CSE 101 FALL 2019 Homework 6 Due: Thursday, 11/21 at 11:59pm

INSTRUCTIONS

For each of the following problems, design the fastest dynamic programming algorithm you can, and give a time analysis.
Note that the following ingredients are needed for all dynamic programming algorithm:

1: Define the sub-problems and corresponding array

2: A description of the base cases

3: the recursion used to fill the array and an explanation of the recursion

4: ordering of the subproblems

5: final form of output

6: pseudocode

7: time analysis

If you don't have the above ingredients, we will not grade your submission.
No justification of correctness is required.

1. Consider the following game: you are given n fair 6-sided dice. You roll them all at once. If none are showing a 6, then you lose. Otherwise, you put all of dice showing a 6 to the side and roll the remaining dice. Continue in this manner until all the dice are showing a 6 in which case you win or when in a single roll, you fail to roll at least one 6 in which case you lose.

   (a) Design a Dynamic Programming algorithm that computes the probability of winning this game with n dice.

   - **Definition of the subproblems:**
     Let $A[i]$ be probability of playing the game with i dice.
   - **Base Case:**
     $A[1] = 1/6 \approx 16.67\%$ because, there is a 1 in 6 chance of getting a 6 by throwing one die. We can say that $A[0] = 1$ since, trivially you have already won since there are no dice left.
   - **Recursion and Case analysis:** To compute $A[i]$, we can do a case analysis based on how many sixes you rolled in the first roll.
     Case 0: you rolled 0 sixes, in which case you lost. The probability of winning in this case is 0. The probability of this happening is $(5/6)^i$

     $$P_0 = (5/6)^i * 0$$

     Case 1: you rolled 1 six, the probability of winning in this case is $A[i-1]$. The probability of this happening is $(1/6)(5/6)^{i-1}\binom{i}{1}$

     $$P_1 = (1/6)(5/6)^{i-1}\binom{i}{1}A[i-1]$$

     $$\dots$$

     Case k: you rolled k sixes, the probability of winning in this case is $A[i-k]$. The probability of this happening is $(1/6)^k(5/6)^{i-k}\binom{i}{k}$

     $$P_k = (1/6)^k(5/6)^{i-k}\binom{i}{k}A[i-k]$$

...

Case i: you rolled all i sixes, then you have won so the probability of winning is $A[0] = 1$. The probability of this happening is $(1/6)^i$

$$P_i = (1/6)^i A[0]$$

Then each case is disjoint and the union of all cases is the entire sample space, we add the probabilities:

$$A[i] = P_0 + P_1 + \ldots + P_i = 0 + \left(\sum_{k=1}^{i} (1/6)^k (5/6)^{i-k} \binom{i}{k} A[i-k] \right)$$

- **Ordering of the subproblems:** We can order the subproblems from 0 to n.
- **Final output:** $A[n]$
- **Implementation:**

  procedure `Sixers`(n)
  (1) $A[0] = 1$
  (2) **for** $i = 1, \ldots, n$
  (3) $\quad A[i] = 0$
  (4) $\quad$ **for** $k = 1, \ldots, i$
  (5) $\quad\quad A[i] = A[i] + (1/6)^k (5/6)^{i-k} \binom{i}{k} A[i-k]$
  (6) **return** $A[n]$
- **Runtime:** The outer loop runs n times each time the inner loop runs i times so the algorithm runs in $O(n^2)$.

(b) In my neighborhood bar, you can bet one dollar to play this game with 7 dice and if you win, you win the jackpot which can be anywhere from \$ 20 to \$200.

What is the probability of winning this game with 7 dice?

Solution: $0.01463 \approx 1$ in 68

(c) What happens when you play with more and more dice? Does the probability seem to decrease without bound? Or to reach some limit? What is the approximate limit?

Solution: It appears that as you use more and more dice, the limit approaches $0.00903 \approx 1$ in 110.

2. Suppose you are a pirate ship with a cannon that can shoot any amount of distance. In a river, there are n merchant ships at distances $m_1, \ldots, m_n$ from the mouth of the river. Each merchant ship has a value $v_1, \ldots, v_n$. There is one constraint. If you sink a merchant ship in position p, then it will alarm all other merchant ships within d distance of p and will empty their cargo in effect, changing their value to zero. (in other words, all merchant ships in the interval $[p-d, p+d]$ will be alarmed and will not be of value.)

   You wish to maximize the value of the merchant ships you sink. Design an algorithm that takes as input positions $m_1, \ldots, m_n$, values, $v_1, \ldots, v_n$ (all positive), and an alarming distance d, and returns the maximum value you can get by sinking merchant ships with the above constraints.

   - **Definition of the subproblems:**
     Let $M[i]$ be the maximum value you can get by sinking merchant ships from the set $m_1, \ldots, m_i$.
   - **Base Case:**
     $M[1] = v_1$ since, if there is only one merchant ship then sinking it would yield the maximum value.

