COMP 7270 Assignment 3 10 Problems 200 points No late submissions!

Due by 11:59 PM Tuesday 4.14.15

Upload your submission well before this deadline. Even if you are a few minutes late, as a result of which Canvas marks your submission late, your assignment may not be accepted.

Instructions:

1. This is an individual assignment. You should do your own work. **Any evidence of copying will result in a zero grade and additional penalties/actions.**
2. Late submissions **will not** be accepted unless prior permission has been granted or there is a valid and verifiable excuse.
3. **Think carefully; formulate your answers, and then write them out concisely** using English, logic, mathematics and pseudocode (no programming language syntax).
4. Algorithms should be provided in numbered pseudocode steps.
5. **Type your answers in this Word document and submit it. If that is not possible, use a word processor to type your answers as much as possible (you may hand-write/draw equations and figures), turn it into a PDF document and upload**.

All questions carry equal weight. Each question carries 20 points.

1. Use the Detailed Method to determine the precise T(n) of the Iterative MSS algorithm (Algorithm-2, slide 28, 7270-09-AlgorithmDesign.pptx). You must show your work below to get any credit.

Note: Since T(n) needs to be calculated in terms of n, we need to use the fact that q-p+1=n and write the limits of the summations in terms of 1 to n instead of p to q. This causes no problems because p to q are just indexes and whether these are 1 to n or 5 to (5+n-1) do not affect the analysis.

Line #	Step	Single execution cost	# times executed
1	sum = max = 0	2	1 exact value required
2	for i = p to q	1	n+1 exact value required
3	sum = 0	1	n exact value required
4	for j = i to q	1	$\sum_{i=1}^{i=n} n - i + 2 = \sum_{i=1}^{i=n} n - \sum_{i=1}^{i=n} i + \sum_{i=1}^{i=n} 2 = n^2 - \frac{n(n+1)}{2} + 2n = \frac{n^2}{2} + \frac{3n}{2}$ summation or an expression with a n^2 term and a n term must be provided
5	sum = sum + A[j]	6	$\sum_{i=1}^{i=n} n - i + 1 = \sum_{i=1}^{i=n} n - \sum_{i=1}^{i=n} i + \sum_{i=1}^{i=n} 1 = n^2 - \frac{n(n+1)}{2} + n = \frac{n^2}{2} + \frac{n}{2}$ as above
6	if sum > max then	3	$\frac{n^2}{2} + \frac{n}{2}$ as above
7	max = sum	2	$\frac{n^2}{2} + \frac{n}{2}$ as above
8	return max	2	1 exact value required

1 point for any reasonable constant in the third column = total 8 points; 1 point for each entry in the fourth column as per instructions in this column= total 8 points

$T(n) = 2 + (n+1) + n^2/2+3n/2+6(n^2/2+n/2)+ 3(n^2/2+n/2)+2(n^2/2+n/2)+2 = 2+n+1+ n^2/2+3n/2+ 11n^2/2+11n/2+2 = 6n^2+8n+5$

1 point for showing simplification, 1 point for the n^2 term in T(n), 1 point for the n term in T(n), 1 point for the constant term in T(n), numerical coefficients may be different, total 4 points.

2. Develop, state and solve the recurrence relations of the Recursive Divide & Conquer iterative algorithm (Algorithm-3, slide 30, 7270-09-AlgorithmDesign.pptx) by answering the following questions. You must state costs in terms of n with numerical coefficients, and not using a complexity order notation, to get credit. You may assume that the for loops on lines 9-12 and 13-16 are executed n/2 times.

Which statements are executed when the input is a base case (provide line #s)? ___1,2,3,4__ 2 points: deduct 0.5 point for each step missing from the list

What is the total cost of these? _____12_____ 1 point for any constant value; 0 point if there is an n-term

Which statements are executed when the input is not a base case (provide line #s)? __1, 5-23___2 points: deduct 0.5 point if step 1 is missing from the list; deduct 1 point if multiple steps are missing

What is the total cost of these?

