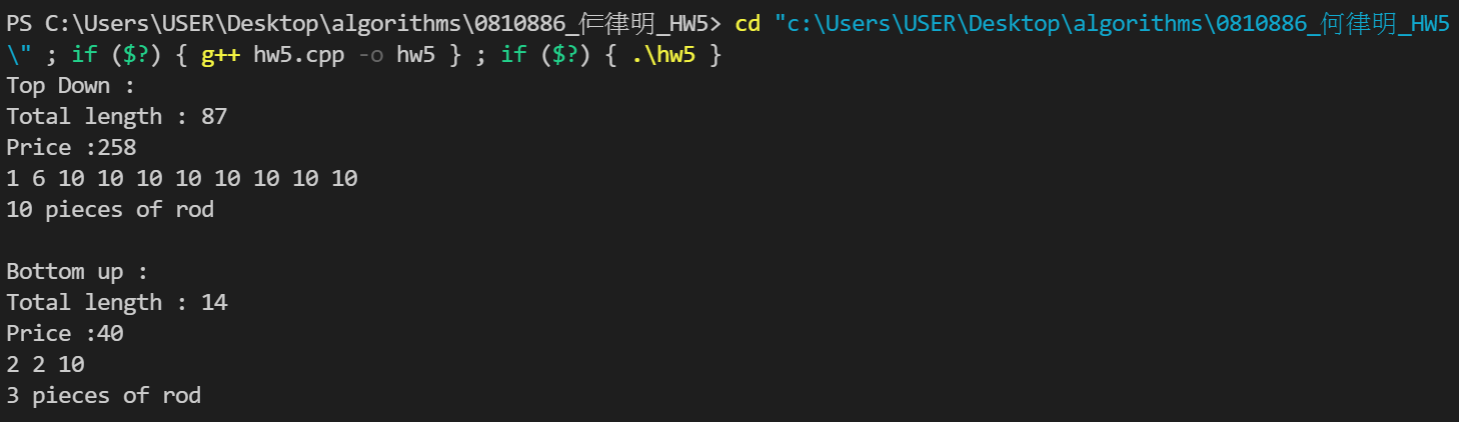
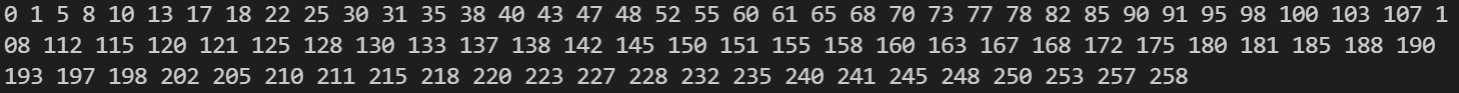
Result:



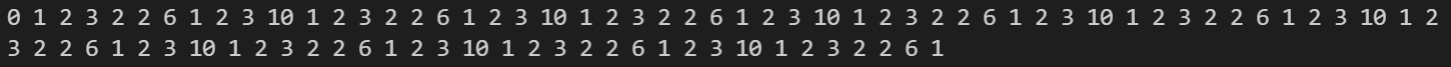
根據結果顯示長度n = 87時，最好的切法為切成1, 6, 10, 10, 10, 10, 10, 10, 10, 10，總共售出金額為1 + 17 + 30\*8 = 258(元)，至於當長度為n = 14時，最好的切法為切成2, 2, 10，總共售出金額為5\*2 + 30 = 40 (元)。我也試過了當n = 87時用Bottom up，以及當n = 14時用Top down，不出意料之外，答案也跟上圖一樣。如下圖:



而從n = 0至n = 87，各長度最佳銷售值如下(r):



從n = 0至n = 87，各長度之最佳切法的第一刀切的長度如下(s):



