Day 40/180

Binary Search Hard Interview Problem 📑

Trapping Rain Water

```cpp
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        // Variable to store the total trapped water
        int water = 0;
        // Variables to track the maximum height, its index, and the
maximum heights encountered from the left and right
        int leftmax = 0, rightmax = 0, maxheight = height[0], index = 0;

        // Iterate to find the index of the maximum height building
        for (int i = 1; i < n; i++) {
            if (maxheight < height[i]) {
                maxheight = height[i];
                index = i;
            }
        }

        // Calculate trapped water for the left part
        for (int i = 0; i < index; i++) {
            if (leftmax > height[i])
                water += leftmax - height[i]; // Add trapped water
            else
                leftmax = height[i]; // Update leftmax if a higher
building is encountered
        }

        // Calculate trapped water for the right part
        for (int i = n - 1; i > index; i--) {
            if (rightmax > height[i])
                water += rightmax - height[i]; // Add trapped water
            else
                rightmax = height[i]; // Update rightmax if a higher
building is encountered
```

```
        }

        // Return the total trapped water
        return water;
    }
};
```

Code Explanation and Complexity

1. Finding Max Height Building:
    - Iterate through the height vector to find the maximum height building (maxheight) and its index (index).
2. Calculate Trapped Water - Left Part:
    - Iterate from the start of the height vector up to the index of the maximum height building.
    - Calculate the trapped water for each building in this left part.
    - Update leftmax as the maximum height encountered so far.
3. Calculate Trapped Water - Right Part:
    - Iterate from the end of the height vector down to the index of the maximum height building.
    - Calculate the trapped water for each building in this right part.
    - Update rightmax as the maximum height encountered so far.
4. Result:
    - Return the total trapped water.
5. Time Complexity:
    - O(n), where n is the size of the height vector.
    - The code iterates through the height vector twice (finding the maximum height and calculating trapped water).
6. Space Complexity:
    - O(1).

3 Sum

```
class Solution {
public:
    // Function to find if there exists a triplet in the
    // array A[] which sums up to X.
    bool find3Numbers(int A[], int n, int X) {
        // Sorting the array using Bubble Sort
        for (int i = n - 2; i >= 0; i--) {
            for (int j = 0; j <= i; j++) {
```

```
                if (A[j] > A[j + 1])
                    swap(A[j], A[j + 1]);
            }
        }

        // Iterate through the array to find the triplet
        for (int i = 0; i < n - 2; i++) {
            int ans = X - A[i];
            int start = i + 1;
            int end = n - 1;

            // Using two-pointer approach to find a pair with sum equal
to 'ans'
            while (start < end) {
                if (A[start] + A[end] == ans)
                    return true; // Triplet found
                else if (A[start] + A[end] > ans)
                    end--; // Reduce sum by moving the end pointer to a
smaller element
                else
                    start++; // Increase sum by moving the start pointer
to a larger element
            }
        }
        return false; // No triplet found
    }
};
```

Code Explanation and Complexity

1. Sorting using Bubble Sort:
    - The array A is sorted in non-decreasing order using the Bubble Sort algorithm.
    - The sorting is done in reverse order, starting from the second-to-last element, to the first element.
2. Finding Triplet:
    - Iterate through the array to find a triplet.
    - For each element at index i, calculate the target sum ans as X - A[i].
3. Two-Pointer Approach:
    - Initialize two pointers start and end to find a pair with the sum equal to ans.
    - Move the pointers based on the comparison of the sum of elements at start and end with the target sum ans.
        - If the sum is equal to ans, return true as a triplet is found.

- If the sum is greater than ans, decrease the sum by moving the end pointer to a smaller element.
- If the sum is less than ans, increase the sum by moving the start pointer to a larger element.

4. Result:
    - If a triplet is found, return true.
    - If no triplet is found, return false.
5. Time Complexity:
    - O(n^2) due to the sorting step (Bubble Sort - O(n^2)) and the two-pointer approach (O(n)).
    - The dominant factor is the sorting operation.
6. Space Complexity:
    - O(1).
    - The code uses only a constant amount of extra space, regardless of the input size.

4 Sum

```
bool find4Numbers(int A[], int n, int X)
{
    // Sorting the array in ascending order
    sort(A, A + n);

    // Iterate through the array for the first two elements of the
combination
    for (int i = 0; i < n - 3; i++)
    {
        // Iterate through the array for the third element of the
combination
        for (int j = i + 1; j < n - 2; j++)
        {
            // Initialize two pointers for the fourth element
            int start = j + 1;
            int end = n - 1;

            // Calculate the target value for the sum of four elements
            int target = X - A[i] - A[j];

            // Use two-pointer approach to find the fourth element
            while (start < end)
            {
                int sum = A[start] + A[end];
```

```
                // If the sum is equal to the target, a combination is
found
                if (sum == target)
                    return true;

                // If the sum is less than the target, move the start
pointer to a larger element
                if (sum < target)
                    start++;
                // If the sum is greater than the target, move the end
pointer to a smaller element
                else
                    end--;
            }
        }
    }
    // No combination found
    return false;
}
```

Code Explanation and Complexity

1. Sorting:
    - The array A is sorted in ascending order using the sort function.
2. Finding Combinations:
    - The function uses a nested loop to iterate through pairs of elements in the array (i and j).
3. Two-Pointer Approach:
    - For each pair of elements (i and j), the function uses a two-pointer approach to find the remaining two elements.
    - Two pointers (start and end) are initialized to find the fourth element.
4. Target Value:
    - The target value for the sum of four elements is calculated as X - A[i] - A[j].
5. Comparisons and Movements:
    - The pointers are moved based on the comparison of the sum of elements at start and end with the target sum.
        - If the sum is equal to the target, a combination is found, and the function returns true.
        - If the sum is less than the target, the start pointer is moved to a larger element.
        - If the sum is greater than the target, the end pointer is moved to a smaller element.
6. Result:
    - If a combination is found during the iteration, the function returns true.