_2T(n/2)+4n+27_ 3 points: 1 point if 2T(n/2) term is present; 1 point for the n-term; 1 point for the constant term; numerical coefficients can be different

Provide the complete and precise two recurrence relations characterizing the complexity of Algorithm-3:

T(n) = __12_ when n=1 1 point for any constant value; 0 point if there is an n-term

T(n) = ___2T(n/2)+4n+27__ when n>1 3 points: 1 point if 2T(n/2) term is present; 1 point for the n-term; 1 point for the constant term; numerical coefficients can be different

Now simplify the recurrence relations by:

1. If your recurrence relation for the non base case input has multiple terms in it besides the term representing the recursive calls, keep only the largest n-term from them and drop the others; if your recurrence relation for the non base case input has only one other term besides the term representing the recursive calls, keep it.
2. Take the largest numerical coefficient of all terms (excluding the term representing the recursive calls) in your two recurrence relations, round it up to the next integer if it is not an integer, and replace the numerical coefficients of all other terms (excluding the term representing the recursive calls) with this coefficient.

Provide the simplified recurrence relations below.

T(n) = __12_ when n=1 1 point for any constant value; 0 point if there is an n-term

T(n) = __2T(n/2)+12n _ when n>1 3 points: 1 point if 2T(n/2) term is present; 1 point for the n-term; 1 point for the numerical coefficient (can be different from 12) being the SAME in both relations

Solve these recurrence relations using the Recursion Tree method, determine and state the T(n) of the algorithm. You must show your work below to get any credit.

Level #	# of recursive executions at this level as a function of level #	Input size to each execution	Additional work done by each execution	Total work done at this level as a function of level #
0	2^0	$n/2^0=n$	$12(n/2^0)=12n$	$2^0*12n=12n$
1	2^1	$n/2^1=n/2$	$12(n/2^1)=12n/2$	$2^1*12(n/2^1)=12n$
2	2^2	$n/2^2=n/4$	$12(n/2^2)=12n/4$	$2^2*12(n/2^2)=12n$
lgn-1	$2^{lgn-1}=n/2$	$n/2^{lgn-1}=2$	$12(n/2^{lgn-1})=24$	$2^{lgn-1}*12(n/2^{lgn-})=12n$
lgn	$2^{lgn}=n$	$n/2^{lgn}=1$	12	$2^{lgn}*12(n/2^{lgn})=12n$

$T(n)=12n(lgn+1)=12nlgn+12$ 4 points: 1 point for the nlogn term in T(n); 1 point for the constant term (numerical coefficients can be different); 2 points for showing the calculations (does not have to be a table like above)

The table below is given for explanatory purposes only. This is not required as part of your answer.

Line #	Step	Single execution cost	# times executed
1	if p==q then	3	1
2	if A[p] > 0 then	4	1
3	return A[p]	4	1
4	else return 0	1	1
5	left-partial-sum = right-partial-sum = max-right = max-left = left-max-sum = right-max-sum = 0	6	1
6	center = floor((p+q)/2)	6 (cost of floor = 1)	1
7	max-left = Algorithm-3(A[p..center])	T(n/2)+1	1
8	max-right = Algorithm-3(A[center+1..q])	T(n/2)+2	1
9	for i from center downto p do	1	n/2+1
10	left-partial-sum = left-partial-sum + A[i]	6	n/2
11	if left-partial-sum > left-max-sum then	3	n/2
12	left-max-sum = left-partial-sum	2	n/2
13	for i from center+1 to q do	1	n/2+1
14	right- partial-sum = right-partial-sum + A[i]	6	n/2
15	if right- partial-sum > right-max-sum then	3	n/2
16	right-max-sum = right- partial-sum	2	n/2
17	if max-left≤max-right then	4	1
18	if max-right≤left-max-sum+right-max-sum then	6	1
19	return left-max-sum+right-max-sum	4	1
20	else return max-right	2	1
	else		
21	if max-left<left-max-sum+right-max-sum then	5	1
22	return left-max-sum+right-max-sum	4	1
23	else return max-left	2	1

Cost of steps 1-4 = 12

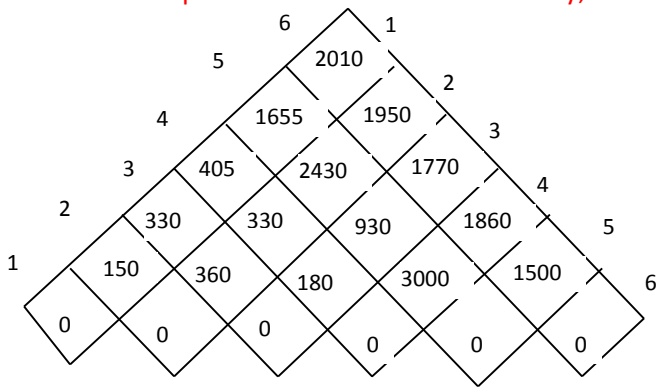
Cost of steps 1 + 5-23 = 3+6+6+T(n/2)+1+T(n/2)+2+n/2+1+3n/2+n/2+1+3n/2+7 = 2T(n/2)+4n+27

