INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (Mid Semester)										**SEMESTER (Spring)**			
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**

CS21003 Algorithms–I, Spring 2018–2019

Mid-Semester Test

21–February–2019 NR123/124/321/322/421/422, 02:00pm–04:00pm Maximum marks: 40

Write your answers in the question paper itself. Be brief and precise. Answer all questions.
If you use any algorithm/result/formula covered in the class, just mention it, do not elaborate.

Questions start from the next page.

1. An array A of n distinct integers resides in a blackbox. At any time, you can read $A[i]$ for $0 \leqslant i \leqslant n-1$. The only type of modification allowed on A is requesting the blackbox to swap $A[i]$ and $A[j]$ for any two valid indices i and j in A. Your task is to sort A. You make a copy of A to a local array, and sort your local array (by any algorithm of your choice). Call this sorted array B. This exercise starts after this. As per the rules, you cannot copy B back to A. You need to issue a sequence of swap requests to the blackbox in order to convert A to (a copy of) B. You are required to achieve this in $O(n \log n)$ time. Moreover, you are allowed to make at most $O(n)$ swap requests to the blackbox. Propose an algorithm to solve this problem. **(10)**

Solution Use a local array $pos[0 \ldots n-1]$ such that if $A[i] = B[j]$, store $pos[i] = j$. Since B is sorted, you can find the j corresponding to each i in $O(\log n)$ time by performing binary search for $A[i]$ in B. The final position of $A[i]$ is j, so if $j \neq i$, $A[i]$ must be relocated. This is carried out by swapping $A[i]$ and $A[j]$. After this, the old $A[i]$ does not need to be relocated again. But the new $A[i]$ may need to be relocated again.

1. For $i = 0, 1, 2, \ldots, n-1$, repeat:
   (a) Use binary search to locate the index j of $A[i]$ in B.
   (b) Set $pos[i] = j$.
2. Initialize $i = 0$.
3. While $i < n$, repeat:
   (a) Let $j = pos[i]$.
   (b) If $j = i$, increment i,
   else do:
      (i) Swap $A[i]$ and $A[j]$.
      (ii) Set $pos[i] = pos[j]$.
      (iii) Set $pos[j] = j$.

Notes

1. This algorithm can work without explicitly using the array $pos[\,]$.
2. If you have $A[i] = B[j]$ initially, and you store $pos[j] = i$, this means that the position j in the final A should be occupied by $A[i]$. In that case, in Step 3, you run a loop on j. Take $i = pos[j]$. If $i = j$, then $A[i]$ is occupied by the correct element, so do nothing. Otherwise, swap $A[i]$ and $A[j]$; this will bring $A[i]$ to its correct position. In both the cases, increment i.

2. Let $A = a_0 a_1 a_2 \ldots a_{n-1}$ be a string of length n. Your task is to express A as a concatenation $A = \alpha_1 \alpha_2 \ldots \alpha_k$, where each substring α_i is a palindrome,[1] and k is as small as possible. For example, the string *acbaabcaca* has two minimum decompositions into three substrings: *acbaabca c a*, and *a cbaabc aca*. In this exercise, you are required to develop a dynamic-programming algorithm to solve this problem. Answer the following two parts. Your algorithm for each part must run in $O(n^2)$ time. Do not use memoization.

(a) Build a two-dimensional table P. For $0 \leqslant i \leqslant j \leqslant n-1$, the element $P[i][j]$ should store the information whether the substring $A[i \ldots j] = a_i a_{i+1} \ldots a_j$ is a palindrome or not. Clearly mention how P is iteratively populated, and in which order (that is, in which sequence of choosing the indices i and j). **(5)**

Solution Every string of length one is a palindrome, so we set $P[i][i] = 1$ for $0 \leqslant i \leqslant n-1$. For $j = i+1$, the substring $a_i a_j$ is a palindrome if and only if $a_i = a_j$, so for $0 \leqslant i \leqslant n-2$, we set $P[i][i+1] = \begin{cases} 1 & \text{if } a_i = a_j, \\ 0 & \text{if } a_i \neq a_j. \end{cases}$ Now, suppose that $j - i \geqslant 2$. Then, $A[i \ldots j]$ is a palindrome if and only if $a_i = a_j$ and $A[i+1 \ldots j-1]$ is again a palindrome. This recursive formulation indicates that for deciding whether a string of length l is a palindrome, we need to check whether a string of length $l-2$ is a palindrome. This, in turn, prescribes that P may be filled in the diagonal-major order. A couple of other possibilities are: (i) row-major order with descending row index i (j may be chosen in any order, but the $(i+1)$-st row should be ready for populating the i-th row), and (ii) column-major order with ascending column index j (i can now be chosen in any order, but we require the $(j-1)$-st column be ready for populating the j-th column).

