Binary Search_1

@kbbhatt04

@June 19, 2023

Binary Search

- Approach
  - Optimal
    - Initialize `start = 0` and `end = n - 1` where `n = len(nums)`
    - Set `mid = (start + end) / 2`
    - Check if key == nums[mid], return mid
    - Else if `key < nums[mid]` then update `end = mid - 1`
    - Else `key > nums[mid]` then update `start = mid + 1`
    - Repeat the above steps till start ≤ end index
    - Time Complexity: $O(logn)$
    - Space Complexity: $O(1)$ for Iterative Solution and $O(logn)$ for Recursive Solution for auxiliary space

```
# Python3
# Optimal Solution
# Iterative Solution
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l = 0
        r = len(nums) - 1

        while (l <= r):
            m = (l + r) // 2
            if (nums[m] == target):
                return m
            elif (nums[m] < target):
                l = m + 1
            else:
                r = m - 1

        return -1
```

```
# Python3
# Optimal Solution
# Recursive Solution
class Solution:
    def binarySearch(self, nums, target, start, end):
        if start > end: return -1

        mid = (start + end) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            return self.binarySearch(nums, target, mid + 1, end)
        else:
            return self.binarySearch(nums, target, start, mid - 1)

    def search(self, nums: List[int], target: int) -> int:
        l = 0
        r = len(nums) - 1

        return self.binarySearch(nums, target, l, r)
```

```
// C++
// Optimal Solution
// Iterative Solution
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int l = 0;
```

```
        int r = nums.size() - 1;

        while (l <= r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] == target) {
                return mid;
            }
            else if (nums[mid] < target) {
                l = mid + 1;
            }
            else {
                r = mid - 1;
            }
        }
        return -1;
    }
};
```

```
// C++
// Optimal Solution
// Recursive Solution
int binarySearch(int arr[], int start, int end, int k) {

  if (start > end) {
    return -1;
  }
  int mid = (start + end) / 2;

  if (k == arr[mid]) {
    return mid;
  } else if (k < arr[mid]) {
    return binarySearch(arr, start, mid - 1, k);
  } else {
    return binarySearch(arr, mid + 1, end, k);
  }
}
```

Floor in a Sorted Array (Lower Bound)

Given a sorted array arr[] of size N without duplicates, and given a value x. The floor of x is defined as the largest element K in arr[] such that K is smaller than or equal to x. Find the index of K(0-based indexing).

- Approach
  - Brute-force
    - Linear Search
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$
  - Optimal
    - Binary Search
    - Time Complexity: $O(logn)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findFloor(self,A,N,X):
        if A[0] > X:    return -1
        i = 0
        while i < N and A[i] <= X:
            i += 1
        return i-1
```

```
# Python3
# Optimal Solution
class Solution:
    def findFloor(self,A,N,X):
        l = 0
        r = N - 1
        ans = -1
```

```
        while l <= r:
            mid = (l + r) // 2
            if A[mid] <= X:
                ans = mid
                l = mid + 1
            else:
                r = mid - 1
        return ans
```

```
// C++
// Optimal Solution
class Solution{
  public:
    int findFloor(vector<long long> v, long long n, long long x){
        long long l = 0;
        long long r = n - 1;
        long long ans = -1;
        while (l <= r) {
            long long mid = l + (r - l) / 2;
            if (v[mid] <= x) {
                ans = mid;
                l = mid + 1;
            }
            else {
                r = mid - 1;
            }
        }
        return ans;
    }
};
```

Search insert position in sorted array

- Approach
  - Brute-force
    - Linear Search
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$
  - Optimal
    - Binary Search
    - Time Complexity: $O(logn)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        for i in range(len(nums)):
            if nums[i] >= target:
                return i
        return i+1
```

```
# Python3
# Optimal Solution
class Solution:
    def searchInsert(self, nums, target):
        l , r = 0, len(nums)-1
        while l <= r:
            mid=(l+r)//2
            if nums[mid] >= target:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

