1) ***Contains Duplicate***
   Given an array of integers, return true if any value appears at least twice and return false if every element is unique.
   Solve it in linear time.
2) ***Contains Duplicate Part 2***
   Given an integer array *nums* and an integer *k*, return true if there are two distinct indices *i and j* in the array such that nums[i] == nums[j] and abs(i - j) <= k.
   Solve it in linear time.
   Example 1:
   Input: nums = [1, 2, 3, 1], k = 3
   Output: true (3-0 <= 3)

   Example 2:
   Input: nums = [1, 2, 3, 1, 2, 3], k = 2
   Output: false
3) ***Longest substring with distinct characters***
   Given a string str, find the length of the longest substring of str with distinct characters only (without repeating characters).
   Solve it in linear time.

   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.

   Example 3:
   Input: s = "dvdf"
   Output: 3
   Explanation: The answer is "vdf", with the length of 3.

4) *Topmost frequent integers*

Given an integer array nums and an integer k, return the *k* most frequent integers.
Return the answer in any order.
It is guaranteed that the answer is unique.
Your algorithm's time complexity must be better than O(n log n), where n is the array's size (basically don't sort anything)

Example 1:
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

Example 2:
Input: nums = [1,2], k = 2
Output: [1,2]

5) *LRU Cache*

Design a cache that follows the Least Recently Used (LRU) principal.

Implement the LRUCache class of generic type <K, V>:

- LRUCache(int capacity)
  Initialize the LRU cache with positive size capacity.
- V get(T key)
  Return the value of the key if the key exists, otherwise return null.
- void put(K key, V value)
  Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.

The functions get and put must each run in O(1) average time complexity.