Programming Assignment 4 of CSCI 311

California State University, Chico

Question1: "Matrix Chain Multiplication Problem: Implementation and Analysis"

Test Case 1 Input: p = [10, 20, 30, 40, 30] The above input array implies that the following matrices are under consideration: A1 (10 × 20) • A2 (20 × 30) • A3 (30 × 40) • A4 (40 × 30) Output: 30000 scalar multiplications.

Using Recursive:

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question1
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Minimum multiplications (Recursive): 30000
Number of recursive calls: 0
Execution time: 1µs
```

Using Dynamic Programming:

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question1
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Minimum multiplications (Dynamic Programming): 30000
Execution time: 0µs
```

Comparison:

We can see the dime difference between recursive approach and dynamic programming approaches. Time taken by dynamic approach is less compared to recursive approach.

Test Case 2 Input:

p = [30, 35, 15, 5, 10, 20, 25]

Output: 15125 scalar multiplications.

Explanation: The most efficient way to multiply these matrices requires 15125 scalar multiplications.

Using recursive:

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question1
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Minimum multiplications (Recursive): 15125
Number of recursive calls: 0
Execution time: 4µs
```

Using Dynamic Programming:

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question1
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Minimum multiplications (Dynamic Programming): 15125
Execution time: 2µs
```

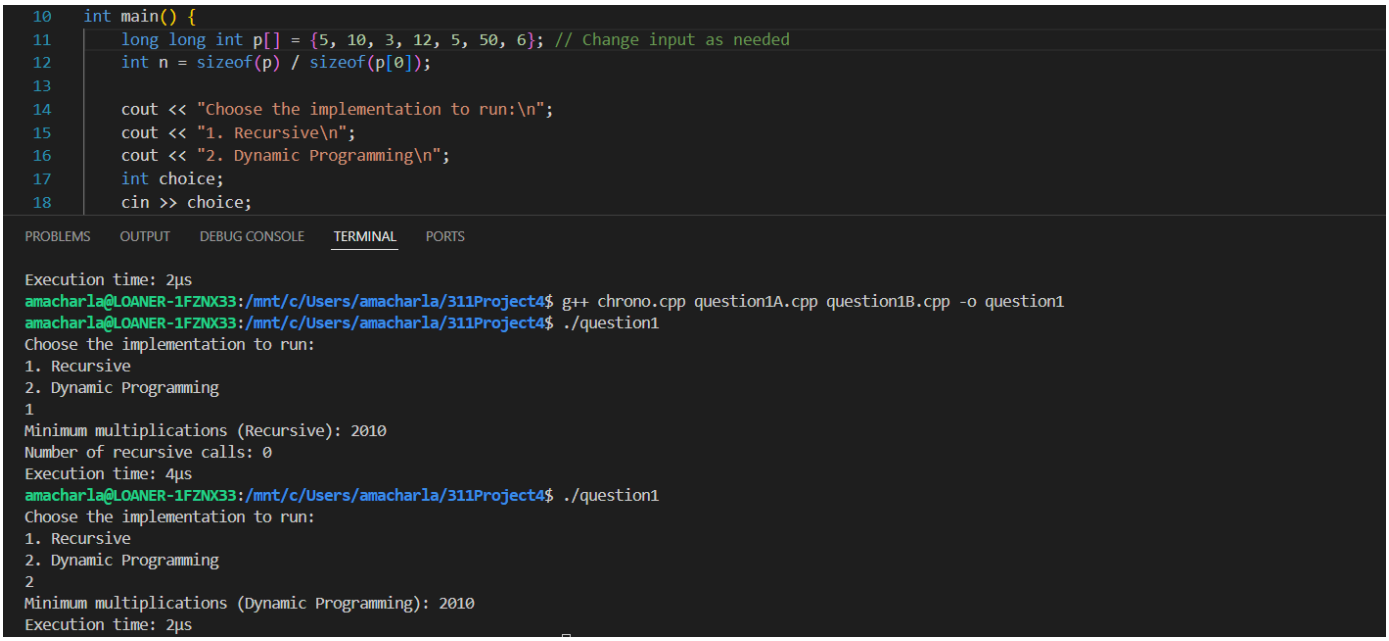
Comparison:

We can see the dime difference between recursive approach and dynamic programming approaches. Time taken by dynamic approach is less compared to recursive approach.

