INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (Mid Semester)										**SEMESTER (Spring 2019-2020)**			
Roll Number										**Section**		**Name**	
Subject Number	C	S	2	1	0	0	3	**Subject Name**		*ALGORITHMS – I*			
Department / Center of the Student											**Additional sheets**		

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items or any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Marks Obtained											

Marks obtained (in words)	**Signature of the Examiner**	**Signature of the Scrutineer**

Indian Institute of Technology Kharagpur
Department of Computer Science and Engineering

Algorithms-I (CS21003) **Mid-Semester [Maximum Marks: 60]** **Spring Semester, 2019-2020**

Date: 20-Feb-2020 (DAY) || **Time: 09:00am – 11:00am** || **Venue: NR – 111/112/323/324/423/424**

Instructions:

- *Answer AS MANY questions as you can. The Maximum Marks that you can obtain in* 60*.*
- *Write your answers in proper places mentioned in the question paper itself. If you need additional space to complete the answer, please use the blank pages provided at the end of this booklet.*
- *There are* 5 *questions in total, each having different total marks. Some questions have multiple parts.*
- *Be brief and precise. If you use any algorithm / result / formula covered in class, please mention it and do not elaborate / derive.*
- *Do not use plain English to write / express steps of your algorithm. Write readable pseudocodes (or precise formulations of the solution). If it is necessary, justify the steps of your pseudocode in English.*

Questions start from the next page.

Q1. Let $f(n), g(n), r(n), s(n)$ be non-negative asymptotic functions. If $f(n) = O(r(n))$ and $g(n) = O(s(n))$, then *prove* or *disprove* whether the following always holds: **[Marks: 4 × 3 = 12]**

(a) $f(n) + g(n) = O(r(n) + s(n))$
(b) $f(n) - g(n) = O(r(n) - s(n))$
(c) $f(n) \times g(n) = O(r(n) \times s(n))$
(d) $f(n) \div g(n) = O(r(n) \div s(n))$

Solution:

Since $f(n) = O(r(n))$, hence $f(n) \leq c_1.r(n) \quad (\forall\, n \geq n_1$ and $c_1 > 0)$; and also
Since $g(n) = O(s(n))$, hence $g(n) \leq c_2.s(n) \quad (\forall\, n \geq n_2$ and $c_2 > 0)$.

(a) $f(n) + g(n) = O(r(n) + s(n)) \quad \rightarrow \quad$ TRUE

$$\begin{aligned} f(n) + g(n) &\leq c_1 r(n) + c_2 s(n), \ \forall\, n \geq \texttt{maximum}\,\{n_1, n_2\} \\ &\leq C.(r(n) + s(n)), \ \textit{where}\ C = \texttt{maximum}\,\{c_1, c_2\} \\ &= O(r(n) + s(n)) \end{aligned}$$

(b) $f(n) - g(n) = O(r(n) - s(n)) \quad \rightarrow \quad$ FALSE

Consider the counter-example, $f(n) = 4n + 3,\ r(n) = n + 1$ and $g(n) = 2n + 1,\ s(n) = n$.

(c) $f(n) \times g(n) = O(r(n) \times s(n)) \quad \rightarrow \quad$ TRUE

$$\begin{aligned} f(n) \times g(n) &\leq c_1 r(n) \times c_2 s(n), \ \forall\, n \geq \texttt{maximum}\,\{n_1, n_2\} \\ &= C.(r(n) \times s(n)), \ \textit{where}\ C = c_1.c_2 \\ &= O(r(n) \times s(n)) \end{aligned}$$

(d) $f(n) \div g(n) = O(r(n) \div s(n)) \quad \rightarrow \quad$ FALSE

Consider the counter-example, $f(n) = n^2,\ r(n) = 2n^2 + 2$ and $g(n) = n,\ s(n) = n^2 + 1$.

