(2) Suppose n activities apply for using a common resource. Activity a_i ($1 \leq i \leq n$) has a starting time $S[i]$ and a finish time $F[i]$ such that $0 < S[i] < F[i]$. Two activities a_i and a_j ($1 \leq i, j \leq n$) are compatible if intervals $[S[i], F[i])$ and $[S[j], F[j])$ do not overlap. We assume the activities have been sorted such that $S[1] \leq S[2] \leq ... \leq S[n]$.

A Design an $\mathcal{O}(n^2)$ dynamic programming algorithm to find a set of compatible activities such that the total amount of time the resource is used by these compatible activities is maximized. You need to define the subproblems, establish the inductive formula and show the initial conditions. Pseudocode is not required.

B Apply your algorithm to the following set of activities:

i	1	2	3	4	5	6	7	8	9	10	11
S[i]	2	3	5	6	7	9	10	12	13	14	16
F[i]	6	5	7	10	8	13	16	14	14	18	20

2.A

Algorithm 1 A dynamic programming algorithm usable to solve the activity problem above in $\mathcal{O}(n^2)$ time. In this algorithm...

```
function MaxActivities(S[1..n], F[1..n])
    M ← P ← ∅
    Initialize(M, P)

    M[1] ← 1
    P[1] ← 0
    for i from 2 to n do                                ▷ Locate max for m_i
        max ← 0
        maxIdx ← 0
        for j from 1 to i - 1 do
            if max < M[j] and F[j] ≤ S[i] then
                max ← M[j]
                maxIdx ← j
            end if
        end for
        M[i] ← max + 1
        P[i] ← maxIdx
    end for

    max ← 1                                             ▷ Find global maximum
    for i from 2 to n do
        if M[max] < M[i] then
            max ← i
        end if
    end for

    return (max, P)
end function
```

With this algorithm in mind, the answer is as follows:

Inductive Formula: $\{m(i) = \max_{1 \leq j \leq i} (m_j) + 1 \mid F[j] < S[i]\}$

Initial Conditions: $m(1) = 1$

The **subproblem** can be thought of as follows. Each activity, i, will be added to a chain of activities resulting in the maximum set that includes activity i. Activity i is always included in its maximum. Its maximum, m_i, must also be based off of previous activities which are compatible.

2.B

After Initialization of M and P.

i	1	2	3	4	5	6	7	8	9	10	11
M[i]	1	0	0	0	0	0	0	0	0	0	0
P[i]	0	0	0	0	0	0	0	0	0	0	0

Upon Algorithm Completion.

i	1	2	3	4	5	6	7	8	9	10	11
M[i]	1	1	2	2	3	4	4	4	5	6	6
P[i]	0	0	2	1	3	5	5	5	6	9	9

3

Algorithm 2 A dynamic programming algorithm usable to solve the chess board problem above.

```
function MaxCoinValue(A, V, n, m)
    P ← ∅
    Initialize(P)

    for i from 1 to n do
        for j from 1 to m do
            if TopValue(A, i, j) > LeftValue(A, i, j) then
                A[i, j] ← V[i, j] + TopValue(A, i, j)
                P[i, j] ← Top(i, j)
            else
                A[i, j] ← V[i, j] + LeftValue(A, i, j)
                P[i, j] ← Left(i, j)
            end if
        end for
    end for

    i ← n
    j ← m
    path ← ∅
    Push(path, (i, j))
    repeat
        previous ← P[i, j]
        Push(path, previous)
        i ← IValue(previous)
        j ← JValue(previous)
    until IsNill(P, i, j)

    return path
end function
```

This algorithm utilizes the following helpers:

Algorithm 3 Returns the value of the square above inputs i and j.

```
function TopValue(A, i, j)
    if IsNotNil(A[i - 1, j]) then
        return A[i - 1, j]
    else
        return -∞
    end if
end function
```

Algorithm 4 Returns the value of the square to the left of inputs i and j.

```
function LeftValue(A, i, j)
    if IsNotNil(A[i, j - 1]) then
        return A[i, j - 1]
    else
        return -∞
    end if
end function
```

and similarly

Algorithm 5 Returns the indices that point to the square atop of square (i, j).

```
function Top(i, j)
    return (i - 1, j)
end function
```

Algorithm 6 Returns the indices that point to the square sitting left of square (i, j).

```
function Left(A, i, j)
    return (i, j - 1)
end function
```

This solution to the stated problem results in the relations:

$$A(i,j) = \max\{A(i-1,j), A(i,j-1)\} + V[i,j]$$
$$A(1,1) = V[1,1]$$

Here the **inductive formula** sits above the **initial condition**. The complexity of this algorithm is quite simple to analyze. No recursive calls are made. The only major contributors to the complexity are the for-loops one of which climbs from 1 to n and the other from 1 to m. Therefore, the **complexity** of the algorithm is as follows:

$$T(n) = \theta(n \cdot m)$$