Test Case 3:

Input: p = [5, 10, 3, 12, 5, 50, 6]

Output: 2010 scalar multiplications



The above pic shows the input we have considered and the time difference between both the approaches.

Comparison:

We can see the dime difference between recursive approach and dynamic programming approaches. Time taken by dynamic approach is less compared to recursive approach.

Final input taking input size very large:

Let's consider a larger output so that the time difference is clear:

Input: p[] = {30, 35, 15, 5, 10, 20, 25, 60, 70, 80, 90, 20, 30}

Output: 120375 scalar multiplications

```
int main() {
    long long int p[] = {30, 35, 15, 5, 10, 20, 25, 60, 70, 80, 90, 20, 30}; // Change input as needed
    int n = sizeof(p) / sizeof(p[0]);

    cout << "Choose the implementation to run:\n";
    cout << "1. Recursive\n";
    cout << "2. Dynamic Programming\n";
    int choice;
    cin >> choice;
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Execution time: 2µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ g++ chrono.cpp question1A.cpp question1B.cpp -o question1
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question1
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Minimum multiplications (Recursive): 120375
Number of recursive calls: 0
Execution time: 914µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question1
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Minimum multiplications (Dynamic Programming): 120375
Execution time: 10µs
```

Time taken by recursive approach for this large input is: 914 microseconds.

Time taken by dynamic approach for this large input is: 10 microseconds.

Comparison:

We can see the dime difference between recursive approach and dynamic programming approaches. Time taken by dynamic approach is less compared to recursive approach.

Conclusion:

From this we can conclude that Dynamic programming approach helps to reduce the calculation time, avoid extra works that get calculated when we use recursive approach.

Question2: Rod Cutting Problem:

Objective: Maximize profit by cutting a rod of length ‘n’ into smaller lengths, given a price table for each length.

Understanding the Problem:

• Basic Concept: Given a rod of length n and a table of prices for rods of different lengths, the goal is to cut the rod into pieces to maximize total profit.

Constraints: – Cuts must be in integer lengths.

Each piece can be sold according to the given price table.

The total length of the pieces must equal the length of the original rod.

Objective: Find the optimal cut configuration that maximizes total profit.

Approach to Solve the Problem:

• Dynamic Programming: Solve the problem using dynamic programming, breaking it down into simpler subproblems.

• Bottom-Up Approach: Start with the smallest lengths, gradually building up to the solution for the original length while tracking the maximum profit.

• Record Keeping: For each length, consider all possible ways to cut the rod and choose the most profitable one.

• Reconstructing the Solution: Use the recorded decisions to determine the specific cuts that yield the maximum profit.

Considering Different test cases:

