1. Ex. 15.1-2

Length i	1	2	3	4
Price p_i	1	20	33	36
Density p_i/i	1	10	11	9

Given a rod with length 4. With the given solution, we make the first cut to get a rod with length 3 which has maximum density. The resulting value is 33(length 3) + 1(length 1) = 34. But the optimal solution is 20(length 2) + 20(length 2) = 40.

2. Ex. 15.3-5

A problem exhibits optimal substructure if its optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve **independently**.

Given the following example:

length i	1	2	3	4
price	15	20	33	36
limit	2	1	1	1

There are only three possible ways to cut a rod with length 4:

1. 36(length 4) = 36
2. 15(length 1) + 33(length 3) = 48
3. 15(length 1) + 15(length 1) + 20(length 2) = 50

Therefore the resulting solution is 50.

Now we look at the subproblem for cutting a rod with length :

1. 15(length 1) + 15(length 1) = 30
2. 20(length 2) = 20

Therefore the resulting solution for the subproblem is 20

But we cannot use this solution for the original solution, because it will violate the limit of the original problem resulting in four rods of length 1.

3. Ex. 16.1-5

Step1:

Let S_{ij} and A_{ij} be defined as in Section 16.1.

A_{ij} is an optimal solution to S_{ij}.

Suppose A_{ij} includes an event a_k.

Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, so the value of maximum-value set $A_{ij} = A_{ik} + A_{kj} + v_k$.

Then we would have the recurrence $val[i,j] = val[i,k] + val[k,j] + v_k$.
Totally, we would have to examine all activities in S_{ij} to find which one to choose, so that

$$val[i,j] = \begin{cases} 0 & if\ S_{ij} = 0 \\ \max\limits_{a_k \in S_{ij}} \{val[i,k] + val[k,j] + v_k\} & if\ S_{ij} \neq 0 \end{cases}$$

Step3:
We create two new activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$.
And we sort jobs with finish time, so that $f_i < f_j, \forall i,j\ \ i < j$.
We will use tables $val[0..n+1, 0..n+1]$, as in recurrence.
And $act[0..n+1, 0..n+1]$, where $act[i,j]$ is the activity that we choose to put into A_{ij}.
We initialize $val[i,i] = 0$, and $val[i,i+1] = 0$.

```
MAX-VALUE-ACTIVITY-SELECTOR($s, f, v, n$)
let $val[0..n+1, 0..n+1]$ and $act[0..n+1, 0..n+1]$ be new tables
for $i = 0$ to $n$
    $val[i,i] = 0$
    $val[i,i+1] = 0$
$val[n+1, n+1] = 0$
for $l = 2$ to $n+1$
    for $i = 0$ to $n - l + 1$
        $j = i + l$
        $val[i,j] = 0$
        $k = j - 1$
        while $f[i] < f[k]$
            if $f[i] \leq s[k]$ and $f[k] \leq s[j]$ and
                    $val[i,k] + val[k,j] + v_k > val[i,j]$
                $val[i,j] = val[i,k] + val[k,j] + v_k$
                $act[i,j] = k$
            $k = k - 1$
```

Step4:

```
PRINT-ACTIVITIES($val, act, i, j$)
if $val[i,j] > 0$
    $k = act[i,j]$
    print $k$
    PRINT-ACTIVITIES($val, act, i, k$)
    PRINT-ACTIVITIES($val, act, k, j$)
```

We will use $PRINT - ACTIVITIES(val,\ act, 0,\ n+1)$ to print result.
The $PRINT - ACTIVITIES$ recursively prints the set of activities placed into the optimal solution A_{ij} until the recursions get bottoms which $val[i, j] = 0$.

Because this algorithm has three nested loop, so that it needs $\Theta(n^3)$ times. The time complexity is $\Theta(n^3)$. Because this algorithm use two (n+2)*(n+2) tables , so that space complexity is $\Theta(n^2)$.

4. Ex. 16.2-2

We can define the following recursive formula:

$$c[i, w] = \begin{cases} 0, & if\ i = 0\ or\ w = 0 \\ c[i-1, w], & if\ w_i > w \\ \max(v_i + c[i-1, w - w_i], c[i-1, w]), & if\ i > 0\ and\ w \geq w_i \end{cases}$$

Where:
c[i, w] denotes the value of the solution for items 0 to i and maximum weight w
v_i is the value of the ith item
w_i is the weight of the ith item

The first and second cases are easy to understand.

