CSE- 207
Algorithms

Lecture: 13
Dynamic Programming-III

Fahad Ahmed
Lecturer, Dept. of CSE
E-mail: fahadahmed@uap-bd.edu

Coin changing problem

Coin Changing Problem

The problem states that "*given a set of coins with several values, it is required to make a change using those coins for a particular amount of cents using the minimum number of coins*".

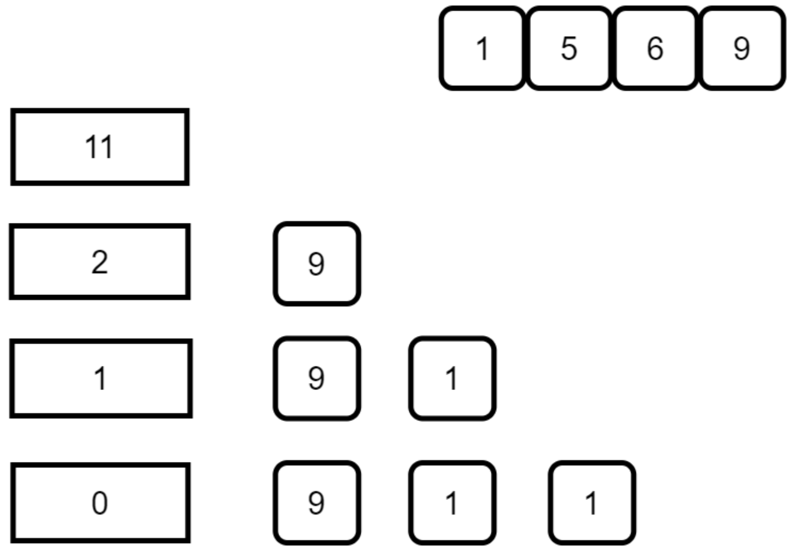
ISSUE WITH GREEDY ALGORITHM APPROACH

While the coin change problem can be solved using Greedy algorithm, there are scenarios in which it does not produce an optimal result.

Input:
coins[] = {9, 6, 5, 1}, V = 11

Output: Minimum 2 coins required

We can use one coin of 6 cents and 1 coin of 5 cents



Coin Change Problem

- But think about the following
  - Change 12 or 16 cent with {1,4,8,10} denomination
    - For 12, Greedy 10+1+1 whereas optimal is 8+4
    - For 16, Greedy 10+4+1+1 whereas optimal is 8+8
  - Change 30 with {1,10,25,50}
    - For 30, Greedy 25+1+1+1+1+1 whereas optimal is 10+10+10
  - Change 16 cent with {1,5,12,25}
    - For 16, Greedy 12+1+1+1+1 whereas optimal is 5+5+5+1

Coin-Change – (Similar to Knapsack)

Unbounded Knapsack	Coin Change
Size of item/array(N)	No of Coins in denomination (k)
Value of items (**Need to maximize**)	No of Coins (**Need to minimize**)
Weight of item array w[i])	Value of coin (v[i])
Maximum weight a Knapsack can carry	Coin needs to change (n)

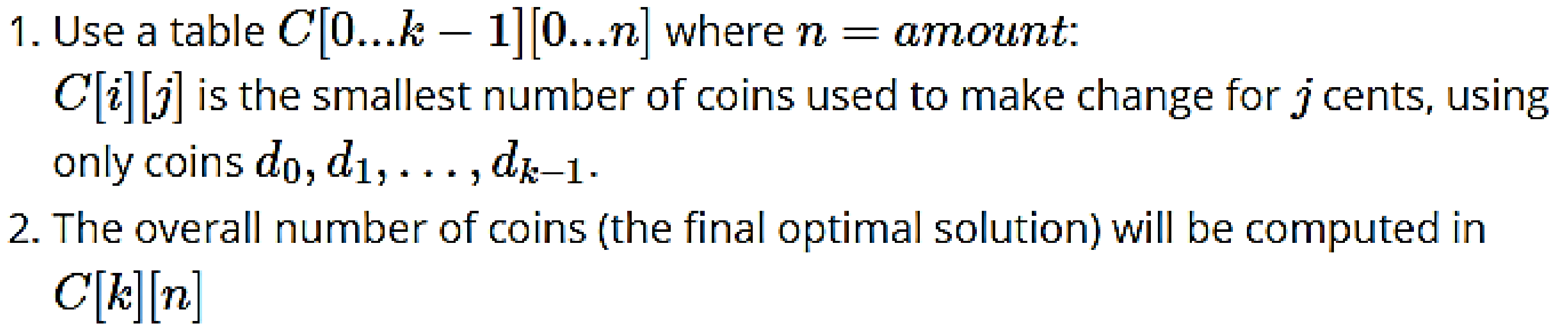
Main concept:

1. For each coin we have **two options**: **include it or exclude it.**

2. Take the one that **minimize** the no of coins.

Coin-Change – (Similar to Knapsack)

Using a table form (2D array).
Let $D=\{d_0,d_1,...,d_{k-1}\}$D={d0,d1,...,dk−1} be the set of coin denominations, arranged such that $d_0=1$ centd0=1 cent. As before, the problem is to make change for n cents using the fewest number of coins.



1. Use a table $C[0...k-1][0...n]$ where $n = amount$:
   $C[i][j]$ is the smallest number of coins used to make change for j cents, using only coins $d_0, d_1, \ldots, d_{k-1}$.
2. The overall number of coins (the final optimal solution) will be computed in $C[k][n]$

Coin-Change – (Similar to Knapsack)

The algorithm looks like this:

1. Making change for j cents with coins $d_0, d_1, \ldots, d_{k-1}$ can be done in two ways:
   1. Don't use coin d_i (even if it's possible):

$$C[i][j] = C[i-1][j]$$

   2. Use coin d_i and reduce the amount:

$$C[i][j] = 1 + C[i][j-d_i].$$

2. We will pick the solution with least number of coins:

$$C[i][j] = min(C[i-1][j], 1 + C[i][j-d_i])$$

Coin-Change – (Similar to Knapsack)

Assume we have to make change for 12 cents with denomination {1,4,8,10}

i	v_i	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	1	0												
2	4	0												
3	8	0												
4	10	0												

← Change

Coin-Change – (Similar to Knapsack)

Assume we have to make change for 12 cents with denomination {1,4,8,10}

i	v_i	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	0	1	2-	3	1	2	3	4	2	3	4	5	3
3	8	0	1	2	3	1	2	3	4	1	2	3	4	2
4	10	0	1	2	3	1	2	3	4	1	2	1	2	2

$\leftarrow$ Change

Coin-Change – (Similar to Knapsack)

Assume we have to make change for 12 cents with denomination {1,4,8,10}

i	v_i	0	1	2	3	4	5	6	7	8	9	10	11	12	← Change
0	0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	Not included
1	1	0	1	2	3	4	5	6	7	8	9	10	11	12	Not included
2	4	0	1	2-	3	1	2	3	4	2	3	4	5	3	**Included 1 time**
3	8	0	1	2	3	1	2	3	4	1	2	3	4	2	**Included 1 time**
4	10	0	1	2	3	1	2	3	4	1	2	1	2	2	Not included

What is the solution?

Using dynamic programming:

- **bottom up manner:**
  - re-computations of same subproblems can be avoided by constructing a temporary array **table[][]**

```
int minCoins(int coins[], int m, int V)
{
    int table[V+1];
    table[0] = 0;

    for (int i=1; i<=V; i++)
        table[i] = INT_MAX;

    for (int i=1; i<=V; i++)
    {
        for (int j=0; j<m; j++)
            if (coins[j] <= i)
            {
                int sub_res = table[i-coins[j]];
                table[i] = min(table[i], 1 + sub_res);
            }
    }
    if(table[V]==INT_MAX)
        return -1;

    return table[V];
}
```

m=number of different coins
V= changing value

table[i] will be storing the minimum number of coins required for i value

Initialize all table values as Infinite

Compute minimum coins required for all values from 1 to V

Add min coins in the table

The time complexity of the above solution is O(m*V).

Reference

- Chapter 15 (15.1 and 15.3) (Cormen)
- http://www.shafaetsplanet.com/?p=3638
- https://www.geeksforgeeks.org/coin-change-dp-7/
- https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/
- https://www.bogotobogo.com/Algorithms/dynamic_programming.php

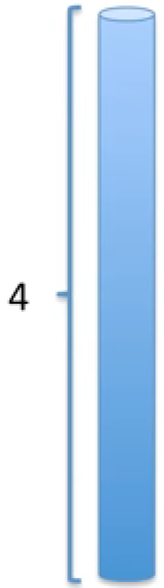
Rod-cutting problem

Rod cutting problem

- Given
  - a rod of length n inches and
  - an array of prices that contains prices of all pieces of size smaller than n.
- Determine
  - the **maximum** value obtainable by **cutting** up the rod and selling the pieces.