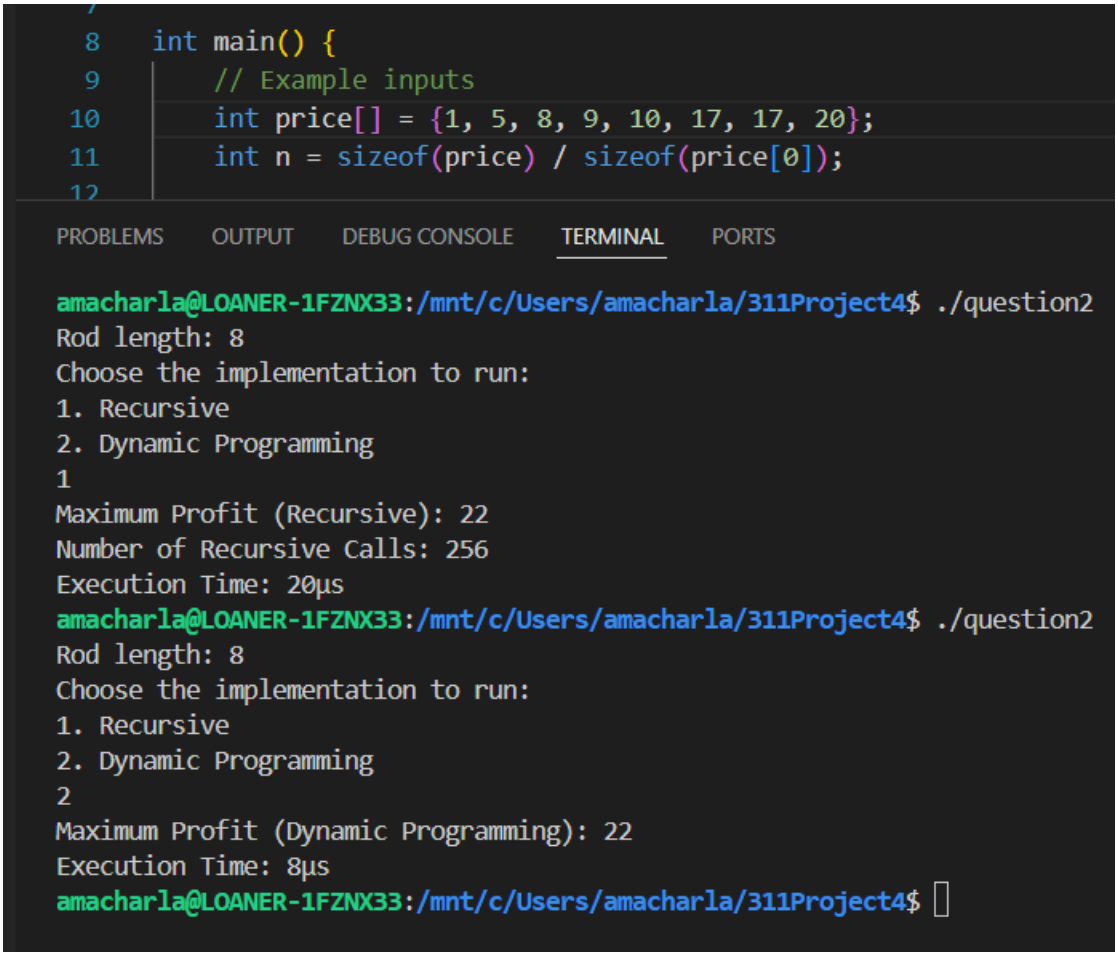
Use Case 1:

• Input: length of the rod = 8 and price array = [1, 5, 8, 9, 10, 17, 17, 20]

• Objective: Maximize the total profit from selling the rod pieces.

• Output: 22

• Explanation: An optimal way is to cut the rod into pieces of lengths 2 and 6, yielding a profit of 22 (5 + 17).



Comparison:

The above pic shows that for the rod length '8' the time taken by dynamic approach is lesser compared to recursive approach.

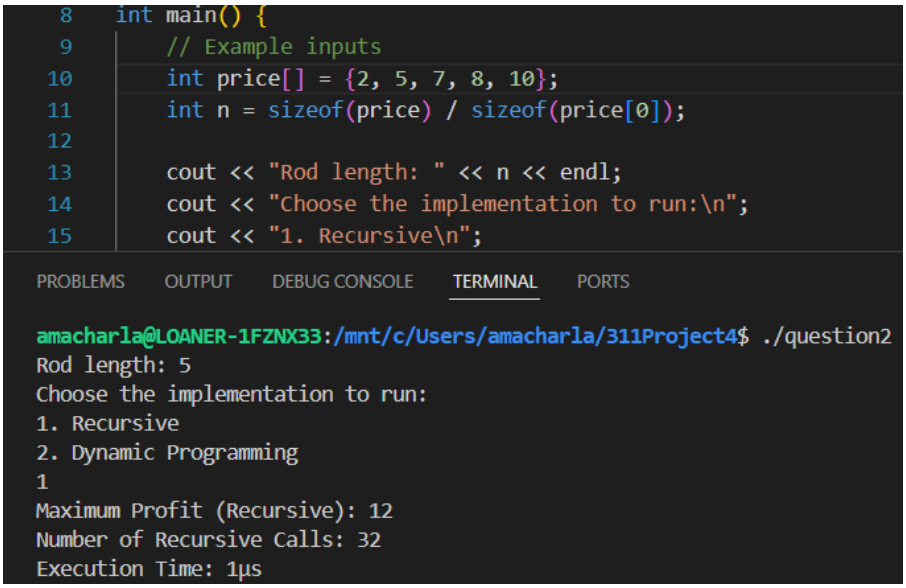
Use Case 2:

