Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Ex.No: 4 Date : 08.08.2024	Implementation of Rod cutting Problem with experiments on the analysis and the efficiency	Name : Venkatesan M Reg.No : 22BAI1259

Aim

We are given a rod of size 'N'. It can be cut into pieces. Each length of a piece has a particular price given by the price array. Our task is to find the maximum revenue that can be generated by selling the rod after cutting into pieces.

Algorithm

The dynamic programming approach to solving the rod-cutting problem involves breaking down the problem into simpler subproblems, solving each subproblem just once, and storing the solutions in a table for future reference. This eliminates redundant calculations and significantly reduces the time complexity.

Memoization Approach

Algorithm:

1. **Initialization:**
   - Create an array `re[]` of size `n+1`, initialized with `-1` to indicate that no subproblem has been solved yet.

2. **Recursive Function (`rodhelp`):**

    - Define a recursive function `rodhelp(n, val, re)` to calculate the maximum revenue for a rod of length `n`.
    - **Base Case:** If the length `n` is `0`, return `0` since no revenue can be obtained from a rod of zero length.
    - **Memoization Check:** If `re[n]` is not `-1`, return the stored value, indicating that the subproblem has already been solved.
    - **Recursive Case:**
        - Initialize a variable `maxre` to `INT_MIN` to store the maximum revenue for the current rod length.
        - Loop through all possible first cuts (`i` from `0` to `n-1`):
            - For each cut, calculate the revenue by adding the price of the first piece (`val[i]`) to the maximum revenue obtained from the remaining rod (`n-(i+1)`).
            - Update `maxre` with the maximum of the current value and the calculated revenue.
        - Store the result in `re[n]` to avoid re-computation.

3. **Function Call:**

    - Call the recursive function `rodhelp(n, val, re)` from `rodcut(n, val)` to initiate the process.

4. **Return Value:**

    - The result in `re[n]` gives the maximum revenue for a rod of length `n`.

Tabulation Approach

Algorithm:

1. **Initialization:**

    - Create an array `res[]` of size `n+1` to store the maximum revenue for each rod length from `0` to `n`.
    - Set `res[0]` to `0`, since no revenue can be obtained from a rod of zero length.