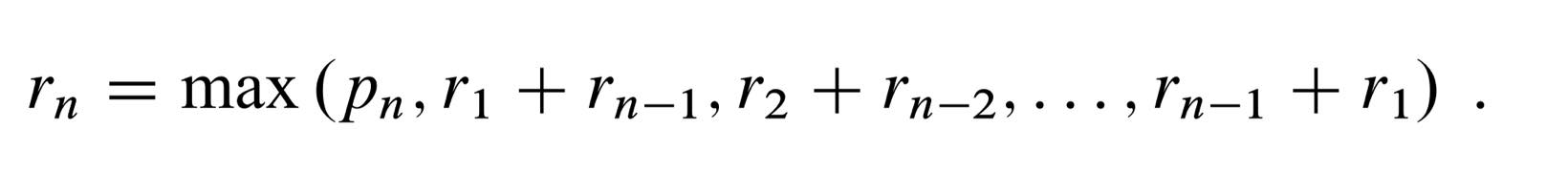
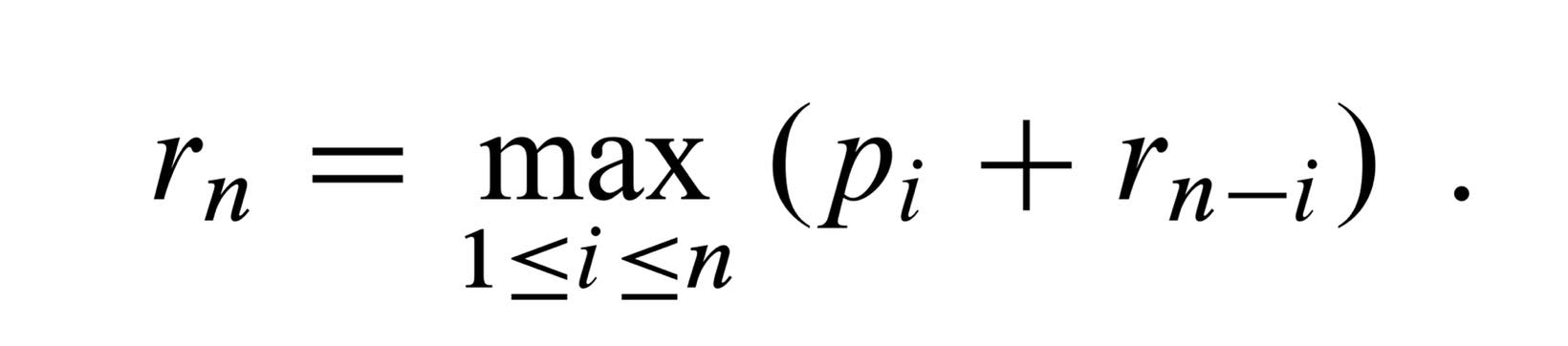
Analysis:

這一次的作業是要我們使用dynamic programming的方式實作，把最終的大問題拆解成小問題，而小問題也可以再更加拆解為小小問題，而大問題拆解成小問題的拆法跟小問題拆解成小小問題的拆法一模一樣。如果不用dynamic programming，用暴力解法去硬算，複雜度將為2的n次方，因為每單位長度都需要決定要切還是不切。而使用dynamic programming則只需要最多n平方的複雜度(因為Bottom up的pseudo code中只有兩層nested loop)。

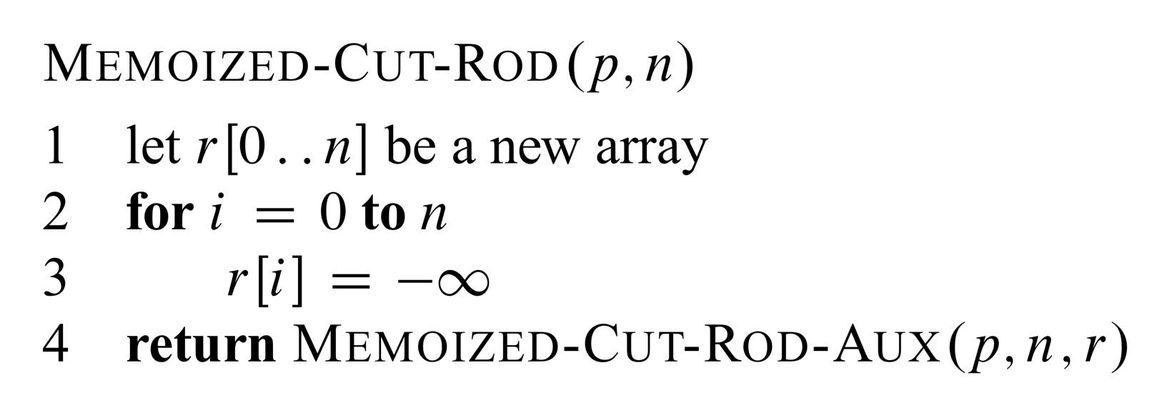
Top down的部分我們是法大問題用recursive的方法去拆解，雖然說很好理解，但因為大問題一直重複拆解成一樣的小問題，所以說比較沒有效率。我們也可以經由把解儲存在table(array)中來解決這個問題。

