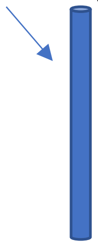
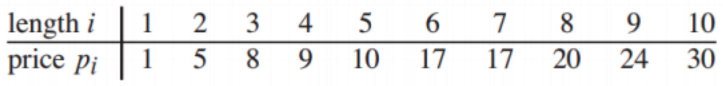
Rod Cutting Problem

Description/interpretation:

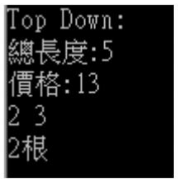
Given a rod of length n units, and the price of all pieces smaller than n, find the most profitable way of cutting the rod. Using Dynamic Programming describe the problem given a rod of length n units and the
Cut one rod and get two rods, which would recursively solve each one.
The result of the first cut yields: one piece to sell whole and a remainder,
The longest piece is to be sold ,while the remainder is recursively solved.
(this one adds an extra parameter, the longest piece we can sell).



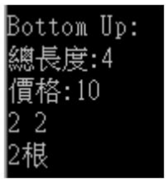


length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

:



```
Top Down:
總長度:5
價格:13
2 3
2根
```



```
Bottom Up:
總長度:4
價格:10
2 2
2根
```

We have two rods of length 89 and length of 14 units respectively.

We are asked to implement the code for **top-down** and **bottom-up** methods to find out how many rods can be cut to sell at the highest price.
Also need to explain lowest price computation process.

Introduction:

Rod cutting problem is one of the one dimensional programming problem.
In this report we provide the walkthrough of such an algorithm as well as details in the implement of our code.
Most importantly, while closely following pseudocodes, we add the important remarks concerning the improvement of the algorithm's efficiency and running cost.

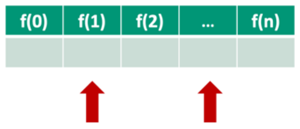
Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. The two main properties of a problem that suggests that the given problem can be solved using Dynamic programming are.

- Memorization (Top Down)
- Tabulation (Bottom Up)

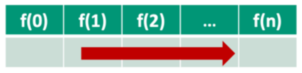
Rod cutting problem has overlapping subproblems and optimal substructures
Therefore it is suitable to be implemented using dynamic programming.
Furthermore, when the number of subproblems is polynomial, the time complexity is polynomial using dynamic programming.

Dynamic Programming algorithm:

Top-down: solves overlapping subproblems recursively with memorization.
Top-Down with Memoization
Solves recursively and proceed to memorise the subsolutions of the subproblems encountered this version of the algorithm encompasses the need to solve all subproblems as it involves memorization.



Bottom-up: builds up solutions to larger and larger subproblems.
Bottom-Up with Tabulation
Fills the table from the smallest index to the largest possible, in this version each small problem are solved individually.



Steps in view:

To achieve a reliable Dynamic program we know that the following must be satisfied:

- Get recursive solution
- Parameter analysis
    - Find how many distinct parameter combinations there are.
    - Determine if there’s few enough answers for each combination of parameters.
- Memorize
    - Allocate a table to hold stored answers.
    - Before running recursive code, check if we have computed answer
- Move to iterative version
    - For a given answer, what answers does it depend upon.
    - Figure out order for indices to fill answers after things they depend upon.
- Garnish
    - Can we reuse space? Optimize for space.
    - Find out if we need to store extra information for a constructive answer.

The problem that arises with the rod cutting algorithm is:
As we have a rod, should we sell the whole rod?
Or should we cut it ? if so then where should we cut it?

The first intuition that we got from those questions would be:

- If we decide to cut into two rods, then we have a problem just like the original problem stated above.
- We want to maximize how much we get for each of them to maximize how much we get for both of them combined. Indeed the fact that these subproblems will look just like the original problem is key for dynamic programing.
    - Two rods, recursively solve each one.
    - One piece to sell whole and a remainder, recursively solve the remainder
    - The longest piece to sell and a remainder, recursively solve the remainder.

    (this one adds an extra parameter, the longest piece can be sold).