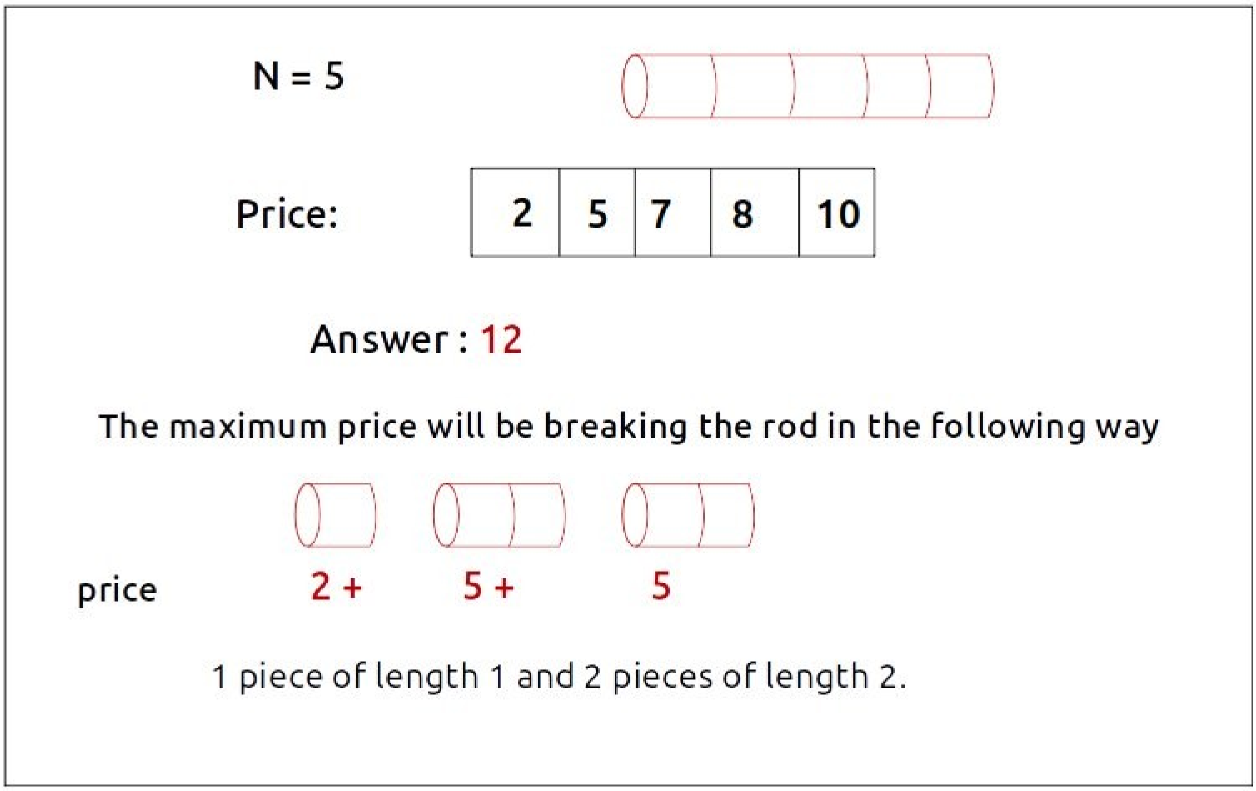
2. **Iterative Computation:**

    - For each length `i` from `1` to `n`, compute the maximum revenue possible:
        - Initialize `res[i]` to `INT_MIN` to find the maximum revenue for this length.
        - Loop through all possible first cuts (`j` from `0` to `i-1`):
            - For each cut, calculate the revenue by adding the price of the first piece (`val[j]`) to the maximum revenue obtained from the remaining rod (`i-(j+1)`).
            - Update `res[i]` with the maximum of the current value and the calculated revenue.

3. **Return Value:**

    - The value in `res[n]` represents the maximum revenue for a rod of length `n`.

Example



Source Code for Memoization Approach

```
#include <iostream>
#include <climits>
#include <iomanip>
using namespace std;

int rodhelp(int n, int val[], int re[]) {
    if (n == 0) {
        return 0;
    }
    if (re[n] != -1) {
        return re[n];
    }
    int maxre = INT_MIN;
    for (int i = 0; i < n; i++) {
        maxre = max(maxre, val[i] + rodhelp(n - (i + 1), val, re));
    }
    re[n] = maxre;
    return re[n];
}
int rodcut(int n, int val[]) {
    int re[n + 1];
    for (int i = 0; i <= n; i++) {
        re[i] = -1;
    }
    return rodhelp(n, val, re);
}
int main() {
    int n = 5;
    int ar[] = {2, 5, 7, 8, 10};
```

```
    cout << "Maximum Revenue: " << rodcut(n, ar) << endl;
    return 0;
}
```

Source Code for Tabulation Approach

```
#include <iostream>
#include <climits>
#include <iomanip>
using namespace std;

int rodcut(int n, int val[]) {
    int res[n + 1];
    res[0] = 0;

    for (int i = 1; i <= n; i++) {
        res[i] = INT_MIN;
        for (int j = 0; j < i; j++) {
            res[i] = max(res[i], val[j] + res[i - (j + 1)]);
        }
    }
    return res[n];
}

int main() {
    int n = 5;
    int ar[] = {2, 5, 7, 8, 10};
    cout << "Maximum Revenue: " << rodcut(n, ar) << endl;
    return 0;
}
```