Example

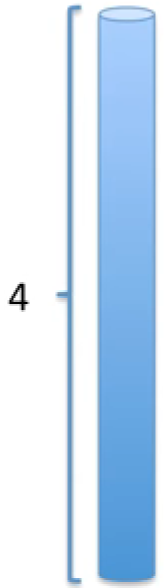
Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30



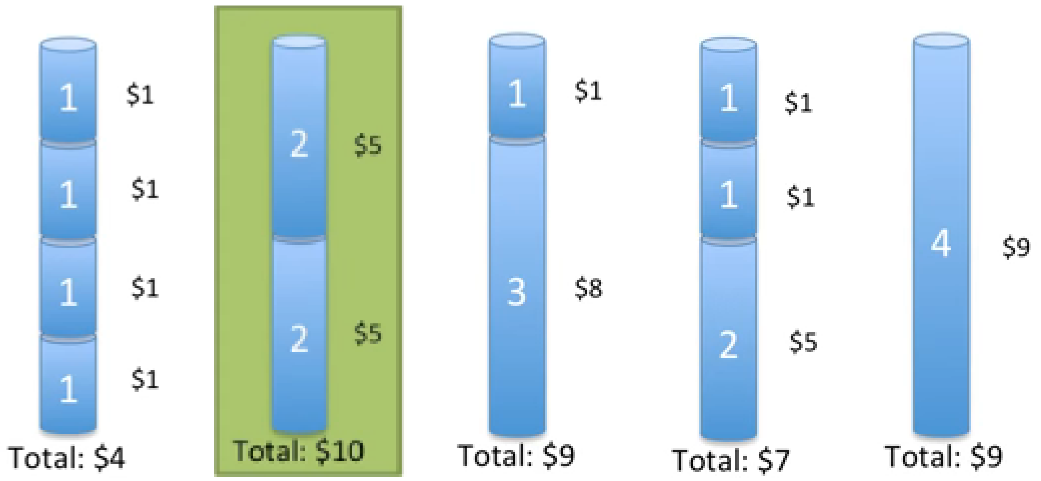
- Assume you have a rod of length 4 inch.
- A table that shows the price of all different length of rod.
- How can we cut the rod so that we can maximize the profit?

Example cont..

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30



Possible combinations:



How to solve

- Brute force
  - Try all possible combination- 2^{n-1} combination

- Dynamic programming
  - Time complexity – n^2

Brute Force approach

The basic idea is that given a length N, the maximum profit is the maximum of the following:

price[i] + max_price(N - i)

which means:
The rod of length N is being split into a part of length i which has price as price[i] and the rest of the rod of length N-i is split further which is captured by the recursive function max_price(N-i).

You need to check for all possible values of i.

Brute Force

```
$\underline{Cut - Rod(price, len):}$
    $if\ \ len == 0\ return\ 0;$
    $set: max\_len = -\infty;$
    $for\ i = 1\ to\ len$
        $max\_len = \max(max\_len, price[i] + Cut - Rod(price, len - i));$
    $return\ \ max\_len;$
```