```
// C++
// Optimal Solution
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int l = 0, r = nums.size() - 1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] >= target) {r = mid - 1;}
            else {l = mid + 1;}
        }
        return l;
    }
};
```

First and Last occurrence of a number in a sorted array

- Approach
    - Brute-force
        - Linear Search
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search
        - Time Complexity: $O(2 * logn)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        start = end = -1
        for i in range(len(nums)):
            if nums[i] == target:
                if start == -1:
                    start = i
                end = i
        return [start, end]
```

```
# Python3
# Optimal Solution
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        if len(nums) == 0:  return [-1, -1]

        start = end = -1
        l = 0
        r = len(nums) - 1
        ans = -1
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] >= target:
                ans = mid
                r = mid - 1
            else:
                l = mid + 1
        start = ans

        if nums[ans] != target: return [-1, -1]

        l = 0
        r = len(nums) - 1
        ans = -1
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] <= target:
                ans = mid
                l = mid + 1
            else:
                r = mid - 1
        end = ans
```

```
        return [start, end]
```

```
// C++
// Optimal Solution
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        return {BinarySearch(nums, target, "FIRST"), BinarySearch(nums, target, "LAST")};
    }

    int BinarySearch(vector<int> nums, int num, string find) {
        int left = 0, right = nums.size() - 1, mid;
        int result = -1;

        while (left <= right) {
            mid = (left + right) / 2;

            if (nums[mid] == num) {
                result = mid;
                (find == "FIRST") ? right = mid - 1 : left = mid + 1;
            }
            else if (nums[mid] > num) {
                right = mid - 1;
            }
            else {
                left = mid + 1;
            }
        }

        return result;
    }
};
```

Count Occurrences in Sorted Array

- Approach
    - Brute-force
        - Linear search
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search for first and last index of that element
        - Time Complexity: $O(2 * logn)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def count(self,arr, n, x):
        cnt = 0
        for i in arr:
            if i == x:
                cnt += 1
        return cnt
        # return arr.count(x)
```

```
# Python3
# Optimal Solution
class Solution:
    def binarySearch(self, arr, n, x, first_or_last_index):
        left = 0
        right = n - 1
        ans = -1

        while left <= right:
            mid = (left + right) // 2
            if arr[mid] == x:
                ans = mid
                if first_or_last_index == "FIRST":
```

```
                right = mid - 1
            else:
                left = mid + 1
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1

    return ans

def count(self,arr, n, x):
    start = self.binarySearch(arr, n, x, "FIRST")
    if arr[start] != x: return 0
    end = self.binarySearch(arr, n, x, "LAST")
    return end - start + 1
```

```
// C++
// Optimal Solution
class Solution{
public:
  int binarySearch(int nums[], int n, int num, string first_or_last_index) {
        int left = 0, right = n - 1, mid;
        int result = -1;

        while (left <= right) {
            mid = (left + right) / 2;

            if (nums[mid] == num) {
                result = mid;
                (first_or_last_index == "FIRST") ? right = mid - 1 : left = mid + 1;
            }
            else if (nums[mid] > num) {
                right = mid - 1;
            }
            else {
                left = mid + 1;
            }
        }

        return result;
    }
  int count(int arr[], int n, int x) {
      int start = binarySearch(arr, n, x, "FIRST");
      if (arr[start] != x) {return 0;}
      int end = binarySearch(arr, n, x, "LAST");
      return end - start + 1;
  }
};
```

Search in Rotated Sorted Array

- Approach
    - Brute-force
        - Linear Search
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Apply Binary Search
        - Check if `nums[mid] == target` then `return mid`
        - Else check if left half is sorted then check if target lies in left sorted half else search in right unsorted half
        - Else if right half is sorted then check if target lies in right sorted half else search in left unsorted half
        - Time Complexity: $O(logn)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        for i in range(len(nums)):
```

```
            if nums[i] == target:
                return i
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        n = len(nums)
        start = 0
        end = n - 1

        while start <= end:
            mid = (start + end) >> 1
            if nums[mid] == target:
                return mid

            # if left half is sorted
            if nums[start] <= nums[mid]:
                # if target lies between this sorted half
                if nums[start] <= target and target <= nums[mid]:
                    end = mid - 1

                # else move to other unsorted half
                else:
                    start = mid + 1

            # else right half is sorted
            else:
                # if target lies between this sorted half
                if nums[mid] <= target and target <= nums[end]:
                    start = mid + 1

                # else move to other unsorted half
                else:
                    end = mid - 1
        return -1
```

```
// C++
// Optimal Solution
int search(vector < int > & nums, int target) {
  int low = 0, high = nums.size() - 1;

  while (low <= high) {
    int mid = (low + high) >> 1;
    if (nums[mid] == target)
      return mid;

    if (nums[low] <= nums[mid]) {
      if (nums[low] <= target && nums[mid] >= target)
        high = mid - 1;
      else
        low = mid + 1;
    } else {
      if (nums[mid] <= target && target <= nums[high])
        low = mid + 1;
      else
        high = mid - 1;
    }
  }
  return -1;
}
```

