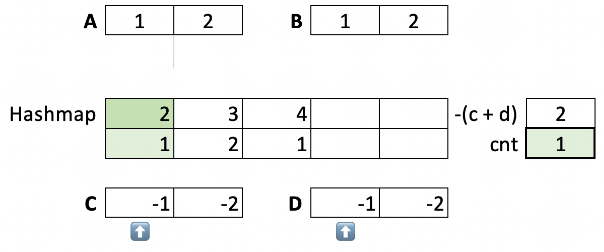
Building further on this idea, we can observe that a + b == -(c + d). First, we will count sums of elements a + b from the first two arrays using a hashmap. Then, we will enumerate elements from the third and fourth arrays, and search for a complementary sum a + b == -(c + d) in the hashmap.

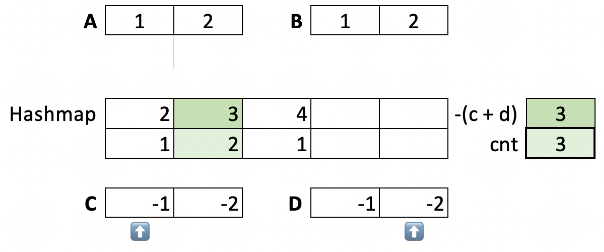


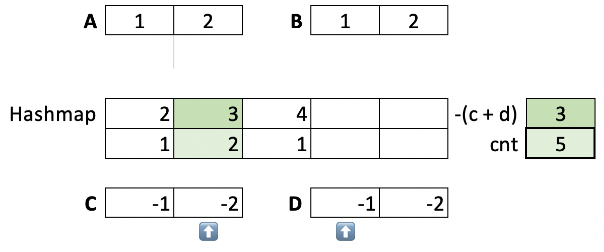


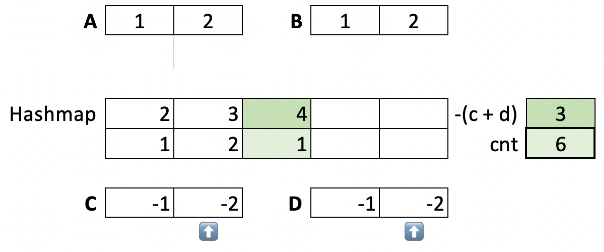












**Algorithm**

1. For each a in A.
   * For each b in B.
     + If a + b exists in the hashmap m, increment the value.
     + Else add a new key a + b with the value 1.
2. For each c in C.
   * For each d in D.
     + Lookup key -(c + d) in the hashmap m.
     + Add its value to the count cnt.
3. Return the count cnt.

|  |
| --- |
| public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {  int cnt = 0;  Map<Integer, Integer> m = new HashMap<>();  for (int a : A)  for (int b : B)  m.put(a + b, m.getOrDefault(a + b, 0) + 1);  for (int c : C)  for (int d : D)  cnt += m.getOrDefault(-(c + d), 0);  return cnt;  } |

**Complexity Analysis**

* Time Complexity: O(n^2). We have 2 nested loops to count sums, and another 2 nested loops to find complements.
* Space Complexity: O(n^2) for the hashmap. There could be up to O(n^2) distinct a + b keys.

#### **Approach 2: kSum II**

After you solve 4Sum II, an interviewer can follow-up with 5Sum II, 6Sum II, and so on. What they are really expecting is a generalized solution for k input arrays. Fortunately, the hashmap approach can be easily extended to handle more than 4 arrays.

Above, we divided 4 arrays into two equal groups, and processed each group independently. Same way, we will divide *k* arrays into two groups. For the first group, we will have k/2 ​ nested loops to count sums. Another k/2​ nested loops will enumerate arrays in the second group and search for complements.

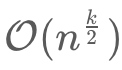
**Algorithm**

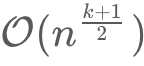
We can implement k/2 nested loops using a recursion, passing the index i of the current list as the parameter. The first group will be processed by addToHash recursive function, which accumulates sum and terminates when adding the final sum to a hashmap m.

The second function, countComplements, will process the other group, accumulating the complement value. In the end, it searches for the final complement value in the hashmap and adds its count to the result.

|  |
| --- |
| public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {  return kSumCount(new int[][]{A, B, C, D});  }  public int kSumCount(int[][] lists) {  Map<Integer, Integer> m = new HashMap<>();  addToHash(lists, m, 0, 0);  return countComplements(lists, m, lists.length / 2, 0);  }  void addToHash(int[][] lists, Map<Integer, Integer> m, int i, int sum) {  if (i == lists.length / 2)  m.put(sum, m.getOrDefault(sum, 0) + 1);  else  for (int a : lists[i])  addToHash(lists, m, i + 1, sum + a);  }  int countComplements(int[][] lists, Map<Integer, Integer> m, int i, int complement) {  if (i == lists.length)  return m.getOrDefault(complement, 0);  int cnt = 0;  for (int a : lists[i])  cnt += countComplements(lists, m, i + 1, complement - a);  return cnt;  } |

**Complexity Analysis**

* Time Complexity:  or O(n^2) for 4Sum II. We have k/2​ nested loops to count sums, and another k/2 nested loops to find complements.

If the number of arrays is odd, the time complexity will be . We will pass k/2 arrays to addToHash, and (k+1)/2 arrays to kSumCount to keep the space complexity  .

* Space Complexity:  for the hashmap. The space needed for the recursion will not exceed k/2​.

#### **Further Thoughts**

For an interview, keep in mind the generalized implementation. Even if your interviewer is OK with a simpler code, you'll get some extra points by describing how your solution can handle more than 4 arrays.

It's also important to discuss trade-offs with your interviewer. If we are tight on memory, we can move some arrays from the first group to the second. This, of course, will increase the time complexity.

In other words, the time complexity can range from  to  and the memory complexity ranges from  O(1) to   accordingly.

**Top K Frequent Elements**

Given a non-empty array of integers, return the ***k*** most frequent elements.

**Example 1:**

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

**Output:** [1,2]

**Example 2:**

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

**Output:** [1]

**Note:**

* You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
* Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.
* It's guaranteed that the answer is unique, in other words the set of the top k frequent elements is unique.
* You can return the answer in any order.