- Complexity: $\Theta(2^n)$

Brute Force approach

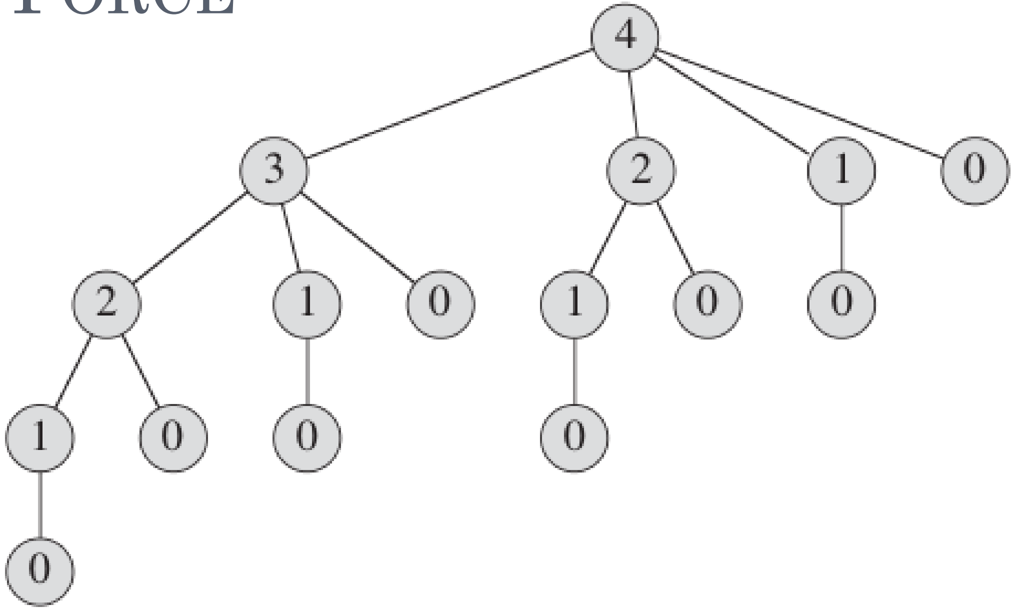
```
#include<bits/stdc++.h>
using namespace std;
int rodCutting(int price[], int len)
{
    if(len<=0)
        return 0;
    int max_len=INT_MIN;
    for(int i=0; i<len; i++)
    {
        max_len = max(max_len, price[i] + rodCutting(price, len-(i+1)));
    }
    return max_len;
}

int main(){
   int price[10] = {1,5,8,9,10,17,17,20,24,30};
   int rod_len=4;
   cout<<rodCutting(price,rod_len,10)<<'\n";
}
```

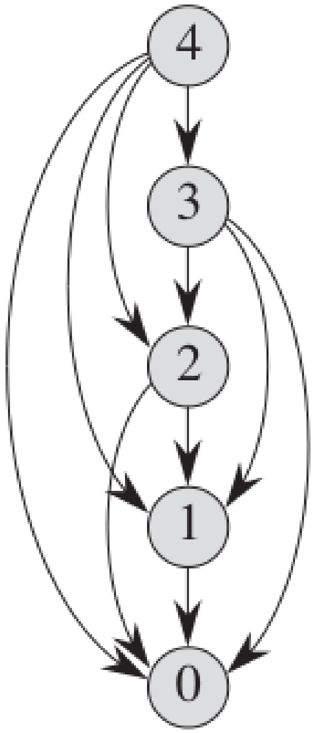
Brute Force

The variable max_len is initialized to minimum value possible. Then, we run a loop until the given rod length (in this case 4), to find the maximum value between the previous max_len and the value obtained by adding the current price and result of recursively calling the function for len-i+1 value.

For rod length 4, there are 2^(3) i.e 8 ways of cutting it, we can cut it in (3+1), (2+2), (1+1+1+1)....ways.

BRUTE FORCE





The recursion tree shows a recursive call resulting from rodCutting(price,4).
This figure clearly explains why the computation time for the algorithm is so ridiculous. We keep calling the function again and again even though the result has already been calculated.

For example **rodCutting(1) has been calculated 4 times.** In order to avoid that **we use dynamic programming**.

ROD CUTTING USING DP

The idea is same as the previous approach with the only difference that the values of **max_price(N-i)** in:

price[i] + max_price(N - i)

are **precomputed** and stored in an array.

This removes all duplicate calls and optimizes our solution greatly.

DP SOLUTION- 1 INCH ROD

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price										

$$r_1 = p_1 = 1 \quad (\textit{base case. No cut possible})$$

$$r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$

DP SOLUTION- 2 INCH ROD

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1									

$$r_2 = \max \begin{cases} p_1 + r_1 = 1 + 1 = 2 \\ p_2 = 5 \end{cases}$$

$$= 5$$

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

DP solution- 3 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5								

$$r_3 = \max \begin{cases} p_1 + r_2 = 1 + 5 = 6 \\ p_2 + r_1 = 5 + 1 = 6 \\ p_3 = 8 \end{cases}$$