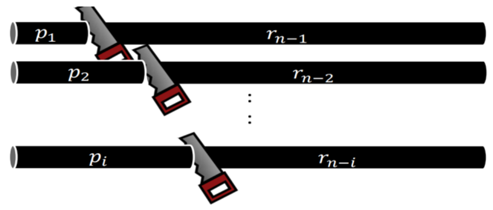
So as enumerated above we need to recursively solve for each one of given i inches rod
 (The whole rod is i inches long.)

Recursive procedure:

▪ Focus on the left-most cut.
▪ Assume that we always cut from left to right from the first cut.
▪When we get to a smaller problem we just assume that its solved same way, thinking conceptually of the base case which is original rod.

Optimal solution *Optimal solution subproblems*

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right)$$



naïve version:

Without caring about efficiency we can obtain the desired result by calling out the naive version of the algorithm by this line:

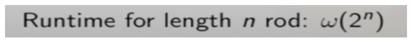
```
190:     Outcome = Naive_CUT_ROD(price,n);
```

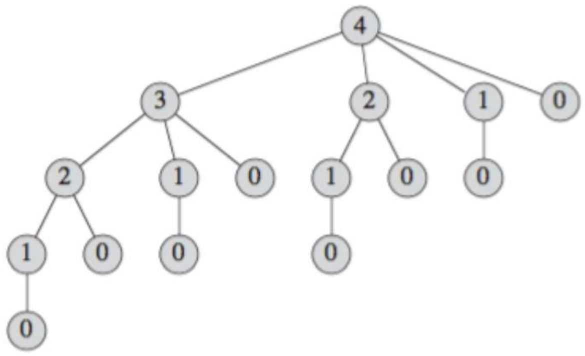
Time for running:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(1) + \sum_{i=0}^{n} T(n-i) & \text{if } n \geq 2 \end{cases} \Rightarrow T(n) = \Theta(2^n)$$

In the graph below, each nodes conceptually tells us the length of the rod, and it stores the best answer it gets from all the possibilities that strikes, when we finish putting the final value in the edge leading from the recursive call.
 Even in particularly small examples the algorithm solves a length two rods four times.
Therefore for a 4 inch rod we would get to the zero leaf in our tree 8 times!
Which yields for a rod of size n then there are 2^n-1 possibilities, because there are n − 1 places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut.

Thus the runtime can be derived as


Runtime for length n rod: $\omega(2^n)$



As can be noticed this version is calling overlapping subproblems which result in a relatively poor efficiency.
The time complexity is not ideal and in order to improve that some action has to be taken.
The bottom Up, and Top down method fit this profile.
The former is called through

```
    Outcome = BOTTOM_UP_CUT(price,n,size);
```

While the latter is called by:

```
    Outcome = TOP_DOWN_VERSION(price,n,size);
```

Top down version with memoization:

Memoization is a technique to avoid repeated computation on the same problems.
From the textbook it is mentioned, about the memoized version: before moving in perform any calculations looks at the table if it has already solved the best price for this height rod.
If so we don't need to recompute the answer from the beginning as before, just return.
If not we compute the answer but store it before returning.
Every node that isn't in the spine of the tree has its children pruned, because all the values get computed on that spine.
We store a linear number of answers, and our computation goes much quicker.
Much larger sizes gives less larger size span of the three than the naïve version of the same algorithm. Notably because we skip some steps in between thanks to the table.

Disclaimer:
The box or array we are using to store the values to be checked is filled to the left, because it directly makes recursive calls to all of the elements, so there is only one way to fill it:
From left to right.

Runtime is:

$$T(n) = \Theta(n^2)$$

Advantages:
▪ Better when some subproblems not be solved at all
▪ Solve only the required parts of subproblems

Its run time is much better than the exponential time we had before.
Since we are not looking for the price we can get but instead for the cost we need to get that price, it makes total sense that we tend toward the improved version of the algorithm.

Steps:

1. We store some information to reconstruct the cuts we need.
2. we make a second table and also stores which cut we use to get that amount.

This table gets updated whenever the price gets updated.

3. When we finish that table we can reconstruct the cuts we need to make.

In this instance for example if we have a rod length 8 then we should cut 2 and sell it, leaving us with 6.

4. The table then tells us that for 6 we need to cut 3 to sell.

And the remaining three also gets a size 3 rod to sell, leaving us with 0.