3. Let Result[1..2] be an array of two integers. Modify steps of the Iterative MSS algorithm (Algorithm-2, slide 28, 7270-09-AlgorithmDesign.pptx) to return the starting and ending indexes of the optimal (maximum) subsequence it found in its last line instead of the maximum sum value. Provide your modified algorithm below. Make only the minimum number of modifications necessary. You will need to add additional steps.

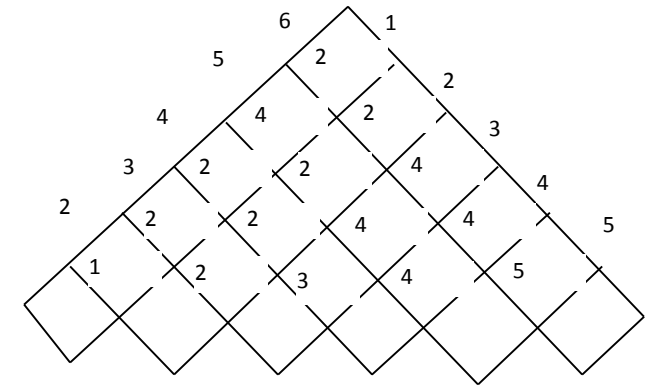
```
Algorithm-2 (A:array[p..q] of integer)
        sum, max: integer       result[1...2]: array of integer     //result can be a parameter instead
1       sum = max = result[1] = result[2] = 0
2       for i = p to q
3               sum = 0
4               for j = i to q
5                       sum = sum + A[j]
6                       if sum > max then
7                               max = sum
8                               result[1]=i                 9 points
9                               result[2]=j                 9 points
10      return result                                       2 points
```

4. Problem 15.2-1 in the text. Show the s and m matrices (like Figure 15.5) and then provide the optimal parenthesization.

m matrix 0.5 point for each correct table entry; 21 entries=10.5 points



s matrix 0.5 point for each correct table entry; 15 entries=7.5 points



Optimal parenthesization: $((A_1A_2)(A_3A_4)(A_5A_6))$ 2 points

5. Convert the recursive characterization of equations (16.2) in text into a recursive algorithm and provide the algorithm below.

In the algorithm below, a 2Xm array A[1..2, 1..m] of activities similar to the one shown on p.415 of the text is assumed to be globally available. The row index 1…m provides the activity number. The first row of A contains starting times and second row finishing times, and activities are numbered in the increasing order of finishing times, again as in p.415 so that if i<j then $f_i \leq f_j$.

```
ACTIVITY-SELECTION(i,j)
1 Sij=Ø  //Sij is the set of activities that begin after ai ends and ends before aj starts
2 for k = i+1 to j-1        //loop adding all activities that begin after ai ends and ends before aj starts to Sij
3       if A[1,k] ≥ i and A[2,k] ≤ j then
4               Sij = Sij U {k}
5 if Sijis empty then
6       return 0        //no activities available
  else
7       max-count = 0
8 for each activity k in Sij
9       count = 1 + ACTIVITY-SELECTION(i,k) + ACTIVITY-SELECTION(k,j)
10      if count > max-count then
11              max-count = count
12 return max-count
```

There are other ways of writing this algorithm. So grade as follows:
1 point for explicitly checking for activities that are available between a_i and and a_j like steps 2-4
2 points for the base case of returning 0 if the activity set is empty, like in steps 5-6
1 point for initializing the variable that is returned, as in step 7
5 points for a loop that iterates through each activity in S_{ij} as in step 8
5 points for making two recursive calls and incrementing the count of activities by 1+the values returned by each recursive call, as in step 9
5 points for updating the variable that is returned if the current count is greater, as in steps 10-11
1 point for returning the max count, as in step 12

