Project 4

Aditya Anil Raut
012648884

Dynamic Programming Problems Analysis

1. Matrix Chain Multiplication

Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices together. The problem is not to perform the multiplications, but rather to decide in which order to perform the multiplications to minimize the total number of scalar multiplications.

Naive Approach

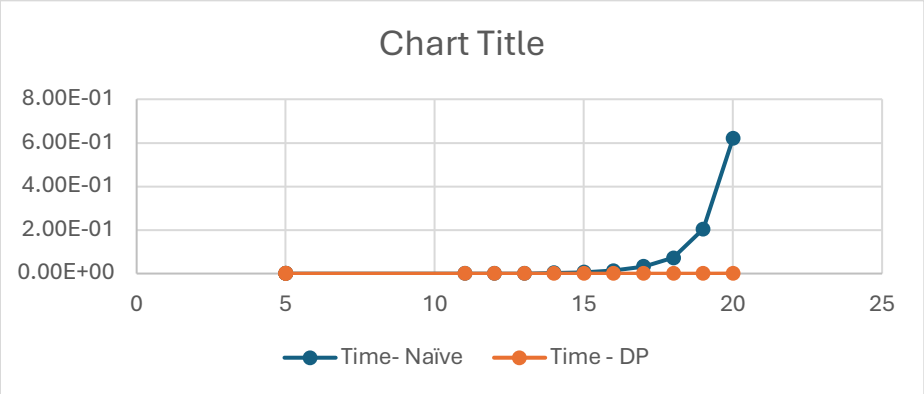
- Uses recursive strategy to try all possible combinations of matrix multiplications
- Repeatedly computes the same subproblems
- No storage of intermediate results
- Higher time complexity due to redundant calculations

Dynamic Programming Approach

- Uses bottom-up approach storing intermediate results
- Builds solution from smaller subproblems
- Eliminates redundant calculations

Complexity Comparison

Aspect	Naive Approach	DP Approach
Time Complexity	O(2^n)	$O(n^3)$
Space Complexity	O(n)	$O(n^2)$



Conclusion

The DP approach significantly improves performance by reducing time complexity from exponential to polynomial, making it practical for larger matrices. The trade-off of additional space complexity is justified by the dramatic improvement in execution time.

2. Rod Cutting Problem

Given a rod of length n inches and a table of prices for different lengths, determine the maximum value obtainable by cutting up the rod and selling the pieces.

Naive Approach

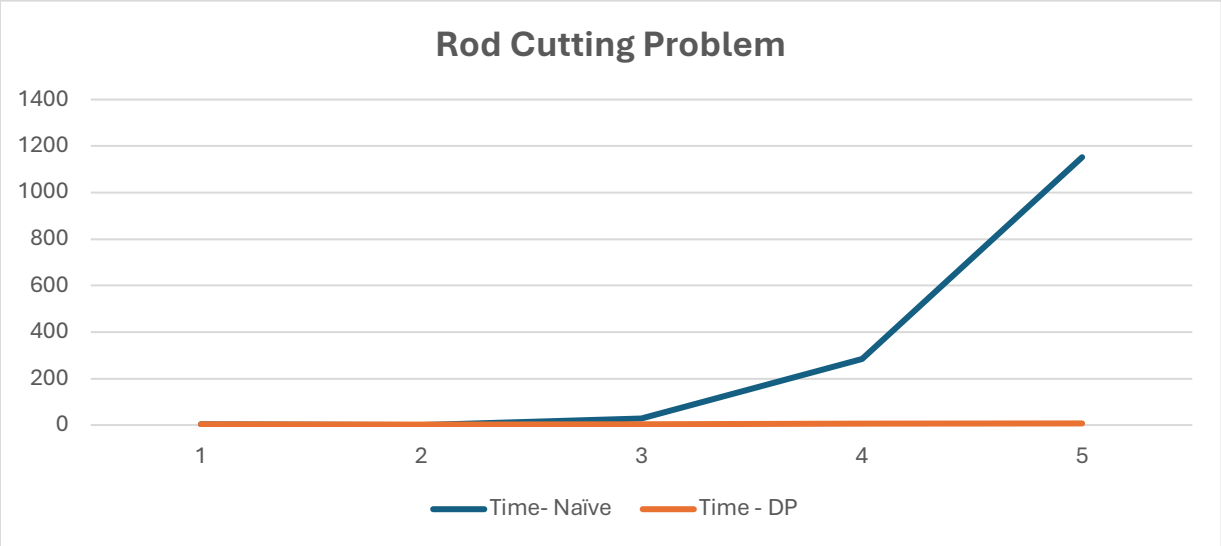
- Recursively explores all possible cutting points
- Generates redundant subproblems
- Simple to implement but inefficient
- Exponential growth with input size

Dynamic Programming Approach

- Uses 1D DP table to store maximum profits
- Bottom-up computation avoiding redundancy
- Efficient handling of larger inputs

Complexity Comparison

Aspect	Naive Approach	DP Approach
Time Complexity	O(2^n)	$O(n^2)$
Space Complexity	O(n)	O(n)



Conclusion

The DP solution maintains the same space complexity while drastically reducing time complexity. Provides a solution to manage time complexity and performs better with larger data.