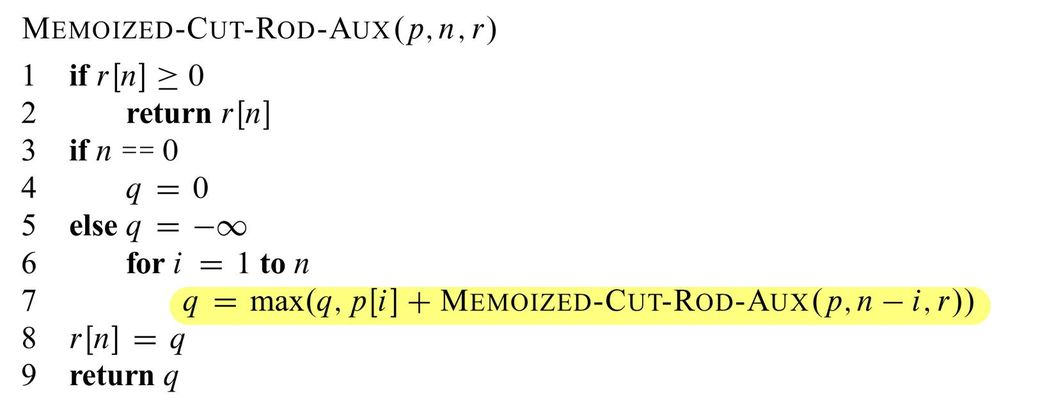
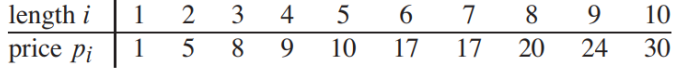
Bottom up則是先給定一個最小問題的初始條件，然後再一層一層計算出更大問題的解，並不需要用到recursive，而每一次的解都儲存在一個table(或是array)中，當下一次遇到同樣的問題的時候，只需要直接查表即可找出最佳解，並不需要再次用recursive的方法去計算下一個問題，所以說每一個subproblem都只也被計算過一次而已，也正因為如此，Bottom up比起一般的recursive Top down更有效率許多。

Thinking:

根據課本: 

我們只需要找出長度為n時切的最左邊那一刀即可，剩下的長度也可以套用在這個方法上。去比較每一組的最大值，此時的i即為下刀的長度，以下為課本Top down的pseudo code:



黃色螢光筆畫的部分P[i]的地方有可能會overflow因為Price table, P[i]只有給10個值，所以說當i大於等於10的時候會overflow不知道取到的是否為正確的值，所以我在我的程式中做了一點修改，如下:

int Memorized\_Cut\_Rod(int price[], int n, int r[], int s[]){

    r[0] = 0;

    for(int i=1 ; i<=n ; ++i){

        r[i] = INT\_MIN;

    }

    return Memorized\_Cut\_Rod\_Aux(price, n, r, s);

}

int Memorized\_Cut\_Rod\_Aux(int price[], int n, int r[], int s[]){

    int q;

    if(r[n] >= 0){

        return r[n];

    }

    else{

        q = INT\_MIN;

        for(int i=1 ; i<=n ; ++i){

            int tmp, m;

            if(i>10){

                tmp = r[i];

            }

            else

                tmp = price[i-1];

            m = Memorized\_Cut\_Rod\_Aux(price, n-i, r, s);

            if(q < tmp+m){

                q = tmp+m;

                s[n] = i;

            }

        }

    }

    r[n] = q;

    return q;

}

其中這一段:

            if(i>10){

                tmp = r[i];

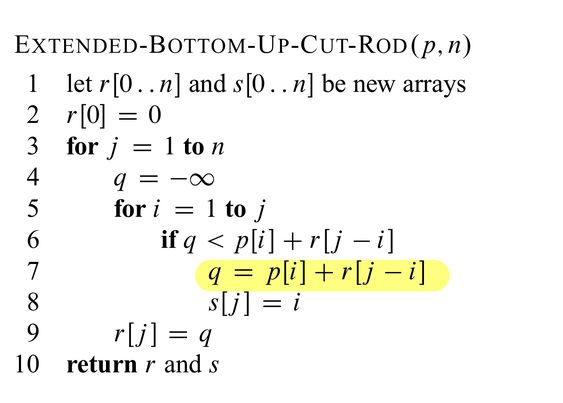
            }

            else

                tmp = price[i-1];