6. Problem 16.1-2 in the text. To prove that the stated approach yields an optimal solution, we have to prove two things: (1) that the choice being made is the greedy choice (this proof will be along the lines of the proof of Theorem 16.1 in the text; but do not copy that proof; your proof will be different, yet similar in structure), and (2) that the resulting solution has optimal substructure.
10 points for each proof that is reasonably close to the corresponding proof below; else give partial credit; total 20 points

(1) Proof of greedy choice property: Consider an optimal solution A with k activities $a_1 \ldots a_k$. Suppose the last activity to start is a_l. If $a_l = a_k$ then the optimal solution already contains the greedy choice. If not, suppose we delete a_k from A and instead add a_l. The size of the solution A is still optimal, with k activities. Since a_l is the last activity to start, a_k must have started before or at the same time as a_l. Since a_k was part of a solution, it did not overlap with any other activity in A. That is, every other activity in A would have finished by the time a_k started. So a_l, which starts at the same time or after a_k could not possibly overlap with any other activity in A. Therefore we have another optimal solution containing the greedy choice. Thus, there is always an optimal solution with the greedy choice.

(2) Optimal substructure: Consider an optimal solution A with k activities $a_1 \ldots a_l$. If we remove a_l, the very last activity to start, what remains in A is the solution to the subproblem of finding a maximal set of activities do between the start time and the time a_l begins. If this solution is not optimal, i.e., if more activities can be done in this interval than there are left in A after removing a_l, then that can be substituted and we will get a larger set of compatible activities. This violates the assumption that A was optimal. Therefore, the solution to the subproblem contained in A must be optimal.

7. Modify the FASTEST-WAY algorithm for two-line Assembly Line Scheduling to suit a factory with three assembly lines with n stations. Use a similar notation for matrices a, e, x, f and l. Matrix t is to be interpreted as follows: it is a 3-dimensional matrix with entry t_{ijk}, $1 \le i \le (n-1)$, $1 \le j,k \le 3$ is the transfer cost of the product to move it, after work on it at station i is finished on line j, to line k so that work on it at station (i+1) will be done on line k.

The algorithm below is stated at a somewhat higher level than a typical algorithm, but sufficient to understand the solution The actual algorithm should have more details, such as a detailed if-then-else statement in place of lines 5 & 6, 7 & 8, 9 & 10, and 11 & 12. Grade as follows.

FASTEST-WAY(a, t, e, x, n)

1 $f_1[1] = e_1 + a_{11}$ 1 point for initializing $f_1[1]$

2 $f_2[1] = e_2 + a_{21}$ 1 point for initializing $f_2[1]$

3 $f_3[1] = e_3 + a_{31}$ 1 point for initializing $f_3[1]$

4 **for** $j = 2$ to n 1 point for the loop statement

5 $f_1[j]$ = minimum of $\{f_1[j-1]+a_{1j}, f_2[j-1]+t_{(j-1)21}+a_{1j}, f_3[j-1]+t_{(j-1)31}+a_{1j}\}$

6 $l_1[j]$ = 1, 2 or 3 depending on which term produced the minimum in the previous step

7 $f_2[j]$ = minimum of $\{f_2[j-1]+a_{2j}, f_1[j-1]+t_{(j-1)12}+a_{2j}, f_3[j-1]+t_{(j-1)32}+a_{2j}\}$

8 $l_2[j]$ = 1, 2 or 3 depending on which term produced the minimum in the previous step

9 $f_3[j]$ = minimum of $\{f_3[j-1]+a_{3j}, f_1[j-1]+t_{(j-1)13}+a_{3j}, f_2[j-1]+t_{(j-1)23}+a_{3j}\}$

10 $l_3[j]$ = 1, 2 or 3 depending on which term produced the minimum in the previous step

11 $f_* = \text{minimum of } \{f_1[n]+x_1, f_2[n]+x_2, f_3[n]+x_3\}$

12 l_* = 1, 2 or 3 depending on which term produced the minimum

2 points for an if-then-else statement or calling a min function to find the minimum of three values to write into $f_1[j]$, as in step 5.
2 points for writing the line number corresponding to this minimum value into $l_1[j]$ as in step 6.
Grade steps 7 & 8 the same way.
Grade steps 9 & 10 the same way.
2 points for computing f* as the minimum of three values as in step 11.
2 points for computing l* as the minimum of three values as in step 12.

8. Write a memorized recursive algorithm RECURSIVE-MEMOIZED-LCS-LENGTH(X,Y) to compute the length of the LCS of X and Y based on equations (15.9), p. 393.

