CS 5592: Design and Analysis of Algorithms
Homework 3
Author: Hayden McParlane

(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...

```
1:  function MaxDuration(S[1..n], F[1..n])
2:      M ← P ← ∅
3:      Initialize(M, P)
4:
5:      M[1] ← F[1] − S[1]
6:      P[1] ← NIL
7:      for i from 2 to n do                          ▷ Locate max for m_i
8:          max ← 0
9:          previous ← NIL
10:         for j from 1 to i - 1 do
11:             if max < M[j] and F[j] ≤ S[i] then
12:                 max ← M[j]
13:                 previous ← j
14:             end if
15:         end for
16:         M[i] ← max + (F[i] − S[i])
17:         P[i] ← previous
18:     end for
19:
20:     max ← 1                                       ▷ Find global maximum
21:     for i from 2 to n do
22:         if M[max] < M[i] then
23:             max ← i
24:         end if
25:     end for
26:
27:     return max and P
28: end function
```

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

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

Initial Conditions: $m(1) = F[1] - S[1]$

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 always has its duration 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) There is a coin in each cell A[i, j] of an n x m chess board, $1 \leq i \leq n$ and $1 \leq j \leq m$, with the value of V[i, j] > 0. Assume the rows are labeled from 1 to n, top down, and columns are labeled 1 to m, left to right. Now, we wish to find a path from cell A[1, 1] to A[n, m] that only moves one cell right or one celldown each step such that every coin on the path will be picked and the total value of these coins can be maximized. Please design a dynamic programming algorithm to solve this problem. You need to show the inductive formula and pseudocode. What is the complexity of your algorithm?

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

```
 1: function MaxCoinValue(A, V, n, m)
 2:     P ← ∅
 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] ← V[i, j] + TopValue(A, i, j)
 9:                 P[i, j] ← Top(i, j)
10:             else
11:                 A[i, j] ← V[i, j] + LeftValue(A, i, j)
12:                 P[i, j] ← Left(i, j)
13:             end if
14:         end for
15:     end for
16:
17:     i ← n
18:     j ← m
19:     path ← ∅
20:     Push(path, (i, j))
21:     repeat
22:         previous ← P[i, j]
23:         Push(path, previous)
24:         i ← IValue(previous)
25:         j ← JValue(previous)
26:     until IsNill(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 -∞
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:

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

(4) Consider the problem of welding a sequence of n steel tubes, $a_1, a_2, a_3, ..., a_n$, into a single tube. You may choose any two adjacent pieces to weld at each step. Suppose that tube a_i has a weight W[i], $1 \leq i \leq n$. Moreover, we assume that the cost for welding two pieces is proportional to the heavier weight of the two pieces (you may assume a dollar per unit weight).

A Design a dynamic programming algorithm to find the order to weld the n pieces such that the total cost is minized. You need only to show the formula and initial conditions from which an algorithm can be developed.

B Apply your algorithm to the following sequence.

$$W[1] = 6, W[2] = 2, W[3] = 7, W[4] = 5, W[5] = 8$$

4.A

Algorithm 7 A dynamic programming implementation used to find the welding order.

```
1:  function MinimizeWeldCost(W, n)
2:      M ← P ← ∅
3:      Initialize(M, P)                          ▷ P indicates weld points
4:
5:      for level from 2 to n do
6:          for i from 1 to n - level + 1 do
7:              j ← i + level − 1
8:              M[i, j] ← ∞
9:              for k from i to j - 1 do
10:                 q ← M[i, k] + M[k + 1, j] + CostToWeld(W, i, k, j)
11:                 if q < M[i, j] then
12:                     M[i, j] ← q
13:                     P[i, j] ← k
14:                 end if
15:             end for
16:         end for
17:     end for
18:
19:     return M and P
20: end function
```

TODO: Part 4.A needs inductive formula and initial conditions.

Algorithm 8 Function to determine cost of a particular weld.

```
1: function CostToWeld(W, i, k, j)
2:     cost ← Cost(W, i, k)
3:     if cost > Cost(W, k + 1, j) then
4:         cost ← Cost(W, k + 1, j)
5:     end if
6:
7:     return cost
8: end function
```

4.B TODO