Recursion Problems Solutions

Here's the **complete, cleanly formatted write-up** for **1D Array Problems and Solutions**, with proper sections, pseudocode, dry runs, and complexity analysis. Everything is rewritten for clarity and correctness:

1D Array Problems and Solutions

---

1. Find Max and Min with Minimum Comparisons

Approach 1: Brute Force

- Traverse the array linearly.
- Maintain two variables: `max_val` and `min_val`, updating as needed.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Approach 2: Tournament Method

- Recursively divide array into halves and find min/max in each half.
- Merge results from both sides.
- **Time Complexity:** O(n)
- **Space Complexity:** O(log n) (due to recursion stack)

Approach 3: Pair Comparison

- Compare elements in pairs instead of individually.
- Reduces total comparisons to ~3n/2.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Pseudocode:

```
cpp

pair<int, int> findMinMax(int arr[], int n) {
    int max_val = INT_MIN, min_val = INT_MAX;
    int i = 0;

    while (i < n) {
        if (i + 1 < n) {
            if (arr[i] > arr[i + 1]) {
                max_val = max(max_val, arr[i]);
                min_val = min(min_val, arr[i + 1]);
            } else {
                max_val = max(max_val, arr[i + 1]);
                min_val = min(min_val, arr[i]);
```

```
            }
            i += 2;
        } else {
            max_val = max(max_val, arr[i]);
            min_val = min(min_val, arr[i]);
            i++;
        }
    }
    return {max_val, min_val};
}
```

2. Sort 0, 1, 2 (Dutch National Flag Problem)

Approach 1: Brute Force

- Count number of 0s, 1s, and 2s.
- Rewrite the array using those counts.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Approach 2: Two-Pointer / Dutch National Flag

- Use three pointers: **`low`**, **`mid`**, **`high`**.
- Arrange 0s in front, 2s at end, 1s in between.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Pseudocode:

```
cpp

void sort012(vector<int>& arr) {
    int low = 0, mid = 0, high = arr.size() - 1;

    while (mid <= high) {
        if (arr[mid] == 0) {
            swap(arr[low++], arr[mid++]);
        } else if (arr[mid] == 1) {
            mid++;
        } else {
            swap(arr[mid], arr[high--]);
        }
    }
}
```

3. Union and Intersection of Arrays

Approach 1: Brute Force

- Use nested loops for union and intersection.
- **Time Complexity:** O(n * m)
- **Space Complexity:** O(n + m)

Approach 2: Using Sets (Optimal)

- For union: insert both arrays in a set.
- For intersection: check each element of `arr2` in `set1`.
- **Time Complexity:** O(n + m)
- **Space Complexity:** O(n + m)

Pseudocode:

```
cpp

vector<int> findUnion(vector<int>& arr1, vector<int>& arr2) {
    set<int> unionSet(arr1.begin(), arr1.end());
    unionSet.insert(arr2.begin(), arr2.end());
    return vector<int>(unionSet.begin(), unionSet.end());
}

vector<int> findIntersection(vector<int>& arr1, vector<int>& arr2) {
    set<int> set1(arr1.begin(), arr1.end());
    vector<int> result;
    for (int num : arr2) {
        if (set1.count(num)) result.push_back(num);
    }
    return result;
}
```

---

4. Kadane's Algorithm (Max Subarray Sum)

Approach 1: Brute Force

- Check all subarrays.
- **Time Complexity:** O(n^2)
- **Space Complexity:** O(1)

Approach 2: Kadane's Algorithm

- Track `current_sum` and `max_sum`.
- Reset `current_sum` if it becomes negative.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Pseudocode:

```
cpp

int maxSubarraySum(vector<int>& arr) {
    int max_sum = arr[0], current_sum = 0;
    for (int num : arr) {
        current_sum = max(num, current_sum + num);
        max_sum = max(max_sum, current_sum);
    }
    return max_sum;
}
```

5. Find Duplicates in an Array

Approach 1: Brute Force

- Compare each element with every other.
- **Time Complexity:** O(n^2)
- **Space Complexity:** O(1)

Approach 2: Hash Map

- Count occurrences using unordered_map.
- **Time Complexity:** O(n)
- **Space Complexity:** O(n)

Pseudocode:

```cpp
void findDuplicates(vector<int>& arr) {
    unordered_map<int, int> freq;
    for (int num : arr) freq[num]++;
    for (auto [num, count] : freq) {
        if (count > 1) cout << num << " ";
    }
}
```

Approach 3: In-Place Marking

- Mark index corresponding to value negative.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Pseudocode:

```cpp
void findDuplicates(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        int index = abs(arr[i]) - 1;
        if (arr[index] < 0) {
            cout << abs(arr[i]) << " ";
        } else {
            arr[index] = -arr[index];
        }
    }
}
```

6. Merge Overlapping Intervals

Approach: Sort and Merge

- Sort by start times, then merge overlapping intervals.
- **Time Complexity:** O(n log n)
- **Space Complexity:** O(n)

Pseudocode:

```
cpp

vector<pair<int, int>> mergeIntervals(vector<pair<int, int>>& intervals) {
    if (intervals.empty()) return {};
    sort(intervals.begin(), intervals.end());
    vector<pair<int, int>> merged;
    merged.push_back(intervals[0]);

    for (int i = 1; i < intervals.size(); i++) {
        auto& last = merged.back();
        if (intervals[i].first <= last.second) {
            last.second = max(last.second, intervals[i].second);
        } else {
            merged.push_back(intervals[i]);
        }
    }
    return merged;
}
```

---

7. Count Inversions in Array

Approach: Brute Force

- Count all pairs `(i, j)` where `i < j` and `arr[i] > arr[j]`.
- **Time Complexity:** $O(n^2)$
- **Space Complexity:** O(1)

Pseudocode:

```
cpp

int countInversions(vector<int>& arr) {
    int count = 0;
    for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++) {
            if (arr[i] > arr[j]) count++;
        }
    }
    return count;
}
```

---

8. Next Permutation

- Rearrange the array into its next lexicographical permutation.
- **Time Complexity:** O(n)
- **Space Complexity:** O(1)

Pseudocode:

```cpp
void nextPermutation(vector<int>& nums) {
    int i = nums.size() - 2;
    while (i >= 0 && nums[i] >= nums[i + 1]) i--;

    if (i >= 0) {
        int j = nums.size() - 1;
        while (nums[j] <= nums[i]) j--;
        swap(nums[i], nums[j]);
    }

    reverse(nums.begin() + i + 1, nums.end());
}
```

---

9. Minimize the Maximum Pair Sum

Problem: Pair elements such that the max sum of any pair is minimized.

Approach: Sorting

- Sort and pair smallest with largest.
- Track max sum of all pairs.
- **Time Complexity:** O(n log n)
- **Space Complexity:** O(1)

Pseudocode:

```cpp
int minimizeMaxPair(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int maxPairSum = 0;
    for (int i = 0, j = nums.size() - 1; i < j; i++, j--) {
        maxPairSum = max(maxPairSum, nums[i] + nums[j]);
    }
    return maxPairSum;
}
```

Dry Run:
Input: [3, 1, 4, 2] → Sorted: [1, 2, 3, 4]
Pairs: (1+4)=5, (2+3)=5 → Max = 5
Output: 5

Let me know if you'd like:

- Actual C++ implementation for all.
- Dry runs for more examples.
- MCQs or coding questions based on this.