$$= 8$$

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

DP SOLUTION- 4 INCH ROD

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5	8							

$$r_4 = \max \begin{cases} p_1 + r_3 = 1 + 8 = 9 \\ p_2 + r_2 = 5 + 5 = 10 \\ p_3 + r_1 = 8 + 1 = 9 \\ p_4 = 9 \end{cases}$$

$$= 10$$

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

DP SOLUTION- 5 INCH ROD

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5	8	10						

$$r_5 = \max \begin{cases} p_1 + r_4 = 1 + 10 = 11 \\ p_2 + r_3 = 5 + 8 = 13 \\ p_3 + r_2 = 8 + 5 = 13 \\ p_4 + r_1 = 9 + 1 = 10 \\ p_5 = 10 \end{cases}$$

$$= 13$$

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

DP solution- 6 to10 inch rod

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30
Opt. price	1	5	8	10	13	17	18	22	25	30

- With similar calculation we can fill out the remaining table
  - r_6 -> 17 (no cut)
  - r_7 -> 18 (1+6 or 2+2+3)
  - r_8 -> 22 (2+6)
  - r_9 -> 25 (3+6)
  - r_{10} -> 30 (no cut)

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

ROD CUTTING USING DP: BOTTOM UP MANNER

```
int rodCutting(int price[], int len)
{
    int val[len+1];
    val[0]=0;
    int i,j;
    for (i = 1; i<=len; i++)
    {
        int max_val = INT_MIN;
        for (j = 0; j < i; j++)
            max_val = max(max_val, price[j] + val[i-j-1]);
        val[i] = max_val;
    }
    return val[len];
}
```

Time Complexity: O(n^2)

This time complexity is till better than the worse time complexity of the brute force approach.

Rod Cutting Using DP: bottom up manner

For filling this table we run a nested loop to calculate val[i], where i represents the length of the rod, and the val[i] contains the optimal revenue that would be obtained if the rod were to be cut in i pieces.

This table solves the overlapping sub-problems part very quick, as we already save the intermediate results, we don't have to re-calculate it.

Rod Cutting Using DP: bottom up manner

max_val = max(max_val, price[j] + val[i-j-1])

The price[j] + val[i-j-1] simply calculates the current price plus the previous(already calculated optimum value)value from the val[] array.

The max_val calculation is done by comparing the maximum value between previous max_val and the current price[j] + val[i-j-1].