Minimum in Rotated Sorted Array

- Approach
    - Brute-force
        - Linear Search
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Apply Binary Search

- At least one half will be sorted
- So if the left side is sorted, then the leftmost element is the smallest in that part and check on the right side
- If the right part is sorted then mid is going to be the smallest value for that part
- To check if the left part is sorted check nums[left] ≤ nums[mid] else right part is sorted
- Time Complexity: $O(logn)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findMin(self, nums: List[int]) -> int:
        return min(nums)
```

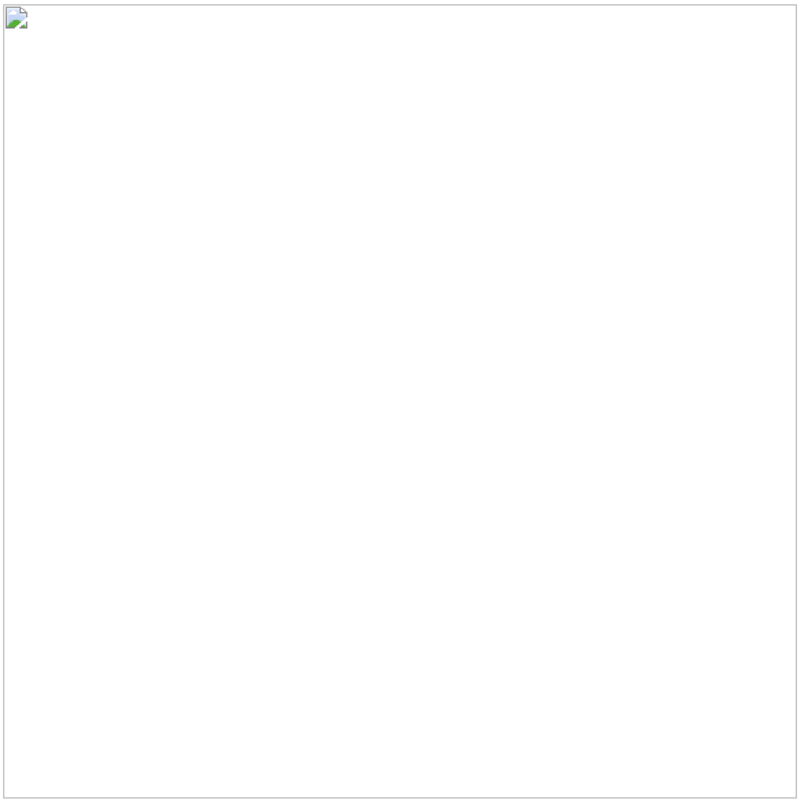
```
# Python3
# Optimal Solution
class Solution:
    def findMin(self, nums: List[int]) -> int:
        l = 0
        r = len(nums) - 1
        ans = nums[0]

        while l <= r:
            mid = (l + r) // 2

            # if search space is sorted
            # nums[l] will be min element in that search space
            if nums[l] <= nums[r]:
                return min(ans, nums[l])

            # if left half is sorted
            # update ans to min element of left half
            # then search in right half
            if nums[l] <= nums[mid]:
                ans = min(ans, nums[l])
                l = mid + 1

            # else if right half is sorted
            # update ans to min element of right half
            # then search in left half
            else:
                ans = min(ans, nums[mid])
                r = mid - 1
        return ans
```

```
// C++
// Optimal Solution
class Solution {
public:
    int findMin(vector<int>& nums) {
        int l = 0;
        int r = nums.size() - 1;
        int ans = nums[0];

        while (l <= r) {
            int mid = l + (r - l) / 2;

            // if search space is sorted
            // nums[l] will be min element in that search space
            if (nums[l] <= nums[r]) {
                return min(ans, nums[l]);
            }