[1] A palindrome is a string which reads the same forward and backward. Examples: **a**, **cc**, **denned**, **racecar**, **madamimadam**.

(b) Using the table P of Part (a), solve the minimum palindrome-decomposition problem for A. **(5)**

Solution Build two one-dimensional tables S and T, where for $j = 0, 1, 2, \ldots, n-1$, the elements $S[j]$ and $T[j]$ store the information about a minimum palindromic decomposition of $A[0 \ldots j] = a_0 a_1 a_2 \ldots a_j$. More precisely, $S[j]$ stores the number of strings in a minimum decomposition of $A[0 \ldots j]$, whereas $T[j]$ stores the start index of the last substring in this decomposition. These tables can be populated as follows.

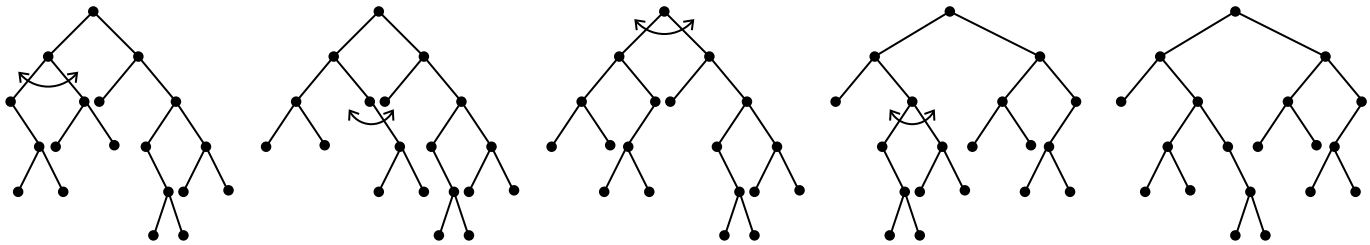
For $j = 0, 1, 2, \ldots, n-1$, repeat:
  If $P[0][j] = 1$, then do:
    Set $S[j] = 1$ and $T[j] = 0$,
  else do:
    Initialize $S[j] = \infty$.
    For $i = 0, 1, 2, \ldots, j-1$, repeat:
      If $P[i+1][j] = 1$, then do:
        Compute $s = S[i] + 1$.
        If $s < S[j]$, update $S[j] = s$ and $T[j] = i+1$.

After S and T are populated, $S[n-1]$ stores the number K of strings in a minimum palindromic decomposition of A. The substrings in this decomposition can be computed as follows. In an array $U[0 \ldots K-1]$, store the start indices of the constituent substrings.

Set $K = S[n-1]$, $k = K-1$, and $j = n-1$.
While $k \geqslant 0$, repeat:
  Set $U[k] = T[j]$ and $j = T[j] - 1$, and decrement k.
For $k = 0, 1, 2, \ldots, K-1$, repeat:
  Set $i = U[k]$ and $j = \begin{cases} U[k+1] - 1 & \text{if } k \leqslant K-2, \\ n-1 & \text{if } k = K-1. \end{cases}$
  Print $A[i \ldots j]$.

Note: Greedily choosing the maximum-length palindrome at the beginning or end will not work. Take $A =$ *ababaab*. The greedy solution (longest at beginning) is *ababa a b*, whereas the optimal solution is *aba baab*. For the longest-at-end greedy strategy, consider the reverse string: $baababa = b\ a\ ababa = baab\ aba$. This example also illustrates that the longest-first-anywhere greedy strategy will not work either.

3. Let T be a binary tree, and v a node in T. A *child swap* at v is defined as the swapping of the left-child and the right-child pointers of v. This operation swaps the entire left and right subtrees of v. We are interested only in the binary-tree structures, so we do not consider, in this context, any data stored in the nodes. Two binary trees T_1 and T_2 are called (structurally) *isomorphic* if T_1 can be converted to (a copy of) T_2 by a (finite) sequence of child-swap operations. The following figure shows the effects of four child-swap operations. All the five trees of the figure are therefore isomorphic to one another.



