

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 $S[1] \leq S[2] \leq \dots \leq S[n]$.

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 sub-problems, 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...

```

1: function MAXACTIVITIES(S[1..n], F[1..n])
2:    $M \leftarrow P \leftarrow \emptyset$ 
3:   INITIALIZE( $M, P$ )
4:
5:    $M[1] \leftarrow 1$ 
6:    $P[1] \leftarrow 0$ 
7:   for  $i$  from 2 to  $n$  do                                      $\triangleright$  Locate max for  $m_i$ 
8:      $max \leftarrow 0$ 
9:      $maxIdx \leftarrow 0$ 
10:    for  $j$  from 1 to  $i - 1$  do
11:      if  $max < M[j]$  and  $F[j] \leq S[i]$  then
12:         $max \leftarrow M[j]$ 
13:         $maxIdx \leftarrow j$ 
14:      end if
15:    end for
16:     $M[i] \leftarrow max + 1$ 
17:     $P[i] \leftarrow maxIdx$ 
18:  end for
19:
20:   $max \leftarrow 1$                                       $\triangleright$  Find global maximum
21:  for  $i$  from 2 to  $n$  do
22:    if  $M[max] < M[i]$  then
23:       $max \leftarrow i$ 
24:    end if
25:  end for
26:
27:  return ( $max, P$ )
28: 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 TODO: Problem Statement

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

```

1: function MAXCOINVALUE(A, V, n, m)
2:    $P \leftarrow \emptyset$ 
3:   INITIALIZE( $P$ )
4:
5:   for  $i$  from 1 to  $n$  do
6:     for  $j$  from 1 to  $m$  do
7:       if TOPVALUE(A,  $i$ ,  $j$ ) > LEFTVALUE(A,  $i$ ,  $j$ ) then
8:          $A[i, j] \leftarrow V[i, j] + \text{TOPVALUE}(A, i, j)$ 
9:          $P[i, j] \leftarrow \text{TOP}(i, j)$ 
10:      else
11:         $A[i, j] \leftarrow V[i, j] + \text{LEFTVALUE}(A, i, j)$ 
12:         $P[i, j] \leftarrow \text{LEFT}(i, j)$ 
13:      end if
14:    end for
15:  end for
16:
17:   $i \leftarrow n$ 
18:   $j \leftarrow m$ 
19:   $path \leftarrow \emptyset$ 
20:  PUSH( $path, (i, j)$ )
21:  repeat
22:     $previous \leftarrow P[i, j]$ 
23:    PUSH( $path, previous$ )
24:     $i \leftarrow \text{IVALUE}(previous)$ 
25:     $j \leftarrow \text{JVALUE}(previous)$ 
26:  until ISNIL( $P, i, j$ )
27:
28:  return  $path$ 
29: end function

```

This algorithm utilizes the following helpers:

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

```

1: function TOPVALUE(A,  $i$ ,  $j$ )
2:   if ISNOTNIL( $A[i - 1, j]$ ) then
3:     return  $A[i - 1, j]$ 
4:   else
5:     return  $-\infty$ 
6:   end if
7: end function

```

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

```
1: function LEFTVALUE(A, i, j)
2:   if ISNOTNIL(A[i, j - 1]) then
3:     return A[i, j - 1]
4:   else
5:     return -∞
6:   end if
7: end function
```

and similarly

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

```
1: function TOP(i, j)
2:   return (i - 1, j)
3: end function
```

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

```
1: function LEFT(A, i, j)
2:   return (i, j - 1)
3: 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:

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

⑤ TODO: Problem Statement