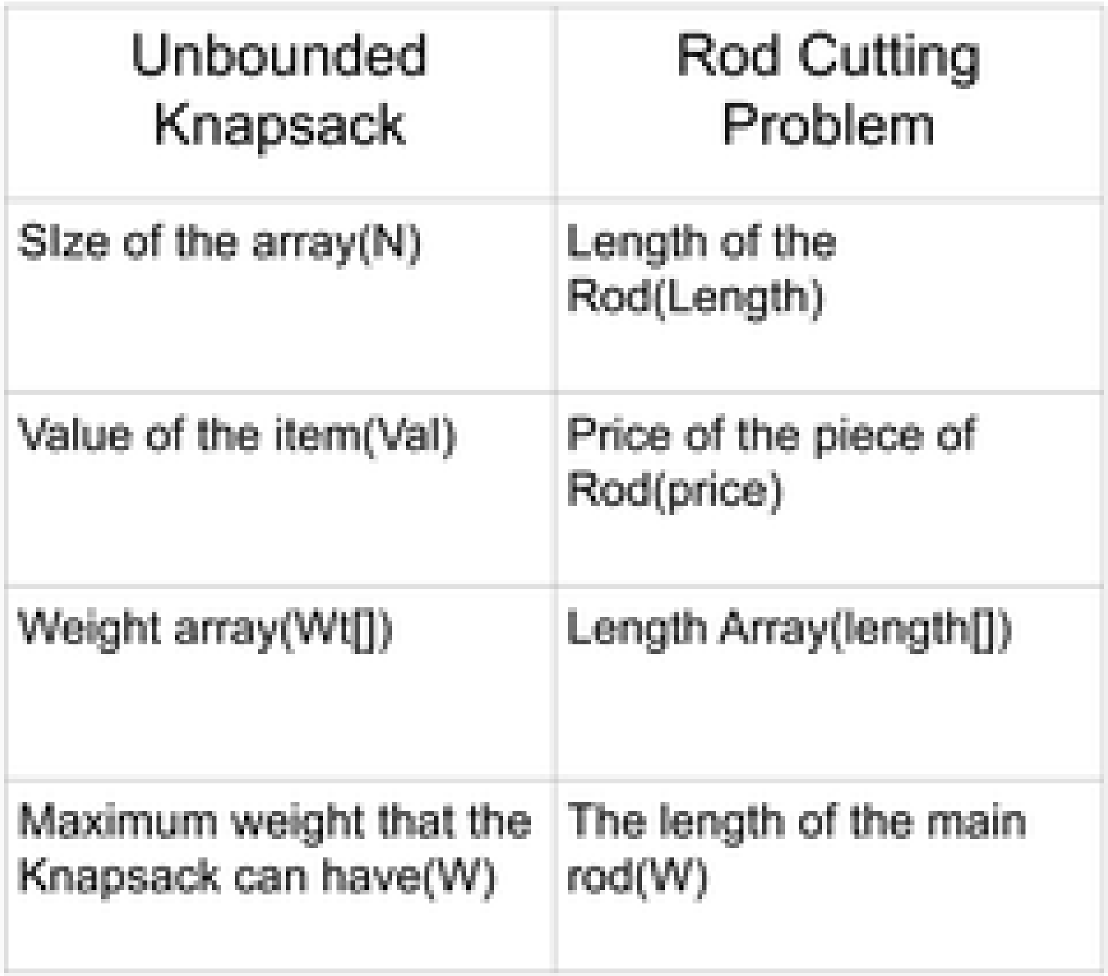
- For val[1] we run the j loop from 0 to 1, counting in all the 2^(1-1) possibilities.
- For val[2] we run the j loop from 0 to 2, counting in all 2^(2-1) possibilities i.e. (1+1), (2). since we already computed val[1] we won't be doing it again.
- For val[3] we build the solution in the same way by considering 2^(3-1) approaches(1+2),(1+1+1),(2+1),(3). Unlike the previous method, since we already have those values (val[1], val[2]) we won't be calculating again

ROD CUTTING USING DP: BOTTOM UP MANNER

The values and their respective implementation of our memoization table i.e val[] is given as follows

- val[0]=0 //a hypothetical situation, there isn't really a rod of length 0
- val[1]=1 //maximum revenue if the rod is of length 1, there isn't much calculation as there is only one answer for a rod length of 1
- val[2]=5//we need to find max of the total 2^(1) possibilities, the only difference here is instead of recalculating we use the values that are already existing.
- val[3]=8 // calculated in the same way by taking in all the 4 possibilities and returning 8
- val[4]=10 // calculation is done in the same way, as the previous

Rod-Cutting – (Similar to Knapsack)



Unbounded Knapsack	Rod Cutting Problem
SIze of the array(N)	Length of the Rod(Length)
Value of the item(Val)	Price of the piece of Rod(price)
Weight array(Wt[])	Length Array(length[])
Maximum weight that the Knapsack can have(W)	The length of the main rod(W)

Rod-Cutting – (Similar to Knapsack)

```
if (w[i]>c)
  T[ i, c ] = T[i-1,c]
else
  T[i, c] = max( T[i − 1, c], v[i] + T[i,c−w[i]])
```

v_i	i	0	1	2	3	4
0	0	0	0	0	0	0
1	1	0	1	2	3	4
5	2	0	1	5	6	10
8	3	0	1	5	8	10
9	4	0	1	5	8	10

Rod-Cutting – (Similar to Knapsack)

```
if (w[i]>c)
    $T[i, c] = T[i-1,c]$
else
    $T[i, c] = \max(T[i-1, c], v[i] + T[i,c-w[i]])$
```

v_i	L[i]	0	1	2	3	4	
0	0	0	0	0	0	0	Not part of solution
1	1	0	1	2	3	4	Not part of solution
5	2	0	1	5	6	10	**Included 2 times**
8	3	0	1	5	8	10	Not part of solution
9	4	0	1	5	8	10	Not part of solution

Reference

- Chapter 15 (15.1 and 15.3) (Cormen)
- https://www.geeksforgeeks.org/cutting-a-rod-dp-13/
- https://www.geeksforgeeks.org/c-program-for-cutting-a-rod-dp-13/
- https://www.codesdope.com/course/algorithms-rod-cutting/