Homework 5

0811324 胡政煒

這次作業是利用 top-down 和 bottom-up 的方法，找出要切割成幾根 rod 才能得到最高和最低的價格。因此理論上會有四種主要的 function 產生，以下為四種主要程式碼:

MEMOIZED_CUT_ROD 和 MEMOIZED_CUT_ROD_AUX:

```
// 使用頂部向下的記憶化方法求解切割鋼條問題，最大化收益
// 參數：
//    - p: 切割方案價格的向量
//    - n: 需要切割的鋼條長度
//    - r: 傳遞記憶化數據的向量，存儲已計算的子問題解的收益
//    - s: 傳遞記憶化數據的向量，存儲每個子問題的最佳切割方案
int MEMOIZED_CUT_ROD_AUX(vector<int>& p, int n, vector<int>& r, vector<int>& s)
{
    // 如果已經計算過 r[n]，則直接返回該值，避免重複計算
    if (r[n] >= 0)
        return r[n];

    int q;
    if (n == 0)
        q = 0;
    else
    {
        q = INT_MIN;
        for (int i = 1; i <= n; i++)
        {
            // 遞迴調用 MEMOIZED_CUT_ROD_AUX 函數計算子問題的最優解
            // 將問題的規模減小，求解 n-i 長度的子問題的最優解 val
            int val = MEMOIZED_CUT_ROD_AUX(p, n - i, r, s);
            // 判斷當前切割方案的收益是否大於目前最大收益 q
            if (q < p[i - 1] + val)
            {
                // 更新最大收益 q
                q = p[i - 1] + val;
                // 更新 n 長度的最佳切割方案 s[n]
                s[n] = i;
            }
        }
    }
    // 將計算結果保存到記憶化數據中
    r[n] = q;
    return q;
}

// 使用頂部向下的記憶化方法求解切割鋼條問題，最大化收益
int MEMOIZED_CUT_ROD(vector<int>& p, int n, vector<int>& r, vector<int>& s)
{
    // 調用 MEMOIZED_CUT_ROD_AUX 函數求解切割鋼條問題，並返回最大化的收益
    return MEMOIZED_CUT_ROD_AUX(p, n, r, s);
}
```

這段程式碼使用頂部向下的記憶化方法求解切割鋼條問題，目標是最大化收益。

MEMOIZED_CUT_ROD_AUX 函數是一個輔助函數，用於計算子問題的最優解和記錄最佳切割方案。MEMOIZED_CUT_ROD 函數是主函數，MEMOIZED_CUT_ROD_AUX 函數來解決切割鋼條問題。

這些函數的輸入參數包括切割方案價格的向量 p、需要切割的鋼條長度 n，以及傳遞記憶化數據的向量 r 和 s。其中，r 向量用於存儲已計算的子問題解的收益，以避免重複計算，而 s 向量則用於存儲每個子問題的最佳切割方案。

在 MEMOIZED_CUT_ROD_AUX 函數中，首先檢查 r[n] 是否已計算過，如果是則直接返回該值。接下來，根據鋼條長度 n 的不同情況進行處理。如果 n 為 0，表示鋼條已經切割完畢，此時收益為 0。否則，使用迴圈遍歷各種切割方案，並遞迴調用 MEMOIZED_CUT_ROD_AUX 函數計算子問題的最優解，並更新最大收益 q 和最佳切割方案 s[n]。最後，將計算結果保存到記憶化數據中，並返回最大收益 q。

整體而言，這段程式碼以遞迴的方式解決 cutting rod 問題，並利用記憶化（Memoization）方法來避免重複計算，從而提高效率。透過 r 陣列，我們可以在需要時查找先前計算的結果，有效地記錄並重複使用子問題的解答。同時，s 陣列用於記錄每個子問題的最佳切割方案，以便在需要時能夠回溯並獲得完整的切割方案。這樣的設計使得我們能夠在求解過程中充分利用先前的計算結果，避免不必要的重複工作，從而提升整體效能。

MEMOIZED_CUT_ROD_MIN 和 MEMOIZED_CUT_ROD_MIN_AUX:

```
int MEMOIZED_CUT_ROD_MIN_AUX(vector<int>& p, int n, vector<int>& r, vector<int>& s)
{
    // 如果已經計算過 r[n]，則直接返回該值，避免重複計算
    if (r[n] <= 0)
        return r[n];

    int q;
    if (n == 0)
        q = 0;
    else
    {
        q = INT_MAX;
        for (int i = 1; i <= n; i++)
        {
            // 遞迴調用 MEMOIZED_CUT_ROD_MIN_AUX 函數計算子問題的最優解
            // 將問題的規模減小，求解 n-i 長度的子問題的最優解 val
            int val = MEMOIZED_CUT_ROD_MIN_AUX(p, n - i, r, s);
            // 判斷當前切割方案的成本是否小於目前最小成本 q
            if (q > p[i - 1] + val)
            {
                // 更新最小成本 q
                q = p[i - 1] + val;
                // 更新 n 長度的最佳切割方案 s[n]
                s[n] = i;
            }
        }
    }
    // 將計算結果保存到記憶化數據中
    r[n] = q;
    return q;
}

int MEMOIZED_CUT_ROD_MIN(vector<int>& p, int n, vector<int>& r, vector<int>& s)
{
    // 調用 MEMOIZED_CUT_ROD_MIN_AUX 函數求解切割鋼條問題，並返回最小化的成本
    return MEMOIZED_CUT_ROD_MIN_AUX(p, n, r, s);
}
```