## Solution

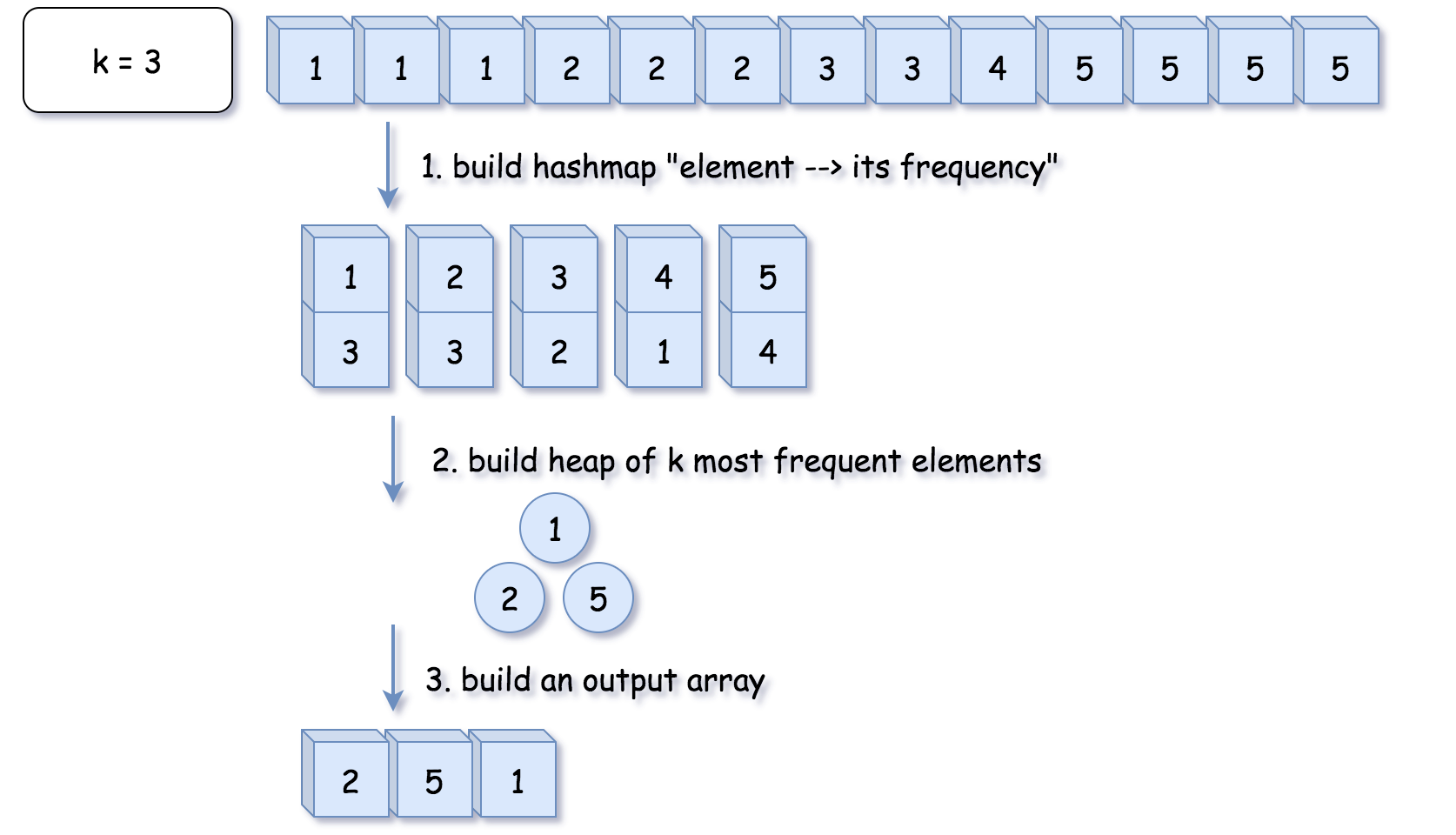
#### **Approach 1: Heap**

Let's start from the simple [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)) approach with O(*N*log*k*) time complexity. To ensure that O(*N*log*k*) is always less than O(*N*log*N*), the particular case *k*=*N* could be considered separately and solved in O(*N*) time.

**Algorithm**

* The first step is to build a hash map element -> its frequency. In Java, we use the data structure HashMap. Python provides dictionary subclass Counter to initialize the hash map we need directly from the input array.  
  This step takes O(*N*) time where N is a number of elements in the list.
* The second step is to build a heap of size k using N elements. To add the first k elements takes a linear time O(*k*) in the average case, and O(log1+log2+...+log*k*)=O(*logk*!)=O(*k*log*k*) in the worst case. It's equivalent to [heapify implementation in Python](https://hg.python.org/cpython/file/2.7/Lib/heapq.py#l16). After the first k elements we start to push and pop at each step, N - k steps in total. The time complexity of heap push/pop is O(log*k*) and we do it N - k times that means O((*N*−*k*)log*k*) time complexity. Adding both parts up, we get O(*N*log*k*) time complexity for the second step.
* The third and the last step is to convert the heap into an output array. That could be done in O(*k*log*k*) time.

In Python, library heapq provides a method nlargest, which [combines the last two steps under the hood](https://hg.python.org/cpython/file/2.7/Lib/heapq.py#l203) and has the same O(*N*log*k*) time complexity.



**Implementation**

|  |
| --- |
| class Solution {  public int[] topKFrequent(int[] nums, int k) {  // O(1) time  if (k == nums.length) {  return nums;  }    // 1. build hash map : character and how often it appears  // O(N) time  Map<Integer, Integer> count = new HashMap();  for (int n: nums) {  count.put(n, count.getOrDefault(n, 0) + 1);  }  // init heap 'the less frequent element first'  Queue<Integer> heap = new PriorityQueue<>(  (n1, n2) -> count.get(n1) - count.get(n2));  // 2. keep k top frequent elements in the heap  // O(N log k) < O(N log N) time  for (int n: count.keySet()) {  heap.add(n);  if (heap.size() > k) heap.poll();  }  // 3. build an output array  // O(k log k) time  int[] top = new int[k];  for(int i = k - 1; i >= 0; --i) {  top[i] = heap.poll();  }  return top;  }  } |

**Complexity Analysis**

* Time complexity : O(*N*log*k*) if *k*<*N* and O(*N*) in the particular case of *N*=*k*. That ensures time complexity to be better than O(*N*log*N*).
* Space complexity : O(*N*+*k*) to store the hash map with not more *N* elements and a heap with *k* elements.

