Comprehensive Report on Solving the Maximum Subarray Problem using Dynamic Programming

1. Understanding the Problem  
The Maximum Subarray Problem is a classical problem in computer science. Given an array of integers, the task is to find a contiguous subarray with the largest possible sum. This problem serves as an excellent example of how dynamic programming can simplify complex tasks.

Formal Statement:  
Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Examples:  
- Input: [-2,1,-3,4,-1,2,1,-5,4] → Output: 6 (from subarray [4,-1,2,1])  
- Input: [1] → Output: 1  
- Input: [5,4,-1,7,8] → Output: 23

2. Naïve Approach (Brute Force)  
Try all possible subarrays and calculate their sums. This results in a time complexity of O(n³), which is inefficient for large arrays.

3. Optimized Brute Force: Cumulative Sum  
Use prefix sums to reduce inner loop, improving time complexity to O(n²), which is still suboptimal.

4. Efficient Approach: Dynamic Programming  
Dynamic Programming (DP) solves problems by breaking them into subproblems, solving each only once, and storing their results.

Features of DP Problems:  
- Overlapping subproblems  
- Optimal substructure

5. Kadane’s Algorithm – Dynamic Programming Approach  
Kadane’s Algorithm efficiently solves the Maximum Subarray Problem in O(n) time by keeping track of:  
- currentMax: maximum sum of subarray ending at current index  
- maxSoFar: overall maximum subarray sum found so far

Recurrence Relation:  
currentMax = max(nums[i], currentMax + nums[i])  
maxSoFar = max(maxSoFar, currentMax)

6. Step-by-Step Example  
Input: [-2,1,-3,4,-1,2,1,-5,4]  
Illustrated via table in full report.

7. C++ Implementation  
```cpp  
class Solution {  
public:  
 int maxSubArray(vector<int>& nums) {  
 int maxSoFar = nums[0];  
 int currentMax = nums[0];  
 for (int i = 1; i < nums.size(); ++i) {  
 currentMax = max(nums[i], currentMax + nums[i]);  
 maxSoFar = max(maxSoFar, currentMax);  
 }  
 return maxSoFar;  
 }  
};  
```

8. Time and Space Complexity  
- Time Complexity: O(n)  
- Space Complexity: O(1)

9. Real-World Analogy  
Visualize walking on a road with ups and downs, trying to find the most rewarding path.

10. When to Use Kadane’s Algorithm?  
- Contiguous subarray problems  
- Optimizing over sums or accumulative metrics  
- Arrays with negative and positive integers

11. Extending the Problem  
- Return the actual subarray  
- Solve 2D version using modified Kadane’s  
- Handle circular arrays

12. Approach Such Problems Systematically  
- Understand the problem and constraints  
- Consider brute-force and identify DP potential  
- Define subproblems and recurrence relations

13. Dynamic Programming’s Value  
DP reduces recomputation and uses previous results to make optimal decisions at each step.

14. General Template for DP Subarray Problems  
Illustrated in Kadane’s Algorithm.

15. Conclusion  
Kadane’s Algorithm exemplifies dynamic programming's strength in reducing time complexity from cubic to linear time. It provides foundational knowledge for solving more advanced optimization problems.