• Input: length of the rod = 5, price array = [2, 5, 7, 8, 10]

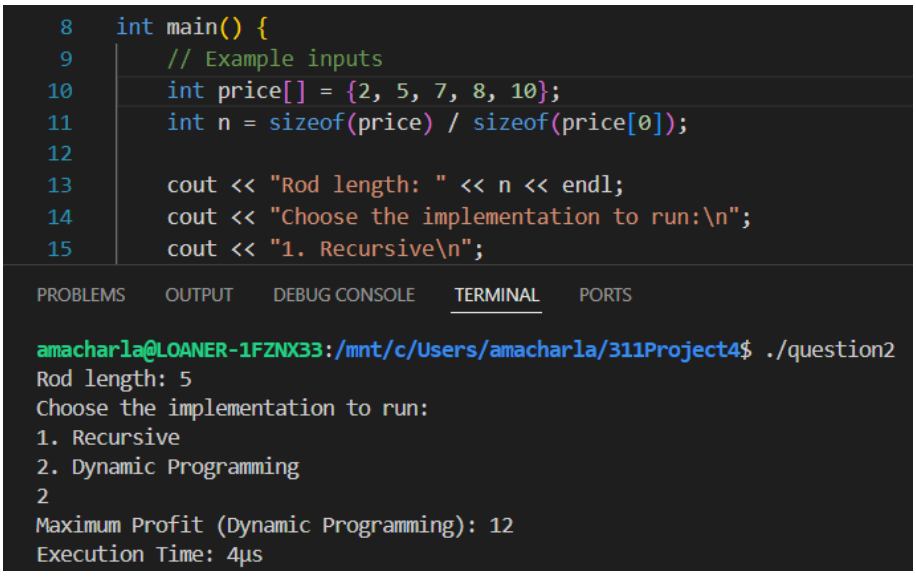
• Output: 12

• Explanation: Cutting the rod into lengths 2 and 3 yields a profit of 12 (5 + 7), which is the maximum profit.

Recursive approach:



Dynamic programming approach:



For the smaller input time taken by recursive approach is less compared to dynamic approach.

Comparison and reason:

Why Recursive Is Faster for Small Inputs

1. **Input Size**:
    - For very small input sizes (e.g., rod length n=8n = 8n=8), the number of recursive calls and operations in the naive recursive approach is minimal.
    - The dynamic programming solution has the overhead of initializing and filling the DP table, even for small inputs.
2. **Overhead in DP**:
    - The dynamic programming approach requires:
        - Additional memory to maintain the DP table.
        - Loop iterations for all possible lengths, even if the input size is small.
    - This results in a slightly higher execution time compared to the recursive approach for small inputs.

We will now test another instance with larger input:

Use Case 3:

• Input: length of the rod = 5

• price array ={1, 5, 8, 9, 10, 17, 17, 20, 24, 30, 35, 40, 50, 55, 60};

• Output: 60

Below are the two approaches Recursive and dynamic.

