1.

(a)

Initialize 2-D array X with dimension n\*n

for each i = 1 … n do

for each j = 1 … n do

if i > j then

X[i, j] = 0

else

p = sum of from i to j

X[i, j] = p

return X

(b)

Initialize 2-D array X with dimension n\*n

for each i = 1 … n do

for each j = 1 … n do

if i > j then

X[i, j] = 0

else if i == j then

X[i, j] =

else

X[i, j] = X[i, j-1] +

return X

2. Assume we have an array named **free** so that **free[n]** indicates whether we get free dinner on day n.

recursive solution:  
food(n):

// base case

if n <= 0 then return 0

// recurrence expressions

if free[n] == true then

cost\_tonight <- food(n-1) // free food

else

cost\_tonight <- 6 + food(n-1) // buy from cafeteria

cost\_grocery <- 20 + food(n-7) // alternatively, buy grocery for a week

return min(cost\_tonight, cost\_grocery)

We see that we have repeated recursive sub-problems when calling food(n-1) and food(n-7), so we can convert the algorithm to an iterative one using dynamic programming.

iterative solution:  
food\_iter(n):

initialize array **cost**with size n

// for simplicity, assume cost[i] return 0 if i <= 0

for i = 1 … n do

if free[n] == true then

cost[i] <- min(cost[i-1], 20 + cost[i-7])

else

cost[i] <- min(6 + cost[i-1], 20 + cost[i-7])

return cost[n]

The running time of iterative solution is , since we only have one for-loop with n iterations and each iteration takes constant time.

3. assume we are given 2 arrays **easy** and **hard** that stores the point value of each homework assignment.

recursive solution:  
homework(n):

// base case

if n <= 0 then return 0

// recurrence expressions

do\_easy = easy[n] + homework(n-1)

do\_hard = hard[n] + homework(n-2)

return max(do\_easy, do\_hard)

We see that we have repeated recursive sub-problems when calling **homework(n-1)** and **homework(n-2)**, so we can convert the algorithm to an iterative one using dynamic programming.

iterative solution:  
homework\_iter(n):

initialize array **point**with size n+1, indexed from 0 to n

point[0] <- 0

point[1] <- hard[1] // hard version guarantee higher point

for i = 2 … n do

do\_easy = easy[n] + point[n-1]

do\_hard = hard[n] + point[n-2]

point[i] <- max(do\_easy, do\_hard)

return point[n]

The running time of iterative solution is , since the algorithm effectively filled the (n+1) size array once, and filling one value in the array takes constant time.