3. Word Break Problem

Given a string and a dictionary of words, determine if the string can be segmented into a space-separated sequence of dictionary words.

Naive Approach

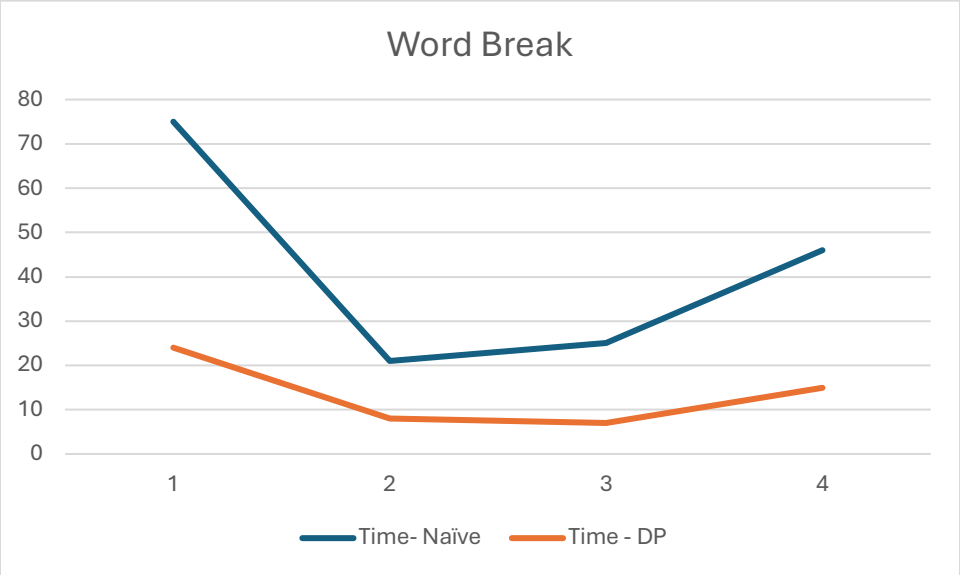
- Recursive string segmentation
- Explores all possible breaking points
- Repeated substring checking
- Inefficient for longer strings

Dynamic Programming Approach

- Uses 1D DP table for segmentation tracking
- Efficient substring validation
- Handles longer strings effectively

Complexity Comparison

Aspect	Naive Approach	DP Approach
Time Complexity	O(2^n)	$O(n^2)$
Space Complexity	O(n)	O(n)



Conclusion

The DP approach provides significant performance improvements while maintaining the same space complexity. Although it can get outperform by naïve approach in some test cases, overall DP is the better approach when it comes to time complexity

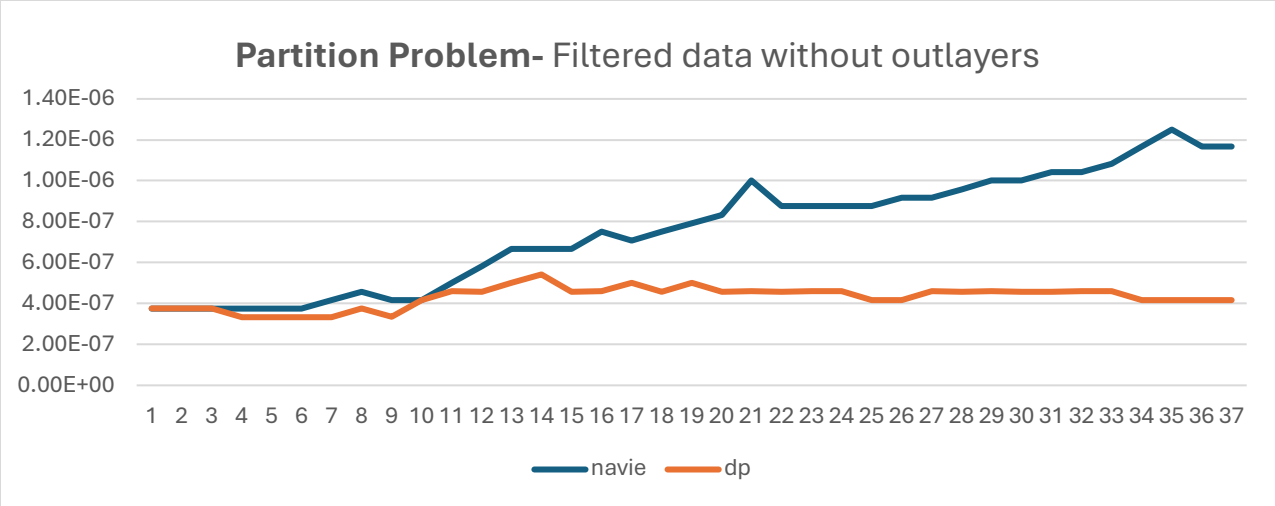
4. Partition Problem

Given a set of numbers, determine if it can be divided into two subsets such that the sum of elements in both subsets is equal.