RECURSIVE-MEMOIZED-LCS-LENGTH(X,Y)

```
1   let c be a (m+1)*(n+1) table where m=length of X and n=length of Y
2       for i = 0 to m
3           for j = 0 to n
4               c[i, j] = infinity
5       return LOOKUP_LENGTH(m,n)
```

5 points for declaring and initializing the table in a separate algorithm as above. Note that the 0th row and column may be initialized to 0 (instead of infinity) here rather than by steps 3-4 in the algorithm below.
There are other ways of writing this algorithm. So grade based on key operations as follows:

LOOKUP_CHAIN(i,j) X, Y and matrix c are globally available; if not, they would also be parameters

```
1       if c[i, j] < infinity          5 points for checking and returning the value from the lookup table
2           then return c[i, j]
3       if i == 0 or j == 0
4       then c[i, j] = 0
5       else    if X[i]==Y[i]  2 points for checking if the last characters of the 2 strings are same
6           then c[i, j] = LOOKUP_CHAIN(i-1,j-1)+1   then 2 points for a SINGLE recursive call
7           else    temp1 = LOOKUP_CHAIN(i,j-1)   else there should be 2 recursive calls
8                   temp2 = LOOKUP_CHAIN(i-1,j)   with 2 points for each – total 4 points
9                   if temp1>temp2 then c[i, j] = temp1 else c[i, j] = temp2 1 point for table updating
10      return c[i, j]      1 point for returning the answer
```

base case , when to stop

9. Do the Questions 1 & 2 on the Thinking Assignment on slide 22, 7270-10-DPandGreedyAlgorithmDesign Part I.pptx The specific input for which you would draw a recursion Tree should be a rod of length 4 inches.

Question 1 on the slide:

(15.1) $r_n=\max(p_n, r_1+r_{n-1}, r_2+r_{n-2},\ldots,r_{n-1}+r_1)$

Appropriate modifications of the Cut-Rod recursive algorithm on p. 363 produce the desired algorithm:

Cut-Rod-Recursive(p,n)

1 if n==1 2 points for the base case of n=1; n=0 can also be a base case, but n=1 must be shown.

2 return p_1

3 q=-∞ 1 point for initializing variable being used in the loop below.

4 for i=1 to (n-1) 1 point for writing a loop

5 q=max(q, Cut-Rod-DP-Recursive(p,i)+ Cut-Rod-DP-Recursive(p,n-i)) 5 points (see below)

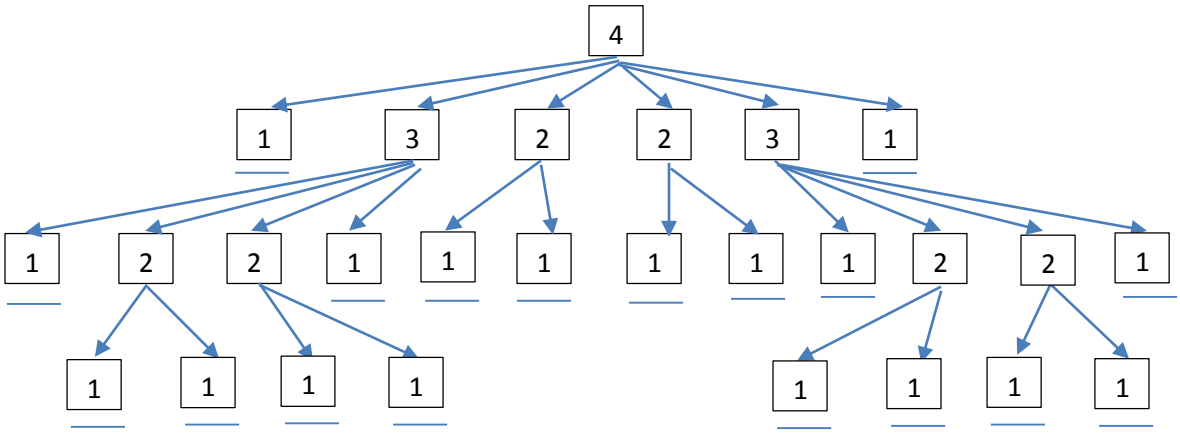
6 return max(q,p_n) 1 point for comparing the value computed by the loop with p_n and returning the max

Grading the loop body (step 5): 2 point for each of the two recursive calls, 1 point for finding the max

Question 2 on the slide:

Recursion tree: (only parameter n shown since p is a static array)

Overlapping subproblems (duplicate executions) are obvious.