(a) You are given two binary trees T_1 and T_2 as input. Assume that the trees are node-disjoint, that is, the nodes of T_1 and T_2 are stored in different memory locations. Each node contains only two child pointers (but no parent pointer). For this exercise, the nodes are not required to store any other item (like a key value). Assume that T_1 and T_2 are immutable (read-only), that is, you can neither change the structures of the trees nor insert new items in the nodes. Propose an efficient algorithm to decide whether T_1 and T_2 are isomorphic. **(5)**

Solution The following recursive function accomplishes the task.

```
int isomorphic ( tree T1, tree T2 )
{
    int LL, LR, RL, RR;

    if ((T1 == NULL) && (T2 == NULL)) return 1;
    if ((T1 == NULL) || (T2 == NULL)) return 0;

    LL = isomorphic(T1->L, T2->L);
    LR = isomorphic(T1->L, T2->R);
    RL = isomorphic(T1->R, T2->L);
    RR = isomorphic(T1->R, T2->R);

    if ((LL == 1) && (RR == 1)) return 1;
    if ((LR == 1) && (RL == 1)) return 1;
    return 0;
}
```

(b) Deduce the running time of your algorithm of Part (a) in terms of m and n (in the big-O notation), where m and n are the numbers of nodes in T_1 and T_2, respectively. (**Note:** The counts m and n are not supplied as input to your algorithm. You can anyway compute them in $O(m+n)$ time, and decide *NO* if you see $m \neq n$. But the case $m = n$ does not lead you to an immediate conclusion.) **(5)**

Solution Denote the running time by $T(m,n)$. Let c denote the time taken by the non-recursive part of the algorithm plus the overhead associated with making (but not executing) four recursive calls. We prove by induction on m,n that $T(m,n) \leqslant (4mn+1)c$. The result holds for $m = 0$ or $n = 0$ [induction basis]. If both m,n are positive, we make four recursive calls, and the running time satisfies the recurrence

$$T(m,n) \leqslant T(l_1,l_2) + T(l_1,r_2) + T(r_1,l_2) + T(r_1,r_2) + c,$$

where l_i, r_i are the numbers of nodes in the left and right subtrees of T_i (for $i = 1,2$). We have $m = l_1 + r_1 + 1$, $n = l_2 + r_2 + 1$, and $(l_1 + r_1) + (l_2 + r_2) \geqslant 0$. It then inductively follows that

$$\begin{aligned} T(m,n) &\leqslant T(l_1,l_2) + T(l_1,r_2) + T(r_1,l_2) + T(r_1,r_2) + c \\ &\leqslant \big[(4l_1l_2+1) + (4l_1r_2+1) + (4r_1l_2+1) + (4r_1r_2+1)\big]c + c \\ &= 4(l_1l_2 + l_1r_2 + r_1l_2 + r_1r_2 + 1)c + c \\ &\leqslant 4(l_1l_2 + l_1r_2 + r_1l_2 + r_1r_2 + l_1 + r_1 + l_2 + r_2 + 1)c + c \\ &= 4(l_1 + r_1 + 1)(l_2 + r_2 + 1)c + c \\ &= (4mn+1)c. \end{aligned}$$

Since c is a constant, we conclude that $T(m,n) = O(m,n)$. In particular, if $m = n$, this running time is $O(n^2)$.

Note: It is tempting to express the running time as $T(n) = 4T(n/2) + O(1)$ in the case $m = n$. This recurrence has the solution $T(n) = O(n^2)$, but this expression stands for the *best case* and fails to depict the correct behavior of the algorithm.

4. You are given a string A of length n, consisting of the symbols (and) only. Your task is to find the longest balanced subsequence (not substring) of A. Here follows an example. The two outputs shown are the same (you should remove all the spaces in the output).

```
               Input:  ) () ) ( () ( () () ) ) ) ) ( ( ( ( () ) ) ( (
Output realization 1:    (  ) ( () ( () ()     ) )   (   ( () ) )
Output realization 2:    ()   ( () ( () (   ) )   ) (   ( (   ) ) )
```

Propose an O(n)-time algorithm to solve this problem using a stack. **(10)**

Solution By using a stack, we can greedily choose the right parentheses as soon as they match earlier unmatched left parentheses. We only need to know the indices of the matching left parentheses. We maintain a stack S of the indices of the left parentheses. We also maintain an array *include* to mark which symbols from the input string A are to be included in a maximum balanced subsequence.

1. For $i = 0, 1, 2, \ldots, n-1$, initialize $include[i] = 0$.
2. Initialize a stack S of integers to empty.
3. For $i = 0, 1, 2, \ldots, n-1$, repeat:

    If $A[i]$ is (, push i to S,
    else if S is not empty, do:
        Let j be the index at the top of S.
        Mark $include[i] = include[j] = 1$.
        Pop from S.

4. Ignore the indices that remain in S after the entire input is read.
5. Write in contiguous memory (output array) those symbols $A[i]$, for which $include[i] = 1$.

For the example given in the question, the output is generated as follows.

```
 Input:  ) () ) ( () ( () () ) ) ) ) ( ( ( ( () ) ) ( (
Output:    ()   ( () ( () () ) )         ( ( () ) )
```