Practice problem : Dynamic Programming

Task 1: Rod cutting problem

A company buys long steel rods (of length n), and cuts them into shorter one to sell. Here,

- lengths are integers only
- cutting is free
- rods of diff lengths sold for diff. price, e.g.,

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

What is the best way to cut the rods?
- n=4: no cutting: $9, 1 and 3: 1+8=$9, 2 and 2: 5+5=$10. So, 2 and 2 is be the best.
- Now how about n=5: ?

Explanation and solution have been discussed in theory class. You can refer to the slide included.

Brute force recursion algorithm:

```
Cut_Rod(p, n) {
    // base case
    if n == 0
        return 0
    // recursive case
    q = -∞
    for i = 1 to n {
        q = max(q, p[i] + Cut-Rod(p, n - i))
    }
    return q
}
```

But this is a $O(2^n)$ solution because of repetitive subproblems. To avoid solving repetitive subproblems we use memoization. Add a global array named *r* to keep the solution of the subproblems.

Recursion with memoization algorithm:

```
init_table(n) {
    // initialize memo (an array r[] to keep max revenue)
    r[0] = 0
    for i = 1 to n
        r[i] = -∞ // r[i] = max revenue for rod with length=i
}

Memoized_Cut_Rod (p, n) {
    if n == 0
        return r[n] // return the saved solution
    q = -∞
```

```
    for i = 1 to n {
        if r[n-i] == -∞ {
            r[n-i] = Memoized-Cut-Rod (p, n-i)
        }
        q = max(q, p[i] +r[n-i])
    }
    r[n] = q // update memo
    return r[n]
}
```

Dynamic programming algorithm:

```
DP_Cut-Rod(p, n) {
    r[0] = 0
    for j = 1 to n { // compute r[1], r[2], ... in order
        q = -∞
        for i = 1 to j {
            q = max(q, p[i] + r[j - i])
        }
        r[j] = q
    }
    return r[n]
}
```

Now, you may want to know how the pieces were cut to produce the maximum revenue. For this we have to extend the program. Add another global array named *cut* to keep the cutting positions.

```
Extended_DP_Cut_Rod(p, n) {
    r[0] = 0
    for j = 1 to n { //compute r[1], r[2], ... in order
    q = -∞
        for i = 1 to j {
            if q < p[i] + r[j - i] {
                q = p[i] + r[j - i]
                cut[j] = i // the best first cut for len j rod
            }
        }
        r[j] = q
    }
    return r[n]
}
```

Finally print the solution using the following function:

```
Print_Cut_Rod_Solution(p, n) {
    i = n
    while i > 0 {
        print cut[i]
        i = i – cut[i] // remove the first piece
    }
}
```

Task 2: 0/1 knapsack problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Explanation and solution have been discussed in theory class.

Brute force recursion algorithm:

```
knapsack(W, v, wt, n) {
    // base case
    if n == 0 || W == 0
        return 0
    // recursive case
    if W - wt[n] < 0 {
        q = knapsack(W, v, wt, n-1)
    }
    Else {
        a = v[n] + knapsack(W- wt[n], v, wt, n-1)
        b = knapsack(W, v, wt, n-1)
        q = max(a, b)
    }
    return q
}
```

But this is a $O(2^n)$ solution because of repetitive subproblems. To avoid solving repetitive subproblems we use memoization. Add a global 2-D array named *table* to keep the solution of the subproblems.

Recursion with memoization algorithm:

```
init_table(n, W) {
    // initialize memo (a 2 -D array to keep max profit)
    for i = 1 to n {
        table[i][0] = 0
    }
    for i = 0 to W {
        table[0][i] = 0
    }

    for i = 1 to n {
        for j = 1 to W {
            table[i][j] = -∞
        }
    }
}
```

```
Memoized_knapsack(W, v, wt, n) {
    if (n == 0 || W == 0)
        return table[n][W] // return the saved solution

    if W - wt[n] < 0 {
        if table[n-1][W] == -∞ {
            table[n-1][W] = knapsack(W, v, wt, n-1)
        }
        q = table[n-1][W]
    }
    else {
        if table[n-1][W] == -∞ {
            table[n-1][W] = knapsack(W, v, wt, n-1)
        }
        if table[n-1][W-wt[n]] == -∞ {
            table[n-1][W-wt[n-1]] = knapsack(W-wt[n], v, wt, n-1)
        }
        a = v[n] + table[n-1][W-wt[n]]
        b = table[n-1][W]
        q = max(a, b)
    }
    table[n][W] = q // update memo
    return table[n][W]
}
```

Dynamic programming algorithm:

```
DP_knapsack(W, v, wt, n){
    for i = 1 to n {
        table[i][0] = 0
    }
    for i = 0 to W {
        table[0][i] = 0
    }
    for i = 1 to n {
        for j = 1 to W {
            if j – wt[i] < 0 {
                table[i][j] = table[i-1][j]
            }
            else {
                a = v[i] + table[i-1][j – wt[i]]
                b = table[i-1][j]
                table[i][j] = max(a, b)
        }
    }
    return table[n][W]
}
```

Now, you may want to know how the pieces were cut to produce the maximum revenue. For this we have to extend the program. Add another global array named *choice* to keep the cutting positions.

```
Extended_DP_knapsack(W, v, wt, n){
    for i = 1 to n {
        table[i][0] = 0
    }
    for i = 0 to W {
        table[0][i] = 0
    }
    for i = 1 to n {
        for j = 1 to W {
            if j – wt[i] < 0 {
                table[i][j] = table[i-1][j]
                choice[i][j] = 0
            }
            else {
                a = v[i] + table[i-1][j – wt[i]]
                b = table[i-1][j]
                if a > b {
                    table[i][j] = a
                    choice[i][j] = 1
                }
                else {
                    table[i][j] = b
                    choice[i][j] = 0
                }
        }
    }
    return table[n][W]
}
```

Finally print the solution using the following function:

```
Print_knapsack_solution(W, v[], wt[], n) {
    i = n
    j = W
    while i > 0 {
        if choice[i][j] == 1 {
            print “item i taken”
            j = j – wt[i]
            i = i - 1
        }
        else {
            print “item i not taken”
            i = i - 1
        }
    }
}
```