10 points for a correct Recursion Tree that looks like this, with the correct values for parameter n. If the complete tree is not shown, some parameter values are missing or incorrect, give partial credit. 0 point if the tree is completely wrong.

10. Do the Questions 3 & 4 on the Thinking Assignment on slide 22, 7270-10-DPandGreedyAlgorithmDesign Part I.pptx As part of your answer for Q.4, you must explain the lookup table – what it's dimensions are and the order in which its cells will be filled by the algorithm.

Question 3 on the slide: 10 points total

Appropriate modifications of the Memoized-Cut-Rod algorithm and Memoized-Cut-Rod-Aux algorithm on p. 363-364 produce the desired algorithms:

Memoized-Cut-Rod-Recursive(p,n)

1 let r[1...n] be a new array

2 for i=1 to n

3 r[i]= -∞
4 return Memoized-Cut-Rod-Recursive-Aux(p,n,r)
1 point for declaring and initializing the table in a separate algorithm as above.
There are other ways of writing this algorithm. So grade based on key operations as follows:
Memoized-Cut-Rod-Recursive-Aux(p,n,r)
1 if r[n]≥0 1 point for checking and returning the value from the lookup table
2 return r[n]
3 if n==1 1 point for the base case of n=1; n=0 can also be a base case, but n=1 must be shown.
4 q=p_1
5 else q=-∞
6 for i=1 to (n-1) 1 point for writing a loop
7 q=max(q, Memoized-Cut-Rod-Recursive-Aux(p,i,r)+Memoized-Cut-Rod-Recursive-Aux(p,n-i,r) 3 pts
8 q=max(q,p_n) 1 point for comparing the value computed by the loop with p_n and returning the max
9 r[n]=q 0.5 point for updating the table entry
10 return q 0.5 point for returning a value
Grading the loop body (step 7): 1 point for each of the two recursive calls, 1 point for finding the max

This is less efficient than Memoized-Cut-Rod. 1 point. No explanation required.
Why? Because though memorization avoids the recursive computation of a previously computed subproblem solution, whenever a subproblem solution has to be computed for the first time, this algorithm ends up making more recursive calls since each loop execution entails two recursive calls instead of one.

Question 4 on the slide: 10 points total
Problem-subproblem dependencies for the recursive characterizations of the optimal solution given by equations (15.1) and (15.2) are basically the same, so appropriate modifications of the Bottom-Up-Cut-Rod algorithm on p. 366 produce the desired algorithm:
There are other ways of writing this algorithm. So grade based on key operations as follows:
Bottom-Up-Cut-Rod-Modified(p,n)
1 let r[1...n] be a new array
2 r[1]= p_1 1 point for initializing table[1] with value p_1
3 for j=2 to n 1 point for the outer loop
4 q=-∞
5 for i=1 to (j-1) //for i=1 to ceiling[(j-1)/2] is also correct 1 point for the inner loop
6 q=max(q,r[i]+r[j-i]) 1 point for adding two precomputed table values and checking max
7 q=max(p_j,q) 1 point for comparing the value computed by the loop with p_n and returning the max
8 r[j]=q 1 point for updating the table entry
9 return r[n] 1 point for returning a value
Total for algorithm: 7 points
The lookup table r has dimensions 1Xn with the meaning that r[i]=optimal answer for an i-inch rod. 1 point for stating the table dimensions It is filled in the order r[1], r[2]],...,r[n]. 1 point for stating the order

In terms of complexity order both algorithms are equally efficient, $\Theta(n^2)$.

1 point for stating both have the same order of complexity or for stating that the algorithm above is more efficient than Bottom-Up-Cut-Rod in the text. No explanation required.

But a detailed analysis will show that the above algorithm is more efficient than Bottom-Up-Cut-Rod, esp. if step 5 is “for i=1 to ceiling[(j-1)/2]”. This is an interesting example because the recursive characterization of (15.1) is less efficient than (15.2), therefore the corresponding memorized recursive algorithms reflect this efficiency difference. But when the recursive calls are replaced by table look-ups in the nonrecursive DP algorithms, we are able to produce a more efficient nonrecursive algorithm using the recursive characterization of (15.1)!