As for the third case, we choose the solution from the following two situations:

1. We take the item i, resulting in adding the ith item's value v_i and the solution of $c[i-1, w - w_i]$
2. We drop the ith item, resulting in the same solution as c[i – 1, w]

The Algorithm:

```
Dynamic-01-Knapsack(v, w, n, W)
Let c[0..n, 0..W] be a new array
For w = 0 to W
   c[0, w] = 0
for i = 1 to n
   c[i, 0] = 0
   for w = 1 to W
         if $w_i \leq w$
               c[i, w] = $\max(v_i + c[i-1, w - w_i], c[i-1, w])$
   else c[i, w] = c[i – 1, w]
```

The final solution is stored at c[n, W] and you can trace back the solution according to the table.

As for the running time: it takes $\theta(\mathrm{nW})$ to fill the table c and O(n) to trace back the solution.

5. Ex. 16.2-6

(1) First, we use a linear-time median algorithm to calculate median *m* of the value-weight ratio v_i/w_i

(2) Divide the items into 3 group:

$$G = \left\{i : \frac{v_i}{w_i} > m\right\}$$

$$E = \left\{i : \frac{v_i}{w_i} = m\right\}$$

$$L = \left\{i : \frac{v_i}{w_i} < m\right\}$$

Compute the total weight of group G and E, W_G and W_E.

(3) Take the item in the following way. (All we can take weighs W)

If $W_G > W$: take nothing yet. Recurse on the group G and repeat (1) to (3).

If $W_G \le W$:

a) $W_G + W_E \ge W$: take all the item in G, and take as much item in E as the remaining capacity $W - W_G$.

b) $W_G + W_E < W$: after taking all the item in G and E, recurse on the group L and with new capacity $W - W_G - W_E$.

Time complexity analysis:

Each time the recursive call happened, it will be a new problem of at most half of the origin data size. Thus, the running time is given by the recurrence $T(n) \le T\left(\frac{n}{2}\right) + \Theta(n)$, whose solution is $T(n) = O(n)$.

6.

a) A greedy algorithm is used, it works as follow:

- Take $q = \lfloor n/25 \rfloor$ quarters. And the remaining price is $n_q = n \bmod 25$.
- Take $d = \lfloor n_q/10 \rfloor$ dimes. And the remaining price is $n_d = n_q \bmod 10$.
- Take $k = \lfloor n_d/10 \rfloor$ nickes. And the remaining price is $n_k = n_d \bmod 5$.
- Finally, take $p = n_k$ pennies.

That is, with greedy algorithm, the optimal solution must includes one coin of value c, where c is the largest coin value such that $c \le n$.

Proof the correction case by case:

- $1 \le n < 5$: A solution must consist only of pennies, and the greedy choice principle holds.
- $5 \le n < 10$: Assume the optimal solution does not contain a nickel, then we can create a better solution by replace five pennies by a nickel. It contradicts to the assumption, which means an optimal solution must

contain a nickel.

- $10 \leq n < 25$: Assume the optimal solution does not contain a dime, and then we can create a better solution by replace any combination of nickels and pennies that make up 10 cents. It contradicts to the assumption, which means an optimal solution must contain a nickel.
- $25 \leq n$: Assume the optimal solution does not contain a quarter, and then we can create a better solution by replace any combination of dimes, nickels and pennies that make up 25 cents. It contradicts to the assumption, which means an optimal solution must contain a nickel

Time complexity analysis:

Greedy algorithm chooses one coin at a time and then recourses on sub-problem, the running time is $\Theta(k)$, where k is the number of coins used in an optimal solution. Apparently, $k \leq n$, then the running time is $O(n)$. Recalls that we don't actually take one coin at a time, we take $\lfloor n/c \rfloor$ at a time. So we perform a constant number of calculation (which is 4 coin types), and the running time is $O(4) = O(1)$.

c)

- Denomination combination 1 : 1, 10, and 25, n=30.
  greedy: 25, 5, 5, 5, 5
  non-greedy optimal solution: 10, 10, 10.
- Denomination combination 2 : 1, 3, and 4, n=6.

greedy: 4, 1, 1

non-greedy optimal solution: 3, 3.