是為了避免overflow所以改出來的，當i大於10的時候，pseudo code原本採用的p[i]，會被先前算好的r[i]所取代掉並儲存在變數tmp中。

Bottom up:



黃色螢光筆畫的部分還是會出現overflow因為Price table, P[i]只還是有給10個值。

修正:

int Extended\_Bottom\_Up\_Cut\_Rod(int price[], int n, int r[], int s[]){

    r[0] = 0;

    for(int i=1 ; i<=n ; ++i){

        r[i] = INT\_MIN;

    }

    int q;

    for(int j=1 ; j<=n ; ++j){

        q = INT\_MIN;

        for(int i=1 ; i<=j ; ++i){

            int tmp;

            if(i>10)

                tmp = r[i];

            else

                tmp = price[i-1];

            if(q < tmp+r[j-i]){

                q = tmp+r[j-i];

                s[j] = i;

            }

        }

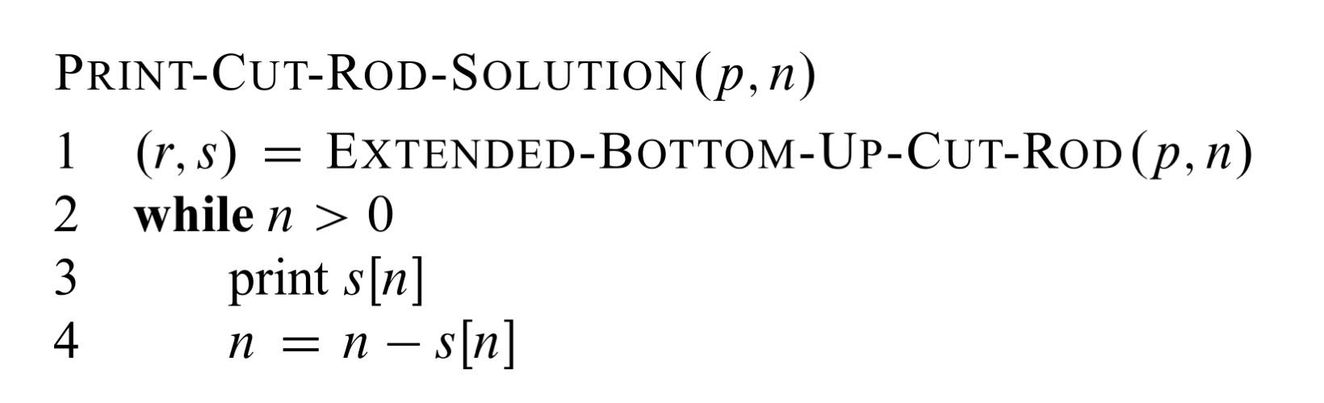
        r[j] = q;

    }

    return r[n];

}

修正部分跟Top down的Memorized\_Cut\_Rod\_Aux一樣，最後還有一個最終輸出的fucntion: Print\_Cut\_Rod\_Solution



int Print\_Cut\_Rod\_Solution(int n, int s[]){

    int time = 0;

    while(n>0){

        cout<<s[n]<<" ";

        ++time;

        n -= s[n];

    }

    cout<<endl;

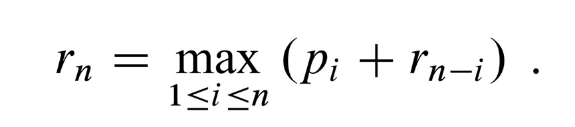
    return time;

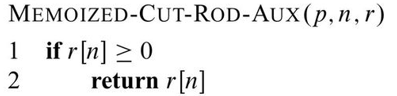
}

其中的time為我額外多加的，是用來計算Rod總共被切成幾段。

Discussion:

演算法根本的架構不變，problem還是可以拆解成更多的subproblems，原本中的max變成min而已，rn變成代表長度為n的最低價格，其他code中的部分也只需要把原來max的地方改成min，從原本的q < tmp + m改成q > tmp + m，還有一開始pseudo code中初始q為負無限大改成正無限大。還有以下:

這個部分改成

if r[n]<INT\_MAX

return r[n]

即可。