這段程式碼使用頂部向下的記憶化方法求解切割鋼條問題，目標是最小化收益。

詳細程式碼內容與最大化收益類似，只是變成找出最小化收益。

Extended_Bottom_Up_Cut_Rod 和 Extended_Bottom_Up_Cut_Rod_MIN:

```c
// 使用底部向上的方法求解切割鋼條問題，最大化收益
int Extended_Bottom_Up_Cut_Rod(int p[], int n, int r[], int s[])
{
    int q;
    // 初始化 r[0] 和 s[0] 為 0，表示鋼條長度為 0 時的解
    r[0] = 0;
    s[0] = 0;
    for (int j = 1; j <= n; j++)
    {
        q = INT_MIN;
        for (int i = 0; i < j; i++)
        {
            // 遍歷各種切割方案，計算最大收益 q 和最佳切割方案 s[j]
            if (q < (p[i] + r[j - i - 1]))
            {
                q = p[i] + r[j - i - 1];
                s[j] = i + 1;
            }
        }
        // 將計算結果保存到 r[j] 中
        r[j] = q;
    }
    // 返回最大收益 q
    return q;
}
```

```c
// 使用底部向上的方法求解切割鋼條問題，最小化成本
int Extended_Bottom_Up_Cut_Rod_MIN(int p[], int n, int r[], int s[])
{
    int q;
    // 初始化 r[0] 和 s[0] 為 0，表示鋼條長度為 0 時的解
    r[0] = 0;
    s[0] = 0;
    for (int j = 1; j <= n; j++)
    {
        q = INT_MAX;
        for (int i = 0; i < j; i++) {
            // 遍歷各種切割方案，計算最小成本 q 和最佳切割方案 s[j]
            if (q > (p[i] + r[j - i - 1]))
            {
                q = p[i] + r[j - i - 1];
                s[j] = i + 1;
            }
        }
        // 將計算結果保存到 r[j] 中
        r[j] = q;
    }
    // 返回最小成本 q
    return q;
}
```

這兩個函數使用底部向上的方法來求解切割鋼條問題，分別用於最大化收益和最小化成本。這些函數遍歷鋼條的不同長度，並計算最大收益或最小成本以及相應的最佳切割方案。

在 Extended_Bottom_Up_Cut_Rod 函數中，首先將 r[0] 和 s[0] 初始化為 0，表示當鋼條長度為 0 時的解。然後使用兩個迴圈遍歷鋼條的不同長度，計算最大收益 q 和最佳切割方案 s[j]。最內層迴圈遍歷各種切割方案，並根據切割的位置 i 和剩餘鋼條的長度（j - i - 1）計算切割方案的收益。如果計算得到的收益 q 更大，則更新 q 和 s[j]。最後，將計算結果保存到 r[j] 中，表示鋼條長度為 j 時的最大收益。函數返回最大收益 q。

Extended_Bottom_Up_Cut_Rod_MIN 函數的邏輯和 Extended_Bottom_Up_Cut_Rod 函數相似，唯一的區別是在計算最小成本 q 和更新最佳切割方案 s[j] 時使用的比較運算符不同。最後，將計算結果保存到 r[j] 中，表示鋼條長度為 j 時的最小成本。函數返回最小成本 q。

