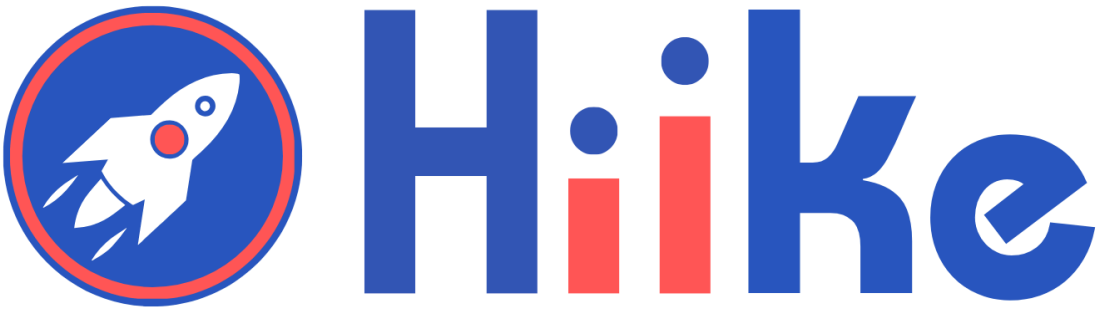
# Class 6

## Detailed Notes



# Problem: Square Root of Integer

Link: [Square Root of Integer]

Solution:

```
class Solution {
public:
    long long int floorSqrt(long long int x) {
        long long int high = 10000, low = 0, mid;
        while (high >= low) {
            mid = low + (high - low) / 2;
            if ((mid * mid) == x) {
                return mid;
            } else if ((mid * mid) < x) {
                if ((mid + 1) * (mid + 1) > x) {
                    return mid;
                } else {
                    low = mid + 1;
                }
            } else {
                high = mid - 1;
            }
        }
        return 0;
    }
};
```

Explanation:

The function uses binary search to find the floor of the square root of `x`. It sets initial bounds and narrows them down based on whether the square of the mid value is less than, greater than, or equal to `x`. If `mid^2` is less than `x` and `(mid+1)^2` is greater than `x`, then `mid` is the floor square root.

Why it works:

Binary search effectively narrows down the range, ensuring that the solution converges quickly to the correct square root or its floor.

Time Complexity: O(log x), because each step of the binary search cuts the search space in half.

Space Complexity: O(1), since the space used does not scale with the size of the input.

# Problem: First and Last Occurrences of X

Link: First and Last Occurrences of X

Solution:

```cpp
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int low = 0, high = nums.size() - 1, m;
        int lowest = -1, highest = -1;
        // First occurrence
        while (low <= high) {
            m = low + (high - low) / 2;
            if (nums[m] == target) {
                lowest = m;
                high = m - 1;
            } else if (nums[m] > target) {
                high = m - 1;
            } else {
                low = m + 1;
            }
        }
        // Last occurrence
        low = 0; high = nums.size() - 1;
        while (low <= high) {
            m = low + (high - low) / 2;
            if (nums[m] == target) {
                highest = m;
                low = m + 1;
            } else if (nums[m] > target) {
                high = m - 1;
            } else {
                low = m + 1;
            }
        }
        return {lowest, highest};
    }
};
```

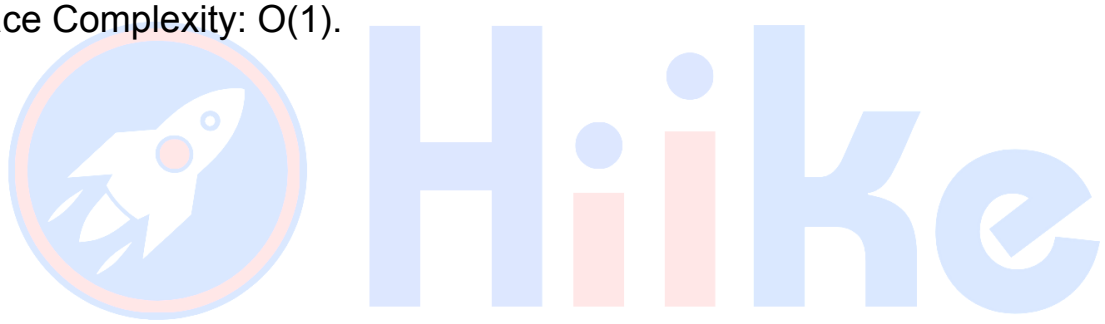
Explanation:

This function finds the first and last occurrences of `target` in a sorted array `nums`. It uses modified binary search twice: once to find the first occurrence (adjusting `high`) and once for the last occurrence (adjusting `low`).