Recursive approach:

```
10      int price[] = {1, 5, 8, 9, 10, 17, 17, 20, 24, 30, 35, 40, 50, 55, 60};
11      int n = sizeof(price) / sizeof(price[0]);
12
13      cout << "Rod length: " << n << endl;
14      cout << "Choose the implementation to run:\n";
15      cout << "1. Recursive\n";
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question2
Rod length: 15
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Maximum Profit (Recursive): 60
Number of Recursive Calls: 32768
Execution Time: 550µs
```

Dynamic Programming approach:

```
10      int price[] = {1, 5, 8, 9, 10, 17, 17, 20, 24, 30, 35, 40, 50, 55, 60};
11      int n = sizeof(price) / sizeof(price[0]);
12
13      cout << "Rod length: " << n << endl;
14      cout << "Choose the implementation to run:\n";
15      cout << "1. Recursive\n";
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question2
Rod length: 15
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Maximum Profit (Dynamic Programming): 60
Execution Time: 11µs
```

Comparison:

Time taken by dynamic programming approach is very less compared to recursive approach.

Conclusion

- **For Small Inputs**:
    - The recursive solution can appear faster due to its minimal overhead and compiler optimizations.
    - The DP approach may take slightly longer due to the overhead of table initialization and iteration.
- **For Larger Inputs**:
    - The recursive approach quickly becomes impractical due to its exponential growth in time complexity (O(2n)O(2^n)O(2n)).
    - The DP approach scales efficiently with polynomial time complexity (O(n2)O(n^2)O(n2)) and handles larger inputs in a reasonable time.
- **Practical Implications**:
    - For small inputs, the difference in time is negligible and should not influence the choice of algorithm.
    - For larger inputs, the DP solution is unequivocally superior and should always be preferred in practical applications.

Question3: Word Break Problem

Objective: Determine if a given string can be segmented into one or more words found in a provided dictionary.

Details of the Problem:

• Input: A string ‘s’ and a list of words ‘wordDict’ that represents the dictionary.

• Task: Check if ‘s’ can be segmented into a space-separated sequence of one or more dictionary words.

• Constraints: – The string ‘s’ and the words in ‘wordDict’ consist only of lowercase English letters. – The same word in the dictionary may be reused multiple times in the segmentation. – The dictionary does not contain duplicate words.

• Output: Return True if the string can be segmented, False otherwise.

Approach to Solve the Problem:

• Dynamic Programming: This problem can be efficiently solved using a dynamic programming approach.

• Subproblem Definition: For a substring of ‘s’, determine if it can be segmented into dictionary words. Use this information to solve for larger substrings.

• Bottom-Up Solution: Start with smaller substrings and build up to the entire string, keeping track of whether each substring can be segmented.

• Backtracking: If the current substring cannot be formed, backtrack and try different segmentations

Use Cases:

Use Case 1:

• Input: s = “applepenapple”, wordDict = [“apple”, “pen”]

• Objective: Determine if “applepenapple” can be segmented into words from the given dictionary.

• Output: True

• Explanation: The string can be segmented as “apple pen apple”, with all words present in the dictionary.

```
 8   int main() {
 9       // Example inputs
10       string s = "applepenapple";
11       vector<string> wordDict = {"apple", "pen"};
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question3
String: applepenapple
Dictionary: [ apple pen ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Can the string be segmented (Recursive)? True
Execution Time: 25µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question3
String: applepenapple
Dictionary: [ apple pen ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Can the string be segmented (Dynamic Programming)? True
Execution Time: 36µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ^C
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$
```

Use Case 2:

• Input: s = “catsandog”, wordDict = [“cats”, “dog”, “sand”, “and”, “cat”]

• Output: False

• Explanation: There is no way to segment “catsandog” into a sequence of words from the dictionary without leaving some letters unused.

Recursive:

```
 8   int main() {
 9       // Example inputs
10       string s = "catsandog";
11       vector<string> wordDict = {"cats", "dog" , "sand", "and", "cat"};
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question3
String: catsandog
Dictionary: [ cats dog sand and cat ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Can the string be segmented (Recursive)? False
Execution Time: 38µs
```

Dynamic:

```
 8   int main() {
 9       // Example inputs
10       string s = "catsandog";
11       vector<string> wordDict = {"cats", "dog" , "sand", "and", "cat"};
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question3
String: catsandog
Dictionary: [ cats dog sand and cat ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Can the string be segmented (Dynamic Programming)? False
Execution Time: 38µs
```

Comparison and Reasoning:

For smaller inputs we get less time for recursion approach compared to dynamic approach.