Bottom-Up with Tabulation:

Now that we understand what exactly the algorithm is about the need for efficiency drives us to look for more improvement in our code, therefore:
The bottom up method use the approach of solving the smaller subproblems first compared to the Top-down method it is better as the solved problems gets visited at least once, and outperform slightly the Top-down method for average case running time use.

Runtime:
$T(n) = \Theta(n^2)$

Advantages:
▪ Better when all subproblems must be solved at least once
▪ Typically outperform top-down method by a constant factor
▪ No overhead for recursive calls
▪ Less overhead for maintaining the table

Summary:

The size of the subproblem graph allows us to estimate the time complexity of the Ddynamic Pprogramming algorithm.
A graph illustrates the set of subproblems involved and how subproblems depend on another
A subproblem is conceptually designed to run only once
$|E|$: sum of #subsubproblems are needed for each subproblem
Time complexity:
linear to $O(E + V)$
Bottom-up: correspond to a Reverse Topological Sort
While
Top-down: performs a Depth First Search

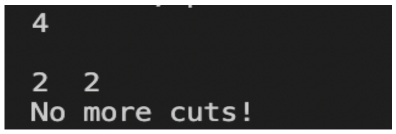
In overall we end up with a quadratic time, linear space algorithm that will be able to reconstruct the answer quickly in time linear in the number of rods of our optimal solution.
In summary the program has got a
Θ(n^2) runtime
Θ(n) space

Results:

Program summary:

As n varies, those are some screenshots of the results:
first line represent the number entered
second line is the location where we cut and the size
for maximum gain.
And so on, the rest is time computed
As n vary those are some values of the program:

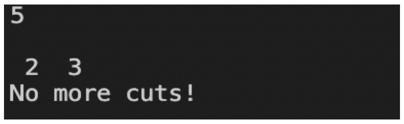
n=4



```
 4

 2  2
No more cuts!
```

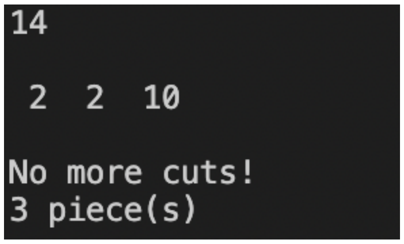
n=5



```
5

 2  3
No more cuts!
```

n=14

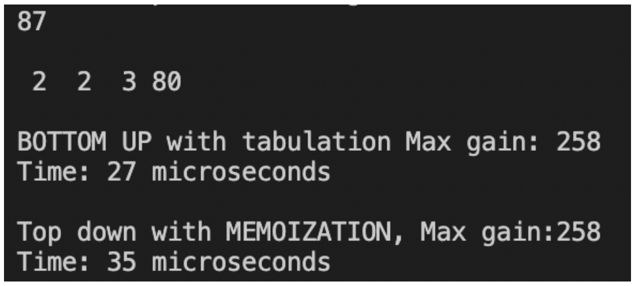


```
14

 2  2  10

No more cuts!
3 piece(s)
```

n=87



```
87

 2  2  3 80

BOTTOM UP with tabulation Max gain: 258
Time: 27 microseconds

Top down with MEMOIZATION, Max gain:258
Time: 35 microseconds
```

Sidenote: *The other versions are also implemented, works with n (0~10) for the sake of experimentation.*

Discussion:

For computing the lowest price:
The array we use in the program to store the prices only record the highest prices for now, assuming the lowest price is the first one that comes out, we can make some modification to the code to yield this result

1. Replace INIT_MIN by INT_MAX
2. And max by min

for each index the program will automatically add the lowest value to the array Prce as can be seen in the example below.

See voidCut function from (51~71) modified.

```
for (i = 1; i<=len; i++)
        {
         int k;
         int B = INT_MAX;
         for (j = 0; j < i && j<s; j++)
             {
              B = min(B, p[j] + Prce[i-j-1]);
              Prce[i] = B;
             }
        }
```

Reference:

https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture12.pdf
https://www.csie.ntu.edu.tw/~yvchen/f106-ada/doc/171012_DynamicProgramming.pdf
https://www.youtube.com/watch?v=re9rF9SqRFc
https://www.youtube.com/watch?v=oBt53YbR9Kk