Why it works:

Using binary search leverages the sorted nature of the array to quickly locate the target values, and adjusting the boundaries ensures that the search focuses on potential boundaries of the target's range.

Time Complexity: O(\log n), where n is the number of elements in `nums`.
Space Complexity: O(1).



# Problem: Find a Peak Element

Link: [Find a Peak Element]

Solution:

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int high = nums.size() - 1, low = 0, mid;
        if (nums.size() == 1) {
            return 0;
        }
        while (low <= high) {
            mid = low + (high - low) / 2;
            if (mid == 0) {
                if (nums[mid + 1] < nums[mid]) {
                    return mid;
                } else {
                    low = mid + 1;
                }
            } else if (mid == nums.size() - 1) {
                if

 (nums[mid - 1] < nums[mid]) {
                    return mid;
                } else {
                    high = mid - 1;
                }
            } else if (nums[mid + 1] < nums[mid] && nums[mid - 1] <
nums[mid]) {
                return mid;
            } else if (nums[mid + 1] > nums[mid]) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return 0;
    }};
```

Explanation:

The function identifies a peak element in an array where a peak is defined as an element greater than its neighbors. It uses binary search to efficiently find a peak, adjusting the search range based on the values of the element's immediate neighbors.

Why it works:

The binary search modification ensures that the search space is narrowed down to where a peak is guaranteed, based on the direction in which elements increase.

Time Complexity: O(log n), where n is the number of elements in the array.
Space Complexity: O(1).

# Problem: Search in a Matrix

Link: [Search in a Matrix]

Solution:

```cpp
class Solution {
public:
    bool search(vector<vector<int>> matrix, int n, int m, int x) {
        int row = matrix.size() - 1, col = 0;
        while (row >= 0 && col < matrix[0].size()) {
            if (x == matrix[row][col]) {
                return true;
            }
            if (x > matrix[row][col]) {
                col++;
            } else {
                row--;
            }
        }
        return false;
    }
};
```

Explanation:

This function searches for a number `x` in a row-wise and column-wise sorted matrix. It starts from the bottom-left corner of the matrix and moves either right (if `x` is larger) or up (if `x` is smaller), effectively narrowing down the potential location of `x`.

Why it works:

The choice of starting point and movement strategy leverages the sorted nature of the matrix, allowing the function to skip over rows and columns that cannot contain `x`.

Time Complexity: O(n + m), where n is the number of rows and m is the number of columns in the matrix.

Space Complexity: O(1), as the search process does not require additional space beyond the input matrix.

# Problem: Matrix Search

Link: [Matrix Search]

Solution:

```
int Solution::searchMatrix(vector<vector<int> > &A, int B) {
    int row = 0, col = A[0].size() - 1;
    while (row < A.size() && col >= 0) {
        if (A[row][col] > B) {
            col--;
        } else if (A[row][col] < B) {
            row++;
        } else {
            return 1;  // Element found
        }
    }
    return 0;  // Element not found
}
```

Explanation:

This function searches for a number `B` in a matrix `A` that is sorted in ascending order both row-wise and column-wise. The search starts from the top-right corner of the matrix. The strategy is to eliminate either a row or a column in each step:
- If the current element is greater than `B`, move left (decrement column index) because all elements in the current column below will be greater than the current element.
- If the current element is less than `B`, move down (increment row index) because all elements in the current row to the left are less than the current element.
- If the current element equals `B`, the search successfully finds the target and returns `1`.

Why it works:

This approach efficiently narrows down the search space by ruling out entire rows or columns with each comparison, exploiting the sorted order of the matrix. By starting at the top-right corner, the solution ensures that it can move in only two directions (left or down) to find the target, thus avoiding unnecessary traversals.

Time Complexity:
- Best Case: O(1), where the element is found at the initial position of the search.
- Worst Case: O(n + m), where `n` is the number of rows and `m` is the number of columns. This occurs when the search path traverses from the top-right corner to the bottom-left corner, effectively walking along the perimeter of the matrix.

Space Complexity: O(1), as the search is done in place without requiring additional space beyond the input parameters for matrix dimensions and the target value.