- **Recursion and Case analysis:** To compute $M[i]$, we can do a case analysis based on whether or not we sink merchant i.
  Case 1: We sink merchant i. In this case we get the value v_i. Then all merchants that are within d distance of m_i will be alarmed so find the last merchant m_j that is not alarmed and collect the value $M[j]$.
  If all of the previous merchants are within distance d of m_i then we will only collect v_i.
  Case 2: We do not sink merchant i. In this case, we get the value $M[i-1]$

  Since we do not know in advance which is better, we calculate both cases and take the maximum:

  $$M[i] = \max(v_i + M[j], M[i-1])$$

  where m_j is the last merchant that is more than d distance from m_i. If all of the previous merchants are within distance d of m_i then we will only collect v_i from Case 1 so the recursion is:

  $$M[i] = \max(v_i, M[i-1])$$

- **Ordering of the subproblems:** We can order the subproblems from 1 to n.
- **Final output:** $M[n]$
- **Implementation:**

  procedure `Merchants`$((m_1, \ldots, m_n), (v_1, \ldots, v_n), d)$
  (1) $M[1] = v_1$
  (2) $j = 1$
  (3) **for** $i = 2, \ldots, n$
  (4) **while** $m_j < m_i - d$
  (5) $j = j + 1$
  (6) **if** $j - 1 == 0$:
  (7) $M[i] = \max(v_i, M[i-1])$
  (8) **else:**
  (9) $M[i] = \max(v_i + M[j-1], M[i-1])$
  (10) **return** $M[n]$
- **Runtime:** The outer loop runs n times each time the inner loop runs i times so the algorithm runs in $O(n^2)$.

3. A *partition* of a positive integer n is a non-increasing sequence of positive integers that add up to n. For example, the *partitions* of 5 are:

   $$(5), (4,1), (3,2), (3,1,1), (2,2,1), (2,1,1,1,1), (1,1,1,1,1)$$

   You wish to find the partition of n for which the product of all the parts is maximized. For example, $3 * 2 = 6$ is the maximum for $n = 5$ and $3 * 3 = 9$ is the maximum for $n = 6$.

   Design an algorithm that takes a positive integer n as an input and returns the maximum product of parts out of all partitions.

   - **Definition of the subproblems:**
     Let $P[i]$ be the value of the maximum product of parts over all partitions of the integer i.
   - **Base Case:**
     $P[1] = 1$ since, 1 only has one partition and it is 1.

- **Recursion and Case analysis:** To compute $P[i]$, we can do a case analysis based on what is the biggest part in the product.
  Case 1: one of the parts is 1. Then $P[i] = 1 * P[i-1]$.
  Case 2: one of the parts is 2. Then $P[i] = 2 * P[i-2]$.
  ...
  Case $i-1$: one of the parts is $i-1$. then $P[i] = (i-1) * P[1]$.
  Case i: the only part is i in which case $P[i] = i$.
  Since we do not know in advance which case results in the biggest product, we take the maximum over all cases:

$$P[i] = MAX\left(i, \max_{k=1,\ldots,i-1}(k * P[i-k])\right)$$

- **Ordering of the subproblems:** We can order the subproblems from 1 to n.
- **Final output:** $P[n]$
- **Implementation:**

  procedure `PartitionProduct`(n)
  (1) $P[1] = 1$
  (2) **for** $i = 2, \ldots, n$
  (3) $\quad P[i] = i$
  (4) $\quad$ **for** $k = 1, \ldots, i-1$
  (5) $\quad\quad P[i] = MAX(P[i], k * P[i-k])$
  (6) **return** $P[n]$

- **Runtime:** The outer loop runs n times each time the inner loop runs i times so the algorithm runs in $O(n^2)$.

4. You are given a $n \times m$ matrix of non-negative integers: $(A[1 \ldots n][1 \ldots m])$. You can start at any position in the first column. A valid move is to move one cell directly to the right, one cell diagonal up-right or one cell diagonal down-right. You wish to find the path with valid moves such that the sum of all the cells in the path is maximized.

   Design an algorithm that returns just the maximum sum.