#### **Approach 2: Quickselect**

**Hoare's selection algorithm**

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

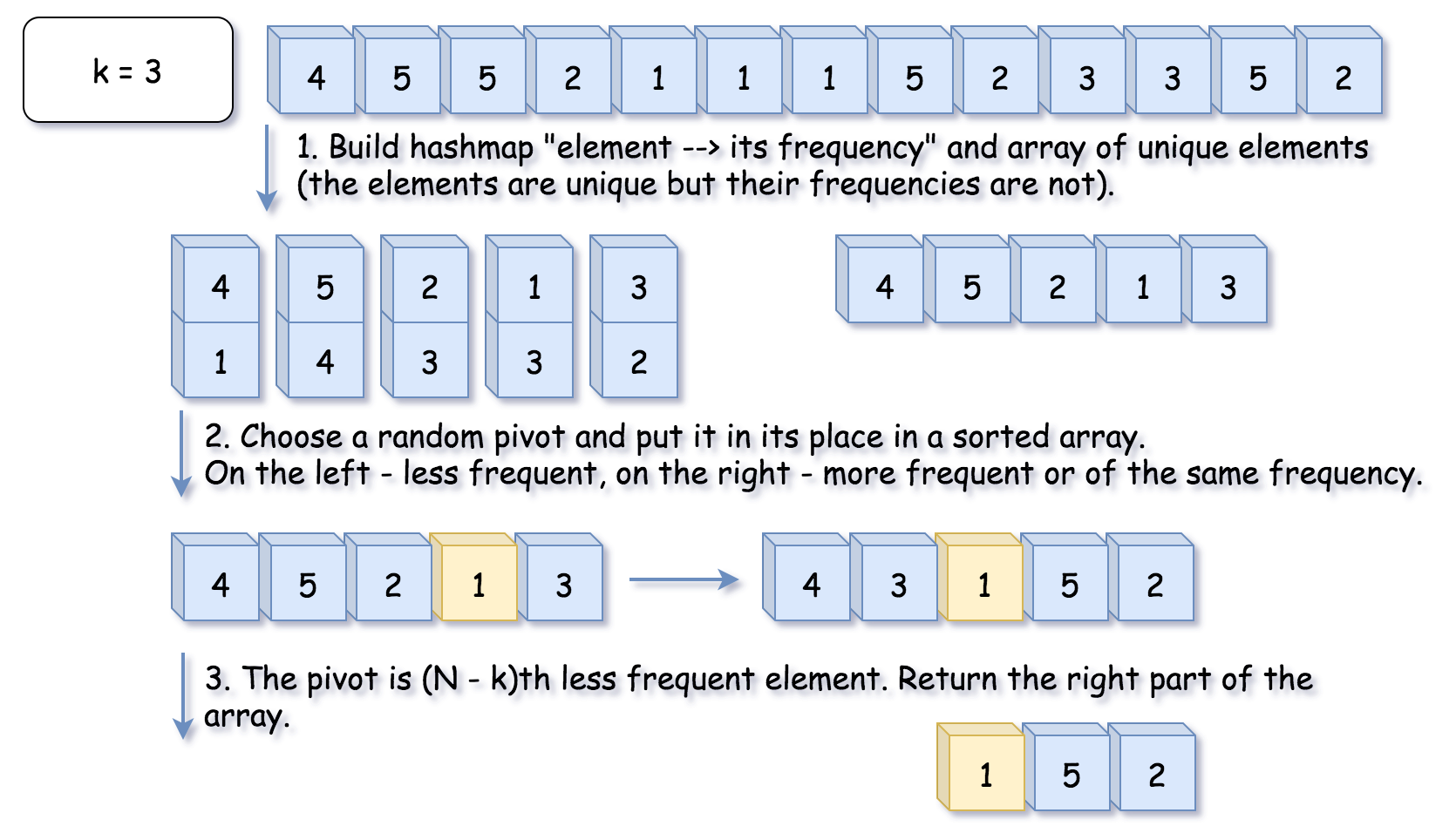
It has O(*N*) average time complexity and widely used in practice. It worth to note that its worth case time complexity is O(N^2), although the probability of this worst-case is negligible.

The approach is the same as for quicksort.

