**Binary Search**

Table of Contents

[**Theory** 2](#_Toc170546621)

[What is Tree data structure? 2](#_Toc170546622)

[What are components of tree? 2](#_Toc170546623)

[Binary Tree 3](#_Toc170546624)

[Tree traversal 3](#_Toc170546625)

[Binary Search Tree 4](#_Toc170546626)

[Use cases of Tree and its use in system design 4](#_Toc170546627)

[Array representation of Binary Tree 5](#_Toc170546628)

[**Important Codes** 6](#_Toc170546629)

[Binary Search Tree Implementation: 6](#_Toc170546630)

[Traversals: 7](#_Toc170546631)

[Breadth First : 8](#_Toc170546632)

[Morris Traversal: 9](#_Toc170546633)

[LEVEL 1: **EASY** 13](#_Toc170546634)

[1. Check if two binary trees are same or not. 13](#_Toc170546635)

[2. Range sum of bst 13](#_Toc170546636)

[3. Merge two binary trees. 13](#_Toc170546637)

[4. Convert a BST to tree with only right nodes. 13](#_Toc170546638)

[5. Average of levels in binary tree. 13](#_Toc170546639)

[6. Sorted array to BST 13](#_Toc170546640)

[7. Binary Tree tilt 13](#_Toc170546641)

[8. Diameter of binary tree 13](#_Toc170546642)

[9. Subtree of another tree 13](#_Toc170546643)

[LEVEL 2: **Medium** 14](#_Toc170546644)

[LEVEL 3: **Difficult** 15](#_Toc170546645)

[**SOLUTIONS:** 16](#_Toc170546646)

[**LEVEL 1:** 16](#_Toc170546647)

# **Theory**

## Definition and Concept

Binary Search is a highly efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

**Concept:**

1. Binary Search compares the target value to the middle element of the list.
2. If the target value is equal to the middle element, the search is successful.
3. If the target value is less than the middle element, the search continues in the left half of the list.
4. If the target value is greater than the middle element, the search continues in the right half of the list.
5. This process repeats, reducing the search interval by half each time.

**Example:** Imagine you have a sorted list of numbers: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] and you want to find the number 7.

1. Compare 7 with the middle element 11.
2. Since 7 is less than 11, search in the left half: [1, 3, 5, 7, 9].
3. Compare 7 with the middle element 5.
4. Since 7 is greater than 5, search in the right half: [7, 9].
5. Compare 7 with the middle element 7.
6. The element is found.

**Time Complexity: O (log (n))**

# **Important Codes**

## Binary Search:

def binary\_search(arr, target):

    left, right = 0, len(arr) - 1

    while left <= right:

        mid = left + (right - left) // 2

        if arr[mid] == target:

            return mid

        elif arr[mid] < target:

            left = mid + 1

        else:

            right = mid - 1

    return -1

arr = [1, 2, 3, 4, 5]

target = 4

print(binary\_search(arr, target))  # Output: 3

## Lower and Upper bound:

The lower bound of an element in a sorted array is the first position where the element can be inserted without violating the order of the array. In other words, it is the index of the first element that is not less than the given value.

The upper bound of an element in a sorted array is the first position where an element greater than the given value can be inserted without violating the order of the array. It is the index of the first element that is greater than the given value.

**Example:** Consider the same sorted array [1, 2, 4, 4, 5, 6] and the target value 4.

* The lower bound of 4 is the first position where 4 can be inserted or the first occurrence of 4.
* In this array, the lower bound of 4 is at index 2.
* The upper bound of 4 is the first position where an element greater than 4 can be inserted.
* In this array, the upper bound of 4 is at index 4.

def lower\_bound(arr, target):

    left, right = 0, len(arr)

    while left < right:

        mid = left + (right - left) // 2

        if arr[mid] < target:

            left = mid + 1

        else:

            right = mid

    return left

def upper\_bound(arr, target):

    left, right = 0, len(arr)

    while left < right:

        mid = left + (right - left) // 2

        if arr[mid] <= target:

            left = mid + 1

        else:

            right = mid

    return left

# Example Usage

arr = [1, 2, 4, 4, 5, 6]

target = 4

print("Lower Bound of 4:", lower\_bound(arr, target))  # Output: 2

print("Upper Bound of 4:", upper\_bound(arr, target))  # Output: 4

# For target like 9(greater than largest element), left will become right+1 at end. So left becomes len(arr), and that will be upper and lower bound for us

Can do mid as (left+right)//2 also. But if left and right both are large value, left+right can go out of bound. Though this problem won’t happen in python.

# LEVEL 1: **EASY**

### Count negative number in a sorted matrix

Link: <https://leetcode.com/problems/count-negative-numbers-in-a-sorted-matrix/>

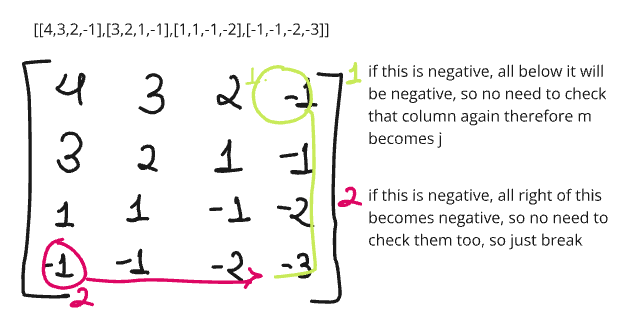
### Find Target Indices after Sorting Array

Link: <https://leetcode.com/problems/find-target-indices-after-sorting-array/>

# **SOLUTIONS:**

## **LEVEL 1:**

1. Count negative number in a sorted matrix



class Solution:

    def countNegatives(self, grid: List[List[int]]) -> int:

        n = len(grid)

        m = len(grid[0])

        ans=0

        for i in range(n):

            for j in range(m):

                if grid[i][j]<0:

                    ans+=(n-i)\*(m-j)

                    m=j

                    break

            if m == 0:

                break

        return ans

Time complexity: O(n+m)

2. Find Target Indices after Sorting Array

We can use lower bound for this. If target exists in nums, lower bound will give us it’s first occurrence. And if number does not exist, it will give index for which nums[index]<target, for this case we will get ans=[] since while condition will not run at all.

class Solution:

    def lower\_bound(self,arr, target):

        left, right = 0, len(arr)

        while left < right:

            mid = left + (right - left) // 2

            if arr[mid] < target:

                left = mid + 1

            else:

                right = mid

        return left

    def targetIndices(self, nums: List[int], target: int) -> List[int]:

        nums.sort()

        ans=[]

        n = len(nums)

        lb = self.lower\_bound(nums,target)

        while(lb<n and nums[lb]==target):

            ans.append(lb)

            lb+=1

        return ans