Q2. A recursive algorithm gives rise to the following recurrence relation:

$$T(n) = \begin{cases} b, & \text{if } n = 1 \\ T(\frac{n}{3}) + T(\frac{2n}{3}) + cn, & \text{if } n > 1 \end{cases} \quad \text{(assume, } b \text{ and } c \text{ are constants)}$$

Solve the recurrence and deduce the running time $T(n)$ in asymptotic Θ-notation. Please note that, you must prove both ends to derive the asymptotic Θ-bound. **[Marks: 8]**

Solution:

Procedure-1: (By Expansion using Recurrence-Tree)

$$\begin{aligned}
T(n) &= T(\tfrac{n}{3}) + T(\tfrac{2n}{3}) + cn \\
&= [T(\tfrac{n}{3^2}) + T(\tfrac{2n}{3^2}) + c.(\tfrac{n}{3})] + [T(\tfrac{2n}{3^2}) + T(\tfrac{4n}{3^2}) + c.(\tfrac{2n}{3})] + cn = T(\tfrac{n}{3^2}) + 2T(\tfrac{2n}{3^2}) + T(\tfrac{4n}{3^2}) + c(\tfrac{n}{3} + \tfrac{2n}{3}) + cn \\
&= \binom{2}{0}.T(\tfrac{2^0.n}{3^2}) + \binom{2}{1}.T(\tfrac{2^1.n}{3^2}) + \binom{2}{2}.T(\tfrac{2^2.n}{3^2}) + 2cn \\
&= \binom{3}{0}.T(\tfrac{2^0.n}{3^3}) + \binom{3}{1}.T(\tfrac{2^1.n}{3^3}) + \binom{3}{2}.T(\tfrac{2^2.n}{3^3}) + \binom{3}{3}.T(\tfrac{2^3.n}{3^3}) + 3cn \\
&= \cdots\ \cdots\ \cdots\ \cdots \\
&= \binom{l}{0}.T(\tfrac{2^0.n}{3^l}) + \binom{l}{1}.T(\tfrac{2^1.n}{3^l}) + \binom{l}{2}.T(\tfrac{2^2.n}{3^l}) + \cdots + \binom{l}{l}.T(\tfrac{2^l.n}{3^l}) + lcn \\
&= \sum_{i=0}^{l} \binom{l}{i} T(\tfrac{2^i.n}{3^l}) + lcn \qquad \text{[Note: You may draw the recurrence tree for better visualization.]}
\end{aligned}$$

Hence, we can say that, when n is asymptotically large (assuming $n = 3^j$),

$$\therefore T(n) \geq T(\frac{n}{3^j}) + jcn = T(1) + (\log_3 n).cn = b + cn\log_3 n \geq c'n\log_2 n$$

Again, we can also say that, when n is asymptotically large (assuming $n = (\frac{3}{2})^k$),

$$\therefore T(n) \leq T(\frac{n}{(\frac{3}{2})^k}) + kcn = T(1) + (\log_{\frac{3}{2}} n).cn = b + cn\log_{\frac{3}{2}} n \leq c''n\log_2 n$$

Therefore, we may conclude that, $c'n\log_2 n \leq T(n) \leq c''n\log_2 n \implies T(n) = \Theta(n\log_2 n)$

Procedure-2: (By Substitution from Initial-Guess)

Let us guess the solution of the relation is, $T(n) \leq dn\log_2 n$ (where $d \geq 0$ is a constant).

$$\begin{aligned}
T(n) &\leq T(\tfrac{n}{3}) + T(\tfrac{2n}{3}) + cn \\
&\leq d(\tfrac{n}{3})\log_2(\tfrac{n}{3}) + d(\tfrac{2n}{3})\log_2(\tfrac{2n}{3}) + cn \\
&= [d(\tfrac{n}{3})\log_2 n - d(\tfrac{n}{3})\log_2 3] + [d(\tfrac{2n}{3})\log_2 n - d(\tfrac{2n}{3})\log_2(\tfrac{3}{2})] + cn \\
&= dn\log_2 n - d((\tfrac{n}{3})\log_2 3 + (\tfrac{2n}{3})\log_2(\tfrac{3}{2})) + cn \\
&= dn\log_2 n - d((\tfrac{n}{3})\log_2 3 + (\tfrac{2n}{3})\log_2 3 - (\tfrac{2n}{3})\log_2 2)) + cn \\
&= dn\log_2 n - dn(\log_2 3 - (\tfrac{2}{3})) + cn \\
&\leq dn\log_2 n \qquad \ldots\ldots \text{ whenever } d \geq \frac{c}{\log_2 3 - \frac{2}{3}}
\end{aligned}$$

Similarly, we can also show from the initial guess that, $T(n) \geq d'n\log_2 n$, for some $d' \leq \frac{c}{\log_2 3 - \frac{2}{3}}$.