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

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

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

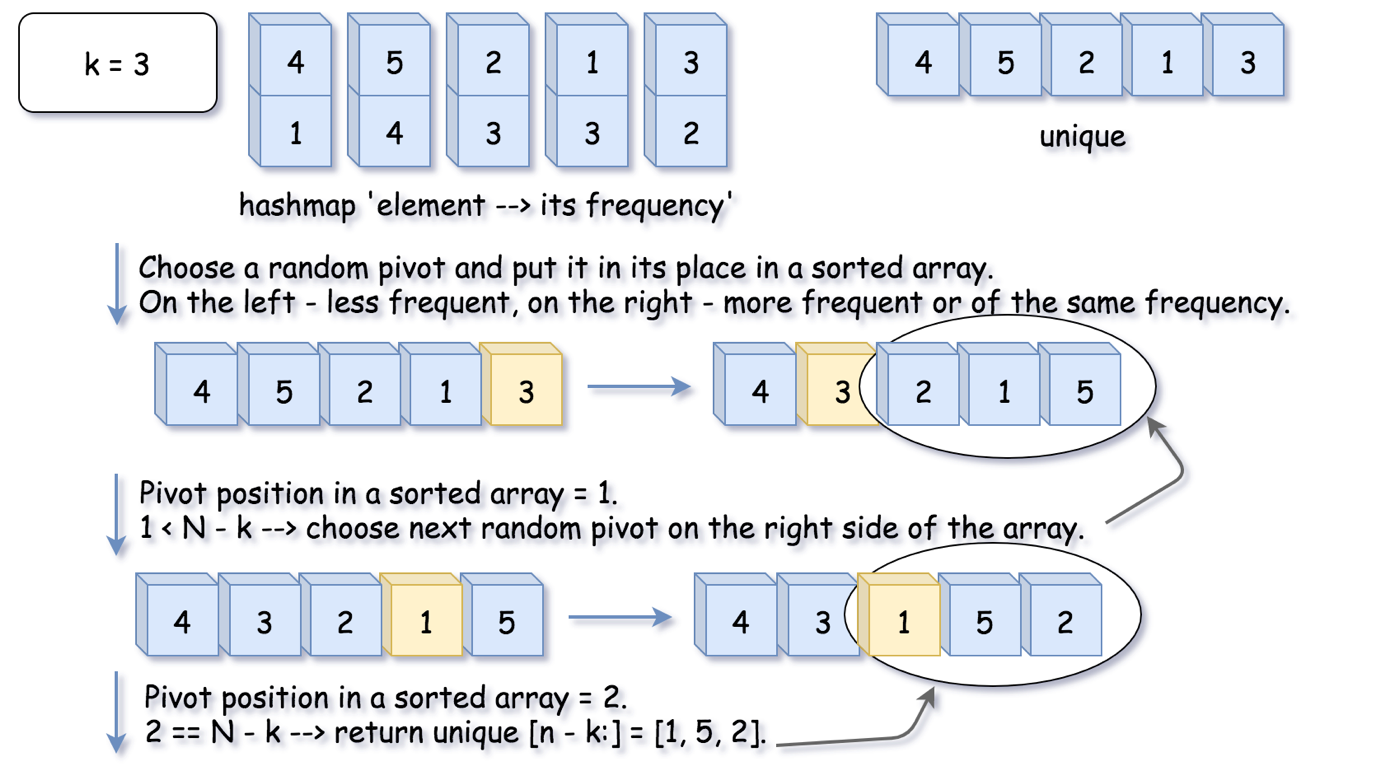


If that were a quicksort algorithm, one would have to process both parts of the array. That would result in O(*N*log*N*) time complexity. In this case, there is no need to deal with both parts since one knows in which part to search for N - kth less frequent element, and that reduces the average time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

**Algorithm**

The algorithm is quite straightforward :

* Build a hash map element -> its frequency and convert its keys into the array unique of unique elements. Note that elements are unique, but their frequencies are not. That means we need a partition algorithm that works fine with duplicates.
* Work with unique array. Use a partition scheme (please check the next section) to place the pivot into its perfect position pivot\_index in the sorted array, move less frequent elements to the left of pivot, and more frequent or of the same frequency - to the right.
* Compare pivot\_index and N - k.
  + If pivot\_index == N - k, the pivot is N - kth most frequent element, and all elements on the right are more frequent or of the same frequency. Return these top k*k* frequent elements.
  + Otherwise, choose the side of the array to proceed recursively.



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

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

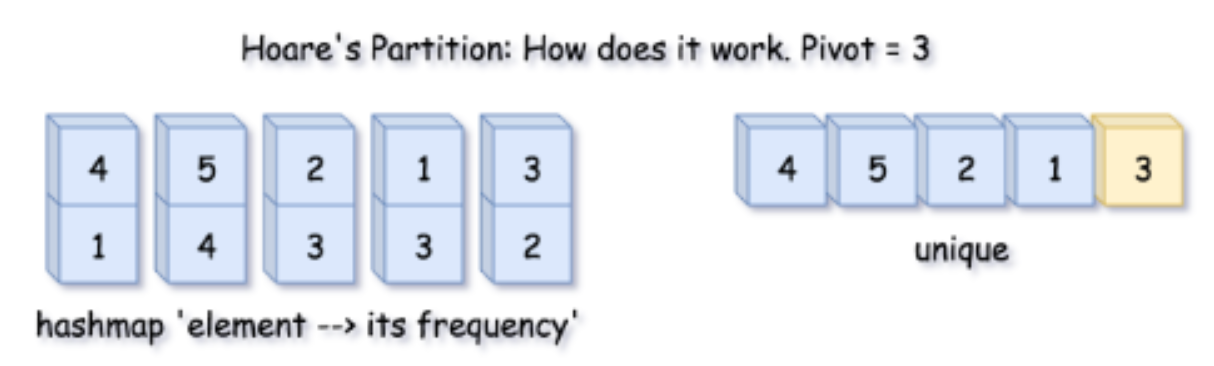
The drawback of Lomuto's partition is it fails with duplicates.

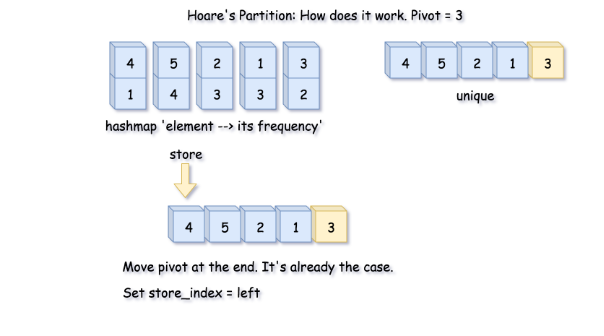
Here we work with an array of unique elements, but they are compared by frequencies, which are not unique. That's why we choose Hoare's Partition here.

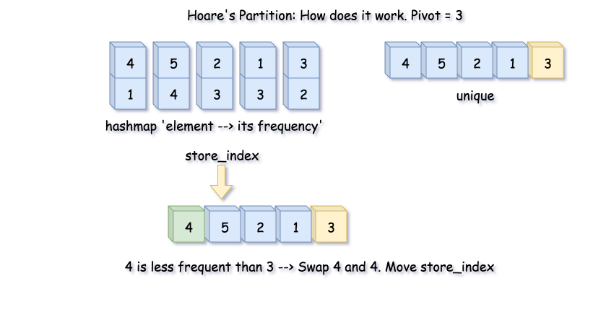
Hoare's partition is more efficient than Lomuto's partition because it does three times fewer swaps on average, and creates efficient partitions even when all values are equal.

