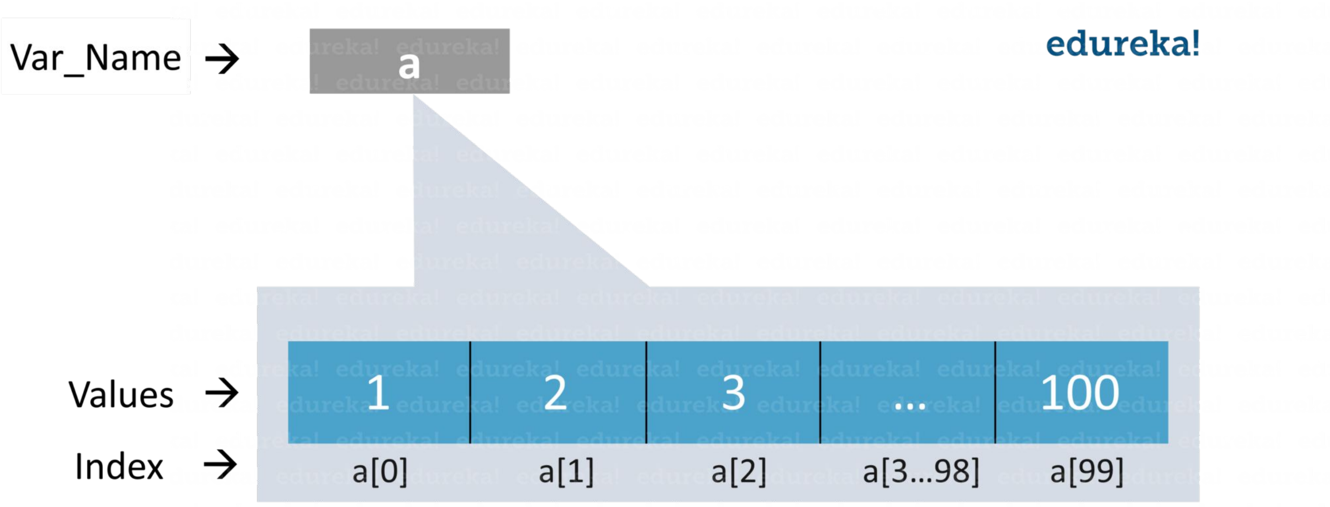
Arrays, Hashing and Binary Search

Arrays

Arrays

- Array: A (usually) contiguous list of values in memory

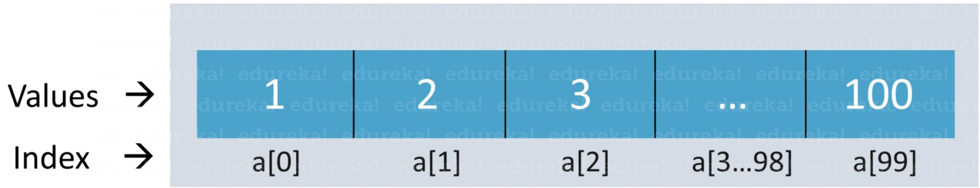


Arrays: Time complexity

- **Time complexity:** How long a program will take to run as a factor of its input size
- We can say an algorithm has a time complexity of ‘big O of x’ (written O(x)) when we refer to the upper limit of a program’s runtime
- In the context of arrays, the length of the array is the ‘n’ when we say an array algorithm is O(n) or O(n^2) time for example
- An array algorithm is O(n) time when we walk through the array once
- It is O(n^2) if we walk through the array once for each element

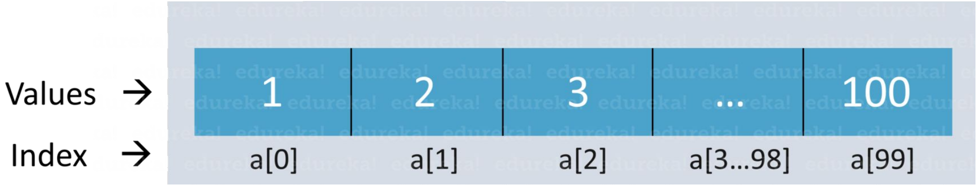
Arrays: Time complexity

- LeetCode usually times out on a $O(n^2)$ algorithm when the input size is over 10^5 units… $10^5 * 10^5 = 100{,}000 * 100{,}000 = 10$ billion operations! (10^4 -> 100 million)
- Here is a link if you want to learn more about finding time complexity: https://youtu.be/D6xkbGLQesk?si=93fEAKMS7Sbphlpd



Arrays: Time complexity

- Time complexity is relevant to us because it lets us know how time-efficient our algorithm is
- In many cases, LeetCode or a competition won't accept your algorithm because it takes too long to run
- In technical interviews, it is common to be asked how you can make your solution more efficient



Arrays: Time complexity

- Generally, $O(n^2)$ runtime is considered to be bad. So how do we avoid it?
- First, let's talk about when might be tempted to use $O(n^2)$ time with an example problem

Arrays: Time complexity

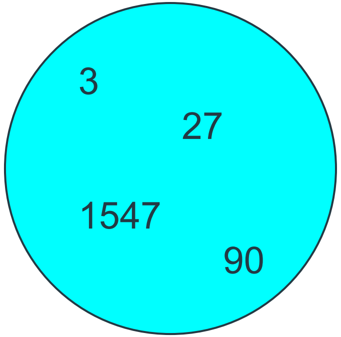
- **Contains Duplicate: Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.**
- How do we brute force this?
  - For each element go through the array to look for duplicates
- How do we speed that up?
  - Sort the array
  - Hash map