   - **Definition of the subproblems:**
     Let $M[i,j]$ be the maximum sum you can get in a path that ends on position (i,j) in the matrix.
   - **Base Case:**
     $M[i,1] = A[i,1]$ since you can start anywhere in the first column.
   - **Recursion and Case analysis:** To compute $M[i,j]$, we can do a case analysis based on what was the last move to get to $M[i,j]$.
     Case 1: the path came from $(i-1, j-1)$. Then: $M[i,j] = M[i-1, j-1] + A[i,j]$.
     Case 2: the path came from $(i, j-1)$. Then: $M[i,j] = M[i, j-1] + A[i,j]$.
     Case 3: the path came from $(i+1, j-1)$. Then: $M[i,j] = M[i+1, j-1] + A[i,j]$.
     Since we do not know in advance which case results in the biggest sum, we take the maximum over all cases:

$$M[i,j] = A[i,j] + MAX\left(M[i-1, j-1], M[i, j-1], M[i+1, j-1]\right))$$

   - **Ordering of the subproblems:** We can order the subproblems from column to column left to right.
   - **Final output:**

$$\max_{0 \le i \le n}(M[i,n])$$

     In other words the maximum of the last column.

- **Implementation:**

  procedure `MatrixSum`(A)
  (1) **for** $i = 1, \ldots, n$
  (2) $\quad M[i, 1] = A[i, 1]$
  (3) **for** $j = 2, \ldots, m$:
  (4) $\quad$ **for** $i = 1, \ldots, n$:
  (5) $\quad\quad M[i, j] = A[i, j] + \max\left(M[i-1, j-1], M[i, j-1], M[i+1, j-1]\right))$
  return $\max_{0 \le i \le n}(M[i, n])$
- **Runtime:** The outer loop runs n times each time the inner loop runs m times so the algorithm runs in $O(nm)$.

5. Skynet is a future AI that sends terminators back in time to find John Conner. The terminators are called T-1, T-2, T-3, and so on. Skynet first sends the terminator T-1 which starts at location 0 and John Conner is at location n. Each position is either nothing or has an explosive that will destroy the terminator. If a terminator T-t is destroyed by an explosive, then skynet will send a more advanced terminator T-$(t+1)$ in the location where the previous terminator died in $t+1$ minutes. The terminator T-1 can only advance 1 location at a time. The terminator T-t can either advance 1 location or jump t locations forward (it can jump over explosives.)

   You are given an array of locations: $L[0, 1, \ldots, n]$ such that $L[i] = N$ if there is nothing on location i, $L[i] = E$ if there is an explosive on location i. You can assume that $L[0] = N$ and $L[n] = N$. Design an algorithm that returns the minimum number of minutes in order to get John Conner. (assume that each advancement and jump takes 1 minute and that the time it takes to send the terminator $T[i]$ takes i minutes.)

   - **Definition of the subproblems:**
     Let $D[i, j]$ be the minimum time it takes to get to position i having the terminator T-j. (assume this is calculated right before the terminator makes his next move so it would include the time it takes for the new terminator to appear if necessary.)
   - **Base Case:**
     $D[0, 1] = 0$ because the clock starts with the T-1 at position 0.
     $D[0, j] = \infty$ for $j > 1$ because it is impossible for any terminator T-j to start at position 0.
     We will assume that if the first index is negative that its value is ∞ since it is impossible to go backward.
     $D[i, 0] = \infty$ because there is no T-0 terminator.
   - **Recursion and Case analysis:** To compute $D[i, j]$, we need to consider whether $L[i] = N$ or $L[i] = E$.
     Case 1: $L[i] = N$.
     Then the terminator T-j traveled to position i from position $i-1$ or from position $i-j$. Each takes 1 minute. We take the minimum of the two choices:

     $$D[i, j] = 1 + \min(D[i-1, j], D[i-j, j]).$$

     Case 2: $L[i] = E$.
     Then the terminator T-$(j-1)$ traveled to position i from position $i-1$ or from position $i-(j-1)$. Each takes 1 minute. Then that terminator would be destroyed and the new terminator T-j would appear in j minutes. We take the minimum of the two choices:

     $$D[i, j] = 1 + j + \min(D[i-1, j-1], D[i-(j-1), (j-1)]).$$

   - **Ordering of the subproblems:** We can order the subproblems row by row from top to bottom.
   - **Final output:** $\min_{j=1\ldots,n}(D[n, j])$
   - **Implementation:**

procedure `Terminator`$(L[0,\ldots,n])$

(1) **for** $j = 0,\ldots,n$
(2) $\quad D[0,j] = \infty$
(3) $\quad D[i,0] = \infty$
(4) $D[0,1] = 0$
(5) **for** $i = 1,\ldots,n$:
(6) $\quad$ **for** $j = 1,\ldots,n$:
(7) $\quad\quad$ **if** $L[i] == N$:
(8) $\quad\quad\quad D[i,j] = 1 + \min(D[i-1,j], D[i-j,j])$.
(9) $\quad\quad$ **if** $L[i] == E$:
(10) $\quad\quad\quad D[i,j] = 1 + j + \min(D[i-1,j-1], D[i-(j-1),(j-1)])$.
(11) **return** $\min_{j=1\ldots,n}(D[n,j])$

- **Runtime:** The outer loop runs n times each time the inner loop runs n times so the whole algorithm will run in $O(n^2)$ time.