            //  if left half is sorted
            //  update ans to min element of left half
            // then search in right half
            if (nums[l] <= nums[mid]) {
                ans = min(ans, nums[l]);
                l = mid + 1;
            }

            // else if right half is sorted
            // update ans to min element of right half
            // then search in left half
            else {
                ans = min(ans, nums[mid]);
```

```
                r = mid - 1;
            }
        }
        return ans;
    }
};
```

Search in rotated sorted array with duplicates

- Approach
    - Brute-force
        - Linear Search
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Apply Binary Search
        - Check if `nums[mid] == target` then `return mid`
        - Also check for edge case where if nums[low] == nums[mid] == nums[high] then increment low by 1 and decrement high by 1



        - Else check if left half is sorted then check if target lies in left sorted half else search in right unsorted half
        - Else if right half is sorted then check if target lies in right sorted half else search in left unsorted half
        - Time Complexity: $O(logn)$ for Average Case and $O(n/2)$ for Worst Case (when all elements in the array are same)
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        return (target in nums)
```

```
# Python3
# Optimal Solution
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        n = len(nums)
        start = 0
        end = n - 1

        while start <= end:
            mid = (start + end) >> 1
            if nums[mid] == target:
                return True

            # Edge case:
            if nums[start] == nums[mid] and nums[mid] == nums[end]:
                start += 1
                end -= 1
                continue

            # if left half is sorted
            if nums[start] <= nums[mid]:
                # if target lies between this sorted half
                if nums[start] <= target and target <= nums[mid]:
                    end = mid - 1

                # else move to other unsorted half
                else:
                    start = mid + 1

            # else right half is sorted
            else:
                # if target lies between this sorted half
                if nums[mid] <= target and target <= nums[end]:
                    start = mid + 1

                # else move to other unsorted half
                else:
                    end = mid - 1
        return False
```

```
// C++
// Optimal Solution
bool searchInARotatedSortedArrayII(vector<int>&arr, int k) {
    int n = arr.size(); // size of the array.
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;

        // if mid points the target
        if (arr[mid] == k) return true;

        // Edge case
        if (arr[low] == arr[mid] && arr[mid] == arr[high]) {
            low = low + 1;
            high = high - 1;
            continue;
        }

        // if left part is sorted
        if (arr[low] <= arr[mid]) {
            if (arr[low] <= k && k <= arr[mid]) {
                // element exists
                high = mid - 1;
            }
            else {
                // element does not exist
                low = mid + 1;
            }
        }
        else { // if right part is sorted
            if (arr[mid] <= k && k <= arr[high]) {
                // element exists
                low = mid + 1;
            }
            else {
                // element does not exist
                high = mid - 1;
            }
        }
    }
```

```
    return false;
}
```

Find the number of times the array is rotated

- Approach
    - Brute-force
        - Linear Search minimum element and its index
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search minimum element and its index
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findKRotation(self, arr, n):
        min_element = arr[0]
        index = 0
        for i in range(n):
            if arr[i] < min_element:
                min_element = arr[i]
                index = i
        return index
```

```
# Python3
# Optimal Solution
class Solution:
    def findKRotation(self, arr, n):
        l = 0
        r = n - 1
        min_element = arr[0]
        index = 0

        while l <= r:
            mid = (l + r) // 2

            if arr[l] <= arr[r]:
                if arr[l] < min_element:
                    min_element = arr[l]
                    index = l
                return index

            if arr[l] <= arr[mid]:
                if arr[l] < min_element:
                    min_element = arr[l]
                    index = l
                l = mid + 1
            else:
                if arr[mid] < min_element:
                    min_element = arr[mid]
                    index = mid
                r = mid - 1
        return index
```

```
// C++
// Optimal Solution
class Solution{
public:
  int findKRotation(int arr[], int n) {
      int l = 0, r = n - 1, min_element = arr[0], index = 0;

      while (l <= r) {
          int mid = l + (r - l) / 2;

          if (arr[l] <= arr[r]) {
              if (arr[l] < min_element) {
                  min_element = arr[l];
```