Therefore, we may conclude that, $d'n\log_2 n \leq T(n) \leq dn\log_2 n \implies T(n) = \Theta(n\log_2 n)$

Q3. An *inversion* of an array $A[1..n]$ of n distinct integer elements is a pair $\langle i, j\rangle$ such that $i < j$ and $A[i] > A[j]$. Your task is to determine the number of inversions present in an array. For example, the array $A[1..8] = \{4,8,9,3,7,6,2,5\}$ has a total of 18 inversions. In particular, the element-pair $\langle 1,4\rangle$ (since $A[1] = 4$ and $A[4] = 3$, so $1 < 4$ but $A[1] = 4 > 3 = A[4]$) presents one such inversion in $A[1..8]$.

Answer the following three parts: **[Marks: (4+2) + (7+2) + 3 = 18]**

(i) Design an algorithm to solve the problem which runs in $\Theta(n^2)$-time. Also, deduce the given time-complexity from your algorithm.

(ii) Design an *efficient* algorithm to solve the same problem and deduce the time-complexity of your newly proposed algorithm. [*Hint*: $O(n\log_2 n)$-time is achievable.]

(iii) Clearly show the working steps of your proposed $O(n\log_2 n)$-time algorithm (above) in the following example with eight elements, $A[1..8] = \{4,8,9,3,7,6,2,5\}$.

Solution-(i):

Algorithm:

```
count = count_inversion_iterative (A[], n)
{
    Initialize count = 0.
    loop for i = 1 to n-1, do:
        loop for j = i+1 to n, do:
            if (A[i] > A[j]), then increment count by 1.
    return count.
}
```

Time-Complexity:
The expensive steps of this algorithm are the two nested loops and it performs $O(1)$-time (say, constant-time c) operations in the innermost region of the loop. The operations outside loop are constant-time (say, d) operations. So, the time-complexity is =

$$T(n) = d + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c = d + \sum_{i=1}^{n-1} c(n-i) = d + cn(n-1) + \frac{n(n-1)}{2} = \Theta(n^2)$$

Solution-(ii):

Algorithm:

```
count = count_inversion_recursive (A[low..high])
{
    Initialize count = 0.
    Initialize a list TEMP of size (high-low+1).
    if (low < high), then do:
        set mid to (low + high)/2.
        count = count + count_inversion_recursive (A[low..mid]).
        count = count + count_inversion_recursive (A[mid+1..high]).
        count = merge_count (A[low..mid], A[mid+1..high], TEMP).
        copy back TEMP into A[low..high].
    return count.
}
merge_count = merge_count (A[ll..lr], A[rl..rr], TEMP)
{
    if (ll > lr) AND (rl > rr) return 0.
    else if (ll > lr), then
        return merge_count (A[ll..lr], A[rl+1..rr], TEMP)
    else if (rl > rr), then
        return merge_count (A[ll+1..lr], A[rl..rr], TEMP)
    else, do:
        if (A[ll] <= A[rl]), then do:
            append A[ll] to TEMP.
            return merge_count (A[ll+1..lr], A[rl..rr], TEMP).
        else, do:
            append A[rl] to TEMP.
            return (lr-ll+1) + merge_count (A[ll..lr], A[rl+1..rr], TEMP).
}
```