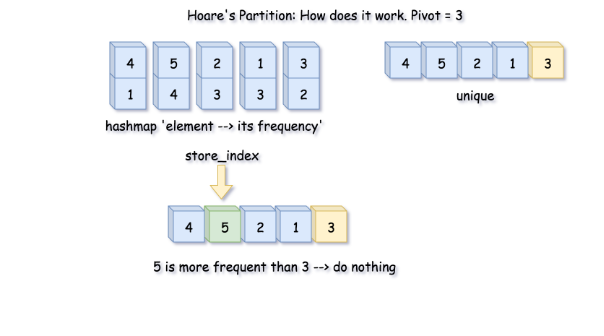
Here is how it works:

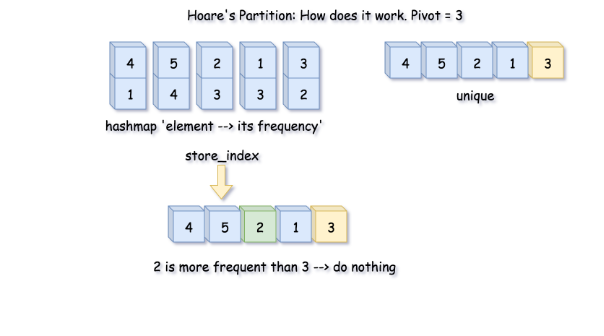
* Move pivot at the end of the array using swap.
* Set the pointer at the beginning of the array store\_index = left.
* Iterate over the array and move all less frequent elements to the left swap(store\_index, i). Move store\_index one step to the right after each swap.
* Move the pivot to its final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

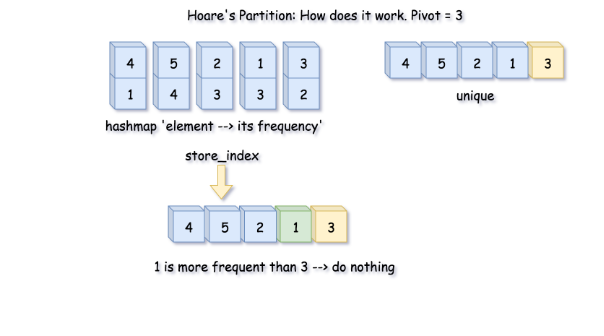


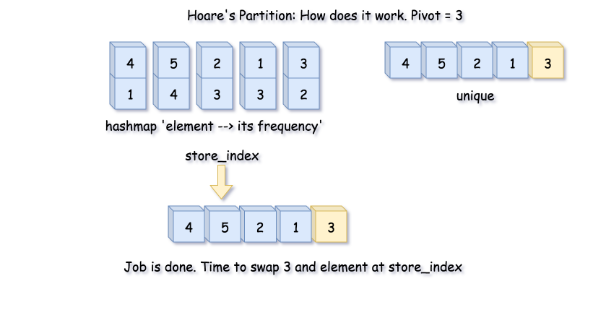


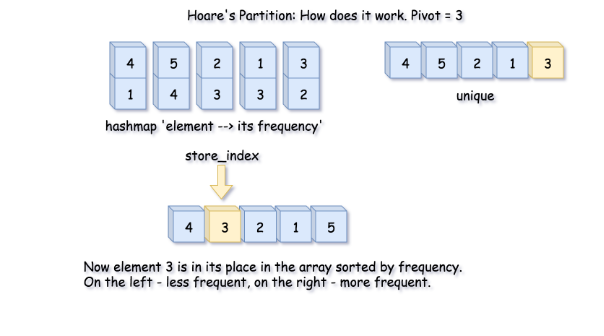


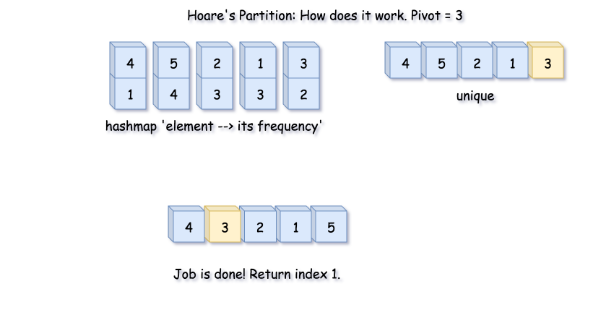












|  |
| --- |
| public int partition(int left, int right, int pivot\_index) {  int pivot\_frequency = count.get(unique[pivot\_index]);  // 1. move pivot to end  swap(pivot\_index, right);  int store\_index = left;  // 2. move all less frequent elements to the left  for (int i = left; i <= right; i++) {  if (count.get(unique[i]) < pivot\_frequency) {  swap(store\_index, i);  store\_index++;  }  }  // 3. move pivot to its final place  swap(store\_index, right);  return store\_index;  } |

**Implementation**

Here is a total algorithm implementation.

|  |
| --- |
| class Solution {  int[] unique;  Map<Integer, Integer> count;  public void swap(int a, int b) {  int tmp = unique[a];  unique[a] = unique[b];  unique[b] = tmp;  }  public int partition(int left, int right, int pivot\_index) {  int pivot\_frequency = count.get(unique[pivot\_index]);  // 1. move pivot to end  swap(pivot\_index, right);  int store\_index = left;  // 2. move all less frequent elements to the left  for (int i = left; i <= right; i++) {  if (count.get(unique[i]) < pivot\_frequency) {  swap(store\_index, i);  store\_index++;  }  }  // 3. move pivot to its final place  swap(store\_index, right);  return store\_index;  }    public void quickselect(int left, int right, int k\_smallest) {  /\*  Sort a list within left..right till kth less frequent element  takes its place.  \*/  // base case: the list contains only one element  if (left == right) return;    // select a random pivot\_index  Random random\_num = new Random();  int pivot\_index = left + random\_num.nextInt(right - left);  // find the pivot position in a sorted list  pivot\_index = partition(left, right, pivot\_index);  // if the pivot is in its final sorted position  if (k\_smallest == pivot\_index) {  return;  } else if (k\_smallest < pivot\_index) {  // go left  quickselect(left, pivot\_index - 1, k\_smallest);  } else {  // go right  quickselect(pivot\_index + 1, right, k\_smallest);  }  }    public int[] topKFrequent(int[] nums, int k) {  // build hash map : character and how often it appears  count = new HashMap();  for (int num: nums) {  count.put(num, count.getOrDefault(num, 0) + 1);  }    // array of unique elements  int n = count.size();  unique = new int[n];  int i = 0;  for (int num: count.keySet()) {  unique[i] = num;  i++;  }    // kth top frequent element is (n - k)th less frequent.  // Do a partial sort: from less frequent to the most frequent, till  // (n - k)th less frequent element takes its place (n - k) in a sorted array.  // All element on the left are less frequent.  // All the elements on the right are more frequent.  quickselect(0, n - 1, n - k);  // Return top k frequent elements  return Arrays.copyOfRange(unique, n - k, n);  }  } |

**Complexity Analysis**

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

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

* Space complexity: up to O(*N*) to store hash map and array of unique elements.