```
                index = l;
            }
            return index;
        }

        if (arr[l] <= arr[mid]) {
            if (arr[l] < min_element) {
                min_element = arr[l];
                index = l;
            }
            l = mid + 1;
        }
        else {
            if (arr[mid] < min_element) {
                min_element = arr[mid];
                index = mid;
            }
            r = mid - 1;
        }
    }
    return index;
  }
};
```

Single element in a Sorted Array

- Approach
  - Brute-force
    - Compare each one with its next adjacent element
    - If the next element is not equal, we know that the current element has occurred only once and thus return it as answer
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$
  - Optimal
    - We can observe that for every element that appears twice will be at even and odd position until a single element is encountered
    - After that it is reversed and first occurrence will be at odd position and second occurrence will be at even position
    - Thus apply binary search and check if mid element is unique i.e. `nums[mid-1] ≠ nums[mid] ≠ nums[mid+1]` then return nums[mid]
    - Else if mid is even then check if nums[mid] == nums[mid+1] or if mid is odd then check if nums[mid-1] == nums[mid] then the unique number cannot be in this half and search in right half
    - Else repeat searching in this half
    - Time Complexity: $O(logn)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def singleNonDuplicate(self, nums):
        for i in range(0, len(nums)-1, 2):
            if nums[i] != nums[i+1]:
                return nums[i]
        return nums[-1]
```

```
# Python3
# Optimal Solution
class Solution:
    def singleNonDuplicate(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            if (mid > 0 and mid % 2 == 1 and nums[mid - 1] == nums[mid]) or (mid < len(nums)-1 and mid%2 == 0 and nums[mid] == nums[mi
                left = mid + 1
            else:
```

```
            right = mid - 1
        return nums[left]
```

```
// C++
// Optimal Solution
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        int n = nums.size();

        if (n == 1) {return nums[0];}
        if (nums[0] != nums[1]) {return nums[0];}
        if (nums[n - 1] != nums[n - 2]) {return nums[n - 1];}

        int l = 1, r = n - 2;
        while (l <= r) {
            int mid = l + (r - l) / 2;

            if (nums[mid - 1] != nums[mid] && nums[mid] != nums[mid + 1]) {
                return nums[mid];
            }
            if ((mid % 2 == 1 && nums[mid - 1] == nums[mid]) || (mid % 2 == 0 && nums[mid] == nums[mid + 1])) {
                l = mid + 1;
            }
            else {
                r = mid - 1;
            }
        }
        return nums[l];
    }
};
```

Find a peak element in array

- Approach
  - Brute-force
    - Linear Search
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$
  - Optimal
    - There are multiple sorted parts in the array
    - Thus we can apply binary search
    - Check if mid is the peak else search in the direction where the sequence is increasing i.e. search on left side if nums[mid-1] > nums[mid] else search on right side
    - Time Complexity: $O(logn)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 1:  return 0
        if nums[0] > nums[1]:   return 0
        if nums[-1] > nums[-2]: return n - 1

        for i in range(1, n - 1):
            if nums[i] > nums[i - 1] and nums[i] > nums[i + 1]:
                return i
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 1:  return 0
```

```
        if nums[0] > nums[1]:   return 0
        if nums[-1] > nums[-2]: return n - 1

        l = 1
        r = n - 2
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] > nums[mid - 1] and nums[mid] > nums[mid + 1]:
                return mid

            if nums[mid] < nums[mid + 1]:
                l = mid + 1
            else:
                r = mid - 1

        return -1
```

```
// C++
// Optimal Solution
int peakEleOptimal(int arr[], int n) {
  int start = 0, end = n - 1;

  while (start < end) {
    int mid = (start + end) / 2;

    if (mid == 0)
      return arr[0] >= arr[1] ? arr[0] : arr[1];

    if (mid == n - 1)
      return arr[n - 1] >= arr[n - 2] ? arr[n - 1] : arr[n - 2];

    // Cheking whether peak element is in mid position
    if (arr[mid] >= arr[mid - 1] && arr[mid] >= arr[mid + 1])
      return arr[mid];

    // If left element is greater then ignore 2nd half of the elements
    if (arr[mid] < arr[mid - 1])
      end = mid - 1;

    // Else ignore first half of the elements
    else
      start = mid + 1;
  }

  return arr[start];
}
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```python
# Python3
# Better Solution
```

```python
# Python3
# Optimal Solution
```

```cpp
// C++
// Optimal Solution
```