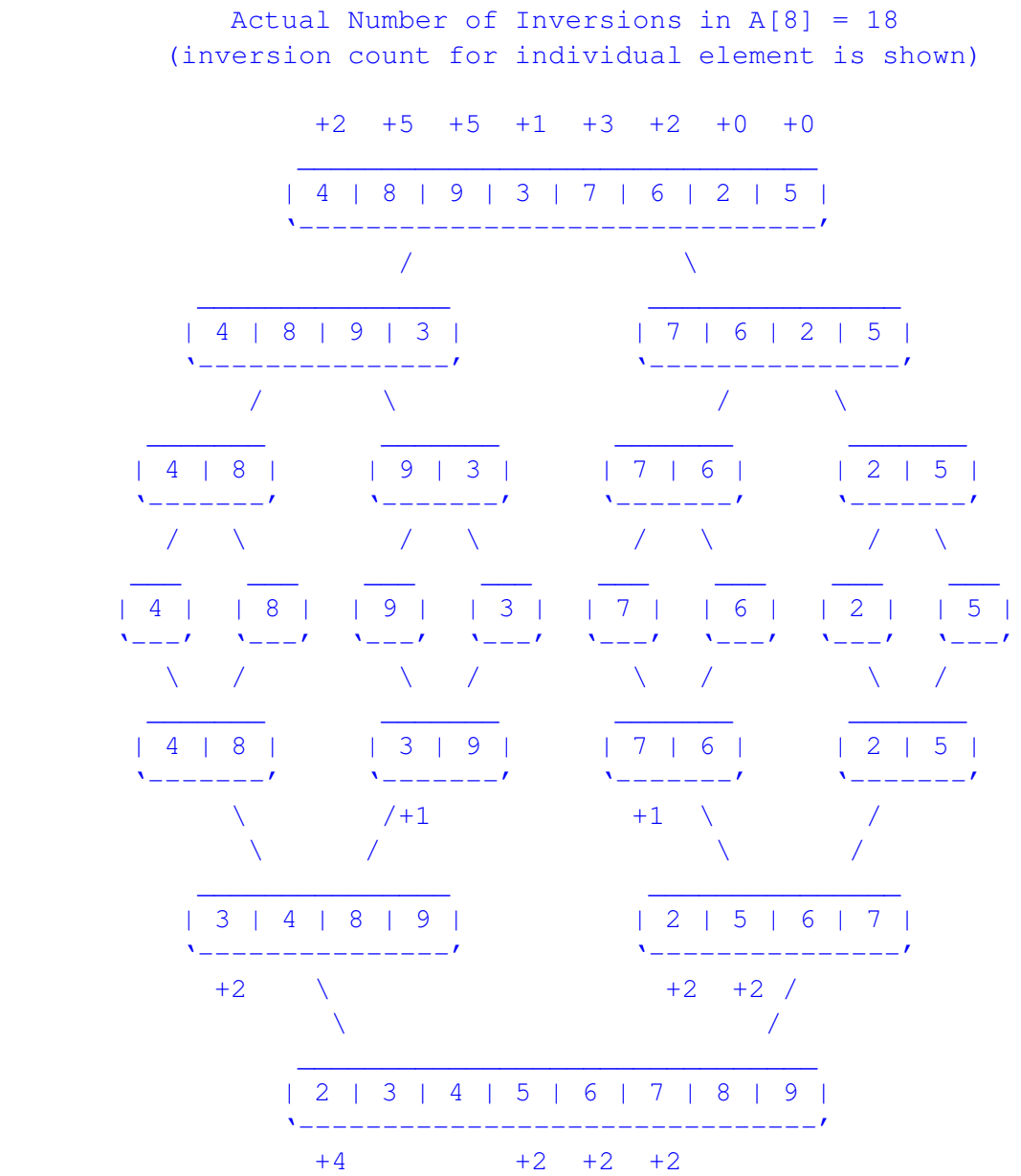
Explanation:
A modified merge-sort can be used to count the number of inversions. It may be observed that the only way in which array elements change their positions is within the `MERGE` procedure. Moreover, the change of position occurs only when an element in left sorted half is greater than some element in right sorted half. Also note that, once a pair of number $\langle x, y \rangle$ $(x > y)$ such that $x \in$ `Left Half` and $y \in$ `Right Half` have exchanged positions, their relative order never changes again as they belong to the same sorted list from there on. Now, let us look at counting the number of inversions while merging. For every $y \in$ `Right Half`, select the smallest x such that $x > y$ and $x \in$ `Left Half`. Now, the total number of inversions induced by this y is equal to the inversion with x plus its inversions with all elements greater than x – that is, only those which resides in the sorted left half after/right-to x. Thus, we can obtain the total inversions while merging by summing it for every such y. Therefore, the above algorithm (similar to merge-sort) realizes the total number of inversions in an array.