#### **Further Discussion: Could We Do Worst-Case Linear Time?**

In theory, we could, the algorithm is called [Median of Medians](https://en.wikipedia.org/wiki/Median_of_medians).

This method is never used in practice because of two drawbacks:

* It's outperformer. Yes, it works in a linear time *αN*, but the constant *α* is so large that in practice it often works even slower than N^2.
* It doesn't work with duplicates.

**Unique Word Abbreviation**

The **abbreviation** of a word is a concatenation of its first letter, the number of characters between the first and last letter, and its last letter. If a word has only two characters, then it is an **abbreviation** of itself.

For example:

* dog --> d1g because there is one letter between the first letter 'd' and the last letter 'g'.
* internationalization --> i18n because there are 18 letters between the first letter 'i' and the last letter 'n'.
* it --> it because any word with only two characters is an **abbreviation** of itself.

Implement the ValidWordAbbr class:

* ValidWordAbbr(String[] dictionary) Initializes the object with a dictionary of words.
* boolean isUnique(string word) Returns true if **either** of the following conditions are met (otherwise returns false):
  + There is no word in dictionary whose **abbreviation** is equal to word's **abbreviation**.
  + For any word in dictionary whose **abbreviation** is equal to word's **abbreviation**, that word and word are **the same**.

**Example 1:**

**Input**

["ValidWordAbbr", "isUnique", "isUnique", "isUnique", "isUnique"]

[[["deer", "door", "cake", "card"]], ["dear"], ["cart"], ["cane"], ["make"]]

**Output**

[null, false, true, false, true]

**Explanation**

ValidWordAbbr validWordAbbr = new ValidWordAbbr(["deer", "door", "cake", "card"]);

validWordAbbr.isUnique("dear"); // return false, dictionary word "deer" and word "dear" have the same abbreviation

  // "d2r" but are not the same.

validWordAbbr.isUnique("cart"); // return true, no words in the dictionary have the abbreviation "c2t".

validWordAbbr.isUnique("cane"); // return false, dictionary word "cake" and word "cane" have the same abbreviation

// "c2e" but are not the same.

validWordAbbr.isUnique("make"); // return true, no words in the dictionary have the abbreviation "m2e".

validWordAbbr.isUnique("cake"); // return true, because "cake" is already in the dictionary and no other word in the dictionary has "c2e" abbreviation.

**Constraints:**

* 1 <= dictionary.length <= 3 \* 104
* 1 <= dictionary[i].length <= 20
* dictionary[i] consists of lowercase English letters.
* 1 <= word.length <= 20
* word consists of lowercase English letters.
* At most 5000 calls will be made to isUnique.

## Summary

This problem has a low acceptance rate for a reason. The logic in isUnique can be a little tricky to get right due to the number of cases you need to consider. We highly recommend that you practice this similar but easier problem first - [Two Sum III - Data structure design](https://leetcode.com/problems/two-sum-iii-data-structure-design/).

## Solution

#### **Approach #1 (Brute Force)**

Let us begin by storing the dictionary first in the constructor. To determine if a word's abbreviation is unique with respect to a word in the dictionary, we check if all the following conditions are met:

1. They are not the same word.
2. They both have equal lengths.
3. They both share the same first and last letter.

Note that [Condition #1](https://leetcode.com/problems/unique-word-abbreviation/solution/#condition-1) is implicit because from the problem statement:

A word's abbreviation is unique if no ***other*** word from the dictionary has the same abbreviation.

public class ValidWordAbbr {

private final String[] dict;

public ValidWordAbbr(String[] dictionary) {

dict = dictionary;

}

public boolean isUnique(String word) {

int n = word.length();

for (String s : dict) {

if (word.equals(s)) {

continue;

}

int m = s.length();

if (m == n

&& s.charAt(0) == word.charAt(0)

&& s.charAt(m - 1) == word.charAt(n - 1)) {

return false;

}

}

return true;

}

}

**Complexity analysis**

* Time complexity : *O*(*n*) for each isUnique call. Assume that *n* is the number of words in the dictionary, each isUnique call takes *O*(*n*) time.

#### **Approach #2 (Hash Table) [Accepted]**

Note that isUnique is called repeatedly for the same set of words in the dictionary each time. We should pre-process the dictionary to speed it up.

Ideally, a hash table supports constant time look up. What should the key-value pair be?

Well, the idea is to group the words that fall under the same abbreviation together. For the value, we use a Set instead of a List to guarantee uniqueness.

The logic in isUnique(word) is tricky. You need to consider the following cases:

1. Does the word's abbreviation exists in the dictionary? If not, then it must be unique.
2. If above is yes, then it can only be unique if the grouping of the abbreviation contains no other words except word.

public class ValidWordAbbr {

private final Map<String, Set<String>> abbrDict = new HashMap<>();

public ValidWordAbbr(String[] dictionary) {

for (String s : dictionary) {

String abbr = toAbbr(s);

Set<String> words = abbrDict.containsKey(abbr)

? abbrDict.get(abbr) : new HashSet<>();

words.add(s);

abbrDict.put(abbr, words);

}

}

public boolean isUnique(String word) {

String abbr = toAbbr(word);

Set<String> words = abbrDict.get(abbr);

return words == null || (words.size() == 1 && words.contains(word));

}

private String toAbbr(String s) {

int n = s.length();

if (n <= 2) {

return s;

}

return s.charAt(0) + Integer.toString(n - 2) + s.charAt(n - 1);

}

}

#### **Approach #3 (Hash Table) [Accepted]**

Let us consider another approach using a counter as the table's value. For example, assume the dictionary = ["door", "deer"], we have the mapping of {"d2r" -> 2}. However, this mapping alone is not enough, because we need to consider whether the word exists in the dictionary. This can be easily overcome by inserting the entire dictionary into a set.