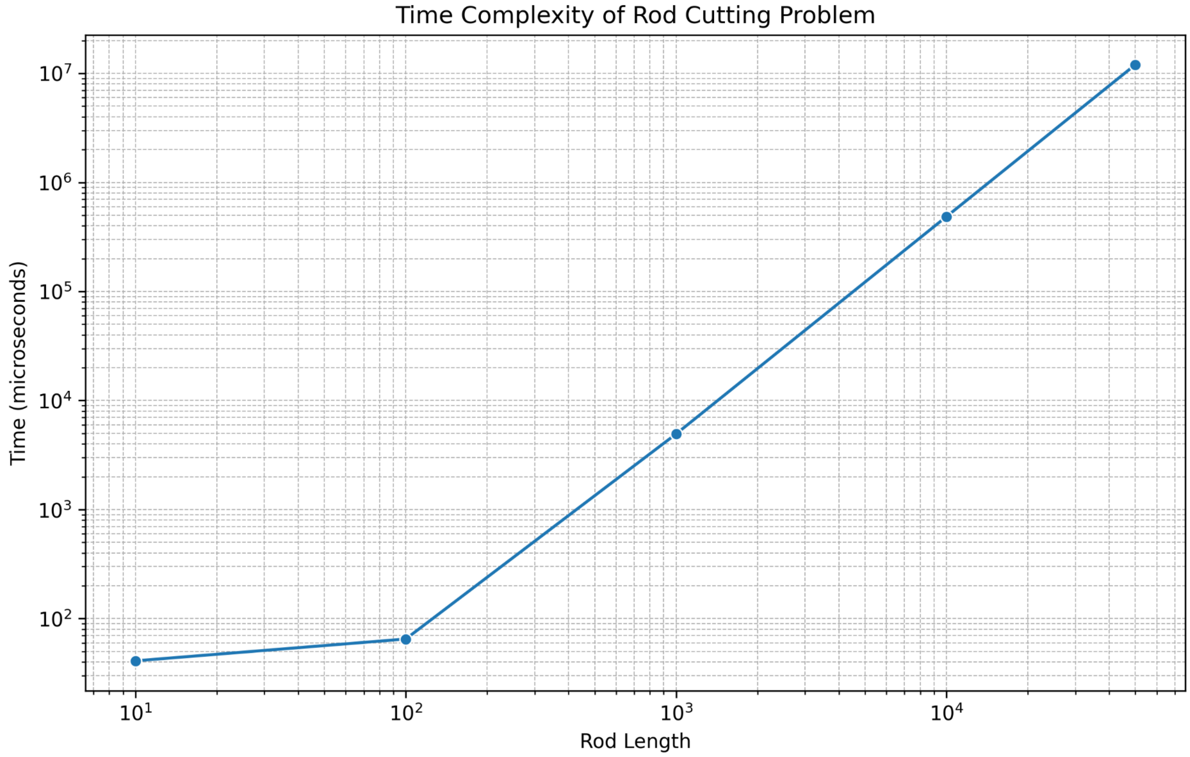
Source Code for Memoization Approach to measure time

```
int main() {
    vector<int> lengths = {10, 100, 1000, 10000, 50000};
    vector<int> times;

    for (int n : lengths) {
        vector<int> prices(n);
        for (int i = 0; i < n; i++) prices[i] = i + 1;

        auto start = high_resolution_clock::now();
        cout << "Max Revenue for rod length " << n << ": " << rodcut(n, prices) << endl;
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(stop - start);
        times.push_back(duration.count());
        cout << "Time taken: " << duration.count() << " microseconds" << endl;
    }

    return 0;
}
```

Output

```
Max Revenue for rod length 10: 10 Time taken: 41 microseconds
Max Revenue for rod length 100: 100 Time taken: 65 microseconds
Max Revenue for rod length 1000: 1000 Time taken: 4935 microseconds
Max Revenue for rod length 10000: 10000 Time taken: 485323 microseconds
Max Revenue for rod length 50000: 50000 Time taken: 12021903 microseconds
```

Graph between varying size of inputs and time



By observing the graph, it confirms an exponential time complexity. In this case, the time complexity of the rod cutting problem with memoization is typically $O(n^2)$, where n is the length of the rod

Complexity Analysis for Memoization Approach:

Time Complexity: $O(n^2)$because each length requires solving all sub-problems.
Space Complexity: $O(n)$ for storing intermediate results.