Why Recursive Is Faster for Small Inputs

1. **Minimal Input Size**:
    - For small strings (like "applepenapple") with short dictionaries, the number of recursive calls is small.
    - The overhead of maintaining the DP table and iterating through substrings in nested loops in the DP approach can outweigh the cost of recursive calls.
2. **No Table Initialization in Recursion**:
    - The recursive approach directly processes the string and does not allocate or initialize a DP table.
    - The DP approach must allocate and iterate over the DP table, even for small inputs, introducing some overhead.

What Happens for Larger Inputs:

For larger inputs (e.g., strings with lengths >30 > 30>30 and larger dictionaries):

- **Recursive Approach**:
    - The number of recursive calls grows exponentially O(2^n) leading to significant performance degradation.
    - Recursive calls may eventually lead to stack overflow for large inputs.
- **Dynamic Programming Approach**:
    - The DP approach has a time complexity of O(n^2)), which grows much slower than the exponential growth of recursion.
    - It efficiently handles large inputs and avoids redundant computations.

Question 4: Partition Problem

Objective: Determine whether a given set of integers can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Details of the Problem:

• Input: A set of integers.

• Task: Check if it's possible to divide the set into two subsets with equal sums.

• Constraints: – The set may contain positive, negative, or zero values. – The number of elements in the set is not fixed. – The subsets do not need to have an equal number of elements, only equal sums. • Output: Return True if such a partition exists, False otherwise.

Use Cases:

Use Case 1:

• Input: set = [1, 5, 11, 5]

• Output: True

• Explanation: The set can be partitioned into two subsets [1, 5, 5] and [11], both with a sum of 11.

```
 8   int main() {
 9       int nums[] = {1, 5, 11, 5};
10       int size = sizeof(nums) / sizeof(nums[0]);
11
12       cout << "Input set: [ ";
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ g++ chrono4.cpp question4A.cpp question4B.cpp -o question4
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question4
Input set: [ 1 5 11 5 ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Can the set be partitioned (Recursive)? True
Execution Time: 0µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question4
Input set: [ 1 5 11 5 ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Can the set be partitioned (Dynamic Programming)? True
Execution Time: 1µs
```

Comparison: Time taken by Dynamic approach is slightly greater than recursive approach.

Reason:

Minimal Problem Size:

- In the test case ({1, 5, 11, 5}), the size of the input is small (n=4n = 4n=4), so the number of recursive calls required is limited.

No Table Initialization in Recursion:

- The recursive approach directly computes results without precomputing or initializing a table, saving time for small problems.

Main Answer:

Yes, this test case can be patitioned.

Use Case 2:

• Input: set = [1, 2, 3, 5]

• Output: False • Explanation: There is no possible partition that divides the set into two subsets with equal sums.

```
 8  int main() {
 9      int nums[] = {1, 2, 3, 5};
10      int size = sizeof(nums) / sizeof(nums[0]);
11
12      cout << "Input set: [ ";
13      for (int i = 0; i < size; i++) cout << nums[i] << " ";
14      cout << "]" << endl;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question4
Input set: [ 1 2 3 5 ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Can the set be partitioned (Recursive)? False
Execution Time: 0µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question4
Input set: [ 1 2 3 5 ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Can the set be partitioned (Dynamic Programming)? False
Execution Time: 0µs
```

Explanation:

This set cannot be partitioned.

Use Case 3:

• Input: set = [2, 3, 5]

• Output: True

• Explanation: The first set can have 2 and 3 which sums to 5 and the second set can have 5 by itself.

```
 8   int main() {
 9       int nums[] = {2, 3, 5};
10       int size = sizeof(nums) / sizeof(nums[0]);
11
12       cout << "Input set: [ ";
13       for (int i = 0; i < size; i++) cout << nums[i] << " ";
14       cout << "]" << endl;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question4
Input set: [ 2 3 5 ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Can the set be partitioned (Recursive)? True
Execution Time: 0µs
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question4
Input set: [ 2 3 5 ]
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Can the set be partitioned (Dynamic Programming)? True
Execution Time: 1µs
```

Explanation: This set can be partitioned.