Time-Complexity:
The `merge_count` routine has similar time-complexity as the procedure `MERGE` which is $O(n)$. So, the above recursive formulation of the overall inversion-count algorithm gives rise to similar recurrence relation as in merge-sort.

$$T(n) \;=\; T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + O(n) \;\approx\; 2T(\tfrac{n}{2}) + O(n) \;=\; O(n \log_2 n)$$

Solution-(iii):

Example:



Total Inversions = (+1 +1) + (+2 +2 +2) + (+4 +2 +2 +2) = 18

Q4. Suppose you are given a set, $S = \{a_1, a_2, \ldots, a_n\}$, of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the completion time of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $\frac{1}{n}\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2 with $p_1 = 5$ and $p_2 = 3$, and consider the schedule in which a_1 runs first, followed by a_2. Then, $c_1 = 5$ and $c_2 = 8$, and the average completion time is $\frac{(5+8)}{2} = 6.5$ units. Whereas, if we schedule a_2 first, followed by a_1. Then, $c_2 = 3$ and $c_1 = 8$, and the average completion time is $\frac{(3+8)}{2} = 5.5$ units.

Answer the following two parts: **[Marks: (3+1) + 4 = 8]**

(i) Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i is started, it must run continuously for p_i units of time. Also, state the running time of your algorithm in Big-O notation.

(ii) Prove the optimality of your algorithm, that is, it minimizes the average completion time.

Solution-(i):

Algorithm:
Step-1: Sort the tasks according to their processing times in ascending order.
Step-2: Schedule the tasks one-by-one starting from the shortest processing time.

Time-Complexity:
Since sorting using an efficient algorithm (say, merge-sort) takes $O(n\log_2 n)$ time, so the above greedy procedure takes an overall $O(n\log_2 n)$ time.

Solution-(ii):

Proof for Optimality:
This algorithm uses a greedy strategy. It is shown to be optimal as follows. Suppose, we schedule n tasks in the following order, $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ (where, $\forall j,\ i_j \in [1,n]$), and the completion times are,

$$c_{i_1} = p_{i_1},\ c_{i_2} = (p_{i_1} + p_{i_2}),\ \ldots,\ c_{i_n} = (p_{i_1} + p_{i_2} + \cdots + p_{i_n})$$

Now, the average completion time can be expressed as,

$$\begin{aligned} c_{avg} = \frac{c_{i_1} + c_{i_2} + \cdots + c_{i_n}}{n} &= \frac{1}{n}[p_{i_1} + (p_{i_1} + p_{i_2}) + \cdots + (p_{i_1} + p_{i_2} + \cdots + p_{i_n})] \\ &= \frac{1}{n}[n.p_{i_1} + (n-1).p_{i_2} + \cdots + 1.p_{i_n})] \end{aligned}$$

So, we find that p_{i_1} is added in the most times (n times), then p_{i_2} for $(n-1)$ times, and so on. As a result, to minimize c_{avg}, p_{i_1} should have the shortest processing time, followed by p_{i_2}, and so on. Otherwise, you could rearrange (swap) the task schedule according to the shorter processing times and produce a faster algorithm. As a result, the greedy property holds and the above algorithm produces optimal result.

Q5. Given two strings, $X = [x_1 x_2 \ldots x_n]$ (having length n) and $Y = [y_1 y_2 \ldots y_m]$ (having length m), the *shortest common supersequence* (SCS) is a minimum length string Z such that both X and Y are subsequences of Z. For example, if $X = [abcbdab]$ (length 7) and $Y = [bdcaba]$ (length 6), a SCS is $Z = [abcbdcaba]$ (length 9). Your task is to find out the length of the SCS from two input strings of length n and m.