Source Code for Tabulation Approach to measure time

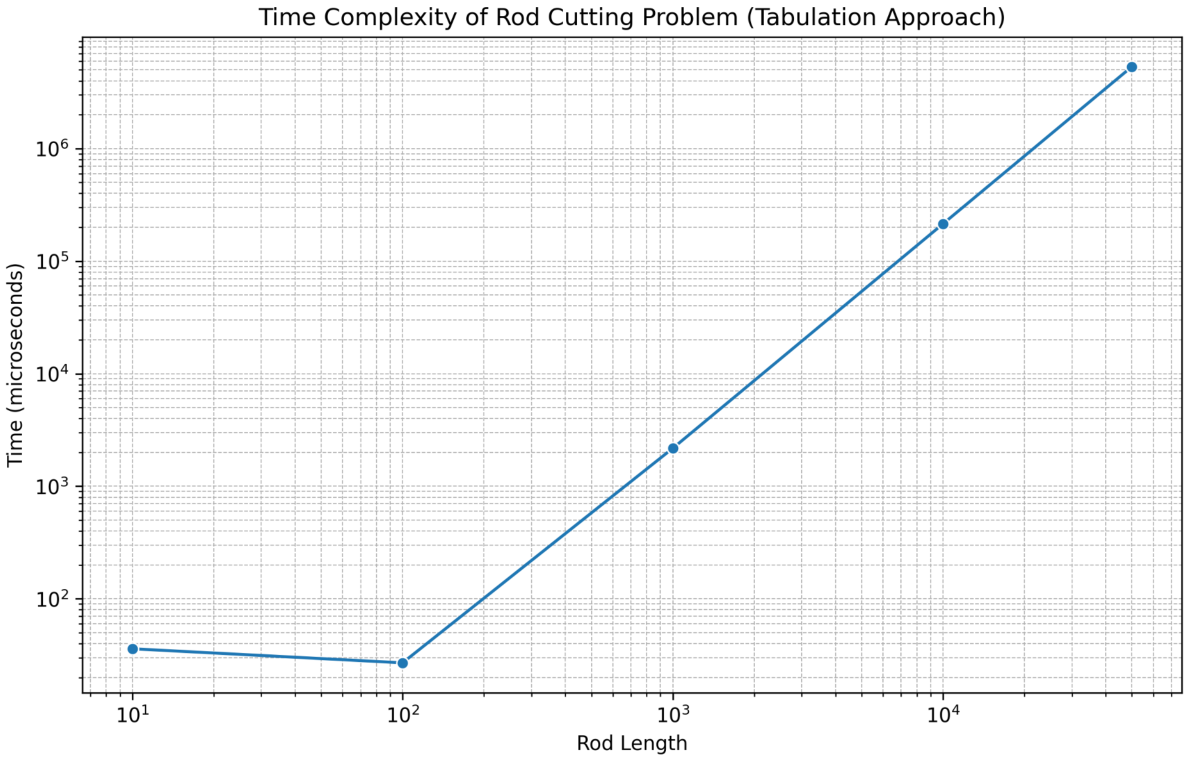
```
int main() {
    vector<int> lengths = {10, 100, 1000, 10000, 50000};
    vector<long long> times;
```

```
    for (int n : lengths) {
        int* prices = new int[n];
        for (int i = 0; i < n; i++) {
            prices[i] = i + 1;
        }

        auto start = high_resolution_clock::now();
        cout << "Maximum Revenue for rod length " << n << ": " << rodcut(n, prices) << endl;
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
        times.push_back(duration.count());
        cout << "Time taken: " << duration.count() << " microseconds" << endl;
        delete[] prices;
    }
    return 0;
}
```

Output

```
Maximum Revenue for rod length 10: 10 Time taken: 36 microseconds
Maximum Revenue for rod length 100: 100 Time taken: 27 microseconds
Maximum Revenue for rod length 1000: 1000 Time taken: 2175 microseconds
Maximum Revenue for rod length 10000: 10000 Time taken: 214356 microseconds
Maximum Revenue for rod length 50000: 50000 Time taken: 5339872 microseconds
```

Graph between varying size of inputs and time



The tabulation approach, while more efficient than a naive recursive approach, still shows exponential-like growth for large input sizes. the time complexity of this tabulation approach is generally considered $O(n^2)$

Complexity Analysis:

- **Time Complexity:** $O(n^2)$ for computing maximum revenue for each rod length.
- **Space Complexity:** $O(n)$ for the results table.

Results

- **Memoization Approach:** The function `rodhelp` is called recursively and computes the maximum revenue that can be obtained by cutting up the rod. It utilizes a memoization array `re[]` to store results of subproblems to prevent redundant calculations.

- **Tabulation Approach:** The function `rodcut` computes the maximum revenue in a bottom-up manner using a results array `res[]`. It iteratively calculates the best revenue for each possible rod length up to `n`.

Top-Down Approach (Memoization)

Uses recursion to solve the problem by breaking it into subproblems and stores the results in a memoization array to avoid redundant calculations. Easier to implement and understand, especially for problems with a natural recursive structure. May have higher memory usage due to recursion stack and additional space for memoization.

Bottom-Up Approach (Tabulation):

Iteratively solves the problem by filling out a table from the smallest subproblems to the largest, building up the solution incrementally. Generally more space-efficient, avoids recursive overhead, and often faster in practice. Can be more complex to implement and requires a predefined table size.