When an abbreviation's counter exceeds one, we know this abbreviation must not be unique because at least two different words share the same abbreviation. Therefore, we can further simplify the counter to just a boolean.

public class ValidWordAbbr {

private final Map<String, Boolean> abbrDict = new HashMap<>();

private final Set<String> dict;

public ValidWordAbbr(String[] dictionary) {

dict = new HashSet<>(Arrays.asList(dictionary));

for (String s : dict) {

String abbr = toAbbr(s);

abbrDict.put(abbr, !abbrDict.containsKey(abbr));

}

}

public boolean isUnique(String word) {

String abbr = toAbbr(word);

Boolean hasAbbr = abbrDict.get(abbr);

return hasAbbr == null || (hasAbbr && dict.contains(word));

}

private String toAbbr(String s) {

int n = s.length();

if (n <= 2) {

return s;

}

return s.charAt(0) + Integer.toString(n - 2) + s.charAt(n - 1);

}

}

**Complexity analysis**

* Time complexity : *O*(*n*) pre-processing, *O*(1) for each isUnique call. Both [Approach #2](https://leetcode.com/problems/unique-word-abbreviation/solution/#approach-2) and [Approach #3](https://leetcode.com/problems/unique-word-abbreviation/solution/#approach-3) above take *O*(*n*) pre-processing time in the constructor. This is totally worth it if isUnique is called repeatedly.
* Space complexity : *O*(*n*). We traded the extra *O*(*n*) space storing the table to reduce the time complexity in isUnique.

**Insert Delete GetRandom O(1)**

Implement the RandomizedSet class:

* bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
* bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
* int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

**Follow up:** Could you implement the functions of the class with each function works in **average** O(1) time?

**Example 1:**

**Input**

["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]

[[], [1], [2], [2], [], [1], [2], []]

**Output**

[null, true, false, true, 2, true, false, 2]

**Explanation**

RandomizedSet randomizedSet = new RandomizedSet();

randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomizedSet.remove(2); // Returns false as 2 does not exist in the set.

randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.

randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 2.

**Constraints:**

* -231 <= val <= 231 - 1
* At most 105 calls will be made to insert, remove, and getRandom.
* There will be **at least one** element in the data structure when getRandom is called.

## Solution

#### **Overview**

We're asked to implement the structure which provides the following operations in average O(1) time:

* Insert
* Delete
* GetRandom

First of all - why this weird combination? The structure looks quite theoretical, but it's widely used in popular statistical algorithms like [Markov chain Monte Carlo](https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo) and [Metropolis–Hastings algorithm](https://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm). These algorithms are for sampling from a probability distribution when it's difficult to compute the distribution itself.

Let's figure out how to implement such a structure. Starting from the Insert, we immediately have two good candidates with O(1) [average insert time](https://wiki.python.org/moin/TimeComplexity):

* Hashmap (or Hashset, the implementation is very similar): [Java HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) / [Python dictionary](https://docs.python.org/3/tutorial/datastructures.html#dictionaries)
* Array List: [Java ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) / [Python list](https://docs.python.org/3/tutorial/datastructures.html)

Let's consider them one by one.

Hashmap provides Insert and Delete in average constant time, although has problems with GetRandom.

The idea of GetRandom is to choose a random index and then to retrieve an element with that index. There is no indexes in hashmap, and hence to get true random value, one has first to convert hashmap keys in a list, that would take linear time. The solution here is to build a list of keys aside and to use this list to compute GetRandom in constant time.

Array List has indexes and could provide Insert and GetRandom in average constant time, though has problems with Delete.

To delete a value at arbitrary index takes linear time. The solution here is to always delete the last value:

* Swap the element to delete with the last one.
* Pop the last element out.

For that, one has to compute an index of each element in constant time, and hence needs a hashmap which stores element -> its index dictionary.

Both ways converge into the same combination of data structures:

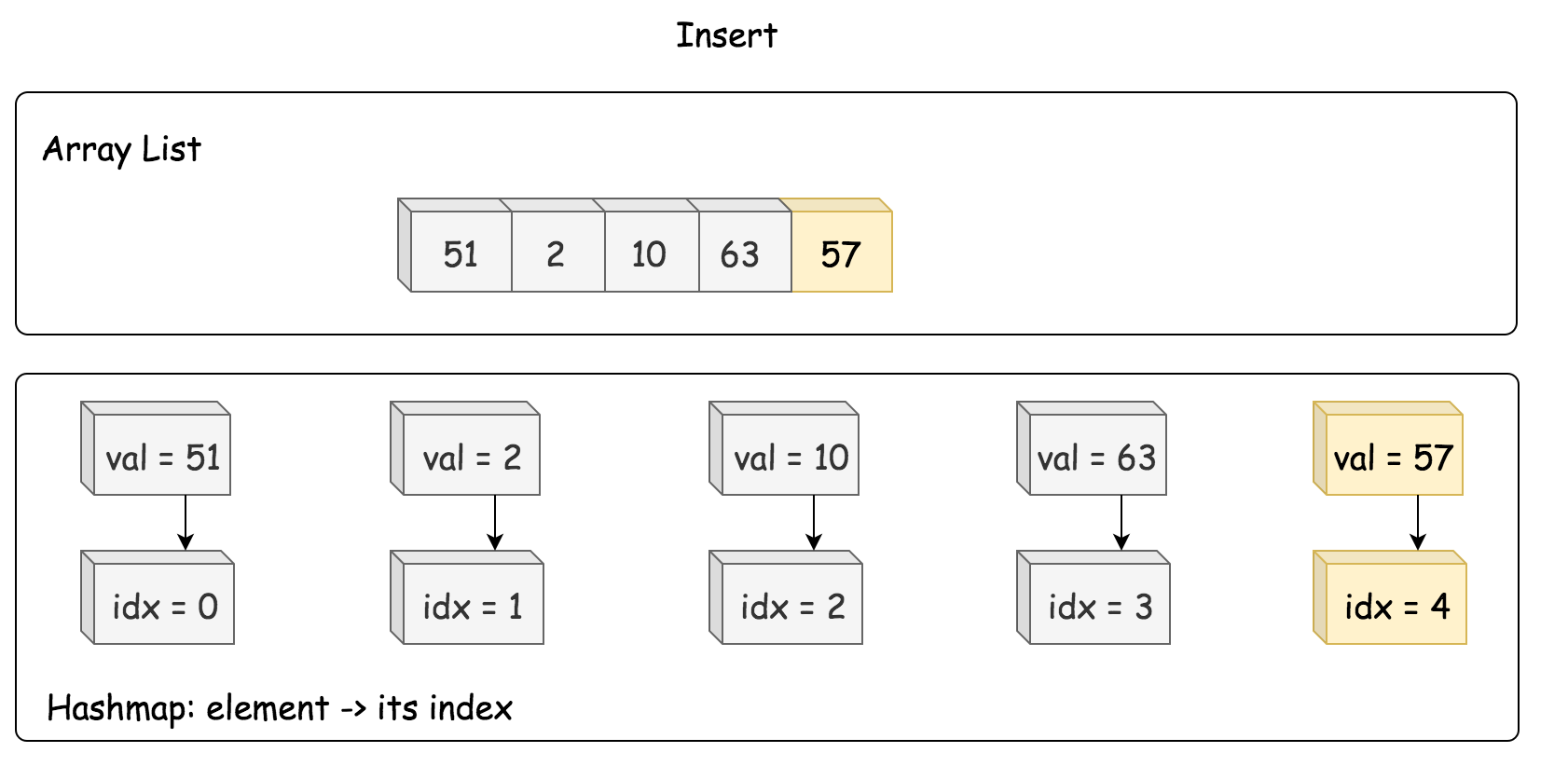
* Hashmap element -> its index.
* Array List of elements.