Answer the following five parts: **[Marks: 3 + (2 + 3) + 4 + (4 + 4) + 4 = 24]**

(i) Provide a recursive definition to compute the length of the SCS as given in the problem statement.
(ii) Develop a recursive algorithm translating the above definition, without declaring additional space. Also, derive the time-complexity of your algorithm in asymptotic Big-O notation.
(iii) Improve this *top-down* recursive algorithm with the help of *Memoization* (using additional space).
(iv) Now, propose an *iterative (bottom-up)* algorithm for the same problem. Also, provide the time and space complexity of your algorithm in asymptotic Big-O notation (give tight bounds).
(v) Clearly show the working steps of your proposed iterative bottom-up algorithm (above) in the given example strings, $X = [abcbdab]$ and $Y = [bdcaba]$.

Solution-(i):

Let us denote the length of SCS of two strings, $X = [x_1 x_2 \ldots x_n]$ (having length n) and $Y = [y_1 y_2 \ldots y_m]$ (having length m), as $L[n][m]$. Therefore, we can recursively define the length of SCS as –

$$L[n][m] = \begin{cases} n, & \text{if } n \geq 0 \text{ and } m = 0 \\ m, & \text{if } n = 0 \text{ and } m \geq 0 \\ L[n-1][m-1] + 1, & \text{if } m,n > 0 \text{ and } x_n = y_m \\ \texttt{MINIMUM}\{L[n-1][m], L[n][m-1]\} + 1, & \text{if } m,n > 0 \text{ and } x_n \neq y_m \end{cases}$$

Solution-(ii):

Recursive Algorithm:

```
length = SCS_length (X[], n, Y[], m)
{
    if (m == 0) return n.
    if (n == 0) return m.
    if (X[n-1] == Y[m-1]), then return 1 + SCS_length (X[], n-1, Y[], m-1).
    else, return 1 + MINIMUM { SCS_length (X[], n-1, Y[], m),
                               SCS_length (X[], n, Y[], m-1) }.
}
```

Time-Complexity:
The performance of the recursive algorithm above is *exponential* in nature with respect to n and m. We prove by induction on n,m that $T(n,m) \leq \binom{n+m-1}{m} - c$. The result holds for $n = 0$ or $m = 0$ [induction basis]. If both n and m are positive, we make two recursive calls and the running time satisfies the recurrence,

$$\begin{aligned} T(n,m) &= T(n-1,m) + T(n,m-1) + c \\ &\leq [\binom{n+m-2}{m} - c) + (\binom{n+m-2}{m-1} - c] + c \\ &= [\binom{n+m-2}{m} + \binom{n+m-2}{m-1}] - c \;=\; \binom{n+m-1}{m} - c \end{aligned}$$

Since c is a constant, we obtain: $T(n,m) = O[\binom{n+m-1}{m}]$.
Similarly, if we choose $T(n,m) \leq \binom{n+m-1}{n} - c$ as our induction hypothesis, we get: $T(n,m) = O[\binom{n+m-1}{n}]$.
Considering both of these cases, we may conclude that, $T(n,m) = O(2^{\min(n,m)})$.

Solution-(iii):

Memoized Top-down Algorithm:

We use additional table, `LEN[ ][ ]`, of $n \times m$ dimension (initialized with all `INVALID` entries), where $\langle i, j \rangle$-th entry in `LEN` stores the length $L[i][j]$ of SCS already computed recursively once. We may use this to look-up when there is again a need for the same value, thereby not solving the overlapping sub-problems repeatedly.

```
length = SCS_length_memoize (X[], n, Y[], m, LEN[][])
{
    if (m == 0) return n.
    if (n == 0) return m.
    if (X[n-1] == Y[m-1]), then
        if (LEN[n-1[m-1] is INVALID), then
            LEN[n-1[m-1] = SCS_length_memoize (X[], n-1, Y[], m-1).
        return 1 + LEN[n-1[m-1].
    else, do:
        if (LEN[n-1][m] is INVALID), then
            LEN[n-1][m] = SCS_length_memoize (X[], n-1, Y[], m).
        if (LEN[n][m-1] is INVALID), then
            LEN[n][m-1] = SCS_length_memoize (X[], n, Y[], m-1).
        return 1 + MINIMUM { LEN[n-1][m], LEN[n][m-1] }.
}
```

Solution-(iv):

