1. Brute Force Approach

The brute force approach involves sorting the array and then finding the third distinct maximum.

Steps:

1. **Sort the Array:** Sort the array in descending order.
2. **Find the Third Distinct Maximum:** Iterate through the sorted array to find the third distinct element.
3. **Return the Result:** If the third distinct maximum exists, return it; otherwise, return the first maximum.

Time Complexity:

- Sorting the array takes O(n log n), and finding the third distinct maximum takes O(n).
- Overall, the time complexity is O(n log n).

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int thirdMax(vector<int>& nums) {
        sort(nums.begin(), nums.end(), greater<int>());
        int distinctCount = 1;
        int firstMax = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] != nums[i - 1]) {
                distinctCount++;
            }
            if (distinctCount == 3) {
                return nums[i];
            }
        }

        return firstMax;
    }
};
```

2. Better Approach

The better approach involves using a set to store distinct elements and then finding the third maximum.

Steps:

1. **Use a Set:** Insert all elements into a set to automatically remove duplicates.
2. **Check the Size of the Set:**
    - If the set has three or more elements, return the third maximum.
    - If not, return the maximum element in the set.

Time Complexity:

- Inserting elements into the set takes O(n log n) time due to the underlying balanced tree structure.
- The overall time complexity is O(n log n).

```cpp
#include <iostream>
#include <vector>
#include <set>
using namespace std;

class Solution {
public:
    int thirdMax(vector<int>& nums) {
        set<int> distinctNums(nums.begin(), nums.end());

        if (distinctNums.size() < 3) {
            return *distinctNums.rbegin(); // Return the maximum element
        }

        auto it = distinctNums.rbegin();
        advance(it, 2); // Move the iterator to the third last element

        return *it;
    }
};
```

3. Best Approach (O(n) Solution)

The best approach uses three variables to track the first, second, and third maximums, all in one pass through the array.

Steps:

1. **Initialize Three Variables:** Initialize three variables to track the first, second, and third maximums.
2. **Single Pass Through the Array:**
    - Update the three variables based on the current element.
    - Skip elements that are the same as any of the three tracked maximums.
3. **Return the Third Maximum:**
    - If the third maximum exists, return it.
    - Otherwise, return the first maximum.

Time Complexity:

- This approach only requires a single pass through the array, making the time complexity O(n).

```cpp
#include <iostream>
#include <vector>
#include <climits>  // For LONG_MIN
using namespace std;

class Solution {
public:
    int thirdMax(vector<int>& nums) {
        long firstMax = LONG_MIN, secondMax = LONG_MIN, thirdMax = LONG_MIN;

        for (int num : nums) {
            if (num == firstMax || num == secondMax || num == thirdMax)
continue;

            if (num > firstMax) {
                thirdMax = secondMax;
                secondMax = firstMax;
                firstMax = num;
            } else if (num > secondMax) {
                thirdMax = secondMax;
                secondMax = num;
            } else if (num > thirdMax) {
                thirdMax = num;
            }
        }

        return thirdMax == LONG_MIN ? firstMax : thirdMax;
    }
};
```