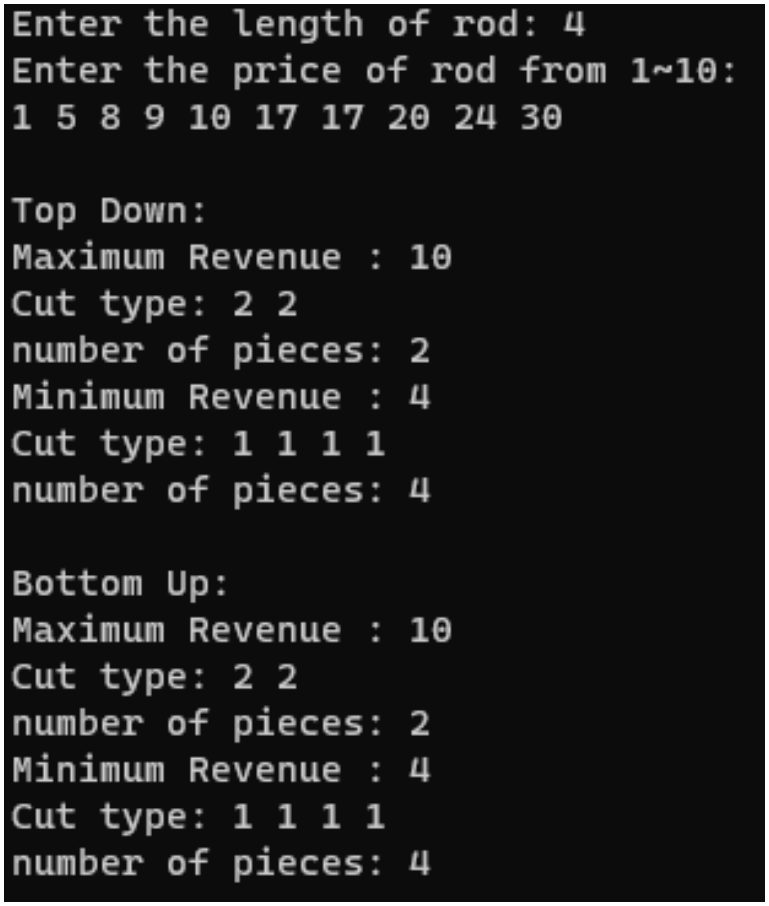
總體來說，這段程式碼使用迭代的方式、採用 bottom-up 方法求解 cutting rod 問題。它從鋼材長度為 0 開始，逐步計算出更大鋼材長度的最優解，同時記錄切割方案。最終，從 r 陣列中可以獲得整個鋼材的最大價值，並從 s 陣列中回溯得到最優的切割方案。相較於 Top-down 方法，這種 bottom-up 方法的優點在於避免了遞迴和記憶化的使用，以更高效的方式解決問題，因此具有更快的執行速度。

Print_Cut_Rod_Solution 和 Print_Cut_Rod_Solution_vec:

```
//    - s: 存儲每個子問題的最佳切割方案的陣列
//    - n: 需要切割的鋼條長度
void Print_Cut_Rod_Solution(int s[], int n)
{
    cout << "Cut type: ";
    int count = 0;
    while (n > 0)
    {
        // 輸出最佳切割方案
        cout << s[n] << " ";
        // 更新鋼條長度 n，繼續輸出切割方案
        n = n - s[n];
        // 統計切割的鋼條片數
        count++;
    }
    // 輸出切割片數
    cout << endl <<"number of pieces: " << count << endl;
}
```

```
//    - s: 存儲每個子問題的最佳切割方案的向量
//    - n: 需要切割的鋼條長度
void Print_Cut_Rod_Solution_vec(vector<int>& s, int n)
{
    cout << "Cut type: ";
    int count = 0;
    while (n > 0)
    {
        // 輸出最佳切割方案
        cout << s[n] << " ";
        // 更新鋼條長度 n，繼續輸出切割方案
        n = n - s[n];
        // 統計切割的鋼條片數
        count++;
    }
    // 輸出切割片數
    cout << endl <<"number of pieces: " << count << endl;
}
```

這兩個函數用於輸出切割鋼條的解答，一個接受陣列作為輸入，另一個接受向量作為輸入。



以上為輸出結果，利用 top-down 和 bottom-up 的方法，分別找出要切割成幾根 rod 才能得到最高和最低的價格

Top-down 和 Bottom-up 方法之間的詳細差異：

- 遞迴結構：

Top-down 方法使用遞迴結構，從原始問題開始，通過遞迴地解決較小的子問題，直到達到基本情況或者遞迴終止條件。
Bottom-up 方法則使用迭代結構，從基本情況開始，逐步計算和解決更大規模的子問題，直到解決原始問題。

- 計算順序：

Top-down 方法以自頂向下的方式計算和解決子問題，首先計算最頂層的子問題，然後根據需要遞迴地計算更小規模的子問題。
Bottom-up 方法以自底向上的方式計算和解決子問題，首先計算最基本的子問題，然後根據計算結果逐步計算和解決規模更大的子問題。

- 記憶化：

Top-down 方法通常使用記憶化技術，即使用數組或向量來存儲計算過的子問題的結果，以避免重複計算，提高效率。
Bottom-up 方法不需要額外的記憶化，因為它按照計算順序從小到大計算和解決子問題，並直接利用先前計算的結果。

- 時間複雜度：

Top-down 方法的時間複雜度通常取決於需要計算的子問題數量，如果沒有適當的記憶化，則可能存在大量的重複計算，導致指數級的時間複雜度。
Bottom-up 方法的時間複雜度通常取決於問題的規模，因為它按照順序計算子問題並存儲結果，避免了重複計算，因此具有較好的時間複雜度。

- 適用場景：

Top-down 方法適用於具有遞迴結構且需要從頂部開始解決子問題的情況。它提供了更自然的思考方式，易於理解和實現。
Bottom-up 方法適合用於問題結構可通過迭代的方式從基本情況到原始問題進行計算的情況。它不需要遞迴，並且可以在計算過程中直接使用先前計算的結果，從而提高效能。

然而，這兩種方法並非互斥，有時可以結合使用。例如，可以使用 Top-down 方法確定問題的結構和遞迴關係，然後使用 Bottom-up 方法進行實際的計算。這樣可以在保持清晰度和可讀性的同時，兼顧效能優化。