Iterative Bottom-up Algorithm:

```
length = SCS_length_iterative (X[], n, Y[], m)
{
    Initialize LEN[n+1][m+1].
    loop for i = 0 to n, do:
        loop for j = 0 to m, do:
            if (j == 0), then LEN[i][j] = i.
            else if (i == 0), then LEN[i][j] = j.
            else if (X[i-1] == Y[j-1]), then LEN[i][j] = 1 + LEN[i-1][j-1].
            else, LEN[i][j] = 1 + MINIMUM { LEN[i-1][j], LEN[i][j-1]) }.
    return LEN[n][m];
}
```

Time-Complexity:

There are two nested loops running a total of nm iterations and inside the innermost loop, only constant-time (say, c) operations are being carried out in worst-case. Also, there are constant-time (say, b) operations performed outside the loop. So, the overall time-complexity is formally given as,

$$T(n,m) = b + \sum_{0}^{n}\sum_{0}^{m} c = b + \sum_{0}^{n} cm = b + cnm = O(nm)$$

Space-Complexity:

Apart from the two input strings used (having $O(n+m)$ space in total) inside the algorithm, we also declare additional space of size $d.(n+1).(m+1)$ for keeping `LEN[ ][ ]` (assuming each data takes d units of storage space). Also, there are only constant variable spaces (say, a) being used. So, the overall space-complexity is formally given as,

$$S(n,m) = a + d(n+1)(m+1) = O(nm)$$

Solution-(v):

Example:

```
Gien two example strings,
  X = [ a b c b d a b ] (having length, n = 7) and
  Y = [ b d c a b a   ] (having length, m = 6)

Table LEN[8][7] will get constructed as follows:
       0     1     2     3     4     5     6
    ---------------------------------------
0   |  0  |  1  |  2  |  3  |  4  |  5  |  6  |
    ---------------------------------------
1   |  1  |  2  |  3  |  4  |  4  |  5  |  6  |
    ---------------------------------------
2   |  2  |  2  |  3  |  4  |  5  |  5  |  6  |
    ---------------------------------------
3   |  3  |  3  |  4  |  4  |  5  |  6  |  7  |
    ---------------------------------------
4   |  4  |  4  |  5  |  5  |  6  |  6  |  7  |
    ---------------------------------------
5   |  5  |  5  |  5  |  6  |  7  |  7  |  8  |
    ---------------------------------------
6   |  6  |  6  |  6  |  7  |  7  |  8  |  8  |
    ---------------------------------------
7   |  7  |  7  |  7  |  8  |  8  |  8  |  9  |
    ---------------------------------------

Therefore, the length of SCS = LEN[7][6] = 9

SCS string will get constructed in SCS_STR[8][7] as follows:
        0         1         2         3         4         5         6
    ------------------------------------------------------------------------
0   |    -    |    b    |   bd    |   bdc   |  bdca   |  bdcab  | bdcaba  |
    ------------------------------------------------------------------------
1   |    a    |   ab    |   abd   |  abdc   |  bdca   |  bdcab  | bdcaba  |
    ------------------------------------------------------------------------
2   |   ab    |   ab    |   abd   |  abdc   |  abdca  |  bdcab  | bdcaba  |
    ------------------------------------------------------------------------
3   |   abc   |   abc   |  abcd   |  abdc   |  abdca  | abdcab  | abdcaba |
    ------------------------------------------------------------------------
4   |  abcb   |  abcb   |  abcbd  |  abdcb  | abdcba  | abdcab  | abdcaba |
    ------------------------------------------------------------------------
5   |  abcbd  |  abcbd  |  abcbd  | abcbdc  | abcbdca | abdcabd |abdcabda |
    ------------------------------------------------------------------------
6   | abcbda  | abcbda  | abcbda  | abcbdac | abcbdca |abcbdcab |abdcabda |
    ------------------------------------------------------------------------
7   | abcbdab | abcbdab | abcbdab |abcbdabc |abcbdcab |abcbdcab |abcbdcaba|
    ------------------------------------------------------------------------

Therefore, the SCS string = SCS_STR[7][6] = [ a b c b d c a b a ]
```