***Question 5: Palindrome Partitioning*:**

Objective: Find the minimum number of cuts needed to partition a string such that every substring is a palindrome.

The goal is to split a given string into the fewest possible substrings where each substring is a palindrome (reads the same backward and forward). If the entire input string is already a palindrome (like “racecarracecar”), the minimum number of cuts required is zero because no partitioning is needed to meet the objective.

Approach to Solve the Problem

• Dynamic Programming: Use dynamic programming to minimize redundant computations.
• Palindrome Check: Implement a function to check if a substring is a palindrome.

• Subproblem Definition: Define a subproblem as finding the minimum cuts for a substring.

• Bottom-Up Solution: Start with small substrings, expanding to the entire string, optimizing the number of cuts.

• Optimization: Store results of subproblems to avoid repeated calculations.

Use Cases :

Use Case 1:

• Input: s = “racecar”

• Output: 0

• Explanation: The entire string “racecar” is a palindrome; hence no cuts are needed.

```
8   int main() {
9       string s = "racecar";
10
11      cout << "Input string: " << s << endl;
12
13      // Check if the input string is a palindrome
14      if (isPalindromeString(s)) {
15          cout << "The string is already a palindrome. No cuts needed." << endl;
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ g++ chrono5.cpp question5A.cpp question5B.cpp -o question5
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question5
Input string: racecar
The string is already a palindrome. No cuts needed.
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$
```

Explanation:

This sting is already a palindrome so there is no need to cut.

Use Case 2:

• Input: s = “noonabbad”

• Output: 2

• Explanation: An optimal solution is “noon — abba — d”, needing two cuts.

Recursive approach:

```
 8   int main() {
 9       string s = "noonabbad";
10
11       cout << "Input string: " << s << endl;
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question5
Input string: noonabbad
The string is not a palindrome.
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
1
Minimum cuts (Recursive): 2
Execution Time: 4µs
```

Dynamic approach:

```
 8   int main() {
 9       string s = "noonabbad";
10
11       cout << "Input string: " << s << endl;
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
amacharla@LOANER-1FZNX33:/mnt/c/Users/amacharla/311Project4$ ./question5
Input string: noonabbad
The string is not a palindrome.
Choose the implementation to run:
1. Recursive
2. Dynamic Programming
2
Minimum cuts (Dynamic Programming): 2
Execution Time: 38µs
```

Explanation:

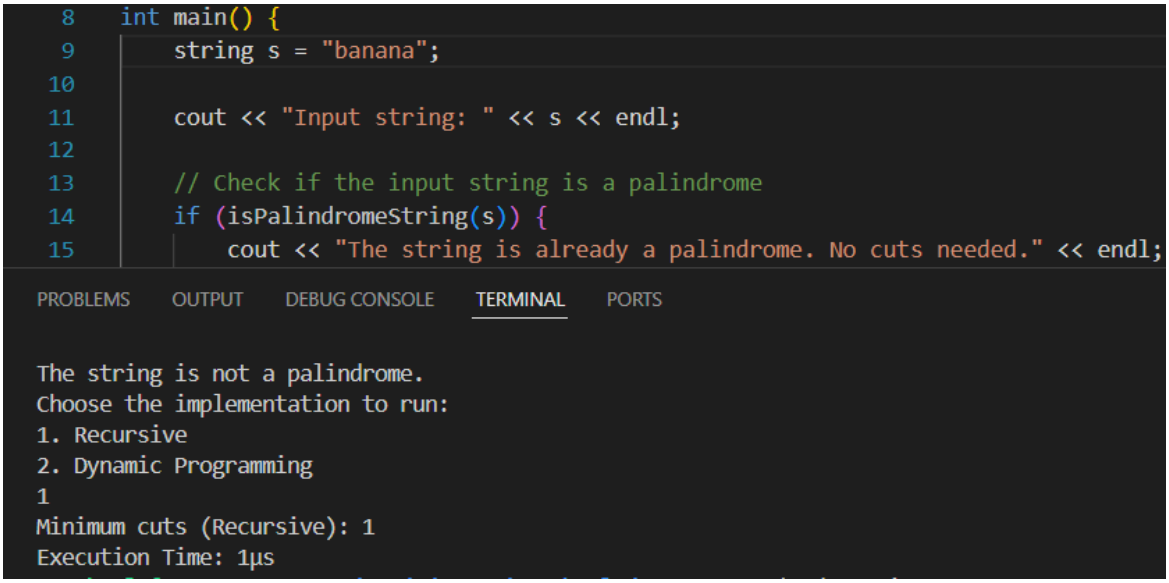
This string is not a palindrome, and it takes two cuts.