- If no combination is found, the function returns false after exhausting all possibilities.

7. Time Complexity:
    - O(N^3) due to the nested loops and the two-pointer approach.
    - The dominant factor is the cubic growth of the input size.
8. Space Complexity:
    - O(1) as no additional space is used other than a few variables.

Remove Element from Array

```
int Solution::removeElement(vector<int> &A, int B) {
    int n = A.size();
    int i = 0, j = 0, ans = 0;

    // Using two pointers i and j
    while (j < n) {
        // If the current element is equal to B, move the j pointer to
the next element
        if (A[j] == B)
            j++;
        else {
            // If the current element is not equal to B, swap A[i] and
A[j], and move both pointers
            swap(A[i++], A[j++]);
            ans++;
        }
    }

    // Return the number of elements left in the array after the
operation
    return ans;
}
```

Code Explanation and Complexity

1. Initialization:
    - Two pointers are initialized - i and j.
    - i represents the position where elements not equal to B should be placed.
    - j is a fast-moving pointer that iterates through the array.
2. Two-Pointer Approach:
    - A two-pointer approach is used to traverse the array efficiently.
3. Removing Elements:
    - If the current element at j is equal to B, increment j to skip this element.

- If the current element at j is not equal to B, swap the elements at positions i and j.
- Increment both pointers (i and j) to move forward.
- Increment the ans variable to count the number of elements left after the operation.

4. Result:
    - The function returns the number of elements left in the array after the operation, which is stored in the variable ans.
5. Time Complexity:
    - O(n), where n is the size of the array.
    - The algorithm performs a single pass through the array using two pointers.
6. Space Complexity:
    - O(1).
    - The algorithm uses constant additional space, regardless of the input size. The elements are rearranged in-place without using any extra data structures.

Array 3 Pointers

```
int Solution::minimize(const vector<int> &A, const vector<int> &B, const
vector<int> &C) {
    // Initialize the minimum difference to the maximum possible value
    int ans = INT_MAX;

    // Initialize pointers for arrays A, B, and C
    int i = 0, j = 0, k = 0;

    // Iterate through all three arrays
    while (i < A.size() && j < B.size() && k < C.size()) {
        // Find the maximum and minimum values among A[i], B[j], and C[k]
        int high_no = max(A[i], max(B[j], C[k]));
        int low_no = min(A[i], min(B[j], C[k]));

        // Move the pointer corresponding to the minimum value
        if (A[i] == low_no) {
            i++;
        } else if (B[j] == low_no) {
            j++;
        } else {
            k++;
        }

        // Update the minimum difference
        ans = min(ans, high_no - low_no);
    }
```

```
    // Return the minimum difference
    return ans;
}
```

Code Explanation and Complexity

- Iterating Through Arrays:
  - Use a while loop to iterate through all three arrays simultaneously.
- Finding Maximum and Minimum:
  - For the current indices (i, j, k), find the maximum and minimum values among A[i], B[j], and C[k].
- Moving Pointers:
  - Move the pointer corresponding to the minimum value to the next element.
  - If A[i] is the minimum, increment i.
  - If B[j] is the minimum, increment j.
  - If C[k] is the minimum, increment k.
- Updating Minimum Difference:
  - Update the minimum difference ans with the difference between the maximum and minimum values found.
- Result:
  - After the loop, the function returns the minimum difference among all possible combinations of elements from arrays A, B, and C.
- Time Complexity:
  - O(N), where N is the total number of elements across all three arrays.
  - The algorithm iterates through the arrays once.
- Space Complexity:
  - O(1).
  - The algorithm uses constant additional space. The space used does not depend on the size of the input but only on the number of pointers used.

Container With Most Water

```
int Solution::maxArea(vector<int> &A) {
    // Get the size of the input array
    int n = A.size();

    // Initialize two pointers, one at the beginning and the other at the
end
    int i = 0, j = n - 1;

    // Initialize variable to store the maximum area
    int ans = 0;
```

```
    // Continue until the two pointers meet
    while (i < j) {
        // Calculate the height of the container (minimum of the two
vertical lines)
        int h = min(A[i], A[j]);

        // Calculate the base of the container (distance between the two
vertical lines)
        int b = (j - i);

        // Calculate the area of the container
        int area = (h * b);

        // Update the maximum area if the current area is larger
        ans = max(ans, area);

        // Move the pointer that points to the smaller vertical line
        if (A[i] <= A[j]) {
            i++;
        } else {
            j--;
        }
    }

    // Return the maximum area
    return ans;
}
```

Code Explanation and Complexity

1. Iterating with Two Pointers:
    - Use a while loop to iterate until the two pointers meet (i < j).
    - The pointers represent the two ends of the potential container.
2. Calculating Height and Base:
    - Calculate the height of the container (h) as the minimum of the heights at positions i and j.
    - Calculate the base of the container (b) as the distance between the two positions (j - i).
3. Calculating Area:
    - Calculate the area of the container as the product of height and base (area = h * b).
4. Updating Maximum Area:

- Update the maximum area (ans) if the current area is larger than the previous maximum.

5. Moving Pointers:
    - Move the pointer pointing to the smaller vertical line.
    - This is done to explore the possibility of finding a larger container.
6. Result:
    - After the loop, the function returns the maximum area that can be contained by the given set of vertical lines.
7. Time Complexity:
    - O(N), where N is the number of elements in the input array.
    - The two pointers move towards each other in a single pass through the array.
8. Space Complexity:
    - O(1).
    - The algorithm uses a constant amount of additional space regardless of the input size.