#### **Approach 1: HashMap + ArrayList**

**Insert**

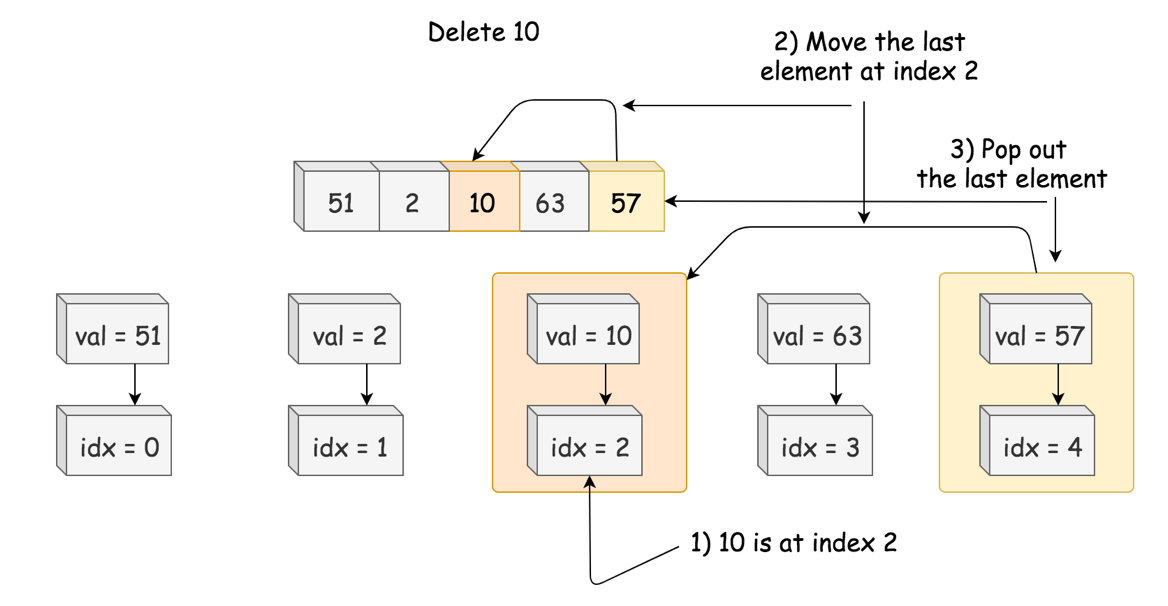
* Add value -> its index into dictionary, average O(1) time.
* Append value to array list, average O(1) time as well.



|  |
| --- |
| /\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/  public boolean insert(int val) {  if (dict.containsKey(val)) return false;    dict.put(val, list.size());  list.add(list.size(), val);  return true;  } |

**Delete**

* Retrieve an index of element to delete from the hashmap.
* Move the last element to the place of the element to delete, O(1) time.
* Pop the last element out, O(1) time.



|  |
| --- |
| /\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/  public boolean remove(int val) {  if (! dict.containsKey(val)) return false;  // move the last element to the place idx of the element to delete  int lastElement = list.get(list.size() - 1);  int idx = dict.get(val);  list.set(idx, lastElement);  dict.put(lastElement, idx);  // delete the last element  list.remove(list.size() - 1);  dict.remove(val);  return true;  } |

**GetRandom**

GetRandom could be implemented in O(1) time with the help of standard random.choice in Python and Random object in Java.

|  |
| --- |
| /\*\* Get a random element from the set. \*/  public int getRandom() {  return list.get(rand.nextInt(list.size()));  } |

**Implementation**:

|  |
| --- |
| class RandomizedSet {  Map<Integer, Integer> dict;  List<Integer> list;  Random rand = new Random();  /\*\* Initialize your data structure here. \*/  public RandomizedSet() {  dict = new HashMap();  list = new ArrayList();  }  /\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/  public boolean insert(int val) {  if (dict.containsKey(val)) return false;  dict.put(val, list.size());  list.add(list.size(), val);  return true;  }  /\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/  public boolean remove(int val) {  if (! dict.containsKey(val)) return false;  // move the last element to the place idx of the element to delete  int lastElement = list.get(list.size() - 1);  int idx = dict.get(val);  list.set(idx, lastElement);  dict.put(lastElement, idx);  // delete the last element  list.remove(list.size() - 1);  dict.remove(val);  return true;  }  /\*\* Get a random element from the set. \*/  public int getRandom() {  return list.get(rand.nextInt(list.size()));  }  } |

**Complexity Analysis**

* Time complexity. GetRandom is always O(1). Insert and Delete both have O(1) average time complexity, and O(*N*) in the worst-case scenario when the operation exceeds the capacity of currently allocated array/hashmap and invokes space reallocation.
* Space complexity: O(*N*), to store N elements.