Use Case 3:

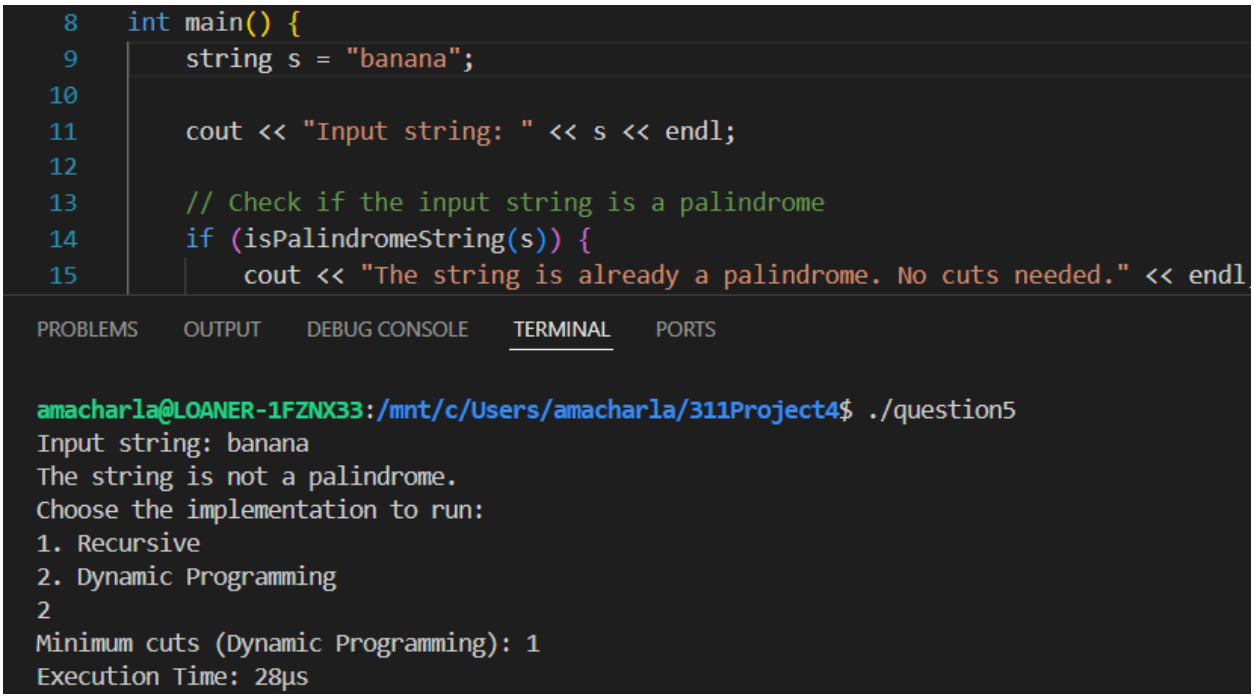
• Input: s = “banana”

• Output: 1 • Explanation: One solution is “b — anana”, requiring one cut.

Recursive approach:



Dynamic approach:



Explanation:

This string is not a palindrome, and it takes 1 cut.