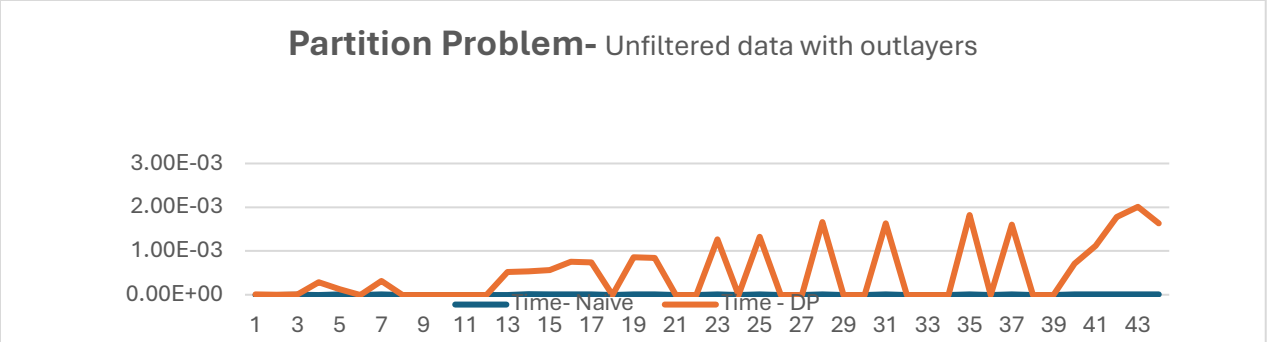
Naive Approach	**Dynamic Programming Approach**
- Explores all possible partitioning options - Simple but highly inefficient - Exponential growth with set size	- Uses DP table for subset sum tracking - Efficient subproblem resolution - Optimal for larger input sets

Complexity Comparison

Aspect	Naive Approach	DP Approach
Time Complexity	O(2^n)	O (n* sum)
Space Complexity	O(n)	O(sum)



In the above chart the data is filtered and only the cases of random array with increasing size are used to show how DP is better than Navie Algorithm.



In the above chart the data is unfiltered raw data is used of random array with increasing size are used. The chart highlights unexpected inefficiencies in the Dynamic Programming (DP) approach for solving the Partition Problem, with execution times occasionally surpassing the naive solution. This anomaly likely stems from issues such as inefficient loop structures, poor handling of large inputs or sums, or redundant computations within the DP implementation.

Conclusion

The DP solution offers a substantial improvement in time complexity, though with slightly increased space requirements. This anomaly such as inefficient loop structures, poor handling of large inputs or sums, or redundant computations within the DP implementation may occur.

5. Palindrome Partitioning

Given a string, partition it into substrings such that each substring is a palindrome. The goal is to find the minimum number of cuts needed for such a partition.

Naive Approach

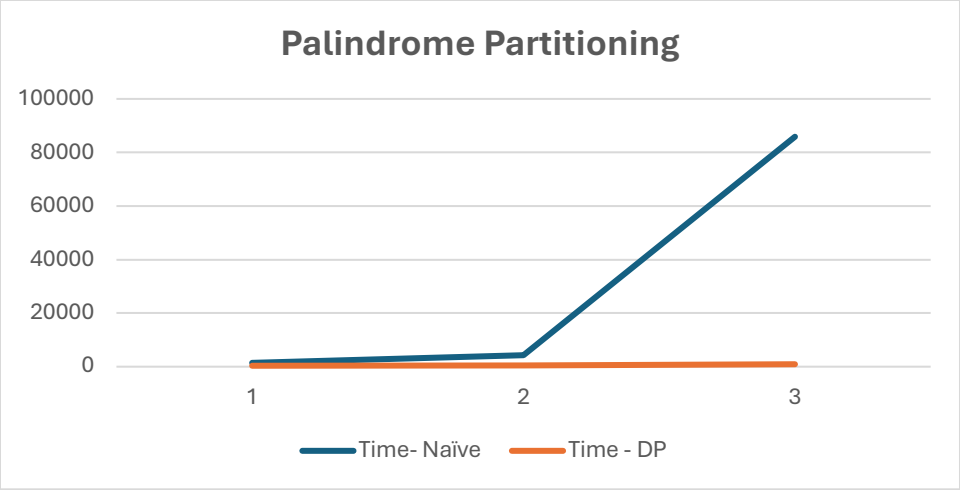
- Recursive cutting and checking
- Redundant palindrome verification
- Simple implementation
- Exponential time growth

Dynamic Programming Approach

- Dual DP tables for optimization
- Precomputed palindrome status
- Efficient cut minimization

Complexity Comparison

Aspect	Naive Approach	DP Approach
Time Complexity	O(2^n)	$O(n^2)$
Space Complexity	O(n)	$O(n^2)$



Conclusion

The DP approach significantly reduces time complexity at the cost of increased space complexity. Although it can get outperform by naïve approach in some test cases, overall DP is the better approach when it comes to time complexity.