Hashing

Hashing

- A way to store values so they can be retrieved with **O(1)** time complexity
- Hashing stores values with no order, and they can't contain duplicates
- Hashing can be used for…

Sets:



Mappings:

hash tables, dictionaries

graduationYear

"Autumn"	→	2026
"Indie"	→	2025
"Sky"	→	2024

(mappings can contain **duplicate values**, but not **duplicate keys**)

Hashing

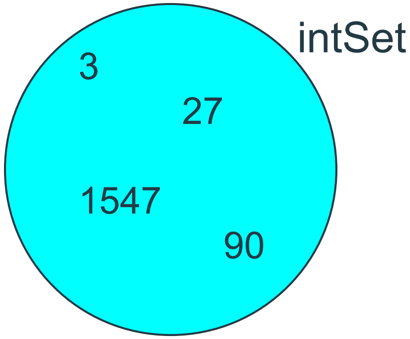
Java:

```
HashSet<Integer> intSet = new HashSet<Integer>();
intSet.put(3);
...
intSet.contains(3);
    -> true
```

Python:

```
intSet = set()
intSet.add(3)
...
27 in intSet
    -> True
```

Sets:



Hashing

Java:

```
HashTable<String,Integer> graduationYear = new HashTable<String,Integer>();
graduationYear.put("Autumn", 2026)
...
```

Python:

```
graduationYear = {}
graduationYear["Autumn"] = 2026
...
```

Mappings:

hash tables, dictionaries

graduationYear

"Autumn"	→	2026
"Indie"	→	2025
"Sky"	→	2024

Java:

```
graduationYear.get("Indie");
     -> 2025
graduationYear.getOrDefault("Bob", 0);
     -> 0
```

Python:

```
graduationYear["Autumn"]
     -> 2026
graduationYear.get("Bob", 0)
     -> 0
```

Binary search

Binary search overview

- Typically used to find the position of a “key” within a sorted array
- Takes O(log(n)) time
- Going through every element of an array takes O(n) time
- Works by repeatedly dividing the array in half and going to the half that contains the “key”

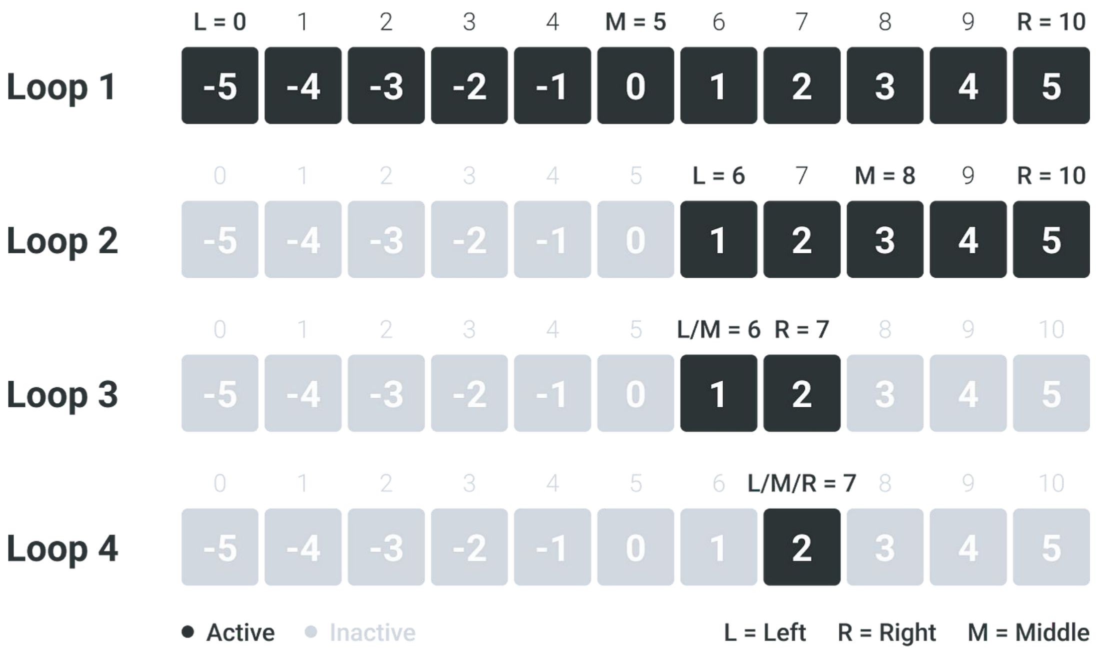
```
jshell> Arrays.binarySearch(new int[]{15, 20, 25, 30}, 25);
$3 ==> 2
```

Cool diagram

To figure out which portion of the array we're searching in, we use variables:

- L for the leftmost index,
- R for the rightmost index
- M for the middle of this portion

Searching for 2:



When to stop

Classic binary search stops when R > L, or when the middle element A[m] is the “key”

Implementing is hard

“Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky” -Donald Knuth

- Edge cases
- Exit conditions not defined correctly
- Overflow error: calculating M=(L+R)/2 can result in overflow (you *probably* won’t have to deal with this)
    - To avoid overflow you can do M = L+(R-L)/2

Code and diagram

```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor(L + (R − L) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
            return m
    return unsuccessful
```